Festschrift

LNCS 13360

Wolfgang Ahrendt
Bernhard Beckert
Richard Bubel
Einar Broch Johnsen (Eds.)

The Logic of Software

A Tasting Menu of Formal Methods

Essays Dedicated to Reiner Hähnle
on the Occasion of His 60th Birthday

Springer

Wolfgang Ahrendt · Bernhard Beckert · Richard Bubel · Einar Broch Johnsen (Eds.)

The Logic of Software

A Tasting Menu of Formal Methods

Essays Dedicated to Reiner Hähnle
on the Occasion of His 60th Birthday

Springer

Editors
Wolfgang Ahrendt
Chalmers University of Technology
Gothenburg, Sweden

Bernhard Beckert
Karlsruhe Institute of Technology
Karlsruhe, Germany

Richard Bubel
TU Darmstadt
Darmstadt, Germany

Einar Broch Johnsen
University of Oslo
Oslo, Norway

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-08165-1 ISBN 978-3-031-08166-8 (eBook)
https://doi.org/10.1007/978-3-031-08166-8

Quo Vadis Formal Methods? I can see clearly now . . .

Preface

Distinguished, multi-valued reader, welcome to this festschrift to celebrate the 60th anniversary of our dear colleague Reiner Hähnle. As your hosts, it is our pleasure to introduce the menu. With a focus on certified modularity and variability, we present you with a history-based and resource-aware, carefully curated experience. The contributions mostly draw on seasonal, local products with the occasional exotic contribution in a cooperative and hopefully bug-free manner. With this festschrift, we can accommodate most allergies with fair constraint merging, except for the abstraction allergy.

For the aperitif, we propose the liberalised adventures of Alice and a dash of dynamic and unbounded clairvoyance, bridging the gap between formal and informal knowledge. For starters, we recommend a transparent treatment of loops to incrementally validate the boundary between the verified and the unverified. This is followed by the orderly resolution of contracts, both their design and their programming. We then consider the symbolic execution of locally abstract, globally concrete semantics of eternal, adaptable and evolving railway operations with anti-links and commodious axiomatization.

The *trou normand* of this festschrift will be saturated by information flow and soundness leaks, inferring secrets by guided experiments in Re-CorC-ing with a focus on naturalness. After that, we present a many-valued symphony of partiality and abstraction, spanning from the Karlsruhe Java Verification Suite, via JML, COSTA, KeY, KIV, Stipula, Łukasiewicz logic, AF-algebras, MaxSAT and MinSAT, to Snap! As for abstraction refinement and incremental validation to enable reuse and transformation, we are afraid that selection is no longer available at this point, without explicit dependency information.

To facilitate symbolic digestion, the grand finale offers a dash of automatic complexity analysis, an injection of sound and complete reasoning about actors and a note on their idleness. By suggesting normal forms for knowledge compilation, the menu is curated to facilitate lifelong learning and to better understand research quality.

We are proud to offer to you a menu with a wide variety of ingredients and preparation styles, reflecting the great repertoire of the master chef Reiner Hähnle, a repertoire of formal methods that he developed during his long journey as a scientist, which brought him from being a young PhD student working on many-valued logics to his long tenure as an eminent researcher in formal methods. Reiner's mastery inspired and influenced many chefs in their cooking, and their commitment to the best end-user gourmet experience.

We hope you will enjoy this tasting menu!

April 2022

Wolfgang Ahrendt
Bernhard Beckert
Richard Bubel
Einar Broch Johnsen

Organization

Editors

Wolfgang Ahrendt	Chalmers University of Technology, Sweden
Bernhard Beckert	Karlsruhe Institute of Technology (KIT), Germany
Richard Bubel	Technische Universität Darmstadt, Germany
Einar Broch Johnsen	University of Oslo, Norway

Reviewers

Ole Jørgen Abusdal	Western Norway University of Applied Sciences, Norway
Elvira Albert	Universidad Complutense de Madrid, Spain
Mads Dam	KTH Royal Institute of Technology, Sweden
Ferruccio Damiani	Università di Torino, Italy
Frank De Boer	Centrum Wiskunde and Informatica, The Netherlands
Crystal Chang Din	University of Bergen, Norway
Samir Genaim	Universidad Complutense de Madrid, Spain
Jürgen Giesl	RWTH Aachen University, Germany
Dilian Gurov	KTH Royal Institute of Technology, Sweden
Marieke Huisman	University of Twente, The Netherlands
Eduard Kamburjan	University of Oslo, Norway
Alexander Knüppel	Technische Universität Braunschweig, Germany
Cosimo Laneve	University of Bologna, Italy
Gary T. Leavens	University of Central Florida, USA
Rustan Leino	Amazon Web Services, USA
Michael Lienhardt	ONERA, France
Felip Manyà	AI Research Institute (IIIA, CSIC), Spain
Wojciech Mostowski	Halmstad University, Sweden
Daniele Mundici	University of Florence, Italy
André Platzer	Carnegie Mellon University, USA
Violet Ka I Pun	Western Norway University of Applied Sciences, Norway
Aarne Ranta	Chalmers University of Technology, Sweden
Wolfgang Reif	University of Augsburg, Germany
Philipp Ruemmer	Uppsala University, Sweden
Ina Schaefer	Karlsruhe Institute of Technology (KIT), Germany
Rudolf Schlatte	University of Oslo, Norway
Bernhard Steffen	University of Dortmund, Germany
Dominic Steinhöfel	CISPA Helmholtz Center for Information Security, Germany

Volker Stolz	Western Norway University of Applied Sciences, Norway
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Mattias Ulbrich	Karlsruhe Institute of Technology (KIT), Germany
Adele Veschetti	University of Bologna, Italy

Contents

I Can See Clearly Now: Clairvoyant Assertions for Deadlock Checking

Ole Jørgen Abusdal[1], Crystal Chang Din[2], Violet Ka I Pun[1], and Volker Stolz[1](✉)

[1] Western Norway University of Applied Sciences, Bergen, Norway
{ojab,vpu,vsto}@hvl.no
[2] University of Bergen, Bergen, Norway
Crystal.Din@uib.no

Abstract. Static analysers are traditionally used to check various correctness properties of software. In the face of refactorings that can have adverse effects on correctness, developers need to analyse the code after refactoring and possibly revert their changes. Here, we take a different approach: we capture the effect of the Hide Delegate refactoring on programs in the ABS modelling language in terms of the base program, which allows us to predict the correctness of the refactored program. In particular, we focus on deadlock-detection. The actual check is encoded with the help of an additional data structure and assertions. Developers can then attempt to discharge assertions as vacuous with the help of a theorem prover such as KeY. On the one hand, this means that we do not require a specific static analyser nor theorem prover, but rather profit from the strength and advances of modern tool support. On the other hand, developers can choose to rely on existing tests to confirm that no assertion is triggered before executing the actual refactoring. Finally, we argue the correctness of our over-approximation.

Keywords: Refactoring · Deadlock detection · Active object languages

1 Introduction

Refactoring is an important software engineering activity. Current tool support in IDEs provides a broad selection of well-known refactorings. These refactorings give no guarantees as to the expected behaviour and have been known to err in this regard in the past, see [21], beyond hopefully still producing compilable code afterwards. We follow Fowler's stipulation that refactorings should preserve behaviour. This is already difficult to check before executing a refactoring at the best of times, and complete support for proving needs sophisticated frameworks such as KeY [1,2,22].

In earlier work, we have introduced assertions during refactoring [9]. These assertions capture the correctness conditions for a refactoring, and place a lighter burden on the developers, in that they do not have to provide proofs in unfamiliar, advanced, incomplete or even non-existing frameworks, but can use their

Partially supported by DIKU/CAPES project "Modern Refactoring".

tools of the trade such as tests and coverage analysis to judge whether the refactored system has been sufficiently exercised to confirm expected behaviour.

These assertions could of course in principle be discharged with program provers. In this paper, we make two contributions: first, we focus on a novel domain of refactoring for active object programs; specifically ABS [14,16][1] programs, where a direct application of well-known object-oriented refactorings potentially leads to surprising results such as deadlocks [23], and second, we present an approach where we insert assertions encoding the correctness conditions in the code *before* refactoring, such that applicability of a refactoring can be checked either *dynamically* (through testing) or *statically* (through proving).

Although, in general, proofs for these conditions can be quite involved in non-trivial settings, such as in programs with unbounded concurrency, it is our standpoint that for easy scenarios, e.g., for a statically known number of objects with a fixed communication topology, the proof-support should be sufficient.

As a proof of concept, we show how to derive the required assertions for the Hide Delegate refactoring. We are motivated by a belief that for the above class of programs we can make use of automated discharging of the assertions (or counter example derivation). For more involved programs, this should at least narrow down the scope for further investigation and guide developers to cases they have to consider before concluding that the refactoring will be correct.

A refactoring is correct and can then safely be applied if all assertions can be discharged. This approach also has the advantage that any remaining assertions will be refactored together with the program, should the developers choose to proceed with applying the refactoring. The assertions then, in the spirit of our earlier results, still serve as runtime checks: a passing assertion indicates that the subsequent synchronisation will not deadlock.

The KeY system [1] has been developed for over two decades. It started in 1998 by Hähnle et al. at Karlsruhe Institute of Technology. The original KeY system supports verification of sequential Java programs. A new version of the KeY system, i.e., KeY-ABS [7], was introduced in 2015. KeY-ABS supports symbolic execution, assertion checking and verification of history-based class invariants for concurrent ABS programs. In this paper, we present a deadlock detection framework for ABS and discuss why KeY-ABS is a suitable tool to implement this analysis approach. We also provide directions on where further effort might be a good investment in the KeY-ABS system.

The remainder of the paper is structured as follows: Sect. 2 briefly introduces the ABS language and how deadlocks can occur. Section 3 describes an assertion transformation to detect deadlocks, and Sect. 4 presents the approach to deadlock detection for *to-be-refactored* programs *before* refactoring. Section 5 discusses how we can use KeY-ABS to reason about the transformed program. Finally, we explore the related work in Sect. 6, and conclude the paper with a discussion on some limitations and future work in Sect. 7. The example presented in this paper is available as a git repository at https://github.com/selabhvl/stolz-srh60-artefact.

[1] https://abs-models.org/.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid fut \mid object \mid invoc \mid cog \mid cn\ cn & cog & ::= & cog(c, act) \\
fut & ::= & fut(f, val) & val & ::= & v \mid \bot \\
object & ::= & ob(o, a, p, q) & a & ::= & T\ x\ v \mid a,\ a \\
q & ::= & \epsilon \mid p \mid q\ q & p & ::= & \{l \mid s\} \mid \text{idle} \\
invoc & ::= & invoc(o, f, m, \overline{v}) & v & ::= & o \mid f \mid b \mid t \\
s & ::= & s; s \mid x = rhs \mid \textbf{suspend} \mid \textbf{await}\ g \mid \textbf{skip} & act & ::= & o \mid \varepsilon \\
 & \mid & \textbf{if}\ b\ \{s\}\ [\ \textbf{else}\ \{s\}\ ] \mid \textbf{while}\ b\ \{s\} \mid \textbf{return}\ e \mid \textbf{cont}(f) & & & \\
rhs & ::= & e \mid \textbf{new}\ [\textbf{local}]\ C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\textbf{get} & & &
\end{array}
$$

Fig. 1. Runtime syntax of ABS, o, f, c are identifiers of object, future, and cog

2 The ABS Language and Deadlocks

In this section, we briefly introduce the ABS language [16], the active object language that the work is based on. We will first present the runtime syntax and then we show how deadlocks can be introduced by code refactoring in the language.

2.1 The ABS Language

ABS is a modeling language for designing, verifying, and executing concurrent software. It has a Java-like syntax and actor-based concurrency model [15], which uses *cooperative scheduling* of method activations to explicitly control the internal interleaving of activities inside a concurrent object group (*cog*). A cog can be conceptually seen as a processor containing a set of objects. An object may have a set of processes, triggered by method invocations, to be executed. Inside a cog, at most one process is *active* while the others are *suspended* in the process pool of the corresponding objects.

Process scheduling is non-deterministic, but is explicitly controlled by the *processor release points* in the language. Such a cooperative scheduling ensures data-race freedom inside a cog. In addition, objects are hidden behind interfaces and all fields are private to an object. Any non-local read or write to fields must be performed explicitly through method invocations. Different cogs can only communicate through asynchronous method calls. Note that a synchronous method call to objects on a different cog will be translated to an asynchronous one that is immediately followed by a blocking **get** operation. Thus, the cog in which the caller resides will be blocked until the method returns. In contrast, synchronous calls within the same cog will only lead to transferring the control of the cog from the caller to the callee, i.e., no cog will be blocked.

The runtime syntax of ABS is presented in Fig. 1[2], where overlined terms represent a (possibly empty) lists over the corresponding terms, and square brackets [ ] denote optional elements. A configuration cn either is empty or consists

[2] We have adopted the new version of the syntax for object creation instead of the one presented in [16].

```
class Client { ...
...{ ...
    d = p.getDept();
    m = d.getManager();
    ...
  } ...
```

(a) Before

```
class Client { ...
...{ ...
    m = p.getManager();
    ...
  } ...
```

```
class Person implements PersonI {
  PersonI getManager() {
    PersonI d=this.getDept();
    PersonI tmp=d.getManager();
    return tmp;
  } ...
```

(b) After

Fig. 2. Before/after Hide Delegate

of futures, objects, invocation messages and cogs. A *future* $fut(f, v)$ has an identifier f and a value v ($\perp$ if the associated method call has not returned). An object is a term $ob(o, a, p, q)$ consisting the object's identifier o, a substitution a representing the object's fields, an active process p and a pool of suspended processes q, where a substitution is a mapping from variable names to values. A process p is idle or consists of a substitution of local variables l and a sequence of statements s, denoted as $\{l \mid s\}$. Most of the statements are standard. The statement **suspend** unconditionally suspends the active process and releases the processor, while the statement **await** g suspends the active process depending on the guard g, which is either Boolean conditions b or return tests $x?$ that evaluate to true if the value of the future variable x can be retrieved; otherwise false. The statement **cont**(f) controls scheduling when local synchronous calls complete their execution, returning control to the caller.

Right-hand side expressions *rhs* for assignments include object creation within the current cog, denoted as **new local** $C(\overline{e})$, and in a new cog, denoted as **new cog** $C(\overline{e})$, asynchronous and synchronous method calls, and (pure) expressions e. An invocation message $invoc(o, f, m, \overline{v})$ consists of the callee o, the future f to which the call returns its result, the method name m, and the actual parameter values $\overline{v}$ of the call. Values are object and future identifiers, Boolean values, and ground terms from the functional subset of the language. For simplicity, classes are not represented explicitly in the semantics, as they may be seen as static tables.

We do not further detail the syntax and semantics of ABS in this paper, but refer the readers to [16] for the complete details.

2.2 Deadlocks Introduced by Refactoring

Figure 2 presents snippets of ABS code before and after a Hide Delegate refactoring that may introduce deadlocks in an actor setting, which is described in [23].

The effect of introducing deadlocks by this refactoring can be summarised by inspecting the difference between the two sequence diagrams in Fig. 3 showing how synchronous calls change, and by considering the possible assignment of objects to cogs shown in Fig. 4.

Figure 3a shows that a Client is first communicating with Person, then with Dept, while Fig. 3b shows that the Client in the refactored program delegates invoking getManager() to Person. Assume that we have a set of at least three

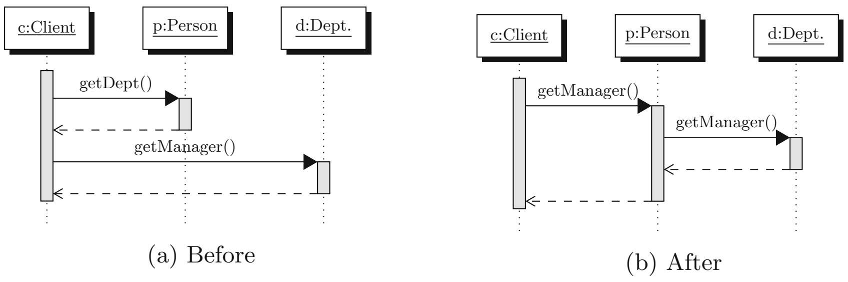


Fig. 3. Effect of Hide Delegate refactoring

objects $\{c, d, p, \ldots\}$ all placed in some cogs. We represent this information by a mapping of object identifiers to cog identifiers. A placement that is without deadlocks before, but with a deadlock after refactoring is $\{c \mapsto 1, d \mapsto 1, p \mapsto 2, \ldots\}$, i.e., objects c and d reside in a cog with identifier 1, while object p is located in a cog with identifier 2.

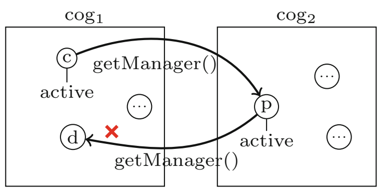


Fig. 4. A deadlock

Figure 4 depicts this placement, under which these three objects can be deadlocked. We observe that object c is blocking cog$_1$ while it is waiting for object p in cog$_2$ to complete processing getManager(), where object p in turn invokes getManager() on object d in cog$_1$. Since cog$_1$ is blocked by object c, object d will not be able to execute the method. Consequently, object p will never finish executing getManager().

2.3 A Wait-For Relation Between Cogs

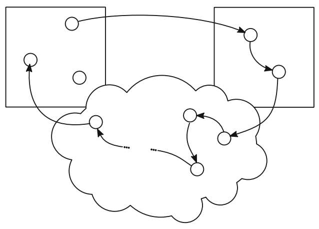


Fig. 5. A deadlocking call chain

Consider the arbitrary call chain shown in Fig. 5 and imagine traversing through the execution path resulting in this chain. Although we cannot yet determine if there exists a deadlock after the first call in the chain, we know that a synchronous call to an object on another cog will block the cog of the caller, i.e., no other object residing in the same cog can proceed. The cog of the caller will remain blocked until the called method returns. After the first call in the chain, we say that the caller cog and the callee cog are in a *wait-for* relation, i.e., the caller cog is waiting for the callee cog. We generalize the *wait-for* relation to also any blocking operation including the waiting for futures to be resolved. Thus, if an object requests the value of a future using the blocking **get** operation, we say its cog and the cog in which the future will be resolved are in a *wait-for* relation. This may be an over-approximation in the case where an **await** statement precedes the **get** operation.

Wait-for Relation and Deadlocks. For each configuration of a given ABS program, we can derive the current wait-for relation indicating if a cog is waiting for another one. A cycle in this relation indicates that there exists a deadlock involving the cogs that form the cycle. Any detection of a cycle can be done prior to any program points that contain a blocking or possibly blocking operation, which are:

– Blocking: Synchronous method calls $\mathsf{x} = \mathsf{o.f}(\overline{e})$ where the caller **this** and the callee o reside in different cogs.
– Possibly blocking: At any statement x = f.**get** irrespective of whether the caller **this** and the future f are in different cogs.

Note that although synchronous calls within the same cog in ABS do not lead to deadlocks, an asynchronous call to an object *o* residing in the same cog as the caller may lead to a deadlock, e.g., **Fut**<Unit> f = o!m(); x = f.**get**. Our analysis will correctly detect this deadlock prior to the **get** operation. However, in the case where an **await** statement precedes the **get** operation, e.g., **Fut**<Unit> f = o!m(); **await** f; x = f.**get**, our analysis will give rise to a *false positive* (see the next section for the details).

3 Program Transformation for Deadlock Checking

In this section, we introduce a general transformation mechanism to inject assertions into the program to detect deadlocks at runtime based on the *wait-for* relation.

3.1 Assertion Transformation

To perform deadlock checking on a program based on the wait-for relation in the form of runtime assertions, the relation needs to be updated along any possible call chain. Such an update requires information about the cog placement of each object, which is not explicitly available in an ABS program, but can be inferred by slightly transforming the program. Figure 6 captures such a transformation, which enables the construction of the wait-for relation in the form of a data structure (w4) and subsequently the detection of deadlocks. In the figure, we have taken some liberties for a denser presentation. We use a pseudo-syntax, e.g., **class** $\mathsf{C}(\overline{\mathsf{T\ e}})\_\{\_\{\_\}\_\}$ refers to a pattern matching any class where C corresponds to a class name. We explain in the following how the transformation is performed.

Any object creation performed through **new** will place the object in a new cog, whereas **new local** will place the object in the same cog as the one executing the constructor call. Thus, to associate every object with a cog, we modify every constructor declaration, **class** $\mathsf{C}(\overline{\mathsf{T\ e}})\_$, such that it is parametrised with

```
class C(T̄ e) _ { _ { _ } _ }
```

```
class C (CogId id, CogMap cogs, T̄ e} _ {
  _ { cogs.add(this,id); _ } _ }
```

(a) Class declaration

```
module M; _
{ _ }
```

```
module M; _
{ CogMap cogs = new CogMap();
  Rel w4 = set[];
  CogId id = cogs.fresh(); _ }
```

(b) The init block

```
T f(p̄) _
```

```
T f(Rel w4, p̄) _
```

(c) Method signatures/declarations

```
x = new C(ē);
```

```
CogId fresh = cogs.fresh();
x = new C (fresh, cogs, ē);
```

(d) Object creation in a new cog

```
x = new local C(ē);
```

```
x = new local C(id, cogs, ē);
```

(e) Object creation in the cog local to the creator object

```
x = o.g(ē);
```

```
w4 = add(w4, Cog(this), Cog(o));
assert cyclefree(w4);
x = o.g (w4, ē);
w4 = remove(w4, Cog(this), Cog(o));
```

(f) Synchronous calls

```
Fut<T> f = o!g(ē);
```

```
w4 = add(w4, Cog(this), Cog(o));
Fut<T> f = o!g(w4, ē);
cogs.add(f, Cog(o));
```

(g) Asynchronous calls

```
x = f.get;
```

```
w4 = addGet(w4, Cog(this), Cog(f));
assert cyclefree(w4);
x = f.get;
w4 = remove(w4, Cog(this), Cog(f));
```

(h) Get

Fig. 6. The assertion transformation, the notation _ is a wildcard match for the expected syntactic entity at its position.

a cog identifier and a map that links object references to cog identifiers i.e., **class** C(CogId id, CogMap cogs, $\overline{\mathsf{T\ e}}$)$_{\_}$, as shown in Fig. 6a. Additionally, in the init block of each class, we inject code to update the cog map to link any class instance to the cog identifier it receives as constructor parameter, as shown in Fig. 6b, where cogs.fresh() is a function returning a fresh cog identity. Note that the cog map is one single object in the program that all other objects, or scopes in the case of the program main block, has a reference to. As such it can dispense of the freshness requirement through also being able to emit fresh identifiers. An empty wait-for relation is also created in the init block of the program, where w4 a functional data structure capturing the wait-for relation on cog identifiers, which can be a set containing pairs of cog identifiers.

Naturally, we must modify every constructor invocation to reflect our change to constructors (see Figs. 6d and 6e). We either record a *fresh* cog identifier for the case of a constructor invocation starting with **new**, or the identifier of the cog where the object invokes **new local**. With our change to the constructor parameters of all classes, we ensure that there is a reference to the cog map in every scope for any updates to the wait-for relation (w4).

The signature of all method definitions is transformed to receive w4 as one of the parameter (see Fig. 6c). Correspondingly, all method invocations are transformed to receive the current value of w4 as the first parameter, as shown in Figs. 6f and 6g Statements are also added to update the wait-for relation. Figure 6f presents the transformation for *synchronous calls*. We first add the call chain information, represented as a pair of cog identifier $\langle Cog(\textbf{this}), Cog(o)\rangle$, to w4 before the synchronous call to object o is invoked, where the function $Cog(o)$ returns the identifier of the cog in which the object o is residing. This pair is removed after the call is made and returns. The update mechanism maintains an *irreflexive* invariant for the wait-for relation for synchronous calls by never adding a pair where $Cog(o_1) = Cog(o_2)$. The wait-for relation is handled similarly for *asynchronous calls*, as shown in Fig. 6g. For each statement **Fut**<T> f = o!g($\bar{\mathsf{e}}$), we register the future variable f in the cog map with the function cogs.add(f, $Cog(o)$), such that the **get** rule (see Fig. 6h) can later retrieve this information. Note that although the call chain information is added to w4 prior to the method invocation, this information is *not* removed because it is unclear when the call returns.

On any retrieval of values in futures, i.e., f.**get**, in Fig. 6h, we first update w4 with addGet to indicate that the current cog is waiting for the cog in which the object that will resolve the future f resides. As opposed to add, addGet does not maintain any irreflexive invariant. Once the value of the future is retrieved, the corresponding chain information is removed from w4 to indicate that the wait is over. We do not have to change the wait-for relation we are carrying forward if we encounter an **await** statement; any of our callers are still blocked and we would have a deadlock if we call back to them.

Finally, we insert the assertion **assert** cyclefree(w4) prior to every synchronous call or blocking **get** expression. This assertion checks whether or not a directed graph (a possible representation of w4) is a directed acyclic graph

```
w4 = add(w4, Cog(this), Cog(p));
assert cyclefree(w4);
m = p.getManager(w4);
w4 = remove(w4, Cog(this), Cog(p));
```

(a) Assertion at call site

```
1  class Person(CogId id, CogMap cogs, ...)
2  implements PersonI {
3   ...
4   PersonI getManager(Rel w4) {
5     PersonI d = this.getDept(w4);
6     w4 = add(w4, Cog(this), Cog(d));
7     assert cyclefree(w4);
8     PersonI tmp = d.getManager(w4);
9     w4 = remove(w4, Cog(this), Cog(d));
10    return tmp;
11  }
```

(b) Assertion in added method

Fig. 7. After applying the assertion transformation to Fig. 3b

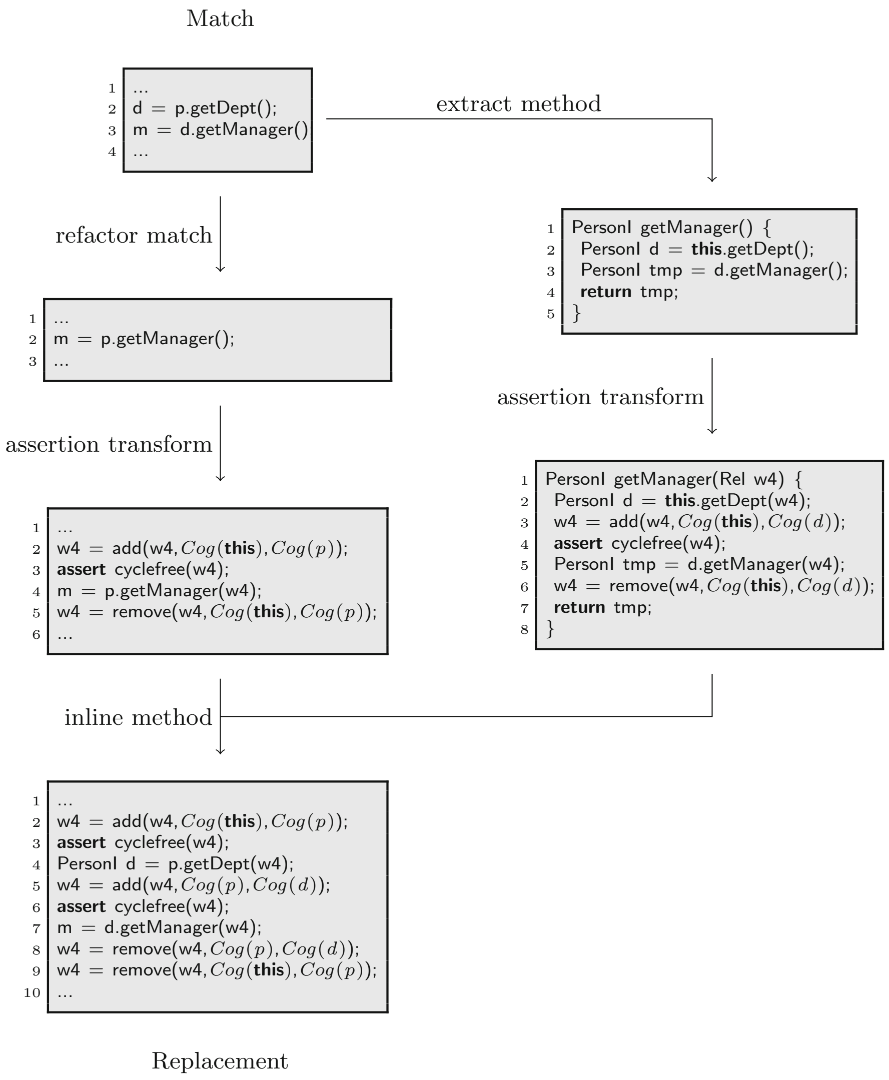
(DAG); if not, the assertion will be triggered. We remark that the statement **assert** e in ABS is equivalent to **skip** when e evaluates to true; otherwise they are equivalent to throwing an exception. This is a pitfall for us as exceptions may be caught by already present exception handling and thus interfering with deadlock detection. Our intended semantics on an assert statement that fails is to indicate that a deadlock will occur on further execution of the program. A complete transformation of a program by the rules shown in Fig. 6 is performed by repetition of any rule that matches on the original program.

Note that our treatment of asynchronous calls gives rise to *false positives*: we propagate the current call chain into an asynchronous call, although the previously recorded chain may no longer be current by the time the callee calls back into the chain (if at all). The objects in the call chain that led up to this asynchronous call may long since have become fully available again through termination of the current computation, or partially available due to an **await** statement.

3.2 Example

In this section, we are going to show the assertion transformation applied to a program resulting from a Hide Delegate refactoring and make some observations about when the assertions would detect deadlocks.

Applying the assertion transformation described in Fig. 6 to a refactored program as shown in Fig. 3b results in the code shown in Figs. 7a and 7b. Next, we are going to observe the wait-for relation in the additional method in Fig. 7b invoked through the call site seen in Fig. 7a. Let us first assume the call site is contained in an object c. Consider the sequence of calls $c \xrightarrow{getManager} p \xrightarrow{getDept} d$ where if we are at line 7 in Fig. 7b, the first call has occurred and the last call is about to occur on execution of line 8. We can see that the w4 relation at line 7 in Fig. 7b may contain the two pairs $\langle Cog(c), Cog(p)\rangle$ and $\langle Cog(p), Cog(d)\rangle$. The case where w4 is a singleton set or an empty set refers to the circumstances in which an object calls either itself or another object residing in the same cog because a cog never waits for itself. If we observe the w4 relation we see that



```
...
d = p.getDept();
m = d.getManager()
...
```

```
PersonI getManager() {
 PersonI d = this.getDept();
 PersonI tmp = d.getManager();
 return tmp;
}
```

```
...
m = p.getManager();
...
```

```
PersonI getManager(Rel w4) {
 PersonI d = this.getDept(w4);
 w4 = add(w4, Cog(this), Cog(d));
 assert cyclefree(w4);
 PersonI tmp = d.getManager(w4);
 w4 = remove(w4, Cog(this), Cog(d));
 return tmp;
}
```

```
...
w4 = add(w4, Cog(this), Cog(p));
assert cyclefree(w4);
m = p.getManager(w4);
w4 = remove(w4, Cog(this), Cog(p));
...
```

```
...
w4 = add(w4, Cog(this), Cog(p));
assert cyclefree(w4);
PersonI d = p.getDept(w4);
w4 = add(w4, Cog(p), Cog(d));
assert cyclefree(w4);
m = d.getManager(w4);
w4 = remove(w4, Cog(p), Cog(d));
w4 = remove(w4, Cog(this), Cog(p));
...
```

Fig. 8. Clairvoyant assertion construction

$Cog(c) \neq Cog(d)$ must be satisfied; otherwise we have a deadlock. This will also ensure that the assertion in line 7 will not be triggered. An important take-away from this example is not so much the former equation, but that we can record the sequence of updates to the w4 relation. For a detailed discussion of all possible object-to-cog allocations for this example see [23].

P	P_A	P_{HD}	$P_{A \circ HD}$	P_{CA}

`C.m(...) {`
⋮
`x = y.f();`
`z = x.g();`
⋮
`}`
	`C.m(...) {`
	⋮
	`w4=add(w4,this,y);`
	`assert cf(w4);`
	`x=y.f(w4);`
	`w4=rem(w4,this,y);`
	`w4=add(w4,this,x);`
	`assert cf(w4);`
	`z=x.g(w4);`
	`w4=rem(w4,this,x);`
	⋮
	`}`
		`C.m(...) {`
		⋮
		`z=y.g();`
		⋮
		`}`
		`C'.g() {`
		`x=this.f();`
		`t=x.g();`
		`return t;`
		`}`
			`C.m(...)`
			⋮
			`w4=add(w4,this,y);`
			`assert cf(w4)`
			`z=y.g(w4);`
			`w4=rem(w4,this,y);`
			⋮
			`}`
			`C'.g() {`
			`x = this.f(w4);`
			`w4=add(w4,this,x);`
			`assert cf(w4);`
			`t = x.g(w4);`
			`w4=rem(w4,this,x);`
			`return t;`
			`}`
				`C.m(...) {`
				⋮
				`w4=add(w4,this,y);`
				`assert cf(w4);`
				`x = y.f(w4)`
				`w4=add(w4,y,x);`
				`assert cf(w4);`
				`z = x.g(w4);`
				`w4=rem(w4,y,x);`
				`w4=rem(w4,this,y);`
				⋮
				`}`

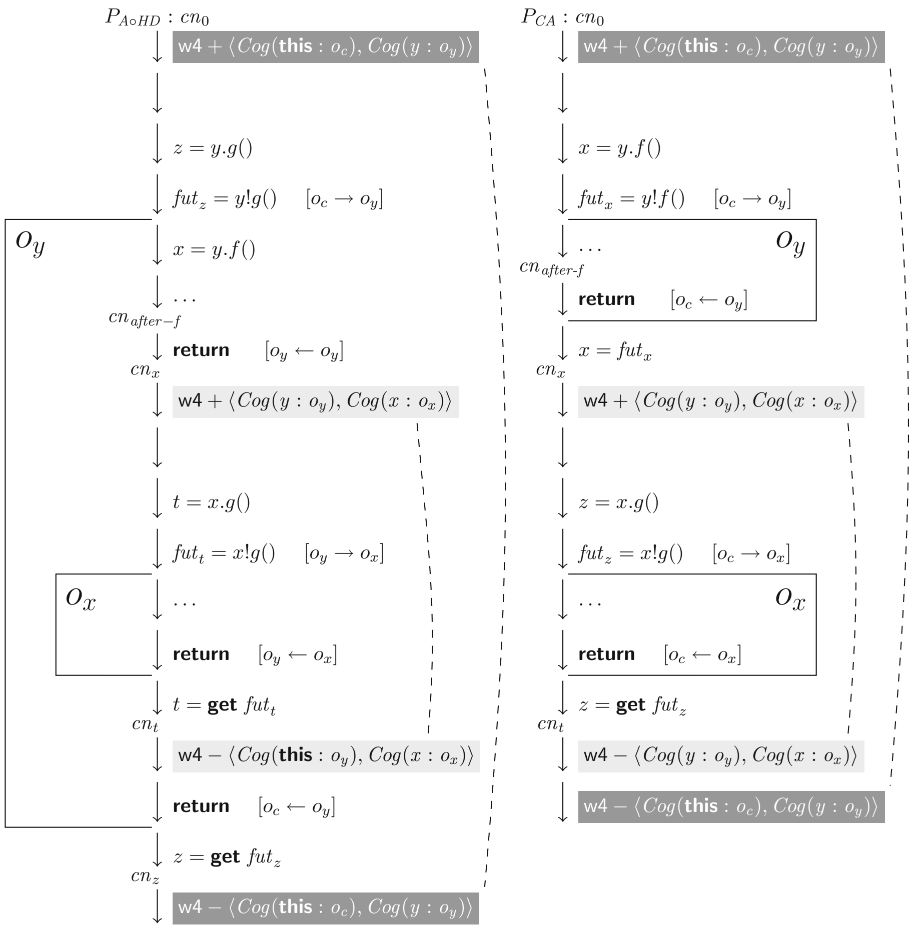
Fig. 9. Effects of the different transformations on P

4 Clairvoyant Assertions

Instead of checking if a program may deadlock using the assertion transformation *after* applying a Hide Delegate refactoring, we can produce a *clairvoyant assertion* transformation for Hide Delegate. It constructs assertions and a modification of the wait-for relation such that it predicts occurrences of deadlocks in a refactored program. We define a *clairvoyant assertion* transformation that is almost identical to the assertion transformation from Fig. 6 with one exception: Instead of applying Hide Delegate refactoring, the method calls in Fig. 8 are handled differently. Normally, the call would be transformed into the code shown in Fig. 7a by rule Fig. 6f, but it is instead transformed into the replacement code shown at the end of the derivation shown in Fig. 8. This clairvoyant assertion transformation will introduce an assertion that predicts whether the Hide Delegate refactoring when applied to the program will introduce a deadlock.

Figure 9 shows the effect of the different transformations, where P refers to a program admissible for the Hide Delegate refactoring, P_A the program after the assertion transformation is applied to P, P_{HD} the refactored version of P, $P_{A \circ HD}$ the program after the assertion transformation is applied to P_{HD}, and P_{CA} the program after the clairvoyant assertion transformation is applied to P. Names have been shortened, e.g., `cf` is the `cyclefree` function.

Equivalence Between $P_{A \circ HD}$ ***and*** P_{CA}**.** In the following, we informally argue that the effects in $P_{A \circ HD}$ and in P_{CA} wrt the injected assertions are the same, by showing that the wait-for relation is the same at the end of the execution in both programs. Different allocations of objects to cogs will give rise to different executions in a program. In the particular execution shown in Fig. 10, the synchronous calls in P_{CA} (and hence P) are always remote (every synchronous call



$P_{A\circ HD} : cn_0$
w4 + ⟨Cog(**this** : o_c), $Cog(y : o_y)$⟩
$z = y.g()$
$fut_z = y!g()$ $[o_c \rightarrow o_y]$
O_y
$x = y.f()$
$\ldots$
$cn_{after-f}$
return $[o_y \leftarrow o_y]$
cn_x
w4 + ⟨$Cog(y : o_y)$, $Cog(x : o_x)$⟩
$t = x.g()$
$fut_t = x!g()$ $[o_y \rightarrow o_x]$
O_x
$\ldots$
return $[o_y \leftarrow o_x]$
t = **get** fut_t
cn_t
w4 − ⟨Cog(**this** : o_y), $Cog(x : o_x)$⟩
return $[o_c \leftarrow o_y]$
z = **get** fut_z
cn_z
w4 − ⟨Cog(**this** : o_c), $Cog(y : o_y)$⟩
$P_{CA} : cn_0$
w4 + ⟨Cog(**this** : o_c), $Cog(y : o_y)$⟩
$x = y.f()$
$fut_x = y!f()$ $[o_c \rightarrow o_y]$
$\ldots$
O_y
$cn_{after\text{-}f}$
return $[o_c \leftarrow o_y]$
$x = fut_x$
cn_x
w4 + ⟨$Cog(y : o_y)$, $Cog(x : o_x)$⟩
$z = x.g()$
$fut_z = x!g()$ $[o_c \rightarrow o_x]$
$\ldots$
O_x
return $[o_c \leftarrow o_x]$
z = **get** fut_z
cn_t
w4 − ⟨$Cog(y : o_y)$, $Cog(x : o_x)$⟩
w4 − ⟨Cog(**this** : o_c), $Cog(y : o_y)$⟩

Fig. 10. Equivalence between $P_{A\circ HD}$ and P_{CA} (one execution).

will be translated into an asynchronous call followed by a blocking **get** operation [16]), i.e., the caller and the callee are always residing in different cogs. This implies that we are in one of two possible scenarios: all three objects are in their own cogs, or o_x and o_y are in the same cog. As the execution in P_{CA} uses a remote call from o_y to o_x, it becomes clear that they must be in different cogs, and we find ourselves in the first of the above two possibilities.

We show on the left an execution from the program $P_{A\circ HD}$ and on the right a corresponding execution of the P_{CA}, where the executions start from some state cn_0. Note that for simplicity, the transitions with respect to method binding and object scheduling are not shown in the figure. Due to the strong concurrent semantics of ABS, we also do not have to consider any interleavings. In the figure, we use the operators $+$ and $-$ to manipulate the wait-for relation (w4), where + denotes the addition of a pair of cog identifiers to w4, while − denotes the removal. We also use $var : o$ to indicate in the manipulations the object identity o to which a variable var refers. For method calls, we annotate the caller and callee objects using $[o_{caller} \rightarrow o_{callee}]$; whereas for returns, we use $[o_{caller} \leftarrow o_{callee}]$ to represent that the method call on the called object terminates and returns back to the caller. Additionally, we indicate the object context the former implies graphically.

This diagram allows us to give the proof idea outlining why the assertions in P_{CA} will always coincide with those in $P_{A \circ HD}$. If we can show that the actual parameters of all w4-manipulations are identical, and that the states of our configurations are equivalent at the end of the refactored code, since the expressions for each pair of assertions are identical, we know that they will give identical results. Although the execution of the refactored program can be different, in a very restricted manner, from that of the original one, they behave equivalently wrt to the w4-relation, which will allow us to draw the necessary conclusions.

As an induction hypothesis, we assume that we have equivalent initial states; and we will see that the same holds for the final states in the end. We note that this is essentially part of the proof that establishes that the equivalence relation $\equiv_{\mathcal{R}}$ between configurations [23] holds between the original program and the refactored program after applying Hide Delegate.

Assuming this, it is obvious that the first assertion checking after the w4-manipulation (or w4-test) uses identical arguments in the respective cn_0-configurations. When both executions reach their respective $cn_{after\text{-}f}$, it is obvious that either has only exactly executed the method `f()` in object o_y. Hence, when they reach cn_x, the variables `x` in object o_y and the one in object o_c have the same value. From this, we conclude that the next (light gray) w4-test again receives identical objects. Next, either program executes method `g()` on object o_x. Correspondingly, in configurations cn_t, local variables z and t refer to the same value. As `x` and `y` remain unchanged, identical information is removed from w4 in either case (light gray). When control returns in $P_{A \circ HD}$ to o_c, `z` has the same value as `t` before, and hence as `z` in P_{CA}. That means the object states have evolved identically in either execution. The final manipulation of w4 (dark gray) is therefor also identical.

5 KeY-ABS

KeY-ABS [7] is a deductive verification system for the concurrent modelling language ABS [14,16]. It is based on the KeY theorem prover [1,2]. KeY-ABS provides an interactive theorem proving environment and allows one to prove properties of object-oriented and concurrent ABS models. The concurrency model of ABS has been carefully engineered to admit a proof system that is modular and permits to reduce correctness of concurrent programs to reasoning about sequential ones [4,8]. The deductive component of KeY-ABS is an axiomatisation of the operational semantics of ABS in the form of a sequent calculus for first-order dynamic logic for ABS (ABSDL). The rules of the calculus that axiomatise program formulae define a symbolic execution engine for ABS.

Specification and verification of ABS models is done in KeY-ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For a sequence of executable ABS statements s and ABSDL formulae P and Q, the formula $P \rightarrow [s]Q$ expresses: If the execution of s starts in a state where the assertion P holds and the program terminates normally, the assertion Q holds in the final state. Verification of an ABSDL formula proceeds by symbolic execution of s, where state modifications are handled by the update mechanism [2]. An expression of the form $\{u\}$ is called an update application, in which u can be an elementary update of the form $a := t$ which assigns the value of the term t to the program variable a, it can also be a parallel update $u_1 \parallel u_2$ that executes the subupdates u_1 and u_2 in parallel. The semantics of $\{u\}x$ is that an expression x is to be evaluated in the state produced by the update u (the expression x can be a term, a formula, or another update). Given an ABS method m with body mb and a class invariant I, the ABSDL formula $I \rightarrow [mb]I$ expresses that the method m preserves the class invariant. Note that the method body mb may contain **assert** statements. KeY-ABS is able to discharge assertions as vacuous (never fire) or show the open proof at such assertion statements. In ABS, the later one is equivalent to throwing an exception. If the proof can be closed at all **assert** statements, it shows that none of the assertions can be violated.

In this work, we focus on deadlock detection. **assert** statements are added before each of the synchronous calls and are used to predict if the refactored version may cause deadlock while the corresponding synchronous calls are invoked. Since each synchronous call has its own call cycle, it is more suitable to express deadlock cycle in assertion conditions than in class invariants. Consequently, we do not consider the use of class invariants in this setting but assertions. Since the assertion depends on the concrete value of the additional `w4` parameter to each method, the required reasoning propagates backwards to call-sites. Eventually this propagation or a proof attempt can result in a contradiction, which indicates a concrete deadlock, although this may be on an infeasible program path. It is then the task of the user to prove this infeasibility, or accept the risk and, for example, resort to testing.

Below is the proof rule for **assert** statement in KeY-ABS.

$$\frac{\Gamma \Longrightarrow \{u\}e = \mathit{true} \qquad \Gamma, \{u\}e = \mathit{true} \Longrightarrow \{u\}[s]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\textbf{assert}\ e; s]\phi, \Delta}$$

where Γ and Δ stand for (possibly empty) sets of side formulae. The expression e in the **assert** statement is evaluated according to the update u. The remaining program s can only be verified when the assertion is evaluated to true, i.e., the assertion is not fired. The predicate ϕ is the postcondition of the method and should be proven upon method termination.

6 Related Work

Using theorem provers to discharge assertions is not new. We rely on this existing feature of KeY-ABS, and other comparable tools such as ESC/Java [5,10] deal with them similarly with varying degrees of automation. An alternative to KeY is Crowbar [18]. Also here we would have to rely on being able to evaluate circularity-queries in the functional fragment of ABS as part of the proof as just like JML the Behavioral Program Logic is not expressive enough to treat them on the level of specifications.

Our encoding in additional data that is only relevant on the specification level is an instance of model variables [6] that model data that goes beyond abstracting the current state. Here, we have introduced data into the program that is intended to be primarily used for reasoning purposes, although they double as concrete program variables for the purpose of runtime checking (a failing assertion indicates an upcoming potential deadlock).

Encoding a static analysis within a theorem proving framework has, to the best of our knowledge, only been done as a proof of concept by Gedell [11]. While his approach also targets the KeY system, the encoding is not in the form of additional data (and properties) thereof in the original program, but as a data structure within the KeY system and an extension of the proof rules for the various syntactic constructs of Java supported by KeY. This has the advantage that no modification of the code is necessary, but requires deeper understanding of the prover framework for the development of the corresponding tactlets. An advantage of our approach is that we are independent from the prover as we embed ourselves within the target language.

That static analysis can in general benefit from relying on theorem provers has already been observed by Manolios and Vroon in [19]. They invoke the ACL2 theorem prover in a controlled manner such that it can be used as a black box when analysing termination. A timeout from the prover gives rise to over-approximation in the static analysis.

The work by Giachino et al. [12] uses an inference algorithm to extract abstract descriptions of methods to detect deadlocks in Core ABS programs, whereas our work captures the potential circular dependencies between cogs by means of a wait-for relation, indicating which method invocations may contribute to deadlock behaviour. Similar to our work, Giachino's approach also over-approximates the occurrence of deadlocks. Albert et al. [3] have developed a comprehensive static analysis based on a may-happen-in-parallel analysis for a very similar language that does not support object groups but treats each object as a singleton member in its own group. They later combined this with a dynamic testing technique that reduces false positives [13].

The work of Kamburjan [17] presents a notion of deadlock for synchronisation on arbitrary boolean conditions in ABS. It supports deadlock detection on condition synchronisation and synchronisation on futures, but it does not consider the cases caused by synchronous method calls as targeted in our work.

Quan et al. formalize refactorings by encoding them as refinement laws in the calculus of refinement of component and object-oriented systems (rCOS) and prove these correct [20], however they do not use a theorem prover and they do not consider concurrent programs.

7 Conclusion

In this paper, we introduce a dynamic deadlock detection mechanism through a program transformation that uses a dependency relation such that assertions can discern deadlocks through inspection of the relation. From the aforementioned transformation we derive a new transformation that manipulates the dependency relation to introduce *clairvoyant* assertions; they predict whether a Hide Delegate refactoring will introduce deadlocks. We argue that in principle our dynamic deadlock detection could be statically discharged by a deductive verification system through resolving the proof goals generated by showing that no assertions trigger.

Discussion and Future Work

While the encoding is certainly useful for runtime checks, using the proof strategies of KeY-ABS to discharge the assertion can be more effective as the proofs cover all the possible execution paths at once.

As a language with interface-based inheritance, it should be clear from the fragments of the transformed ABS programs in Fig. 9 that for every method call there is uncertainty as to which class we are calling into, if the declared type of the callee has more than one implementation. If the classes implementing the same interface have incompatible behaviour, and e.g., only one of the classes will be used at run time, it is again up to the user to provide evidence to the theorem prover that this is the case (and hence eliminate the other classes at this call site). This is however not a particular issue of our approach, but a recurring theme in use of the KeY system, both for ABS and for Java.

We also note that the current version of KeY-ABS does not support **new local**, which is not a problem for the proof, since we explicitly encode the mapping from objects to cogs in the program.

The ABS language does not support object mobility. Integrating this poses a major challenge, since this operation is essentially a side-effect which would mean we would have to give up our model of the deadlock-relation as a purely functional structure. The same will be true for correctly accounting for **await** calls that allow other objects in the same group to make progress concurrently.

To address the shortcoming of false positives (we cannot complete a proof, yet all counterexamples are spurious) in the case where we would need a static analysis to propagate information about the subsequent code backwards into asynchronous calls, we plan to investigate an encoding that uses an oracle in the target language, which ressembles the equivalent encoding of a more precise static analysis in the domain of the prover. It is our goal to remain firmly independent from any particular prover to do our part on encouraging a lively competition between provers.

Clearly working with the code augmented by our assertions has disadvantages for developer in terms of readability. Ideally, such manipulation should be done behind the scenes, preferably in a different view of the model. A natural combination would be to use existing static and dynamic techniques in a first phase, and discharge any assertions that e.g. are not part of a deadlock-cycle reported by this tools.

The clairvoyant assertions introduced here are specific to the Hide Delegate refactoring. Variations will be necessary to predict negative effects of other refactorings, such as other constellations of deadlocks [23]. We have as yet to implement an automated assertion generation to try out our idea and gauge the currently feasible amount of automation.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives. LNCS, vol. 12345. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6
3. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. ACM Trans. Comput. Log. **17**(2), 11:1–11:39 (2016). https://doi.org/10.1145/2824255
4. Bubel, R., Montoya, A.F., Hähnle, R.: Analysis of executable software models. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 1–25. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_1
5. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_16
6. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.H.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**(6), 583–599 (2005). https://doi.org/10.1002/spe.649
7. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
8. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2014). https://doi.org/10.1007/s00165-014-0322-y
9. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 517–531. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_36
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 234–245. ACM (2002). https://doi.org/10.1145/512529.512558
11. Gedell, T.: Embedding static analysis into tableaux and sequent based frameworks. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 108–122. Springer, Heidelberg (2005). https://doi.org/10.1007/11554554_10

12. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. **15**(4), 1013–1048 (2016). https://doi.org/10.1007/s10270-014-0444-y
13. Gómez-Zamalloa, M., Isabel, M.: Deadlock-guided testing. IEEE Access **9**, 46033–46048 (2021). https://doi.org/10.1109/ACCESS.2021.3065421
14. Hähnle, R.: The abstract behavioral specification language: a tutorial introduction. In: Giachino, E., Hähnle, R., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 1–37. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40615-7_1
15. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 235–245. Morgan Kaufmann Publishers Inc. (1973). http://dl.acm.org/citation.cfm?id=1624775.1624804
16. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
17. Kamburjan, E.: Detecting deadlocks in formal system models with condition synchronization. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **76** (2018). https://doi.org/10.14279/tuj.eceasst.76.1070
18. Kamburjan, E., Scaletta, M., Rollshausen, N.: Crowbar: behavioral symbolic execution for deductive verification of active objects. CoRR abs/2102.10127 (2021). https://arxiv.org/abs/2102.10127
19. Manolios, P., Vroon, D.: Integrating static analysis and general-purpose theorem proving for termination analysis. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006), pp. 873–876. ACM (2006). https://doi.org/10.1145/1134285.1134438
20. Quan, L., Zongyan, Q., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 323–338. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_23
21. Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Trans. Softw. Eng. **39**(2), 147–162 (2013). https://doi.org/10.1109/TSE.2012.19
22. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 319–336. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_20
23. Stolz, V., Pun, V.K.I., Gheyi, R.: Refactoring and active object languages. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12477, pp. 138–158. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_9

When COSTA Met KeY: Verified Cost Bounds

Elvira Albert[1,2], Samir Genaim[1,2](✉), Alicia Merayo[1], and Guillermo Román-Díez[3]

[1] Complutense University of Madrid, Madrid, Spain
sgenaim@ucm.es
[2] Instituto de Tecnología del Conocimiento, Madrid, Spain
[3] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. COSTA is an automatic resource analysis tool that given an input Java bytecode program, and a selection of a cost measure, returns an upper bound on the execution cost of the program as a parametric function on the program's input data. KeY is a deductive verification system for Java programs that accepts specifications written in the Java Modeling Language that are transformed into theorems of dynamic logic and then compared against program semantics. COSTA and KeY met 10 years ago during the curse of the EU HATS project. This encounter started up a fruitful collaboration between the two teams on the verification of upper bounds. The initial work within HATS focused on the verification in KeY of cost bounds obtained by the COSTA tool for sequential Java programs using only Integer data. A notable result of the COSTA+KeY cooperation has been that KeY spotted a bug in COSTA, as it failed to prove correct one invariant which was incorrect and provided a concrete counterexample that helped understand, locate and fix the bug. The work was extended later to handle also sequential Java programs working on heap-allocated data. The latest work after HATS has brought these ideas further to the general context of abstract programs which contain placeholder symbols to represent instantiations into concrete programs. This invited paper gives an overview of the four papers jointly written with Reiner and his team on the subject and discusses related work on the matter.

1 Introduction and Overview

One of the most important aspects of a program is its efficiency, i.e., its resource consumption (e.g., amount of execution time, memory allocated, number of instructions executed). The inference of bounds on the resource consumption of programs is a very active field of research since its inception in the seminal work of Wegbreit [33]. Bounding the resource consumption allows ensuring that the amount of resources required to execute the program will never

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31 and by the CM projects S2018/TCS-4314 co-funded by EIE Funds of the European Union.

exceed the inferred upper bound. Depending on the considered resource, this means that the execution of the program will not exceed the upper bound time, or will not use more than the inferred upper bound memory, etc. There are resource analysis tools that are able to infer automatically such upper bounds for today's programming languages (see, e.g., for Integer programs [14,20,29], for Java-like languages [3,21], for concurrent and distributed languages [23], for functional programs [22], for probabilistic programs [26,27]). COSTA [4] is one such resource analyzer that initially was developed for Java bytecode programs. In its input, COSTA takes a compiled Java bytecode, a selection of a cost model among number of executed instructions, amount of memory allocated, or calls to functions, and it returns in the output an upper bound of the resource consumption of the program for the selected cost model.

At the start of the HATS project[1], the COSTA team had manually written soundness proofs for the techniques used in the analysis [2,5]. Basically, the proofs guarantee that (theoretically) the results are correct, i.e., no execution of the program will ever exceed the inferred upper bound. However, the implementation undertaken when implementing the COSTA tool could be buggy, and hence using the inferred bounds for safety critical purposes could be risky. When COSTA [4] met KeY [12] in the context of the HATS project, we had a unique opportunity to combine the strengths of both systems and use KeY to formally verify the soundness of the inferred bounds. The cooperation of COSTA and KeY allowed us to mitigate the aforementioned risks, and have formal guarantees that the results inferred by the tool are correct. For the first time, we could generate *verified upper bounds*.

Our first joint work with Reiner and Richard, published in the Proceedings of PEPM'11 [6], established the basis for the combination of both systems on the simpler setting of sequential Java programs using only Integer data, i.e., we left out concurrency and also heap-allocated data structures. We proposed to formally verify in KeY the information inferred by each of the subcomponents used to produce upper bounds, as their composition in a final step is straightforward. Section 2.2 describes in detail the verification of such three main subcomponents:

1. The ranking functions [2] that provide upper bounds on the number of iterations that loops perform. COSTA infers linear ranking functions that are translated into `decreasing` annotations that KeY can automatically prove correct.
2. The size relations [5] which capture the effect of executing code fragments (of different forms) on program variables and are in the form of linear constraints among the values of variables at different program points. They lead to `assert` annotations to be proven by KeY.
3. The loop invariants that COSTA infers [2] that relate the loop's variables and their initial values, which are transformed into `loop-invariant` annotations whose soundness needs to be formally verified by KeY.

The next natural step was to extend the work to handle heap-allocated data and this way have verified cost bounds for most real sequential Java programs.

[1] https://hats.se.informatik.tu-darmstadt.de/.

This extension was published in a next paper that appeared in the proceedings of FASE'12 [8] in which we presented the new components needed for the verification of bounds for heap-allocated programs. Section 2.3 will give an overview on such extension which essentially needs to:

1. Use a size analysis for heap-allocated data structures that allows us to reason on how their sizes are modified along the execution of the program. We use the path-length analysis [30] for this purpose.
2. Soundness of the path-length analysis requires proving new annotations over the heap-allocated structures, namely, we need to prove complex properties such as acyclicity, reachability and disjointness of heap regions. Annotations for such properties are generated by COSTA and formally verified by KeY.

Together with Reiner and Richard, we also published a journal paper in the *Journal of Software and Systems Modeling* [7] that unifies the two conference papers and extends the experimental evaluation undertaken to assess the effectiveness and efficiency of the implementation. This successfully completed the work of verified upper bounds in the HATS project.

It was only two years ago that COSTA and KeY met again at the Formal Methods conference in Porto. The KeY team was presenting their new work on Abstract Execution [31], a technique which allows verifying properties of *abstract* programs. Abstract programs contain placeholders for specifying parts of the program that have been extracted away so that one considers a more general context. The placeholders can be instantiated with concrete statements and the verification proofs should be valid for the concrete instantiations as well. The original work [31] was applied to prove *functional* properties of abstract programs and in particular to prove functional equivalence of program transformation techniques. We started to discuss in Porto if it would be possible to leverage the technique to prove *quantitative* properties of programs as well. A new joint work published in the Proceedings of FASE'21 [10], this time written with Reiner and Dominic, proposes such an extension under the name: *Quantitative Abstract Execution.*

Section 3 will describe the main ideas of Quantitative Abstract Execution, which have brought in two important extensions:

1. Leveraging the cost analysis of COSTA to work with abstract Java programs. We have redefined the basic notions of a cost analysis for this abstract setting. In particular, the notion of recurrence equations used in cost analysis needs to be adapted, and the upper bounds generated from them will be abstract as well.
2. The verification in KeY has been enabled by means of the notion of cost invariant, which is an inductive expression that defines the cost of loops at each of their iterations. This inductive notion of cost perfectly fits with the abstract execution framework, since it permits compositional proofs about the cost.

We wish that COSTA and KeY will meet many more times and that our fruitful collaboration with Reiner will continue in the coming years!

2 Key+COSTA for Concrete Programs

This section describes how KeY can be used to verify the resource analysis bounds obtained by COSTA for concrete Java programs. This section is organized as follows: Sect. 2.1 describes the different information used by COSTA to compute upper bounds; Sect. 2.2 describes how a Java program, whose resource consumption only depends on local integer data, can be annotated with this information using the Java Modeling Language (JML); and Sect. 2.3 describes new annotations required to model the heap properties needed for handling programs whose cost depends on information stored in the heap memory.

2.1 Resource Analysis: Upper Bounds

We start by describing the information computed by COSTA in the process of inferring an upper bound on the resource consumption of a given program. W.l.o.g. we focus on polynomial upper bounds, but the same information is used to infer logarithmic and exponential upper bounds as well. The computation of the upper bound is performed by considering the *scopes* found in the program, i.e., a code fragment that either corresponds to a loop or to a code fragment before or after a loop. The computation of the upper bound for the whole program is performed by analyzing each scope separately, and then composing the results into a closed-form function expressed in terms of the input parameters of the program. COSTA computes the following information for each scope:

(1) To guarantee termination of each loop COSTA infers a corresponding *ranking function* of the form $\mathsf{nat}(f(\overline{x}))$, where $\mathsf{nat}(v) = \max(0, v)$, such that it is decreasing by 1 at each iteration and is non-negative by definition. Thus, assuming that $\bar{x}_0$ are the initial values of the variables involved in the ranking function, $\mathsf{nat}(f(\overline{x}_0))$ bound the number of iterations. Note that COSTA only supports linear ranking functions, i.e., $f(\bar{x})$ is of the form $a_0 + a_1x_1 + \cdots + a_nx_n$ where each a_i is rational number.

(2) For each scope, COSTA computes *size relations*, denoted by φ, that capture the effect of executing some code fragments on program variables. COSTA models such relations using linear constraints among the (values of) variables at different program points. Concretely, our size relations relate the values of the variables at a certain program point of interest within a scope to their initial values when entering the scope. In addition, COSTA computes *input-output size relations* for the methods, which relate the values of a method parameters to its return value, this is useful to model the effect of a method call. Similarly, this is done also for loops where in this case size relations play the role of loop summaries.

(3) For each loop, using the size relations, COSTA infers a *loop invariant* that relates the values of the loop's variables at each iteration to their initial values (just before entering the loop). The loop invariant is a disjunction between two conjunctions of linear constraints $\psi \equiv \psi^o \vee \psi^n$ where ψ^o corresponds to the first visit to the loop, and ψ^n corresponds to visiting the

loop after executing the loop body at least once (this separation is needed for precision issues). These loop invariants, together with the size relations, are needed to compute the worst-case cost of executing one loop iteration.

Given a Java program and the corresponding information (as described above) that is used to compute the upper bound, COSTA annotates the Java source code with corresponding JML annotations, and then passes it to KeY in order to verify the correctness of all these annotations, and thus the correctness of the corresponding information inferred by COSTA.

Example 1. Consider the Java method depicted in Fig. 1, which corresponds to an implementation of the *Bubble Sort* algorithm. We will use it to show how COSTA annotates the Java code with JML annotations to express the different information used to compute an upper bound for method `bubbleSort`. Note that the call to method `increment` at Line 24 (L24 for short) simply increments the value of variable `i` by 1, and we do it in a separate method to explain why *input-output size relations* are needed. The upper bound of method `bubbleSort` is computed as follows:

1. For the inner loop, COSTA computes size relations, a loop invariant, and the ranking function $\mathsf{nat}(n-j-i)$. Assuming that the cost of executing the loop body once is $c_4 + c_{inc}$, where c_{inc} is the cost of method `increment` and c_4 is the cost of the rest of the instructions, and that the cost of the evaluation of the loop condition is c_3, the upper bound corresponding to the inner loop is $\mathsf{UB}_{in} = c_3 + (c_4 + c_{inc}) * \mathsf{nat}(n - j - i)$.
2. For the outer loop, COSTA computes size relations, a loop invariant, and the ranking function $\mathsf{nat}(n - j)$. Assuming that the cost of executing the outer loop body once is $c_5 + c_{in}$, where c_{in} represents the worst-case cost of the inner loop and c_5 the cost of the rest of the instructions, and that the cost of the evaluation of the loop condition is c_2, the upper bound corresponding to the outer loop is $\mathsf{UB}_{out} = c_2 + (c_{in} + c_5) * \mathsf{nat}(n - j)$.
3. The next step replaces c_{in} in UB_{out} by an expression in terms of the program variables, which is done by maximizing the cost expression UB_{in} in the context of the outer loop. Note that the maximum value of $n-j-i$ in this context (in terms of the initial values of the variables) is n, which happens when $i = 0$ and $j = 0$, and thus c_{in} is replaced by $c_3 + (c_4 + c_{inc}) * \mathsf{nat}(n)$ resulting in $\mathsf{UB}_{out} = c_2 + (c_3 + (c_4 + c_{inc}) * \mathsf{nat}(n) + c_5) * \mathsf{nat}(n - j)$.

Finally, we rewrite UB_{out} in terms of the method parameters. This is done by using the size relations of the first block of the method to maximizes UB_{out} in terms of the input parameters, resulting in the following upper bound for the method: $\mathsf{UB}_{bs} = c_1 + c_2 + \mathsf{nat}(n) * (c_3 + (c_4 + c_{inc}) * \mathsf{nat}(n)) + c_5 * \mathsf{nat}(n)$. Note that c_1 corresponds to the cost of the code fragment before the outer loop.

```
 1 public void bubbleSort(int arr[], int n) {
 2
 3   // @ ghost int n0=n;
 4   int j=0;
 5   int tmp = 0;
 6   //@ assert && j==0 && tmp==0 && n0==n && i==0;
 7   //@ ghost int n1=n; int j1=j; int tmp1=tmp; int i1=i;
 8   //@ ghost int n2=n; int j2=j; int tmp2=tmp; int i2=i;
 9   //@ decreases ( n-j>=0 ? n-j : 0);
10   //@ loop_invariant (i==0 && j==0 && n2==n) || (i>=0 && j>=1 && n>=j);
11   while (j<n) {
12     int i=0;
13     //@ ghost int n3=n; int j3=j; int tmp3=tmp; int i3=i;
14     //@ assert (n-j>=1 && tmp1==tmp && j1==j && i==0 && n1==n);
15     //@ ghost int n4=n; int j4=j; int tmp4=tmp; int i4=i;
16     //@ decreases ( n-j-i>=0 ? n-j-i : 0);
17     //@ loop_invariant (i==0 && n-j>=1 ) || (i>=1 && n-j-i>=0);
18     while (i < n - j) {
19       if (arr[i] > arr[i + 1]) {
20         tmp=arr[i];
21         arr[i]=arr[i + 1];
22         arr[i + 1]=tmp;
23       }
24       i=increment(i);
25       //@ assert i==i3+1 && n3==n && j3==j && n-j-i>=0;
26       //@ set n3=n; j3=j; tmp3=tmp; i3=i;
27     }
28     j++;
29     //@ assert i>=0 && n-j1>=1 && n1==n && j==j1+1;
30     //@ set n1=n; j1=j; tmp1=tmp; i1=i;
31   }
32 }
```

Fig. 1. A Java implementation of the *Bubble Sort* algorithm.

2.2 Integer Java Programs

Recall that the ranking functions computed by COSTA are of the form $\mathsf{nat}(exp)$ where $\mathsf{nat}(v) = \max(0, v)$. To verify the correctness of a ranking function, it is sufficient to ensure, by using the loop invariant and the size relations, that it is decreasing in each iteration. We use the following JML annotation to output the ranking function found by COSTA which instructs to check that *exp* decrease at least on 1 when it is positive:

```
//@ decreases exp > 0 ? exp : 0.
```

Note that the *if-then-else* structure encodes the maximum between *exp* and 0.

Example 2. Method `bubbleSort` of Fig. 1 has two loops, at L11 and L18. The ranking function for the loop at L11 is $\mathsf{nat}(n-j)$ and its corresponding annotation is at L9. Similarly, the annotation at L16 corresponds to the ranking function $\mathsf{nat}(n - j - i)$ of the inner loop at L18.

One of the uses of size relations is to relate the variables at some program point of interest (within a given scope) to their values at the beginning of the same scope. The annotation of such size relations requires two different JML annotations: one at the beginning of the scope to store the initial values into auxiliary variables, and another annotation at the program point of interest to state the actual relation. To do so, we first produce new *ghost* variables by means of an annotation of the form

$$//@ \text{ ghost int } \mathsf{w}_1 = \mathsf{v}_1 \; ; \ldots ; \text{ int } \mathsf{w}_n = \mathsf{v}_n$$

to store the values of variables $\mathtt{v}_1$,...,$\mathtt{v}_n$ at the beginning of the scope into the auxiliary variables $\mathtt{w}_1$,...,$\mathtt{w}_n$, and then use

$$//@ \text{ assert } \varphi$$

to state the corresponding size relations φ at the program point of interest (we assume that φ is given in terms of these variables).

These annotations suffice for non-iterative scopes, however, iterative scopes require a slightly different treatment to update the ghost variables to their new values at the end of the corresponding scope (the initial values in the next iteration). This is done by using the annotation

$$//@ \text{ set } \mathsf{w}_1 = \mathsf{v}_1 \; ; \ldots ; \; \mathsf{w}_n = \mathsf{v}_n$$

at the end of the corresponding scope.

Example 3. The only non-iterative scope in the example of Fig. 1 is trivial, and corresponds to the code before the outer loop. This scope is annotated by means of the `ghost` variables declaration at L3 and its corresponding `assert` at L6. For the iterative scope that corresponds to the body of the inner loop, at L13 the ghost variables are declared, at L25 the size relations are stated, and at L26 the ghost variables are updated to their new values. For the scope that corresponds to the body of the outer loop, the definition and the update of the ghost variables are at L7 and L30 respectively. However, as this scope is split by the inner loop, we have two lines where the size relations are stated: one before the inner loop at L14 and another one at the end of the scope at L29.

As we have explained before, in some cases COSTA relies on *input-output size relations* to model the effect of method calls. In order to verify these relations we make use of the JML method contract to annotate each method with its corresponding input-output size relations.

Example 4. The following is the code of method `increment` (called at L24) and its corresponding JML contract that expresses the fact that the returned value equals the value of the input `x` plus 1.

```
/*@ public behavior
  @ requires true;
  @ ensures \result = x + 1;
  @ signals (Exception) true;
  @ signals_only Exception;
  @*/
int increment(int x) {
  return x + 1;
}
```

This contract is used by KeY to verify the size relation $i = i_3 + 1$ at L25.

Recall that our loop invariants are slightly different from the standard notion since, in addition, they relate the values of the program variables at each iteration to the initial values (just before entering the loop). However, we can handle them using the invariant annotation of JML in a similar way to the size relations: we first capture the initial values of the variables by means of ghost variables, and then, given a loop invariant Ψ inferred by COSTA, we add the annotation

```
    //@loop_invariant Ψ
```

above the loop, which will be verified by KeY. Note that Ψ already refers to the current and initial values of the variables.

Example 5. The annotation for the invariant of the inner loop defines the ghost variables at L15 and states the invariant at L17. Similarly, for the outer loop it defines the ghost variables at L8 and states the invariant at L10.

Now that the program is fully annotated, we pass it to KeY which succeeds to prove the correctness of all the information used by COSTA to generate the upper bound.

2.3 Extension to Handle the Heap

In the previous section we have shown how COSTA annotates a Java program whose cost depends only on integer data, so we can use KeY to verify the correctness of the corresponding information. However, the resource consumption of programs often depends on *heap-allocated* data structures as well, and thus, to handle such programs COSTA relies on inferring related structural heap properties. In this section we describe an extension for verifying the correctness of these heap properties. Note that programs whose cost depends on the size of arrays can be handled using the techniques discussed above, this is simply done by using `x.length` when COSTA refers to the length of an array `x`.

When a program works with variables of reference type, its resource consumption often depends on a *size measure* related to the corresponding data structure rather than the concrete values that are stored in the data structure, e.g., on the length of a list rather than the values stored in the list. Inference of upper bounds for such programs requires information on how the size measures of the different data structures change along the execution of the program. COSTA relies on a size measure called *path-length* [30] that measures the longest path (i.e., depth) of a data-structure (length of a list, depth of a tree, etc.).

```
 1 //@ public behavior
 2 //@ requires (x != null ==> \acyclic(x))
 3 public List reverseList (List x) {
 4
 5         //@ ghost int x0 = \depth(x);
 6         List l = null;
 7
 8         //@ ghost int x1 = \depth(x); int l1 = \depth(l);
 9         //@ assert (\depth(l)==0 && \depth(x)==x0 && \depth(x)>=0);
10         //@ ghost int x3 = \depth(x); int l3 = \depth(l);
11         //@ decreases ( \depth(x)>=0 ? \depth(x) : 0);
12         //@ loop_invariant ((\depth(l)==0 && \depth(x))>=0) ||
13         //@            (l 3=0 && \depth(x)>=0 && x3−\depth(x)>=1) &&
14         //@            (x != null ==>\acyclic(x) && (l != null ==>\acyclic(l)) &&
15         //@            \disjoint(x,l) && \disjoint(l,x) &&
16         //@            (x != null && l != null ==> !\reach(x,l) && !\reach(l,x)));
17         while ( x != null ) {
18             l = new List(x.data,l);
19             x = x.next;
20             //@ assert (l 1>=0 && x1>=\depth(x)+1 && \depth(x)>=0);
21             //@ set x1 = \depth(x); l1 = \depth(l);
22         }
23
24         return l;
25 }
```

Fig. 2. A Java implementation of reversing a list.

Technically, COSTA first abstracts reference variables to their corresponding *path-length* and then infers corresponding size relations between the different reference variables (they might include also relations to integer variables). For example, a linear constraint $x < y$, where x and y are reference variables, means that *the depth of the data structure to which x points to is strictly smaller than the depth of the data structure pointed to by y.* To annotate the program with information that refers to the *path-length* of data structures, we extended JML with the new keyword `\depth` and use it whenever COSTA refers to the *path-length* of a reference variable, e.g., in ranking functions, in invariants and in size relations. This also required extending KeY to support the `\depth` annotation. Supposing additional size-measures for data-structures can be done, however, apart from supporting it at the level of COSTA, it would require corresponding modifications to as we have done for the case of path-length.

Example 6. The method depicted in Fig. 2 is a Java implementation of reversing a list, i.e., it returns a new list with the elements reversed with respect to the input list `x`. The number of iterations of the loop depends on the path-length of the list, and indeed COSTA infers that $\mathsf{nat}(x)$ is a bound on the number of iterations where x is the path-length of the list pointed to by variable `x`. The annotation of this ranking function is at L11, where we use `\depth`(x) to refer to the path-length of variable `x`.

The computation of the upper bound in the presence of heap-allocated data structures relies on some additional structural heap properties: Acyclicity, reachability and disjointness are essential properties both for path-length analysis and for the verification of the path-length assertions. COSTA infers information related to these properties and uses it when inferring corresponding upper bounds.

To model these structural heap properties, we have extended JML with the following new keywords: (1) `\acyclic(x)`, which states that `x` points to an acyclic data structure; (2) `\reach(x, y)`, which states that `y` must be reachable from `x` in zero or more steps; (3) `\reachPlus(x, y)`, which states that `y` must be reachable from `x` by at least one step; and (4) `\disjoint(x, y)`, which states that `x` and `y` do not share any common region in the heap. We use all these new keywords in the JML annotations added by COSTA to produce the resource guarantees for programs whose resource consumption depends on heap allocated data. Note that KeY was also extended with new rules to prove annotations that involve these new keywords.

Example 7. The example of Fig. 2 shows the JML extensions created to feed KeY with structural information about the heap memory of the program. At L2 it can be seen that `reverseList` has one precondition: the list must be acyclic, otherwise COSTA cannot guarantee the termination of the method. Additionally, the loop invariant annotation at L14 states that not only the input list, but also the new list is always acyclic. Another relevant information introduced in the loop invariant at L15 is that the variables `x` and `l` do not alias, and at L16 we can see that `l` is not reachable from `x` and vice versa. All this information is used and verified by KeY so as to guarantee the correctness of information used by COSTA to infer the corresponding upper bound.

3 Verified Cost Bounds for Abstract Programs

This section gives an overview on the verification of cost bounds for abstract programs using the COSTA and KeY systems. Abstract programs are programs containing an abstract context represented by placeholder symbols. As mentioned in Sect. 1, they are required whenever one aims to rigorously analyze program transformation techniques, as in compiler optimization. In a sense, our joint work with Reiner and Dominic on the verification of abstract cost bounds [10] generalizes the results of Sect. 2 in the same way that an abstract program generalizes a concrete one. We have called the framework *Quantitative Abstract Execution* (QAE) and it unifies an automatic abstract cost analysis using COSTA with an automated verifier for the correctness of the inferred abstract bounds using KeY.

```
int i = 0;
//@ loop_invariant i >= 0 && i <= n;
//@ cost_invariant i · (acP (y) + 2) ;
//@ decreases n − i;
while (i < n) {
  //@ assignable x;
  //@ accessible x, y;
  //@ cost_footprint y;
  \abstract_statement P;
  i ++;
}
//@ assert \cost == 2 + n · (acP (y) + 2) ;
```

Precondition: $\mathsf{n} \geq 0$

Fig. 3. QAE annotations

3.1 QAE Annotations

In order to define QAE, an important technical innovation has been defining an abstract cost analysis. The main technical novelty has been the notion of *cost invariant* which expresses *a sound abstract cost bound* at the beginning of each loop iteration.

We describe this concept and the annotations used in QAE informally by means of the example in Fig. 3, in which we have a loop with an abstract statement P in its body. This abstract statement with identifier P, declared as **\abstract_statement P**, abstracts an arbitrary concrete statement. The abstract statement incorporates a specification of its behaviour by means of annotations (those that appear immediately before the abstract statement). These annotations are of three kinds: (1) "`assignable`" variables, the memory variables that may be written by an abstract statement; (2) "`accessible`" variables, the variables that the abstract statement can read; and (3) "`cost_footprint`" variables, a subset of the accessible variables on which the cost of the abstract statement might depend. Note that in our example, the annotations are declaring that the variables involved in the loop guard cannot be accessed (neither in read/write mode) by the abstract statement. Moreover, loops are required to be *neutral* with respect to the cost, i.e., they cannot change the value of any variable that affects the cost, either because it is involved in the guard or it appears in any `cost_footprint` annotation inside the loop. instructions together with these specifications of the permitted behaviour. In the next step QAE produces the other annotations by means of automatic abstract cost analysis. The whole program, including the inferred annotations, constitutes the input to the second phase: the certifier that proves that the cost annotations are correct.

The loop invariant, keyword `loop_invariant`, has the standard meaning. In our example, it infers the relation between final values of n, of variable i, and that i is non-negative. To prove the abstract cost of the loop, QAE also needs to infer the annotation `decreases` that, as for concrete programs, provides (an upper bound on) the number of iterations that the loop executes (automatically inferred from the ranking function of the loop). So far, these two annotations are well known in standard automated cost analysis, as seen in Sect. 2. Besides, abstract cost analysis automatically infers the two remaining annotations: `cost_invariant` and `assert`. Note that unlike what we have done in Sect. 2, the non-negativity of the ranking function is not encoded in the `decreases` annotation, but rather in the loop invariant (and corresponding precondition).

Each abstract statement has an associated *abstract cost*, which is parametric in the variables defined in its cost footprint. In the case of P, with a cost footprint only composed of variable y, the abstract cost is denoted by $\mathsf{ac}_{\mathsf{P}}(y)$, being possible to instantiate this function with any function parametric in y. For example, if we have the following instance for P:

```
j =0;
while (j <y) j++;
```

and assuming a cost model that counts the number of instructions, the precise exact instance of the cost is $\mathsf{ac}_{\mathsf{P}}(y) = 2 + 2 \cdot y$ (two instructions when declaring j and evaluating the guard for the first time, and two instructions when increasing j and re-evaluating the guard in each iteration). The keyword `cost_invariant` specifies the cost invariant of the loop, i.e., a loop invariant expressing a valid abstract cost bound on the cost of all the iterations of the loop up to the beginning of the next iteration. To get this cost invariant, it is necessary to infer the number of iterations executed so far, that we refer to as the *growth* of the loop (the difference between applying the ranking function to the current and the initial values of the corresponding variables). The cost invariant is computed by multiplying this growth by the cost of the body of the loop.

Finally, the keyword `assert` expresses the total accumulated cost of the program, which is known as *cost postcondition*. This cost postcondition is obtained similarly to the cost invariant: instead of multiplying the cost of the body of the loop by the number of performed iterations, we multiply the cost of the body of the loop by the upper bound on the number of iterations, and we add the cost of the statements outside the loop. This corresponds to the standard notion of upper bound as used in Sect. 2. To ensure the soundness of this bound, a cost precondition of $n \geq 0$ is also inferred by the analysis.

3.2 Cost Postconditions

Cost postconditions are computed for two purposes: to handle programs with nested loops and to prove relational properties.

Nested Loops. In nested loops, to compute the cost invariant of the outer loop, we need first to compute the abstract cost of the inner loop after its complete execution. Then, this cost of the inner loop is used to generate the cost invariant of the outer loop, and the approach is fully compositional.

```
int i = 0;
//@ loop_invariant  i ≥ 0 && i ≤ n;
//@ cost_invariant  i · (3 + m · (2 + acP(y)));
//@ decreases n − i;
while (i < n) {
  int j = 0;
  //@ loop invariant  j ≤ m && j ≥ 0;
  //@ cost_invariant  j · (acP (y) + 2) ;
  //@ decreases m − j;
  while (j < m) {
    //@ assignable  x;
    //@ accessible  x, y;
    //@ cost_footprint  y;
    \abstract_statement P;
    j ++;
  }
  //@ assert \cost ==  2 + m · (acP (y) + 2) ;
  i ++;
}
//@ assert \cost ==  2 + n · (3 + m · (2 + acP(y))) ;
```

Precondition: $\mathsf{n} \geq 0$ and $\mathsf{m} \geq 0$

Fig. 4. Nested loops

In Fig. 4, we have an abstract program with a nested loop. Both loops in this example have their corresponding inferred annotations, including the cost postconditions. In the case of the inner loop, this translates to having the cost of the piece of code that, inside the outer loop, goes until the end of the inner loop. Assuming a cost model that counts the number of instructions, for the inner loop we have a cost postcondition of $2 + m \cdot (\mathsf{ac}_{\mathsf{P}}(y) + 2)$, where the first 2 corresponds to the declaration of variable j and the first evaluation of the guard, and the second 2 corresponds to increasing variable j and the new evaluation of the guard in each iteration of the inner loop. This cost, together with the increase of variable i in the body of the outer loop, gives a cost for the body of the outer loop of $3 + m \cdot (2 + \mathsf{ac}_{\mathsf{P}}(y))$, that multiplied by the growth of the outer loop leads to the cost invariant of the outer loop. The KeY system successfully verifies all annotations inferred for the abstract program.

Relational Properties. QAE has been *the first method to analyze the cost impact of program transformations.* For the purpose of comparison of transformed programs, quantitative relational properties are supported. An important remark is that, to compare these cost postconditions, it is needed to have *exact* ones, as in case for having an over-approximation the comparison would not be conclusive, as the over-approximations could be of different magnitude in the compared expressions.

In Fig. 5, we have an example of a comparison of a program transformation. We note that, for conciseness of the presentation, we only write the cost postcondition at the end of the whole program in "Program Before", even if the first loop also has its cost postcondition computed during the analysis.

Program Before

```
int i = 0;
//@ loop_invariant i>=0 && i<=n;
//@ cost_invariant i · (acP (y) + 2) ;
//@ decreases n − i;
while (i < n) {
  //@ assignable x;
  //@ accessible x, y;
  //@ cost_footprint y;
  \abstract_statement P;
  i++;
}

int j = 0;
//@ loop_invariant j>=0 && j<=m;
//@ cost_invariant j · (acP (y) + 2) ;
//@ decreases m − j;
while (j < m) {
  //@ assignable x;
  //@ accessible x, y;
  //@ cost_footprint y;
  \abstract_statement P;
  j++;
}
//@ assert \cost == 4 +
  n · (acP (y) + 2) + m · (acP (y) + 2) ;
```

Precondition: n ≥ 0 and m ≥ 0

Program After

```
int i = 0;
//@ loop_invariant i>=0 &&
      i<=n+m;
//@ cost_invariant
      i · (acP (y) + 3) ;
//@ decreases n + m − i;
while (i < n+m) {
  //@ assignable x;
  //@ accessible x, y;
  //@ cost_footprint y;
  \abstract_statement P;
  i++;
}
//@ assert \cost == 3 +
  (n + m) · (acP (y) + 3);
```

Precondition: n ≥ 0 and m ≥ 0

Relational property

\cost_after $\geq$ **\cost_before**

Fig. 5. Cost postconditions

In this program transformation, two independent loops with a similar body are put together into a single loop that performs as many iterations as the sum of iterations of the two initial loops. Even if the transformation could seem to be an optimization, when computing the abstract cost analysis, we see that counting the number of instructions leads to a larger cost in the transformed program. This is due to the fact that the operation $n+m$ is computed in each evaluation of the guard. Disregarding constants, the cost of "Program After" is greater than the cost of "Program Before" by $n+m$ instructions. This can be proven in QAE by means of the relational property \cost_after $\geq$ \cost_before.

3.3 Other Cost Models

Even if this section has been developed using as cost measure the *number of instructions* of the program, other cost models are allowed in the analysis. This is the case of the program in Fig. 6 for which we aim at inferring its memory consumption. In this program that allocates an array inside the loop, we perform the analysis using the cost measure of allocated memory. In this case, COSTA

```
int i = 1;
//@ loop_invariant  i ≥ 0 && i ≤ n;
//@ cost_invariant  i · (4 · (n − 1) + acP (y));
//@ decreases n − i;
while (i < n) {
  int[] a = new int[i];
  //@ assignable x;
  //@ accessible x, y;
  //@ cost_footprint  y;
  \abstract_statement P;
  i++;
}
//@ assert \cost ≤ (n − 1) · (4 · (n − 1) + acP (y));
```

Precondition: n≥1

Fig. 6. Cost model of memory allocated

assumes the worst-case memory consumption for each iteration, i.e., creating an array of n elements, and thus the cost postcondition is an upper bound rather than an exact result. More precise results could be obtained by a tighter approximation as in [9].

4 Related Work

In order to increase the trust of end-users in static analysis tools, we typically deliver guarantees to ensure that the inferred results are actually correct. These guarantees are supposed to be (easily) verifiable by a minimal set of corresponding trusted tools. Work in this area can be (roughly) divided into two categories:

1. develop the whole static analysis tool using the programming language of a proof assistant – such as Coq [18] or Isabelle/HOL [28] – together with a correctness proof that can be verified by the proof assistant, and then automatically generate a corresponding verified tool in a more efficient programming language such as OCAML or Haskell. This means that the implementation has been formally proven correct, and thus all the results that it produces are correct.
2. the static analysis tool outputs, for each run, a certificate that certifies the correctness of the results of that run. This certificate can be then verified by a trusted tool (a theorem prover, a proof assistant, or by a dedicated checker that has been proven correct by itself).

In the context of resource analysis, apart from our work [6–8, 10] that we have discussed in Sects. 2 and 3, which belongs to the second category, there are few other works that also fall into these categories.

Blazy et al. [13] develop a tool for inferring loop-bound estimations for WCET analysis that is implemented and formally verified in Coq, and integrated in the

CompCert verified C compiler [25] to provide bounds for the generated assembly programs. The corresponding automatically generated tool is shown to be competitive with other loop-bound inference tools that are based on the same underlying theory.

Carbonneaux et al. [17] present a framework for amortized-based resource usage analysis with emphasis on easing the process of certification. The framework is instantiated in a resource usage analysis tool for integer programs that generates Coq objects as certificates, i.e., for each run it produces a corresponding Coq file (with theorems and code stating the correctness of the results) that can be verified by checking its validity using Coq.

Carbonneaux et al. [16] use Coq and the verified CompCert C compiler [25] to derive stack bounds for assembly code that are verified by Coq. The analysis itself is developed for C programs, but the overall framework also verifies that the results are valid for the generated assembly program.

There has been also interest in verified tools in the context of termination analysis, which is very related to resource usage analysis. Probably the most well-known is the CeTA checker [32], which is automatically generated using Isabelle/HOL, and can be used to check the correctness of termination proofs for term rewrite systems. Nowadays, several termination analysis tools for term rewrite systems generate certificates that can be checked by CeTA. It has also been used by Brockschmidt et al. [15] for certifying termination proofs of integer transition systems. Later, CeTA was extended to support certification of complexity proofs for term rewrite systems [11] as well.

5 Conclusions

The use of static analysis tools in the software development process helps programmers to spot runtime errors and unexpected behaviours that are difficult to find manually, and thus to deliver error-free software. However, while these tools are typically based on solid and correct mathematical principles, the implementations can be buggy. Therefore, we need to verify the correctness of the implementations to increase the trust of end-users in these tools. This is particularly important in contexts where erroneous results might have drastic consequences, e.g., in safety critical systems. In this invited paper, we have described our collaboration with Reiner and his team along this research line, that spans over 4 different joint papers since 2011 and concentrated on verifying the resource usage bounds inferred by COSTA using KeY.

Our collaboration started by handling sequential Java programs [5–7], however, instead of verifying the upper bounds directly, we actually verify all intermediate information inferred by COSTA to compute the corresponding upper bounds: ranking functions, invariants, size relations, and structural heap properties such as acyclicity and disjointness. The workflow is as follows: COSTA annotates the Java programs that it analyses by corresponding JML annotations that include the intermediate information, and then KeY verifies their correctness. Apart from modifying COSTA to output these annotations, KeY was also modified to support some new annotations that refer to structural heap properties that were not supported before.

In a recent collaboration [10], these ideas were generalized for verifying resource usage bounds of *abstract* programs. These are programs containing an abstract context represented by placeholder symbols, and they are used, for example, to analyze program transformation techniques, e.g., the effect of a transformation on the resource usage. In this case COSTA was extended to handle such abstract programs, and to output annotations similar to the ones used for Java programs, but, in addition, they include a cost invariant that helps KeY to actually verify the bounds directly. KeY was also modified to support these cost invariants annotations.

For future work, we would like to extend our work to support more complex programs such as recursive methods and non-integer numerical data, as well as to consider other programming paradigms such as concurrent programs. We note that we already have a resource usage analyser [1], with a workflow that is very similar to that of COSTA, for the actor-based concurrent modeling language ABS [24], and that KeY has also been generalized to support ABS programs [19].

References

1. Albert, E., et al.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_46
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reason. **46**(2), 161–203 (2011)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92188-2_5
5. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. **413**(1), 142–159 (2012)
6. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using COSTA and key. In: Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, pp. 73–76. ACM (2011)
7. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: A formal verification framework for static analysis - as well as its instantiation to the resource analyzer COSTA and formal verification tool key. Softw. Syst. Model. **15**(4), 987–1012 (2016)
8. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Román-Díez, G.: Verified resource guarantees for heap manipulating programs. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 130–145. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_10

9. Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. ACM Trans. Comput. Log. **14**(3), 22:1–22:35 (2013)
10. Albert, E., Hähnle, R., Merayo, A., Steinhöfel, D.: Certified abstract cost analysis. In: FASE 2021. LNCS, vol. 12649, pp. 24–45. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_2
11. Avanzini, M., Sternagel, C., Thiemann, R.: Certification of complexity proofs using CeTA. In: 26th International Conference on Rewriting Techniques and Applications, RTA 2015. LIPIcs, vol. 36, pp. 23–39. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
12. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
13. Blazy, S., Maroneze, A., Pichardie, D.: Formal verification of loop bound estimation for WCET analysis. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 281–303. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_15
14. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4):13:1–13:50 (2016)
15. Brockschmidt, M., Joosten, S.J.C., Thiemann, R., Yamada, A.: Certifying safety and termination proofs for integer transition systems. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 454–471. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_28
16. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 270–281. ACM (2014)
17. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated resource analysis with coq proof objects. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 64–85. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_4
18. Coq Development Team: The Coq Proof Assistant Reference Manual - Version 8.7 (2018)
19. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
20. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
21. Frohn, F., Giesl, J.: Complexity analysis for Java with AProVE. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 85–101. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_6
22. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012)
23. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_6

24. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
25. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
26. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
27. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pp. 496–512. ACM (2018)
28. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
29. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reason. **59**(1), 3–45 (2017)
30. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. ACM Trans. Program. Lang. Syst. **32**(3), 8:1–8:70 (2010)
31. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 319–336. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_20
32. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_31
33. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)

Lifelong Learning of Reactive Systems in Practice

Alexander Bainczyk(✉), Bernhard Steffen(✉), and Falk Howar(✉)

TU Dortmund University, 44227 Dortmund, Germany
{alexander.bainczyk,bernhard.steffen,falk.howar}@tu-dortmund.de
http://ls5-www.cs.tu-dortmund.de

Abstract. This paper presents our lifelong learning framework for continuous quality control. The framework integrates automata learning, model checking, and monitoring into a six-phase continuous improvement cycle which is designed to capture entire system life-cycles. The technical backbone of our framework is ALEX, an open source, web-based learning tool for defining adequate test blocks, as well as for serving as test execution environment and as platform for learning Mealy machines. Keys to the industrial success of our framework are a) the guarantee that the level of quality can only increase when using our framework, b) the continuous improvement of originally customer-provided (regression) test suites, c) the maintenance of achieved quality levels even across system changes, and d) the visualization of system changes using automatically generated *difference trees* and *difference automata*. All this is illustrated using an adaptive cruise control system (ACC) that has been implemented in a one year students project.

Keywords: Automata Learning · Learning-Based Testing · Black-Box Testing · Adaptive Cruise Control

1 Introduction

After decades of research and practice, quality assurance is still a bottleneck of software and system development: to this day, e.g., formal methods for system design are not used consistently, formal verification techniques are mostly used in a few safety-critical domains, and even light-weight techniques as integration testing and system-level testing are not done in a principled and automated way in many industrial contexts. The problem is inherently so hard that, despite substantial improvements, we still lack techniques and tools (e.g., for defining test oracles) and automated support (e.g., for test case generation) in order to control the exploding costs of quality assurance. Reiner Hähnle has dedicated his academic work to mitigating this situation through the development of formal methods and the implementation of these methods in robust tools that can be used by software engineers to design, verify, test, and analyze software systems: His group e.g. is one of the groups developing and maintaining the famous *KeY* tool for the deductive verification for Java programs, based on dynamic logic [5].

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 38–53, 2022.
https://doi.org/10.1007/978-3-031-08166-8_3

Two particularly practical applications of KeY are the Symbolic Execution Debugger (SED) and the application of KeY for the automated generation of unit tests from proofs [11]. The EU FP7 project HATS [13], short for *Highly Adaptable and Trustworthy Software using Formal Models*, which Reiner coordinated, aimed at developing a formal underpinning for the design and development of software product families [27]. The key technical result of the project was the Abstract Behavioral Specification language [20] for the precise description of features and components in software product families. Reiner has facilitated the exchange between the software verification and machine learning communities through projects and workshops, aiming at the development of innovative combinations of knowledge inference and formal verification, providing contexts for the work of the authors of this paper: He served as the coordinator of task forces in the Eternals (*Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*) EU FP7 coordinating action [14], organized an ESF Workshop on *Combining Learning and Symbolic Analysis for Software Documentation and Mastering Change*, and was co-organizer of the workshops SARS 2011 and MLSC 2011, held under the auspices of ISoLA 2011 in Vienna [15], as well as a Dagstuhl seminar on *Machine Learning for Dynamic Software Analysis: Potentials and Limits* [6].

In this paper, on the occasion of Reiner Hähnle's 60^{th} birthday, we present the results obtained in a common DFG project *"Constraint-Based Operational Consistency of Evolving Software Systems (COCoS)"* whose initial results were reported in [16] and are now extended to our lifelong learning framework for continuous quality control that combines automata learning, model checking, and monitoring in an integrated fashion. Characteristic for this framework is its six-phase continuous improvement cycle displayed in Fig. 1 and discussed in Sect. 4 and Sect. 5. The technical backbone of our framework is ALEX, an open source, web-based learning tool for defining adequate test blocks, as well as for serving as test execution environment and as platform for learning Mealy machines. Our industrial partners particularly value the following four aspects of our approach:

a) We can guarantee that the level of quality can only increase, a property required for systems that need certification. This is achieved by using customer's entire test suite as a seed for our learning process.
b) We continuously improve the originally customer-provided (regression) test suite. Not only is the test suite refined (redundant tests are eliminated and essential new ones are added), but it is also automatically updated in response to system changes.
c) We continuously maintain the achieved quality level even across system changes by using the newest (regression) test suite as a seed for re-learning.
d) We monitor the impact of system changes using automatically generated *difference trees* and *difference automata* (cf. Sect. 5). This does not only allow one to check whether the intended fix really works, but it also reveals unintended side effects of a fix.

We illustrate the approach using an adaptive cruise control system (ACC) that has been implemented in a one year student project.

Outline. We start by giving an overview of related work in Sect. 2 and continue with background information on learning-based testing in Sect. 3. After that, the lifelong learning cycle is presented on a conceptual basis in Sect. 4 and in Sect. 5, in which we demonstrate its facets on an simulator for an adaptive cruise control system. Finally, in Sect. 6, the presented approaches are evaluated and an outlook on future work is provided.

2 Related Work

Automata learning [2] is a technique that deals with inferring automaton models from formal languages that has been applied to a lot of real world applications by now, for example in the domain of reactive systems such as web applications [4,26]. A widely adapted application of learned automaton models is the generation of test suites for regression testing [9] and conformance testing [8].

Learning reactive black-box systems is a task in which we continuously search for counterexamples to refine the model representing the system. A feasible approach for this is given by run-time monitoring [7] where the system under learning is observed and observations are traced back to states of the inferred model. In case of a mismatch between observed and learned behavior, the trace is transformed into a counterexample which triggers the refinement of the model. Other approaches deal with automata learning during system evolution to continuously ensure the quality of the system. Active continuous quality control [29] aims to learn the system in each iteration and to verify that it changed in a desired way. Therefore, a stable alphabet abstraction is required and intended changes are formulated using temporal logic and verified by model checkers whenever applicable.

In this paper, we describe a continuous cycle that takes up the idea of the never-stop learning approach and tools such as LBTest [23] and extends it in a way that also takes a changing system into account. Further, we eliminate the need to learn the evolved system for verification purposes of desired changes by learning the difference via automata learning directly. Lastly, we introduce a stable alphabet abstraction and use it for generating regression test suites from inferred models that enables us to find mismatches faster in proceeding learning iterations. Closest to our work is a study by Karl Meinke [24] in which learning-based testing and model checking are used for testing the safety of a platoon of vehicles in a simulation environment.

3 Preliminaries

Learning-based testing is a technique that combines active automata learning, model-based testing and model checking. Given an unknown formal language L over some alphabet Σ, active automata learning aims to inferring an automaton model of L [2]. This practice has been adapted for inferring models of reactive systems [28] which can be represented as Mealy machines with an input alphabet Σ and an output alphabet Ω.

Active automata learning follows the *minimally adequate teacher* (MAT) model and is a continuous cycle of model inference and model refinement by counterexamples. This cycle, which typically terminates if no mismatches between the underlying system and its conjured model can be found, contains two types of queries:

Membership Query. A learner poses membership queries, i.e. words $\sigma \in \Sigma^*$, to the system under learning, and records its outputs $\omega \in \Omega^*$ until enough information is gathered to build a model M_{hyp}, also called hypothesis, of the system.

Equivalence Query. Given a reactive black-box system SUL and a learned model M_{hyp}, an *equivalence oracle* is asked if $SUL \equiv M_{hyp}$, i.e. if the output behavior is the same for all words $\sigma \in \Sigma^*$. In the ideal case, the oracle would simple answer with *'yes'* or *'no'* and in the latter case, provide a counterexample which triggers the model refinement. Because of the nature of black-box systems, these type of queries can only be approximated, e.g. by random testing.

When dealing with real systems, we require tools such as LearnLib [19,25] or Tomte [1] that allow us to build a bridge between the formal level of automata learning and the practical level of real world systems. LearnLib, for example, implements a mapper concept that maps abstract inputs $\in \Sigma$ to concrete system actions as well as it interprets system outputs and maps them to symbols $\in \Omega$. Not only that, but the framework also offers various automata learning algorithms and strategies for approximating equivalence queries.

4 Lifelong Learning in Practice

Lifelong learning constitutes a continuous improvement cycle Fig. 1 which consists of the following six mutually supportive phases whose impact will be illustrated in Sect. 5 along the evolution of the ACC system, which is symbolized in Fig. 1 by the car in the center.

4.1 Phase 1

Phase 1 is the initialization phase where a first model is learned from scratch, or, as often in established settings where lifelong learning is later introduced, using a set of given test cases as an initial seed for learning. In later stages of the lifelong learning cycle, after a system repair, this phase exploits the regression suite generated from the model which has been learned before the system was repaired.

The quality of the learned model depends on the employed equivalence oracle (cf. Sect. 3). In Sect. 5, we use simple random testing for this purpose. Other methods, like the W-Method [9] can also be used but are often very expensive.

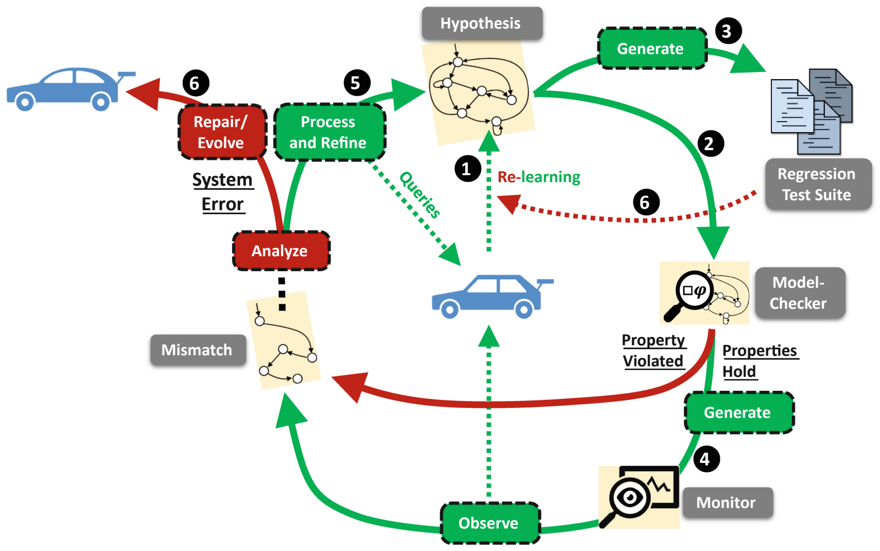


Fig. 1. Lifelong Learning Cycle

4.2 Phase 2

Phase 2 model checks the learned model for essential behavior properties that have been specified in a temporal logic. Found property violations give immediate rise to either a counterexample for improving the hypothesis model, or they reveal a true system error. This interplay between model checking and learning is known as black-box checking [10].

4.3 Phase 3

Phase 3 generates a regression suite from the learned model, or more precisely from the decision tree that serves as the central data structure of the used learning algorithms, that guarantees state coverage and is sufficient to reconstruct the learned model just by analyzing the successors for the states.

Already improving and later maintaining the regression suite during the system evolution this way was considered very valuable by our industrial partners. In Sect. 5, we use the size of the generated test suites as an indication for the impact of the individual steps of the lifelong learning cycle: The more tests we generate from learned models, the more accurate the initial models of subsequent system iterations will get (c.f. Table 1).

4.4 Phase 4

Phase 4 uses a monitor generated from the learned model to check for each individual step of the running system whether it conforms to the learned model, and

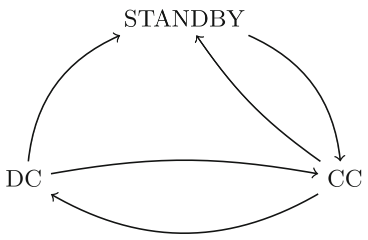


Fig. 2. Simplified state machine of an ACC system

thereby, due to the previous model checking, to all essential system properties. Thus, it serves as a run-time verification process, which, at the same time can be seen as a lifelong equivalence query: found behavioral discrepancies between the system and the hypothesis model provide, like properties violation that are detected in phase 2, either a counterexample for improving the hypothesis model or they reveal a real system error.

4.5 Phase 5

Phase 5 deals with the case that the found discrepancy was identified as a counterexample. It uses the TTT algorithm to refine the hypothesis accordingly, thus closing the continuous improvement cycle.

4.6 Phase 6

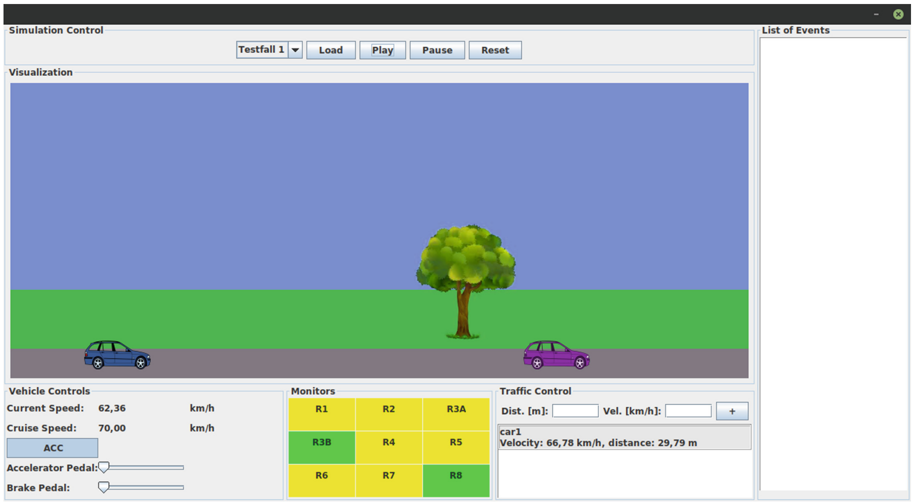
Phase 6 can be regarded as a ‘successful’ case: a true bug has been found. The bug has to be repaired before the lifelong learning cycle can be started again. As mentioned above, in contrast to the initial learning phase, restarting the learning process after repair strongly benefits from the regression suite generated from the original system.

The following section illustrates our lifelong learning approach using an adaptive cruise control system as an example.

5 Learning an Adaptive Cruise Control System

Adaptive Cruise Control (ACC) systems control the velocity of a vehicle while automatically maintaining a safe distance to vehicles in front. The basic behavior of an ACC is sketched in Fig. 2. In *STANDBY* mode, the system is disabled and the driver has full control over the velocity v_{ego} $[m/s]$[1] of the ego vehicle V_{ego} and thus has responsibility for the head distance h $[m]$ to a vehicle V_{obs} driving with velocity v_{obs} $[m/s]$ in front of the ego vehicle and in the same lane.

[1] $[m/s]$ refers to the unit of meters per second.



Current Speed: 62,36 km/h
Cruise Speed: 70,00 km/h

car1
Velocity: 66,78 km/h, distance: 29,79 m

Fig. 3. GUI of the ACC simulator (Color figure online)

Once the driver enables the system at a velocity v_{des} $[m/s]$, it switches to the *cruise control (CC)* mode where the velocity of V_{ego} is maintained automatically until the distance to V_{obs} becomes relevant, in which case the system switches to the *distance control (DC)* mode which is designed to avoid crashes by automatically reducing v_{ego} to maintain a safe distance. In case V_{obs} switches lanes or increases its velocity again, V_{ego} speeds up until v_{des} is reached and switches back to the *CC* mode. In both modes, CC and DC, the ACC system is disabled whenever the driver uses the break pedal. We refer the reader to ISO norm 15622 for detailed technical definitions and requirements on the behavior of an ACC.[2]

We illustrate our lifelong learning approach using an ACC system that has been developed by students during a one year project in which the students developed multiple domain-specific languages for modeling control systems, formal requirements, and test scenarios. Beside the system itself, which comes with a Java API, the students also developed a simulator with a graphical user interface which is displayed in Fig. 3. The simulator can be fed with scenarios that define the initial distance of V_{ego} to V_{obs} as well as their velocities and at which point the ACC system is being enabled. By offering slide controls for the gas and break pedal of V_{ego} the user of the simulator can manipulate the behavior of the car and thereby observe how the ACC system acts in different situations. In addition, the simulator constantly monitors the current situation by checking the system state against nine constraint rules, whose respective status is displayed in nine colored tiles. For example, the monitor "R6" observes if the ACC system will be disabled once the driver hits the break pedal. Initially, all tiles are colored yellow, but as soon as the monitored situation occurs and the

[2] c.f. https://www.iso.org/standard/71515.html.

constraint is met, the corresponding tile turns green, otherwise it turns red. We use the Java API of the ACC system to learn its behavior in terms of reactions to API-based stimulation.

5.1 The Learning Setup

For learning the ACC system, we use the tool ALEX[3] [3]. ALEX is an extension of LearnLib[4] [19] that allows users to model and execute learning experiments for web applications via a web interface. The result of a learning experiment with ALEX is a Mealy machine which, in our example, models the behavior of the ACC system.

ALEX is designed to access the system under learning via an HTTP-based interface that makes the ACC API available over the web. The corresponding interface for the ACC system also implements the functionality of a mapper [21] that maps abstract membership queries produced by active learning algorithms to concrete system inputs and maps the output to such a query accordingly. For example, the input symbol $\mathit{Enable} \in \Sigma$ is mapped to an HTTP POST request to the address

```
http://localhost:8080/acc/enable
```

which in turn calls the method `ACC::enable` of the Java API, which enables the ACC system. Additionally, it returns the current state of the system given a specific input as a string representation. The input alphabet Σ that we use for the learning experiments consists of the following eleven symbols:

Enable, Disable
Enable or disable the ACC system.

Break, Release Break
Hit or release the break pedal, respectively. Since we discretize the system and are only interested in its internal states, finer granular division of the breaking process is omitted. With these symbols, the transitions from CC and DC to $STANDBY$ are triggered.

Drive {60, 80, 120, 150}
Setting the velocity v_{ego} to a concrete value $\in \{60, 80, 120, 150\}$ *kmph* allows us to distinguish the essential internal state transitions of the ACC simulator.

Add Vehicle {Slow, Fast}
Simulate the event that another vehicle V_{obs} is in front of V_{ego}. Since the system lets us set the velocity of V_{obs} only relatively to V_{ego}, it is enough for us to cover the situations that V_{obs} is slower than or as fast as V_{ego}. The case that V_{obs} drives faster than V_{ego} is captured by the final alphabet symbol below.

Remove Vehicle
Simulate the event that V_{obs} switches lanes or accelerates to a velocity greater than v_{des}.

[3] https://learnlib.github.io/alex/.
[4] https://github.com/Learnlib/learnlib.

Table 1. Actions of the lifelong learning cycle performed on the ACC system

Step	Model states	Test cases	Action
First cycle (Broken ACC)			
0	-	5	Learn
1	15	5	Generate test suite
2	15	29	Equivalence test (random)
3	86	29	Generate test suite
4	86	322	Model checking
Second cycle (Fixed ACC)			
5	-	322	Learn
6	81	322	Generate test suite
7	81	292	Equivalence test (random)
8	88	292	Generate test suite
9	88	335	Model checking
10	88	335	Monitoring

The output alphabet consists of the states the ACC system can assume together with a boolean flag *warn* that is true if and only if the distance between V_{obs} and V_{ego} is insufficient for the current speed, i.e.,

$$\Omega = \{\text{STANDBY, STANDBY:warn, CC, CC:warn, DC, DC:warn}\}.$$

As it can be seen in Fig. 4, ALEX presents outputs of the transition of the Mealy machine as "*Ok* $(\omega \in \Omega)$" in order to explicitly indicate that the transition has been successful.

This learning alphabet setup is sufficient for our illustration. Please note, however, that ALEX allows one to easily increase the learning alphabet even during a running learning experiment.

5.2 The Iterative Learning Process

For our industrial partners it was important that our lifelong learning cycle can start with a given set of existing test cases, in order to make sure that a certain testing level is guaranteed. For our ACC study we therefore start with five initially given test cases modeled as sequences of pairs of $(\sigma \in \Sigma, \omega \in \Omega)$ where σ is the input to the system and ω the corresponding expected output.

Intuitively, the list of inputs of a test form a membership query and we define a test as passed if the system produces the expected outputs for each symbol of the test. Please note that it is important for our lifelong learning approach that the modeling of system tests and the learning process use the same underlying alphabet.

Table 1 summarizes our overall lifelong learning experiment. The numbers in each row concern the input setting for the actions mentioned in the fourth

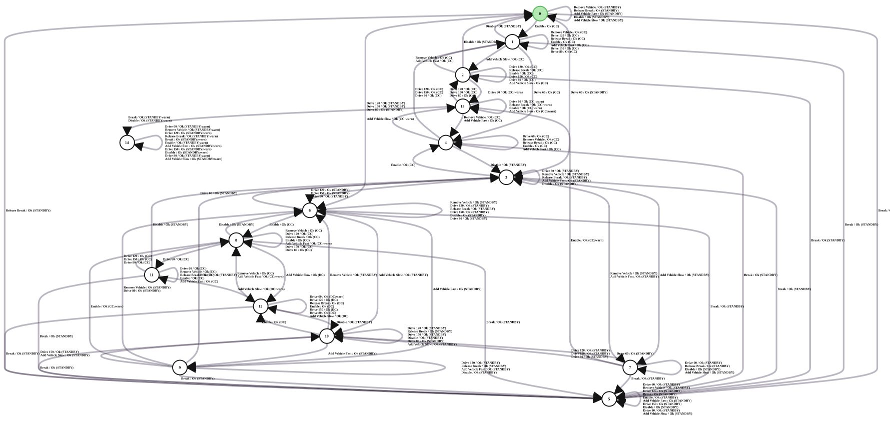


Fig. 4. Mealy machine of the ACC system in step 1 (cf. Table 1)

column. Applying these actions leads to setups with the numbers listed in the next row.

Our initial setting simply consists of the five customer-provided test cases. In the first step, using TTT [18], the current state-of-the-art learning algorithm, these five test cases are automatically complemented with additional test cases generated from the algorithm itself during the learning process. These test cases are required to obtain a consistent and deterministic Mealy machine which, in this case, lead to an initial model with 15 states (cf. Fig. 4).

In a second step one can generate a regression suite from this 15 state Mealy machine which is sufficient to guarantee state coverage for the Mealy machine and to regain the learned Mealy machine from scratch again. This is achieved using the discrimination tree data structure [22] that is used by the TTT algorithm internally to store observations in a redundancy-free fashion. Industrial partners value these regression suites as they allow them to perform their usual regression testing without requiring any learning technology. Still, they benefit from our technology because the generated regression suite of, in this concrete case 29 test cases, guarantees a much better quality.

In the third step, we employed an equivalence oracle based on random testing to refine the learned hypothesis model. This resulted in the detection of 71 more states.

The corresponding regression suite generated during the fourth step already results in 322 test cases, and therefore is in a much better basis for regression testing.

Model checking the 86 state Mealy machine reveals that an essential property of an ACC, which has been specified in temporal logic once and for all, is violated: Whenever V_{ego} is in DC mode and V_{obs} disappears, V_{ego} should switch back to CC mode, meaning that it should stop accelerating as soon as the set velocity is reached.

In order to verify that this property violation of the model is indeed an error of the ACC system, we have to test the corresponding error trace on the ACC system. If the ACC system behaves correctly, the error trace can be used as a counterexample to refine the hypothesis model. Otherwise, the ACC system needs to be corrected.

In our case, the latter has been true and the ACC has to be corrected. After the correction, the lifelong learning process has to be restarted, as the learned model can no longer be used without potentially violating a central invariant of automata learning: the number of state of the hypothesis model is not larger than the number of states of the system under learning. This underlines the importance of the regression suite which can nicely be used as initial test suite for re-learning. Typically, the resulting updated models are of a similar quality as the models for the ACC before their repair.

In this case this means that we first have to learn a hypothesis model of the corrected ACC on the basis of the 322 test cases. In our case this results in a Mealy machine with 81 states. Generating the corresponding regression suite shows that the test suite shrinks. 30 test cases became irrelevant through the correction. The subsequent steps sketched in Table 1 are as before. Only the final model checking in step 9 is this time successful. This indicates that the ACC meets its essential requirements and may be put to 'production' mode, of course, according to our philosophy of lifelong learning, under the control of an accordingly generated monitor.

In the tenth and final step we use a monitor which can be automatically generated form the learned model for run-time verification. In our experiment, this leads to the detection of a discrepancy between the ACC and the learned hypothesis: If the vehicle in fronts reduces its velocity after disabling and re-enabling the ACC, the system switches directly to the DC mode although the model reads that it stays in CC mode. As the observed behavior is desired, we have found a counterexample that allows us to refine the hypothesis model. The corresponding TTT application delivers a 88 state Mealy machine with which a new cycle could continue, although we end the demonstration at this point.

5.3 Controlling the Evolution

Bug fixing often results in new problems. Thus, it is desirable to control the effect of a change at the behavioral level, i.e., at the abstract level given by the input output alphabets. Our lifelong learning approach supports the comparison of a system and its update in two ways (cf. Fig. 5):

First, in terms of a *difference tree* displaying all the test sequences of the two corresponding regression suites that produce a different output for the other system. Figure 6 shows such a difference tree where the suffixes after the first discrepancy are pruned. As we use the entire regression suite of the original Mealy as a 'seed' for the initial learning step, we can collect the test cases of that regression suite where two Mealy machines differ during this very learning step. In contrast, the test cases that are newly entered during the learning of the corrected systems have to be checked for the original system.

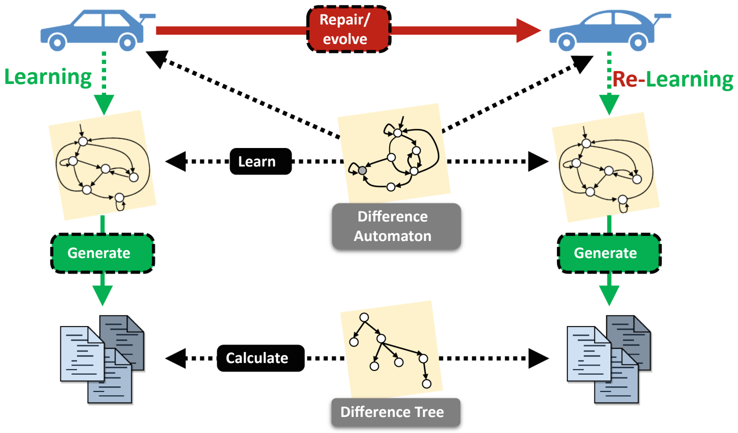


Fig. 5. Views on the difference between iterations

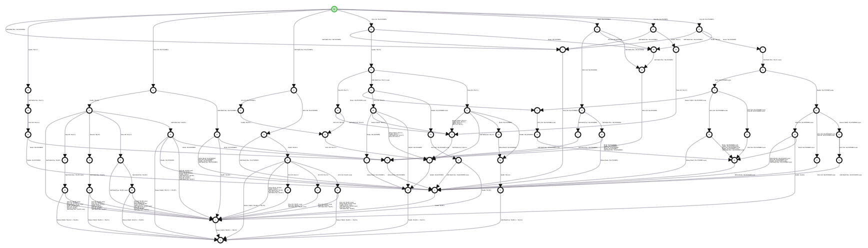


Fig. 6. Difference tree with pruned suffixes

Second, in terms of a *difference automaton* that consists of all sequences that lead to a behavioral difference in the two Mealy machines. Figure 7 shows the difference automaton for the 86-state Mealy machine before the repair and the 88-state Mealy machine after the repair. To indicate the additional power of difference automata we colored the transitions that are not covered by the difference tree in red. In fact, there are 206 unvisited transition in this case.

Difference automata can be constructed in two ways:

– By means of a 'product-like' automata construction in case that both Mealy machines are available, or
– by means of automata learning, where the corresponding mapper feeds both systems with the same sequences, records the outputs of these sequences as long as the outputs coincide, and interrupts the test case execution with success (it found a discrepancy!) as soon as the outputs differ.

The advantage of the learning-based variant is that it can be applied before the corrected system has been learned. Combined with a guided learning approach, where the system exploration prioritizes test cases according to their

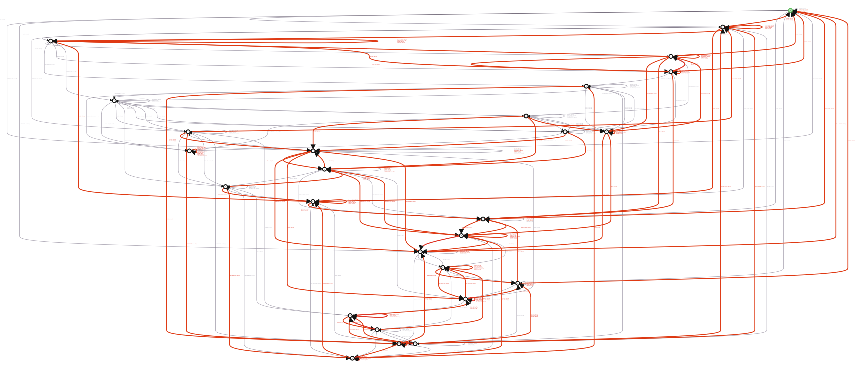


Fig. 7. Superposition of the difference tree on the difference automaton (Color figure online)

probability of touching affected system parts this allows one to obtain fast feedback about the impact of the implemented changes. This works the better the smaller the system changes are. We therefore employ this strategy in our daily system builds.

6 Conclusions and Future Work

We have presented our lifelong learning framework for continuous quality control showing how it integrates automata learning, model checking, and monitoring into a six-phase continuous improvement cycle that is designed to capture entire system life-cycles. The individual phases are all fully automated, except for the repairing process which requires manual work. In particular, phases 1, 2, 3 and 5 can be triggered by simply pushing a button in ALEX, our open source, web-based learning tool. Based on this tool which allows us to define adequate test blocks and which serves as test execution environment and as a platform for learning Mealy machines, our framework guarantees that a) the level of quality can only increase when using our framework, b) the originally customer-provided (regression) test suite is continuously improved, c) the achieved quality levels are maintained even across system changes, and that d) system changes can be visualized using automatically generated *difference trees* and *difference automata*. All these features have been illustrated using an adaptive cruise control system (ACC) that has been implemented in a one year student‘s project.

Currently, we are working on fully integrating our lifelong learning approach into our DevOps-environment for daily system builds, in order to avoid the currently still quite high manual effort for resetting stages in case of errors etc., as well as for starting the tailored re-learning and rebuilding of the system. Besides this mainly managerial automation, we also work on supporting semantic aspects, like semi-automatically adapting the learning alphabets in the course of evolution following the ideas of automated alphabet abstraction refinement [17]. Finally, we are aiming at further optimizing the treatment of extremely long counterexamples as they arise when using monitoring as a means for counterexample detection. The TTT algorithm used by ALEX is the best algorithm when dealing with 'classical' Mealy machines. It can well deal with counterexamples whose lengths are in the thousands. For systems like the one for ACC discussed in this paper, this carries quite far. Our most recent development indicates that using procedural automata, this boundary of counterexample length can be pushed forward by four to five orders of magnitude [12]. We are currently investigating for which application domains the required procedural modeling style is adequate.

Further, we proposed two approaches for the visualization of the difference between the input-output behavior of two reactive systems. While the difference tree that is described in Subsect. 5.3 requires that a formal model of both systems exists, the difference automaton from Subsect. 5.3 can be inferred without learning them in the first place. The advantage of learning the difference automaton becomes clear in Fig. 7. Here, we highlighted all transitions of the difference automaton that have not been covered by the difference tree in red, which amounts to a total of 206 unvisited transitions. Furthermore, the automaton can now again be used for model checking purposes to verify properties of the delta and ensure that the system changed in the desired way.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 10–27. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_4
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. Bainczyk, A., Schieweck, A., Isberner, M., Margaria, T., Neubauer, J., Steffen, B.: ALEX: mixed-mode learning of web applications at ease. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 655–671. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_51
4. Bainczyk, A., Schieweck, A., Steffen, B., Howar, F.: Model-based testing without models: the TodoMVC case study. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 125–144. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_7
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
6. Bennaceur, A., Giannakopoulou, D., Hähnle, R., Meinke, K.: Machine learning for dynamic software analysis: potentials and limits (Dagstuhl seminar 16172). Dagstuhl Rep. **6**(4), 161–173 (2016). https://doi.org/10.4230/DagRep.6.4.161
7. Bertolino, A., Calabrò, A., Merten, M., Steffen, B.: Never-stop learning: continuous validation of learned models for evolving systems through monitoring. ERCIM News **2012** (2012)

8. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, A-MOST 2007, pp. 54–62. ACM, New York (2007). https://doi.org/10.1145/1291535.1291541. http://doi.acm.org/10.1145/1291535.1291541
9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **SE-4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
10. Clarke, E.M., Jr., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
11. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73770-4_10
12. Frohme, M., Steffen, B.: Efficient never-stop context-free runtime verification (2020, under submission)
13. Hähnle, R.: HATS: highly adaptable and trustworthy software using formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 3–8. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_2
14. Hähnle, R.: Task forces in the EternalS coordination action. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 20–22. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_6
15. Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.): ISoLA 2011. CCIS, vol. 336. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34781-8
16. Hähnle, R., Steffen, B.: Constraint-based behavioral consistency of evolving software systems. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 205–218. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_8
17. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_19
18. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
19. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
20. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
21. Jonsson, B.: Learning of automata models extended with data. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 327–349. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_10
22. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
23. Meinke, K., Sindhu, M.A.: LBTest: a learning-based testing tool for reactive systems. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 447–454, March 2013. https://doi.org/10.1109/ICST.2013.62

24. Meinke, K.: Learning-based testing of cyber-physical systems-of-systems: a platooning study. In: Reinecke, P., Di Marco, A. (eds.) EPEW 2017. LNCS, vol. 10497, pp. 135–151. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66583-2_9
25. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_18
26. Raffelt, H., Steffen, B., Margaria, T.: Dynamic testing via automata learning. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77966-7_13
27. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. Computer **44**(2), 82–85 (2011). https://doi.org/10.1109/MC.2011.47
28. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_8
29. Windmüller, S., Neubauer, J., Steffen, B., Howar, F., Bauer, O.: Active continuous quality control. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE 2013, pp. 111–120. ACM, New York (2013). https://doi.org/10.1145/2465449.2465469

A Case Study in Information Flow Refinement for Low Level Systems

Roberto Guanciale[1], Christoph Baumann[2], Pablo Buiras[1](✉), Mads Dam[1], and Hamed Nemati[3,4]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
{robertog,pablo,mfd}@kth.se
[2] Ericsson Research Security, Kista, Sweden
christoph.baumann@ericsson.com
[3] Stanford University, Stanford, USA
hnnemati@stanford.edu
[4] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract. In this work we employ information-flow-aware refinement to study security properties of a separation kernel. We focus on refinements that support changes in data representation and semantics, including the addition of state variables that may induce new observational power or side channels. We leverage an epistemic approach to ignorance-preserving refinement where an abstract model is used as a specification of a system's permitted information flows that may include the declassification of secret information. The core idea is to require that refinement steps must not induce observer knowledge that is not already available in the abstract model. In particular, we show that a simple key manager may cause information leakage via a refinement that includes cache and timing information. Finally, we show that deploying standard countermeasures against cache-based timing channels regains ignorance preservation.

1 Introduction

The last decade has seen a number of formally verified separation kernels [1,18,25], which can provide strong isolation among software components. Nevertheless, their resilience against sophisticated attacks that use low level microarchitectural features [3,24,27,32–34,40] is not proven. For realistic microarchitectures a monolithic analysis of kernel's confidentiality properties is not feasible, since this would require to take into account caches, multiple cores, pipelines, buses, GPUs, devices, and so on. To cope with this complexity, a modular approach based on some form of refinement [2] is essential, since it would allow to handle different security threats at different abstraction levels. The main problem is that standard refinements (e.g., trace inclusion) do not support confidentiality properties [36]: the refined model may provide an observer with information of the execution environment such as execution time or power consumption that may introduce *side channels*.

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 54–79, 2022.
https://doi.org/10.1007/978-3-031-08166-8_4

We have recently developed [11] a theory for secure refinement that supports data refinement, i.e., changes in data representation, reducing nondeterminism and underspecification, and adding state variables that may introduce discriminating power. The key idea is to use the abstract model as a specification of the permitted information flow, and then to ensure that this flow is an upper bound of the corresponding flow in the refined model. We achieve this using knowledge in the sense of [20]. If the progression of observer knowledge on all refined computations is the same or weaker than the one on corresponding abstract computations then the refined model does not leak more information than the abstract model, hence we say it is a *Ignorance-Preserving Refinement*, IPR.

Here we demonstrate our theory by analysing a provably secure low-level system. The system consists of a kernel and processes that use different types of communication, provide services to each other, and operate concurrently. Attackers are compromised processes with unknown behaviour that attempt to acquire secret information from the trusted victims.

For our case study, we formalize a machine with a flat memory. Since verifying functional correctness of the kernel is out of our scope, we axiomatize the expected kernel properties on the abstract model. In order to demonstrate the notion of knowledge, we introduce a simple key manager, which is a process that owns a secret key and allows a potentially malicious client (i.e., another process) to obtain the key-dependent Message Authentication Code of input data.

The goal of IPR is to only protect secrets that are already represented at abstract level, hence IPR does not imply noninterference preservation "out of the box": a concrete model may introduce new types of unrelated implementation information that can be freely leaked. We illustrate the usage of IPR via a refined model, where we add caches and timing information. The question we then need to answer is if this refinement step can cause information to be leaked that would not be possible in the abstract model. Unsurprisingly, given the many results on this topic in the literature [21,40,44], we show that for the key manager the answer is affirmative, due to timing differentials when caches are involved. We then demonstrate that IPR is regained when a simple countermeasure is deployed.

In general, IPR is not sequentially composable, therefore some procedures are secure only if they are executed once. In this paper we also show how sequential compositionality for IPR can be achieved via a kind of relational Hoare logic [12] lifted to refinements. We further apply our proposed solution to verify that IPR is guaranteed when cache colouring [30] or constant time programming, both standard countermeasures against cache-based timing channels, are deployed.

To make presentation easier to follow, we structure the paper by interleaving the theoretical definitions and results of [11] with their application to the case study. In particular, in Sect. 2 we introduce an abstract modeling framework and the epistemic notions of knowledge and ignorance, which allow to formalize the abstract information flows. Section 3 presents instantiation of the abstract framework to our case study. In Sects. 4, 5 and 6 we present our account of refinement and IPR and show the usage of IPR via a refined model. Sections 7 and 8 discuss our solution to make IPR sequentially composable and apply it to some examples countermeasures against cache-based timing attacks. Finally, in Sects. 9 and 10 we discuss related work and give our concluding remarks.

2 Models, Knowledge, and Ignorance

The models we consider in this work are extended transition systems equipped with a store and an observation function. A *model* $\mathcal{M} = (S_0, S, \rightarrow, L, PId, \mathcal{O}, Obs)$ is a labeled transition system with a set of states $S = Var \rightarrow Val$ that map variables from Var to values in Val; $S_0 \subseteq S$ the set of initial states; $\rightarrow \subseteq S \times L \times S$ the transition relation; $\mathcal{O}$ a set of observations, PId a set of process, or observer id's, and for each $p \in PId$ an *observation function* $Obs_p : S \rightarrow \mathcal{O}$ that returns the observations in $\mathcal{O}$ an observer can make in a given state.

We let s, s' range over states, α, β range over observations and write $s \text{-}\alpha_p\!\!\rightarrow s'$, if $Obs_p(s') = \alpha$ and $s \rightarrow s'$. A *final state* is any state s in which no transition starts, i.e., for which no s' exists such that $s \rightarrow s'$. Observation functions Obs_p allow to encode both statically determined sets of variables observed by process p and dynamically varying notions of observability.

In the context of a given transition system, a *run* $\rho \in \mathcal{R}(\mathcal{M})$ is a finite sequence $s_0 \cdots s_n$ such that $s_0 \in S_0$ and $s_{i-1} \rightarrow s_i$ for all $i > 0$ for which s_i is defined. The i'th state of ρ, $\rho(i)$, is s_i, the first (last) element of ρ is $fst(\rho)$ ($lst(\rho)$), $|\rho| \in \mathbb{N}$ is the length of ρ (i.e. number of states in the run), and $\rho(: i)$ is the prefix of ρ having length i. A *complete* run is one that cannot be extended, i.e. there is no s such that $\rho(|\rho| - 1) \rightarrow s$. In that case $lst(\rho) = \rho(|\rho| - 1)$ is final.

The notions of observation trace, observation equivalence, and the epistemic notions of knowledge and its dual, ignorance, are standard. First, two states s_1, s_2 are observationally equivalent as seen by process p, $s_1 \sim_p s_2$, if p has the same observations in the two states, $Obs_p(s_1) = Obs_p(s_2)$. We write $\langle s \rangle_p$ for the equivalence class that contains s. An observation trace for p is the sequence of observation of some run ρ, i.e. $Obs_p(\rho) = Obs_p(\rho(0)) \cdots Obs_p(\rho(|\rho| - 1))$. A complete trace is a trace of a complete run, and the runs ρ_1 and ρ_2 are p-observation equivalent $\rho_1 \sim_p \rho_2$, if p's observations in ρ_1 are the same as p's observations in ρ_2, i.e. $Obs_p(\rho_1) = Obs_p(\rho_2)$. To avoid clutter, we often omit the "p"-subscript when understood from the context.

In the context of a given model $\mathcal{M}$ we view a *property* as a set $\phi \subseteq \mathcal{R}(\mathcal{M})$, namely the set of runs for which the property holds. This allows to define the standard epistemic modality $K_p\phi$ of perfect recall knowledge and its De Morgan dual $I_p\phi$ of "ignorance" on properties ϕ in the following way:

- $\rho \in K_p\phi$, if for all $\rho' \in \mathcal{R}(\mathcal{M})$, if $\rho' \sim_p \rho$ then $\rho' \in \phi$.
- $\rho \in I_p\phi$, if there is $\rho' \in \phi$ such that $\rho' \sim_p \rho$.

In this paper we focus on confidentiality properties, i.e. observer ignorance rather than knowledge.

The set $I_p\phi$ is the set of runs ρ that are "compatible" with some ρ' in ϕ in the sense that p cannot tell ρ' from ρ. Thus, if ϕ holds for ρ', for all p can tell ϕ may hold for ρ as well. Accordingly, we call a set ϕ a *p-ignorance set* if ϕ is closed under $\sim_p$. The *initial ignorance* of a property ϕ is the set of initial states of runs in $I_p\phi$, i.e. $I_p^{\text{init}}\phi = \{\rho'(0) \mid \rho' \in I_p\phi\}$.

If the transition relation is deterministic, for each initial state there is a unique maximal run. We use the notation $\mathcal{R}(s)$ to identify the maximal run starting from state s, and $\mathcal{R}(s, n)$ to identify the run starting from s and taking exactly n steps. These definitions are extended pointwise to sets of states, so if S is a set of states, then $\mathcal{R}(S)$ is the set of maximal runs starting from states in S, and $\mathcal{R}(S, n)$ is the set of runs of length n starting from states in S. Notice that for a deterministic system the standard notion of non-interference can directly expressed in terms of initial ignorance:

Proposition 1. *A system is non-interfering from m to n if for every pair of runs $\rho \sim_p \rho' \in \mathcal{R}(S_0, m)$ it holds that $\mathcal{R}(\rho'(0), n) \subseteq I_p(\mathcal{R}(\rho(0), n))$. A deterministic system is non-interfering from m to n iff for all $I_p(\mathcal{R}(S_0, n)) = \mathcal{R}(I_p^{init}(\mathcal{R}(S_0, m)), n)$.*

That is, after m transitions from the initial state, a system does not leak information for $n - m$ transitions if the ignorance after n transitions is equal to the runs obtained by staring from a state that was in the initial ignorance after m transitions.

3 Case Study: Processor Model and Separation Kernel

We introduce the abstract processor model used in the paper, ignoring things like caches and time related to the refined model introduced in Sect. 6. The example is based on an operating system that allows processes $\{0, \ldots, N\}$ to execute on a sequential processor, where process number 0 represents the kernel and others are unprivileged processes.

A state of the abstract model is a total function $s : \mathit{Var} \to \mathit{Val}$, where $\mathit{Var} = \mathit{regs} \cup \mathit{mem}$, regs is the set of registers (including special purpose registers that control memory protection, program counter, etc.), and mem is the set of memory addresses.

The transition relation $s \xrightarrow{l} s'$ is deterministic and represents the execution of a single machine instruction. We annotate the transition system with label l to capture the list of memory operations performed by the instruction. This list includes all addresses that are involved in the elaboration of instructions such as page tables and instruction memory. Operations (rd, a) and (wt, a) model the reading and writing of address a respectively. We use $R(l)$ and $W(l)$ to extract the set of read/written addresses.

The following notation allows us to abstract from the kernel, exposing common abstractions that depend on special purpose registers and the internal kernel data structures. We use $P(s) \in \mathit{PId}$ to identify the active process in s. The kernel control's memory resources allocation and configures the Memory Management Unit (MMU) accordingly. The sets $W(s, p) \subseteq R(s, p)$ represent the sets of addresses that process p is allowed to write/read. Figure 1 shows an example with three user processes.

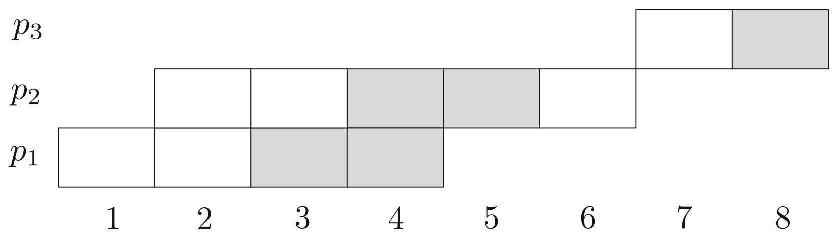


Fig. 1. Access permissions of eight memory pages for three processes. Gray boxes represent addresses that are in W and white boxes represent addresses that are only in R. Processes p_1 and p_2 can directly communicate using addresses in page 4, which can be written by both processes. Page 3 provides a unidirectional channel, since it cannot be modified by p_2. Page 2 is readable by both processes and can be used to store shared libraries. Page tables affect the process behaviour and should not be modifiable by unprivileged processes, therefore they cannot be in $\{3, 4, 5, 8\}$. Process p_3 is isolated and its communication with p_2 must be mediated by the kernel, which may decide to copy data from/to $\{7, 8\}$ or change access permissions.

We make some general assumptions that reflect the above intuition. First, processes cannot violate MMU settings, and they are unable to escalate their access privileges without mediation of the kernel:

Kernel Assumption KA 1. *If* $s \xrightarrow{l} s'$ *and* $P(s) = p \neq 0$ *then*

1. $R(l) \subseteq R(s, p)$ *and* $W(l) \subseteq W(s, p)$
2. $\forall p'.\ R(s, p') = R(s', p')$ *and* $W(s, p') = W(s', p')$
3. $\forall a \in mem \setminus W(l)\ .\ s(a) = s'(a)$

In other words: An unprivileged process p can read and write only addresses for which it has the necessary permissions (1.1). It cannot affect the read/write permissions of any process (1.2), and if p does not write to a given address, then the content of that address remains unchanged (1.3).

Secondly, the behaviour of the active process depends on the registers, the region of memory that can be accessed, and the content of the memory that is read:

Kernel Assumption KA 2. *If* $P(s_1) = P(s_2) = p$, $R(s_1, p) = R(s_2, p)$, $W(s_1, p) = W(s_2, p)$, $s_1 \xrightarrow{l} s_1'$, *and for all* $a \in regs \cup R(l)$ *then* $s_1(a) = s_2(a)$, *then there is some* s_2' *such that* $s_2 \xrightarrow{l} s_2'$ *and for all* $a \in regs \cup W(l)\ .\ s_1'(a) = s_2'(a)$.

These properties are enforced by all secure kernels, cf. [1,18,25], and they do not restrict the kernel design. The kernel is free to change memory grants (i.e. to allocate, free, and change ownership of memory regions) and to copy data among processes. Processes can communicate via shared memory if the kernel allows it. Moreover, these properties do not constrain the presence of microarchitectural communication channels, e.g., due to the insecure usage of caches. Finally, these rules accommodate collaborative as well as preemptive multi-tasking and do not constrain information flows made available by the kernel scheduler.

To apply the epistemic framework introduced in Sect. 2 we need to also provide an observation model. Processes are able to observe the CPU registers when they are active. Moreover, in this example we use a so-called trace driven model [41]. An adversarial process is able to capture the state of its accessible memory (and cache if available) while another process is running. This model allows us to take into account scenarios where memory operations have effects on some other components, like caches or memory mapped devices, that could be controlled by an attacker. In these cases, the intermediary states of the memory that is accessible by the attacker provide a sound overapproximation of the information available to the attacker. Accordingly, the observations of process p in state s, $obs_p(s)$, contain:

1. The identity of the active process, $P(s)$.
2. p's memory rights, $R(s, p)$ and $W(s, p)$.
3. The content of the accessible memory,

$$\{(a, s(a)) \mid a \in R(s, p)\} \ .$$

4. The CPU register contents when p is active,

$$\text{if } (P(s) = p) \text{ then } \{s(r) \mid r \in \mathit{regs}\} \text{ else } \emptyset \ .$$

Notice that a process is able to observe resources that can affect its behaviour only indirectly. For example in Fig. 1, process p_2 can observe that the first memory page is not in $W(s, p_2)$, since writing in this memory page raises a page fault and activates the kernel.

3.1 A Simple Key Manager

To demonstrate the model of knowledge and prepare the ground for later refinements we introduce an abstract model/specification of a simple key manager. The system executes the assumed kernel and two processes: p_1 is the key manager and p_2 is a potentially malicious client. Memory is statically partitioned like in Fig. 1.

The key manager owns a secret key and provides a service that allows other processes to obtain the Message Authentication Code (MAC) of a piece of input data using the secret key. This MAC can be used, for instance, by a client to remotely authenticate the device. Process p_2 cannot directly access the key, which is stored in page 1, and input data and results are communicated via the shared page 4. This system has an intended information leakage: the MAC of the input data with the secret key. In terms of language-based security, the key-manager declassifies the result of this computation, and the goal of the later refinement step is to show that it adds no more information channels than what is already allowed by the abstract model.

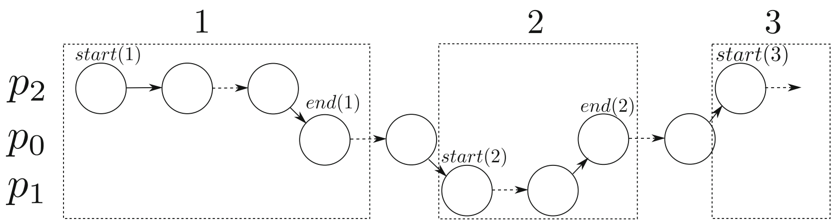


Fig. 2. Three phases and context switches of the example. Dashed arrows represent multiple transitions.

For the purpose of demonstrating our framework it is not important that the key manager uses cryptographically secure primitives. Therefore, a MAC algorithm based on m rounds of a naive Feistel cipher is used: data $d = d_0 d_1$ consists of two bytes, key $k = k_1 \ldots k_m$ consists of m round keys, and the i-th round is computed as follows:

$$\begin{aligned}\mathrm{MAC}_{-1}(d,k) &= d_0\\ \mathrm{MAC}_0(d,k) &= d_1\\ \mathrm{MAC}_i(d,k) &= \mathrm{MAC}_{i-2}(d,k) \oplus T_i[k_i + \mathrm{MAC}_{i-1}(d,k)]\end{aligned}$$

where $+$ is addition modulo 256, and $\oplus$ is bit-wise xor. The MAC is computed using m tables T_i of 256 entries, which implements publicly known byte permutations.

We assume that process p_2 is active in the initial state s_0, and that the system progresses in three phases of Fig. 2:

1. p_2 writes d in page 4, and requests a new service,
2. p_1 computes locally $\mathrm{MAC}_m(d,k)$ and writes the result in page 4,
3. p_2 uses the received MAC.

In between the three phases, the kernel simply context switches between the two processes, without affecting any resource that is observable by the two processes (i.e. it does not change memory permissions and does not modify pages $1 \ldots 6$). For simplicity, we also assume that the context switch requires a constant amount of instructions. Phase i is started after $start(i)$ transitions and completed after $end(i)$ transitions. Let s_0 be the initial state:

– $start(1) = 0$, since we regard the system as starting only when boot is complete and the kernel hands over control to the client, p_2.
– $\mathcal{R}(s_0, end(1))$ is the run ending once p_2 has prepared the request and control has been passed to the kernel.
– $\mathcal{R}(s_0, start(2))$ is the run ending when the kernel has completed the context switch and passed control to p_1.

Our analysis focuses on the ignorance of the client p_2.

Initial Ignorance. $I_{p_2}(\mathcal{R}(s_0, 0)) = I_{p_2}^{\text{init}}(\mathcal{R}(s_0, 0)) = \langle s_0 \rangle_{p_2}$, which is set of initial states where p_2 is active and pages $\{2 \ldots 6\}$ have the same memory content as s_0, independently of the content of pages $\{1, 7, 8\}$. Thus, initially p_2 has no information about k.

Phase 1 and 3. During the first $n \leq \mathit{end}(1)$ transitions, due to KA 1 and KA 2, for every $s_1 \in \langle s_0 \rangle_{p_2}$ let $\rho_1 = \mathcal{R}(s_1)$, it is the case that $\rho_0(n) \sim_{p_2} \rho_1(n)$, therefore $\mathcal{R}(s_0, n) \sim_{p_2} \mathcal{R}(s_1, n)$ and $I_{p_2}(\mathcal{R}(s_0, \mathit{end}(1))) = \mathcal{R}(I_{p_2}^{\text{init}}(\mathcal{R}(s_0, 0)), n)$. Moreover, $I_{p_2}^{\text{init}}(\mathcal{R}(s_0, \mathit{end}(1)) = I_{p_2}^{\text{init}}(\mathcal{R}(s_0, 0))$: as expected, the operating system prevents the client from gaining information without an explicit communication performed by p_1 or by the kernel itself. For the same reason, during the third phase, i.e., for $n \geq \mathit{start}(3)$, it is the case that $I_{p_2}(\mathcal{R}(s_0, n)) = \mathcal{R}(I_{p_2}^{\text{init}}(\mathcal{R}(s_0, \mathit{start}(3))), n)$.

Context Switches. Since we assume that the kernel does not affect any resource that is observable by the process and that the context switch is done in "constant time", then $I_{p_2}(\mathcal{R}(s_0, \mathit{start}(i+1))) = \mathcal{R}(I_{p_2}^{\text{init}}(\mathcal{R}(s_0, \mathit{end}(i))), \mathit{start}(i+1))$ for $i \in \{1, 2\}$.

Phase 2. The client p_2's ignorance now decreases, since process p_2 learns the MAC. In fact, a run ρ_1 is in $I_{p_2}(\mathcal{R}(s_0, \mathit{end}(2)))$, if its initial state s_1 is in $I_{p_2}^{\text{init}}(\rho_0)$ (i.e. $s_1 \sim_{p_2} s_0$) and

$$\mathrm{MAC}_m(d, k) = \mathrm{MAC}_m(d, k')$$

where k' is the key in s_1. In terms of information-theoretic security, by learning the MAC, p_2 also learns a correlation between k and k' that reduces the entropy of k by 8 bits.

In the trace based model, the attacker controlling p_2 can observe its memory while process p_1 is executing. Therefore, if p_1 uses the shared page 4 to temporary store the value of $T_1[k_1 + d_1]$, instead of computing locally the first round, then the ignorance of p_2 is further reduced and the attacker directly learns the first byte of the key.

4 Refinement and Ignorance Preservation

We now compare two models, an abstract model $\mathcal{M}_a$ with states $s \in S$, runs $\mathcal{R}_a$, and transition relation $\rightarrow_a$, typically used to predict the desired behavior, and a concrete, or implementation model $\mathcal{M}_c$, with states $t \in T$, runs $\mathcal{R}_c$, and relation $\rightarrow_c$, that is used to describe how the abstract behavior is realized. We write ρ, ϕ for (sets of) abstract runs and σ, ψ for concrete ones. The two models are connected by a *refinement relation* $s \Downarrow t$, or function $\lceil t \rceil = s$, which for each concrete state t produces one or more abstract states s, which t is intended to refine. We refer to refinement relations of the latter form as *functional.* In this section we set out the basic properties we assume of refinements before we turn in Sect. 4.1 to information flow preservation.

A large section of work in the refinement domain is based on the notion of (forward) simulation, cf. [17,23,45], which in the present synchronous setting can be cast as follows.

Definition 1 (Simulation, Observation Preservation)

1. *The refinement relation* $\Downarrow$ *is a simulation of* $\rightarrow_c$ *by* $\rightarrow_a$*, if* $s \Downarrow t$ *and* $t \rightarrow_c t'$ *implies* $s \rightarrow_a s' \Downarrow t'$ *for some* s'*. Moreover, if* $s \Downarrow t$*, then* s *is initial or final if and only if* t *is.*
2. *The relation* $\Downarrow$ *preserves p-observations, if whenever* $s \Downarrow t$ *it holds that* $Obs_p(t) = Obs_p(s)$*.*

In the functional case the equivalent condition to 1.1 is that $t \rightarrow t'$ implies $\lceil t \rceil \rightarrow \lceil t' \rceil$. The simulation property 1.1 allows abstractions to be point-wise extended to runs by $\rho = s_0 \cdots s_n \Downarrow t_0 \cdots t_n = \sigma$ if $s_i \Downarrow t_i$ for all $i : 0 \leq i \leq |\rho| = |\sigma| = n+1$ and for sets ϕ, ψ, $\phi \Downarrow \psi$, if for all $\rho \in \phi$ there is $\sigma \in \psi$ such that $\rho \Downarrow \sigma$, and vice versa, for all $\sigma \in \psi$ there is $\rho \in \phi$ such that $\rho \Downarrow \sigma$. In the functional case Moreover, we denote by $\Uparrow$ the direct image of $\Downarrow^{-1}$, i.e., $\psi\Uparrow = \{\rho \mid \exists \sigma \in \psi.\ \rho \Downarrow \sigma\}$ and obtain:

Corollary 1. *If* $\Downarrow$ *is a simulation of the concrete model, then 1.* $\phi \Downarrow \psi \Rightarrow \phi \subseteq \psi\Uparrow$ *and 2.* $(\psi\Uparrow) \Downarrow \psi$*.* □

For functional correctness, refinement usually requires both simulation and observation preservation. In this work we rely on the simulation condition as the crucial hook needed to relate computations at abstract and concrete level. Preservation of observations in the sense of 1.2 is used, e.g., in [4] but its necessity appears less clear. For ignorance preservation the key issue is preservation of observable distinctions and not necessarily the observations themselves. Indeed, as we show in this paper it is perfectly possible to conceive of meaningful refinement-like relations that preserve observation distinctions but not the observations themselves.

The key is to shift attention from preservation of observations to preservation of distinctions. In particular we distinguish observational equivalence relation $\sim_p$ on the abstract model from its counterpart (written $\approx_p$) on the concrete model. This motivates the following well-formedness condition:

Definition 2 (Well-formedness). *The refinement relation* $\Downarrow$ *is well-formed, if* $s_1 \Downarrow t_1$ *and* $t_1 \approx_p t_2$ *and* $s_2 \Downarrow t_2$ *implies* $s_1 \sim_p s_2$*.*

For functional refinement relations this becomes the condition that $t_1 \approx_p t_2$ implies $\lceil t_1 \rceil \sim_p \lceil t_2 \rceil$.

Well-formedness reflects the expectation that information content of models should generally increase under refinement. Then, if two abstract states are observationally distinct, we should expect this discriminating power to be preserved to concrete level. We obtain:

Proposition 2. *Suppose that the simulation $\Downarrow$ is well-formed. Then:*

1. *If $\rho_1 \Downarrow \sigma_1$ and $\sigma_1 \approx_p \sigma_2$ and $\rho_2 \Downarrow \sigma_2$ then $\rho_1 \sim_p \rho_2$.*
2. *Suppose $\phi \Downarrow \psi$, then $I_p\phi \supseteq (I_p\psi)\Uparrow$.*

Proof. 1. Follows immediately from Definition 2. 2. If $\rho \in (I_p\psi)\Uparrow$ then we find $\sigma \in I_p\psi$ such that $\rho \Downarrow \sigma$ and a $\sigma' \in \psi$ such that $\sigma \approx_p \sigma'$. By $\phi \Downarrow \psi$ there is $\rho' \in \phi$ with $\rho' \Downarrow \sigma'$ and by well-formedness $\rho \sim_p \rho'$. But then $\rho \in I_p\phi$.

In other words it follows directly from well-formedness and the simulation property that ignorance is preserved from concrete to abstract level. We define:

Definition 3 (Refinement). *The refinement relation $\Downarrow$ is a* refinement*, if $\Downarrow$ is well-formed and a simulation.*

4.1 Ignorance Preservation

One key idea for confidentiality preservation, proposed originally by Morgan [38], is to compare ignorance at abstract level with ignorance at concrete level: If the ignorance at concrete level is "at least as high as" (in [38]: a superset of) the ignorance at abstract level, no more information is learned by executing the protocol at concrete level than what is learned by executing the ideal functionality. While Proposition 2.2 is useful, our interest, however, is in preservation of ignorance in the opposite direction.

However, ignorance at abstract and concrete levels is not readily comparable, as in our setting (as opposed to [38]) the state spaces related by the refinement are different. In general, refinement will reduce nondeterminism and add observational power by implementation choices, e.g., for data representation. Nevertheless, reflecting our view of the abstract model as specifying the desired information flow properties, all information relevant for the analysis of information flow preservation is available already at abstract level. Thus we can use the refinement relation to push epistemic properties between the abstract and concrete levels, as follows:

Definition 4 (Ignorance-Preserving Refinement, IPR). *The refinement $\Downarrow$ is p-ignorance-preserving, if $\Downarrow$ is a well-formed simulation such that $\phi \Downarrow \psi$ implies $I_p\phi \Downarrow I_p\psi$.*

We relativize ignorance preservation to the processes p since this allows to use different abstraction functions for each p, reflecting the potentially different views each process may have of the refinement. It becomes clear that Definition 4 is the desired property if we consider an equivalent formulation:

Proposition 3. *The refinement $\Downarrow$ is p-ignorance-preserving, iff $\phi \Downarrow \psi$ implies $I_p\phi = (I_p\psi)\Uparrow$.*

Proof. By Corollary 1.1, $I_p\phi \Downarrow I_p\psi$ implies $I_p\phi \subseteq (I_p\psi)\Uparrow$ and by well-formedness (Proposition 2.2) $I_p\phi = (I_p\psi)\Uparrow$. The other direction follows directly via $((I_p\psi)\Uparrow) \Downarrow I_p\psi$ by Corollary 1.2.

Thus, IPR means that we have the same ignorance for observer p on both levels, when viewed in terms of the abstract model. In particular, a concrete model observer p cannot distinguish more behaviors than possible on the abstract model, when "re-abstracting" the set of indistinguishable concrete runs. The following is a useful sufficient and necessary condition for ignorance-preserving refinement:

Definition 5 (Paired Refinement). *The refinement $\Downarrow$ is paired, if for all ρ, ρ', σ':*

$$\text{If } \rho \sim_p \rho' \Downarrow \sigma' \text{ then there exists } \sigma \text{ s.t. } \rho \Downarrow \sigma \approx_p \sigma'. \qquad (*)$$

Proposition 4. *The paired refinement condition (∗) holds for refinement $\Downarrow$ if, and only if, $\Downarrow$ is p-ignorance-preserving.*

Proof. The implication IPR $\Rightarrow$ (∗) follows directly from $I_p\phi \Downarrow I_p\psi$ for $\phi = \{\rho'\}$ and $\psi = \{\sigma'\}$. For direction (∗) $\Rightarrow$ IPR, assume that $\phi \Downarrow \psi$ for the refinement $\Downarrow$. For any $\rho \in I_p\phi$ we find $\rho' \in \phi$ such that $\rho \sim_p \rho'$ and a $\sigma' \in \psi$ such that $\rho' \Downarrow \sigma'$. By (∗) we find σ s.t. $\rho \Downarrow \sigma$ and $\sigma \approx_p \sigma'$, i.e. $\sigma \in I_p\psi$. Conversely, if $\sigma \in I_p\psi$ then we find $\sigma' \in \psi$ such that $\sigma \approx_p \sigma'$ and then a $\rho' \in \phi$ such that $\rho' \Downarrow \sigma'$. By the simulation property we find ρ such that $\rho \Downarrow \sigma$ and then by well-formedness, $\rho \sim_p \rho'$, i.e. $\rho \in I_p\phi$, as desired.

Intuitively, (∗) requires that for each pair of indistinguishable abstract runs, if one of them is implemented, so is the other one and the corresponding concrete runs are indistinguishable as well.

Assume models $\mathcal{M}_i$, $0 \leq i \leq 2$, and assume we have ignorance-preserving refinements $\Downarrow_i$, $i \in \{1, 2\}$ from $\mathcal{M}_{i-1}$ to $\mathcal{M}_i$. Then the relational composition $\Downarrow_1 \circ \Downarrow_2$ from $\mathcal{M}_0$ to $\mathcal{M}_2$ should be ignorance-preserving, too. The simulation property and well-formedness properties are easily checked, it remains to show that a vertically composed refinement is an IPR if its component refinements are:

Proposition 5. *If the refinements $\Downarrow_1$ and $\Downarrow_2$ are ignorance-preserving then so is $\Downarrow = \Downarrow_1 \circ \Downarrow_2$.*

Proof. Let $\phi \Downarrow_1 \psi \Downarrow_2 \xi$. Using (∗), assume $\rho_0 \sim \rho_1 \Downarrow \tau_1$. We find σ_1 such that $\rho_1 \Downarrow_1 \sigma_1 \Downarrow_2 \tau_1$. By (∗) there is σ_0 with $\rho_0 \Downarrow_1 \sigma_0 \approx \sigma_1$. Applying (∗) again for $\sigma_0 \sim \sigma_1 \Downarrow_2 \tau_1$, we obtain τ_0 with $\rho_0 \Downarrow \tau_0 \approx \tau_1$ and conclude via Proposition 4.

Vertical composability enables a common verification strategy to deal with perfect recall attackers in epistemic settings: extend the abstract state with an observable history variable that can be computed from existing observations; prove that the abstraction that disregards the history variable is a CPR, and finally analyse a refined model w.r.t. the extended model.

5 Case Study: Adding a History Variable

In the model processes can observe the active process $P(s)$. Therefore we can add an observable variable H that keeps track of the number of transitions performed by each process (this variable simplifies the formalisation of constant time execution in Sect. 8). Let $s \xrightarrow{l} s'$, then $u = (s, H) \xrightarrow{l} (s', H') = u'$ and

$$H'(p) = H(p) + \begin{cases} 1 & \text{if } P(s) = p \\ 0 & \text{otherwise} \end{cases}$$

For this extended model, the simulation simply disregards the history variable.

Lemma 1. *For every process p, $s \Downarrow (s, H)$ is a IPR for the model of Sect. 4.1.*

Proof: The extended model is simulated by the abstract model by construction, similarly well-formedness trivially holds. Therefore it suffices to demonstrate Eq. $*$. Let $t = (s, H)$, $\sigma \in \mathcal{R}(t)$, $\rho \in \mathcal{R}(s)$ such that for every $\sigma(n) = (\rho(n), H_n)$ for some n, $\rho' \in \mathcal{R}(s')$, $\rho' \sim_p \rho$, and $t' = (s', H)$. By construction exist $\sigma' \in \mathcal{R}(t')$ such that for every n exists H'_n such that $\sigma(n) = (\rho(n), H_n)$. Since $\rho(n) \sim_p \rho'(n)$, and since $P(\_)$ is observable we can conclude that $H'_n = H_n$. Therefore $\sigma' \approx_p \sigma$. ∎

6 Case Study: Cache Aware Model

To demonstrate IPR we first introduce a refined version of Sect. 3's processor model. In this model the processes are executed on data-cache enabled hardware and are allowed to measure the time needed to execute their own instructions. The refined state has the form $t = (s, H, c, \tau_0, \ldots, \tau_n)$ where H is the history variable introduced in above, c is a shared cache and τ_i is the clock for the process p_i. In this model, the processes do not have a shared clock, which is reasonable for systems that offer only virtualized time to processes.

To be general we use an abstract model for caches. The cache has $\mathbb{S}$ entries (sets), and c is a total function from $\{0, \ldots, \mathbb{S} - 1\}$ to cache entries. A cache entry $e = (h, d)$ is a pair, where h contains metadata (e.g. validity, tag, state of replacing policy, dirtiness flags, etc.) and d contains the data of the entry. In case of a direct mapped cache this is the complete data stored in the cache line, in multi-way caches the data stored across all the ways. We use the following notation: $idx(a)$ identifies the cache entry corresponding to the address a, $hit(h, a)$ holds if the address a is stored in the entry e, and $get(e, a)$ extracts the content of the address from the entry.

To simplify the notation we introduce the following operators to filter lists of operations accessing the same cache entry i:

$$\varepsilon|_i = \varepsilon \quad \text{and} \quad ((op, a) \circ l)|_i = (op, a) \circ l|_i \text{ if } idx(a) = i \text{ else } l|_i \,,$$

where $\circ$ is the list constructor. To extract metadata of cache entries that collide for a given set of addresses A, we have:

$$c|_A = \{(idx(a), c(idx(a)).h) \mid a \in A\}$$

$\lceil(s, H, c, \tau_0, \ldots, \tau_n)\rceil = (s', H)$ defines the abstraction map for the cache-enabled model, where

$$s'(a) = \begin{cases} get(c(idx(a)), a), & \text{if } hit(c(idx(a)).h, a) \\ s(a), & \text{otherwise} \end{cases}$$

The behaviour of the cache is governed by four model assumptions, similar in spirit to those of Sect. 3.

First, we assume that the kernel abstractions P, R, and W are overloaded for the refined model and invariant w.r.t. the abstraction function:

Kernel Assumption KA 3. *$P(t) = P(\lceil t\rceil)$ and for every process p, $R(t, p) = R(\lceil t\rceil, p)$ and $W(t, p) = W(\lceil t\rceil, p)$.*

Secondly, the cache is transparent:

Cache Assumption CA 1. *If $t \xrightarrow{l} t'$ then $\lceil t\rceil \xrightarrow{l} \lceil t'\rceil$*

In other words: A transition enabled at refined level remains enabled at abstract level once any state information added in the refinement is abstracted away. Note, that this implies that the simulation condition (Definition 1) holds in our model. System software must use special precautions to ensure CA 1 and KA 3, for example by flushing caches and Translation Lookaside Buffer (TLB) when page tables are updated.

Moreover, the metadata of a cache entry does not depend on cache data or accesses to addresses belonging to other entries:

Cache Assumption CA 2. *Let $t_1 \xrightarrow{l_1} t_1'$ and $t_2 \xrightarrow{l_2} t_2'$. For every entry index $i < S$ such that $t_1.c(i).h = t_2.c(i).h$, if $l_1|_i = l_2|_i$ then $t_1'.c(i).h = t_2'.c(i).h$.*

CA 2 expresses that for any two transitions in the cache-aware model, if:

– the metadata associated with a cache entry i is the same in the two prestates, and
– the two transitions read and write the same addresses,

then the cache metadata associated with entry i is the same in the two poststates.

Assumptions CA 1 and CA 2 are general enough to grant a wide scope to our analysis. They accommodate write-through as well as write-back caches, both direct and multi-way associative caches, several types of replacement policies, and do not require inertia (i.e. they allow the eviction of lines even in absence of cache misses for the corresponding entry).

Finally, for the processes' virtualized clocks we make the following assumptions:

Time Assumption TA 1. *If* $t_1 \xrightarrow{l} t'_1$ *and* $p = P(t_1)$ *then*

1. *For all* $p' \neq p$, $t_1.\tau_{p'} = t'_1.\tau_{p'}$.
2. *If* $P(t_2) = p$, $t_2 \xrightarrow{l} t'_2$, $t_1.\tau_p = t_2.\tau_p$, *then for all* $a \in \mathit{regs} \cup R(l)$. $\lceil t_1 \rceil(a) = \lceil t_2 \rceil(a)$, *and* $t_1.c|_{R(l)\cup W(l)} = t_2.c|_{R(l)\cup W(l)}$, *then* $t'_1.\tau_p = t'_2.\tau_p$.

The upshot of TA 1 is that: Only the clock of the active process is incremented (TA 1.1); The execution time of an instruction depends only on the register state, the accessed memory contents, and cache metadata of accessed cache entries (TA 1.2).

For guaranteeing CA 2 and TA 1.2 the cache must provide some sort of *isolation among cache entries*. This is usually the case when the replacement policy does not have state information that is shared among cache entries. An example that violates the requirements is prefetching of adjacent entries in case of cache misses. In this case, the metadata of a cache entry is dependent on the accesses performed in the adjacent entries. Also, for the same type of cache, the prefetching of adjacent cache entries can slow down the execution of a memory access.

In the refined model, a process p can further observe the corresponding private clock and the resources that can indirectly affect it, namely the metadata of the cache elements that can be accessed using the readable memory. Formally, $obs_p(t)$ now contains:

1. The identity of the active process, $P(t)$,
2. p's memory rights, $R(t, p), W(t, p)$
3. The content of accessible memory: $(a, \lceil t \rceil(a)) \mid a \in R(t, p)\}$
4. The CPU registers when p is active: if $(P(s) = p) \wedge a \in \mathit{regs}$ then $s(a)$ else $\bot$.
5. p's local clock, $t.\tau_p$
6. The cache state as seen by p, $t.c|_{R(t,p)}$

Lemma 2. *For the models of Sects. 3 and 6, the function* $\lceil\cdot\rceil$ *is a well-formed refinement.*

Proof: The first part of Definition 1 is obvious because cache and time are transparent on the abstract model and other observations are identity-mapped. For Definition 2, for every $s \sim_p \lceil t_0 \rceil$ let t be a state with the same registers and memory of s, the same timers as t_0, $t.c(i).h = t_0.c(i).h$ for all i, and $t.c(i).d$ such that $get(c(idx(a)), a) = s(a)$ if $hit(t_0.c(idx(a)), a)$. Then t satisfies $s = \lceil t \rceil$ and $t \sim_p t_0$. ■

From KA 3 it follows directly that two states are observation-equivalent if their corresponding abstractions are observation-equivalent, the process clocks are the same, and the metadata of the accessible cache entries are equivalent:

Lemma 3. *If* $\lceil t_1 \rceil \sim_p \lceil t_2 \rceil$, $t_1.\tau_p = t_2.\tau_p$, *and also* $t_1.c|_{R(t_1,p)} = t_2.c|_{R(t_2,p)}$, *then* $t_1 \approx_p t_2$ *holds.*

6.1 Timing Channels in the Refined Model

Caches and timing information pose threats to the key manager. For simplicity, we assume that variables of p_1 and every entry of tables T_i are allocated on different cache entries. Moreover, we assume that in the initial state $\lceil t_0 \rceil = s_0$, clocks are zero, the cache is initially empty, and that process p_2 knows the memory layout of the key manager.

CA 1 guarantees that in the abstract and refined models process p_2 prepares the same inputs and process p_1 provides the same replies. Therefore p_2's knowledge obtained by observing registers, memory, and active process is the same in the two models. However, in the refined model cache and timing effects must be taken into account, as these are not reflected at the abstract level. We assume that p_2 "primes" the cache every time it is executed, filling all entries with data belonging to page 6. For simplicity, we assume that victim p_1 accesses only the addresses needed to implement the key manager, and that the kernel always accesses the same sequence of addresses during context switches.

It is easy to show that IPR holds for the first $start(2)$ transitions. Let ρ_n and σ_n be the runs $\mathcal{R}(s_0, n)$ and $\mathcal{R}(t_0, n)$ respectively.

Initial Knowledge. For every run (consisting only of the initial state) $\rho' \in I_{p_2}(\rho_0)$ there is exactly one corresponding run $\sigma' \in I_{p_2}(\sigma_0)$ where the initial state has empty cache, zero clocks, and such that $\lceil \sigma' \rceil = \rho'$. Therefore, $\lceil I_{p_2}(\sigma_0) \rceil = I_{p_2}(\lceil \sigma_0 \rceil)$, as desired.

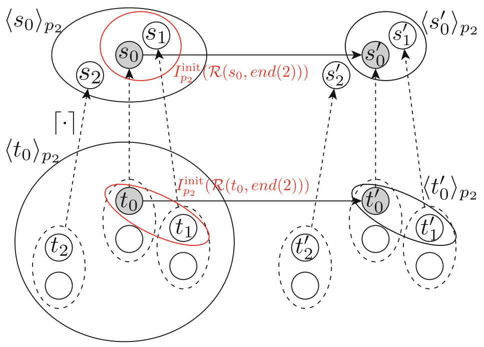
Phase 1. For the first $n \leq end(1)$ transitions, the attacker clock and the cache metadata depend on the initial cache state, which is empty in the initial state of every $\sigma'_0 \in I_{p_2}(\sigma_0)$. Therefore for every $\sigma'_0 \in I_{p_2}(\sigma_0)$ there is $\sigma'_n \in I_{p_2}(\sigma_n)$ such that $\sigma'_n(0) = \sigma'_0(0)$. This means that $I_{p_2}(\sigma_n) = \mathcal{R}(I^{\text{init}}_{p_2}(\sigma_0), n)$, hence $\lceil I_{p_2}(\sigma_n) \rceil = I_{p_2}(\lceil \sigma_n \rceil)$.

Context Switches. For $end(i) < n \leq start(i+1)$ after phases $i \in \{1, 2\}$, the kernel accesses the same sequence of addresses and it cannot modify the process clock. Therefore $I_{p_2}(t_0, start(i+1)) = \mathcal{R}(I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(i))), start(i+1))$ and $\lceil I_{p_2}(\sigma_{start(i+1)}) \rceil = I_{p_2}(\lceil \sigma_{start(i+1)} \rceil)$ if $\lceil I_{p_2}(\sigma_{end(i)}) \rceil = I_{p_2}(\lceil \sigma_{end(i)} \rceil)$.

However, Phase 2 is more challenging. Starting from the last state of $\sigma_{start(2)}$ the key manager accesses the input data, the key, and $T_i[k_i + \text{MAC}_{i-1}(d, k)]$ for $i \in \{1 \dots m\}$. Therefore, after $end(2)$ transitions, the cache entries that are evicted are $idx(T_i + k_i + \text{MAC}_{i-1}(d, k))$, where $T_i + k_i + j$ is the address of the $(k_i + j)$'th element of T_i. On the other hand, if the system had started from the initial state $t'_0 \in I^{\text{init}}_{p_2}(t_0, start(2))$ with key k', then it would have evicted the entries $idx(T_i + k'_i + \text{MAC}_{i-1}(d, k'))$.

In other words, for the last state of $\sigma_{end(2)}$ a dependency of the cache metadata $c(j).h$ on indices $j = idx(T_i + k_i + \text{MAC}_{i-1}(d, k))$ is being introduced, causing $\lceil I_{p_2}(\sigma_{end(2)}) \rceil$ to become (in general) a strict subset of $I_{p_2}(\rho_{end(2)})$, i.e., p_2 in the refined model learns more than in the abstract model.

In practice, this side-channel enables p_2 to discover the key. In fact, for the first round, we cannot guarantee that the cache metadata is the same when



$\langle s_0\rangle_{p_2}$
s_0
s_1
s_2
$I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(2)))$
s'_0
s'_1
s'_2
$\langle s'_0\rangle_{p_2}$
$\lceil\cdot\rceil$
$\langle t_0\rangle_{p_2}$
$I^{\text{init}}_{p_2}(\mathcal{R}(t_0, end(2)))$
t_0
t_1
t_2
t'_0
t'_1
t'_2
$\langle t'_0\rangle_{p_2}$

Fig. 3. Ignorance-Preserving Refinement for p_2, which can distinguish by design the key in $s_2 = \lceil t_2\rceil$. Refined states t_0, t_1 have different key and random seeds. Still, $\lceil I^{\text{init}}_{p_2}(\mathcal{R}(t_0, end(2)))\rceil = I^{\text{init}}_{p_2}(\mathcal{R}(s_0, end(2))) = I^{\text{init}}_{p_2}(\mathcal{R}(\lceil t_0\rceil, end(2)))$.

starting from an initial state where $idx(T_1+k'_1+d_1)$ differs from $idx(T_1+k_1+d_1)$. This can result in different clocks for p_2 after the next context switch (i.e. $n > start(3)$). The same reasoning can be done for the other rounds.

6.2 A Naive Countermeasure for the Key Manager

In this section we show the IPR condition in action by verifying a naive countermeasure, which relies on randomization and the fact that the key manager performs only a single access to each permutation box. We assume that p_2 knows the memory layout of p_1 with the exception of the entries of the tables. Each table T_i has been permuted using a random byte r_i by moving the j'th entry to position $j \oplus r_i$. We assume that p_2 does not know r_i. For this refinement we have the same abstraction map, with exception of the entries of T_i which are mapped as follows: $\lceil t\rceil(T_i+j) = t(T_i+(j\oplus r_i))$. Therefore, in the refined model the MAC is computed as $\text{MAC}_i(d,k) = \text{MAC}_{i-2}(d,k) \oplus T_i[(k_i + \text{MAC}_{i-1}(d,k)) \oplus r_i]$.

We use Fig. 3 to illustrate the scenario. Each state of the refined model is mapped to a unique abstract state via the abstraction $\lceil\cdot\rceil$. The vice-versa is not true: there are at least 256^m refined states (for different values of $r_1 \dots r_m$) that are mapped to the same abstract state. This arises from the fact that refined states have more information than abstract ones.

The function $\lceil\cdot\rceil$ induces an equivalence relation: Two refined states are *abstraction-equivalent* if their abstraction is the same. In our example, states that are abstraction-equivalent must have the same key, but they can have different randomization of the permutation boxes. The equivalence class $\langle t_0\rangle_{p_2}$ can be partitioned into abstraction-equivalent classes (shown as dashed ellipses in the figure), each corresponding to an abstract state of $\langle s_0\rangle_{p_2}$.

To demonstrate IPR for process p_2 we focus on Phase 2, which is the one that violates IPR due to the cache side channel. Let t_0 be a given state with a fixed key k and σ_n a run of n transitions starting from t_0. The argument is essentially relational. Assume a run $\rho'_n \sim \lceil \sigma_n \rceil$. By (5) we have to find a refined state t' such that $\lceil t' \rceil = \rho'_n(0)$ and $\sigma'_n \sim_{p_2} \sigma_n$ for $\sigma'_n \in \mathcal{R}(t', n)$, i.e. such that $\sigma_n(n') \sim_{p_2} \sigma'_n(n')$ for all $n' \leq n$. By the argument of Sect. 6 we must show that we can find t' such that

$$\mathcal{R}(t', end(2)) \sim_{p_2} \mathcal{R}(t_0, end(2)) \ . \tag{1}$$

Once this is established, agreement on the local clocks and cache states allows (1) to extend to arbitrary $n > end(2)$.

Assuming that, except for the table lookup, the MAC is computed using registers only, finding t' is tantamount to identifying a state that satisfies, for every round i,

$$r'_i \oplus (k'_i + \mathrm{MAC}_{i-1}(d, k')) = r_i \oplus (k_i + \mathrm{MAC}_{i-1}(d, k))$$

where r_i and r'_i are the random seeds of t_0 and t', respectively, d is the data in both states, and k' is the key in t'.

In fact, starting from any such t_0 and t' the key manager accesses exactly the same memory locations (even if they have different keys) and therefore the same cache lines have been evicted in $\mathcal{R}(t_0, end(2))$ and $\mathcal{R}(t', end(2))$. This, and the fact that t_0 and t_1 produce the same MAC due to $\rho'_n \approx_{p_2} \lceil \sigma_n \rceil$, guarantees that $\mathcal{R}(t_0, end(2))$ and $\mathcal{R}(t', end(2))$ are indistinguishable by p_2. This completes the argument that the naive countermeasure is sufficient for the key manager to satisfy IPR.

States that are abstraction-equivalent to t_0 or t' are dropped from the initial ignorance set $I_{p_2}(\sigma_0)$ during the computation, if they have a different random seed than t_0 or t', i.e., even though they agree on the resulting MAC they are not in $I^{\mathrm{init}}_{p_2}(\sigma_{end(2)})$ because the refined model reveals the different seed. However, this does not violate IPR.

Unfortunately, the above countermeasure is not compositional: i.e., cannot guarantee IPR if the key manager is invoked multiple times. Let the attacker provide the data d in the first step, and in the third step request a second MAC for data d'. Each step of the key manager satisfies IPR when executed in isolation. However, their composition fails to satisfy IPR if the permutation boxes are not re-randomized. In fact, in the abstract model the attacker learns the value of $\mathrm{MAC}_m(k, d)$ and $\mathrm{MAC}_m(k, d')$. However, in the refined model, the attacker can learn more information regarding the key used for the first round. The first execution of the key manager allows p_2 to additionally learn $r_1 \oplus (k_1 + d_1)$, while the second execution allows the same process to learn $r_1 \oplus (k_1 + d'_1)$. Therefore, the combination of the two executions enables p_2 to discover the value of $(k_1 + d_1) \oplus (k_1 + d'_1)$ which leaks additional bits of k_1 depending on the values of d_1 and d'_1.

7 Relational Verification

In order to handle an example of the size and complexity of our key manager a way to sequentially compose refinements is very useful. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ differ only in that the initial states can be different in the two models, and that final states of $\mathcal{M}_1$ are initial states in $\mathcal{M}_2$. That is:

1. The state spaces of $\mathcal{M}_1$ and $\mathcal{M}_2$ are identical.
2. If $s_1 \rightarrow s_2$ in $\mathcal{M}_1$ (i.e., $s_1 \rightarrow_1 s_2$) then $s_1 \rightarrow_2 s_2$.
3. If $s_1 \rightarrow_2 s_2$ and $s_1 \rightarrow_1 s_2'$ (i.e. s_1 is not final in $\mathcal{M}_1$) then $s_1 \rightarrow_1 s_2$.
4. Final states in $\mathcal{M}_1$ are initial states in $\mathcal{M}_2$.
5. The observations in $\mathcal{M}_1$ and $\mathcal{M}_2$ agree, i.e., $Obs_1(s) = Obs_2(s)$ for all state s in $\mathcal{M}_1$ and $\mathcal{M}_2$.

These properties allow to compose the two models sequentially while preserving observability properties. Note in particular that we allow states, not only initial/final ones, to be present in both $\mathcal{M}_1$ and $\mathcal{M}_2$, which is meaningful for unstructured programs. Also, the definition allows models to be sequentially split and recomposed in a very flexible fashion, by simply stopping execution of $\mathcal{M}_1$ at whichever state is convenient for the analysis. In particular we can define the sequential composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ as the model $\mathcal{M}_1;\mathcal{M}_2$ with the initial states of $\mathcal{M}_1$, states and observations of $\mathcal{M}_1$ (or $\mathcal{M}_2$), and transitions of $\mathcal{M}_2$. We obtain:

Proposition 6

$$\begin{aligned}\mathcal{R}(\mathcal{M}_1;\mathcal{M}_2) &= \mathcal{R}(\mathcal{M}_1);\mathcal{R}(\mathcal{M}_2)\\ &= \{\rho_1;\rho_2 \mid \rho_1 \in \mathcal{R}(\mathcal{M}_1), \rho_2 \in \mathcal{R}(\mathcal{M}_2), \mathit{lst}(\rho_1) = \mathit{fst}(\rho_2)\}\end{aligned}$$

where $(\rho_1 s);(s\rho_2) = \rho_1 s \rho_2$ is the sequential composition of runs.

Under information flow constraints the sequential composition of refinements is generally highly delicate as shown in [43]. Here, we follow the approach of [11] based on ideas from relational Hoare logic [12].

Definition 6 (Relational Refinement). *Let symmetric relations $R_{pre}, R_{post} \subseteq T \times T$ be given such that $R_{pre} \subseteq \approx_p$. The triple $\{R_{pre}\} \Downarrow \{R_{post}\}$ is a* relational refinement*, if $\Downarrow$ is a well-formed refinement from $\mathcal{M}_a$ to $\mathcal{M}_c$ such that:*

1. *If $s_1 \Downarrow t_1$ are initial states, then given any $s_1 \sim_p s_2$, we can find a t_2 such that $s_2 \Downarrow t_2$ and $t_1\ R_{pre}\ t_2$.*
2. *If $\mathit{fst}(\sigma_1)\ R_{pre}\ t_2$, $\rho_1 \Downarrow \sigma_1$, $\rho_1 \sim_p \rho_2$, $\mathit{fst}(\rho_2) \Downarrow t_2$, then a run σ_2 exists with $\mathit{fst}(\sigma_2) = t_2$, $\rho_2 \Downarrow \sigma_2$, $\sigma_1 \approx_p \sigma_2$, and if σ_1 is complete, so is σ_2 and $\mathit{lst}(\sigma_1)\ R_{post}\ \mathit{lst}(\sigma_2)$.*

The triple $\{R_{pre}\} \Downarrow \{R_{post}\}$ expresses that whenever there is a complete run from a concrete state t_1, which is a refinement of an abstract state indistinguishable from s_2, then it is possible to find a complete run from some other concrete state t_2, which is a refinement of s_2, and such that the two runs are indistinguishable, R_{pre} holds on the initial states of the runs, and R_{post} on the final states of the two runs.

By conditioning R_{post} on whether σ_1 and σ_2 are complete, the definition covers both terminating and diverging programs. Clearly, the definition ensures that the IPR condition holds.

Corollary 2. *Any relational refinement is an IPR.* □

Also, we can show that relational refinements provide sequential compositionality. To this end let $\Downarrow$ be a relational refinements from both $\mathcal{M}_{a,1}$ to $\mathcal{M}_{c,1}$ and $\mathcal{M}_{a,2}$ to $\mathcal{M}_{c,2}$ (even if $\Downarrow_1$ and $\Downarrow_2$ are identical as relations they may not both be relational refinements). For clarity we use $\Downarrow_i$ when we want to refer to $\Downarrow$ as a relational refinement from $\mathcal{M}_{a,i}$ to $\mathcal{M}_{c,i}$, and $\Downarrow_1;\Downarrow_2$ when we want to refer to $\Downarrow$ as a relational refinement on $\mathcal{M}_1;\mathcal{M}_2$. Sequential compositionality now follows from the definitions in a straightforward fashion.

Theorem 1 (Sequential Compositionality [11]). *Suppose* $\{R_{pre}\} \Downarrow_1 \{R\}$ *and* $\{R\} \Downarrow_2 \{R_{post}\}$ *are relational refinements. Then* $\{R_{pre}\} \Downarrow_1;\Downarrow_2 \{R_{post}\}$ *is a relational refinement.* □

It follows by Corollary 2 that if $\{R_{pre}\} \Downarrow_1 \{R\}$ and $\{R\} \Downarrow_2 \{R_{post}\}$ are relational refinements then the refinement $\Downarrow_1;\Downarrow_2$ is ignorance preserving.

8 Case Study: Verification of Constant Time Execution and Cache Coloring

In this section we verify security of two widely adopted countermeasures against trace driven cache-based side channels: Constant time execution and cache coloring.

Hereafter we only consider models that have been extended with the history variable H. The following definitions provide a formalization of cache coloring and constant time execution. For each process p we use E_p to identify the indexes of cache entries that process p is allowed to access. The kernel must restrict the process-accessible memory to ensure this property:

Countermeasure C 1. *If* $s \xrightarrow{l} s'$ *and* $P(s) = 0$ *then for all* p*:*

$$idx(R(s,p)) = idx(R(s',p)) = E_p \ .$$

Since processes cannot directly change their access permissions (KA 1), C 1 allows the set of cache indices from the point of view of p to be partitioned into two sets: private entries ($E_p^P = E_p \setminus \cup_{p' \neq p} E_{p'}$) and shared entries ($E_p^S = E_p \cap \cup_{p' \neq p} E_{p'}$). A trusted process p' can perform unrestricted accesses to $E_{p'}^P$ while accesses to $E_{p'}^S$ must satisfy constant time execution. The latter is formalized by the following property, which requires that memory accesses that involve a cache entry accessible by process p depend only on information that is available to p:

Countermeasure C 2. *A system is constant time w.r.t. process p if for every* $s_1 \sim_p s_2$*, if* $s_1 \xrightarrow{l_1} s_1'$*, and* $s_2 \xrightarrow{l_2} s_2'$*, then* $l_1|_{E_p} = l_2|_{E_p}$.

Both C 1 and C 2 can be verified using the abstract model, since they only constrain the list of accessed addresses and addressable indices, and they do not require to explicitly analyse the cache state. C 1 is a kernel invariant to be verified by standard techniques of program analysis. A number of tools exist to check C 2, including relational analysis [7] for binary programs, and abstract interpretation [13] for source code in conjunction with secure compilation [9].

Finally, we can demonstrate that constant time execution and cache coloring (or a mix of the two) prevent side-channels:

Theorem 2. *For a process p, C 1 and C 2 guarantee IPR.*

Proof: In order to prove IPR we show that the two properties ensure a relational refinement for each transition. This allows us to analyze each transition independently and compose the refinements to obtain properties of complete executions. In other words, we look at transition systems that have traces of only one transition per trace and we compose horizontally to obtain the entire transition system. Here the refinement pre- and post-relations are the same: $\approx_p$. Therefore condition (1) holds by definition and (2) holds by well-formedness of $\lceil \_ \rceil$.

By simulation (CA 1), if $t_1 \approx_p t_2$, $\sigma_1 = t_1 \xrightarrow{l_1} t_1'$ then $\rho_1 = \lceil t_1 \rceil \xrightarrow{l_1} \lceil t_1' \rceil$. Since transition systems here are left-total, then exists $\sigma_2 = t_2 \xrightarrow{l_2} t_2'$. Let $\rho_1 \sim_p \rho_2 = \lceil t_2 \rceil \xrightarrow{l_2} s_2'$, by simulation (CA 1) and determinism of the transition system we have that $s_2' = \lceil t_2' \rceil$.

Therefore we must prove that $t_1' \approx_p t_2'$. Hypothesis $t_1 \sim_p t_2$ ensures that $P(t_1) = P(t_2) = p'$. Lemma 3 guarantees that $\approx_p$ is established if $t_1'.\tau_p = t_2'.\tau_p$, and $t_1'.c|_{R(t_1',p)} = t_2'.c|_{R(t_2',p)}$. These equalities are demonstrated for two cases: when p is the active process ($p = p'$) and when it is suspended ($p \neq p'$).

Case $p = p'$. As access permissions are not affected by the cache (KA 3) and cannot be directly changed by the process (KA 1.2) we get for $i \in \{1, 2\}$:

$$R(t_i', p) = R(\lceil t_i' \rceil, p) = R(\lceil t_i \rceil, p) = R(t_i, p) \tag{2}$$

Since the process can only access its own memory (KA 1.1), which is observable, and the same observable access permissions are in place, it performs the same instruction with the same effects on the abstract level (KA 2.1), hence:

$$R(l_1) \cup W(l_1) \subseteq R(\lceil t_1 \rceil, p) = R(t_1, p) \text{ and } l_2 = l_1 \tag{3}$$

Therefore, for cache metadata we have

$$\begin{array}{rr}
t_1 \sim_p t_2 & \\
t_1.c|_{R(t_1,p)} = t_2.c|_{R(t_2,p)} & \text{Def. } \sim_p \\
t_1'.c|_{R(t_1,p)} = t_2'.c|_{R(t_2,p)} & \text{by (3) and CA 2} \\
t_1'.c|_{R(t_1',p)} = t_2'.c|_{R(t_2',p)} & \text{by (2)}
\end{array}$$

Similarly, for the process clock we have:

$$\begin{array}{rlr} & t_1 \sim_p t_2 & \\ \forall a \in R(l_1). \lceil t_1 \rceil (a) = \lceil t_2 \rceil (a) & & \text{Def. } \sim_p \\ \text{and } t_1.c|_{R(l_1) \cup W(l_1)} = t_2.c|_{R(l_1) \cup W(l_1)} & & \text{and (3)} \\ & t'_1.\tau_p = t'_2.\tau_p & \text{by TA 1.2} \end{array}$$

Case $p \neq p'$. For the non-active process, the equality of τ_p follows directly from TA 1.1. By $t_1 \sim_p t_2$ we get:

$$R(t_1, p) = R(t_2, p) \tag{4}$$

Since standard processes ($p' \neq 0$) cannot change access permissions (KA 1.2) and kernel ($p' = 0$) constraints the observable cache entries of p to E_p (C 1), then

$$\begin{aligned} idx(R(t'_1, p)) &= idx(R(t_1, p)) = idx(R(t_2, p)) \\ &= idx(R(t'_2, p)) = E_p \end{aligned}$$

Therefore $t'_1.c|_{R(t'_1,p)} = t'_2.c|_{R(t'_2,p)}$ is equivalent to showing that $t'_1.c(i).h = t'_2.c(i).h$ for every $i \in E_p$:

$$\begin{array}{lr} t_1 \sim_p t_2 & \\ t_1.c(i).h = t_2.c(i).h & \text{Def. } \sim_p \text{ and (4)} \\ t'_1.c(i).h = t'_2.c(i).h & \text{by C 2 and CA 2} \end{array}$$

■

9 Related Work

Information flow security policies that require a complete absence of leakage are usually specified using different variations of noninterference following the approach pioneered by Goguen and Meseguer [22]. Various verification methods for noninterference have been developed including the self-composition method pioneered by Hähnle and others [19].

For realistic systems we need to support communication beyond the multilevel security model. IPR allows to transfer arbitrary information flow properties from specification (the abstract model) to implementation (the refined model) without the need of a specific mechanism for declassification, like the ones proposed in [5,6,8,15,37,37,42]. This allows to verify countermeasures against side-channels without the need of taking into account the mechanism used to analyze declassification in the abstract model.

The intuition behind IPR has been used to analyze information flow security in presence of speculative processors. Both *conditional noninterference* [26] and *speculative noninterference* [28] are restricted forms of IPR that formalize absence of speculative side channels by requiring that states non-interfering under non-speculative semantics are non-interfering under speculative semantics.

A precursor to IPR is the work of Cohen et al. [16] on abstraction in multi-agent systems. They introduce an epistemic simulation relation that is essentially a state-based version of IPR, and use this to show preservation of formulas in the epistemic temporal logic ACTLK.

Morgan and McIver [35,38] propose an instrumented *shadow* semantics for ignorance-preserving program refinement, constructing the ignorance set explicitly for the final values of hidden variables. Moreover, their refinement requires equality for all global variables, allowing the introduction of new observations only as local variables within the scope of the refined program. These local variables cease to exist after the scope of the program has ended, hence the approach does not allow for persistent implementation variables (e.g. state of caches) that can carry data between invocations of different program segments.

One of the features that differentiate IPR from other works is that IPR supports introducing secret dependent observations together with sequential compositionality. Similarly to Sect. 6.2, a compiler may shuffle an array using a random key r and introduce a memory look-up dependent on $s \oplus r$ without compromising the secrecy of s. This type of refinement is not considered in [9,17,23] because it violates the assumption requiring that all refined runs of the same abstract state have the same leakage. Similarly, this is not allowed in [39], since modified variables must either be private, which is not the case for τ_{p_2}, or have their classification decreased, which would prevent the indirect flow of $s \oplus r$ to τ_{p_2}. This makes IPR a more permissive condition justifying information-theoretically secure refinement. Therefore it allows us to raise the question when it is possible to infer that an entire implementation is information flow secure in terms of the information flow security of the specification and the information flow preservation of each step.

Heifer et all [29] have investigate how to formally verify prevention of timing channels in seL4. Similarly to Sect. 8, they provide an abstracted representation of the hardware resources and postulated how they cause timing channels. Their work is specific for seL4 and depends on specific kernel designs. In contract, this work provides a blueprint for this type of analyses by using the general framework of IPR.

Security of constant time programming has been investigated in [10], where Barthe et al. show that this policy protects against cache based side channels in virtualized environments. In [9] the authors show that several compiler optimizations do not affect the constant time policy, by demonstrating that observational noninterference w.r.t. a source state relation results in observational noninterference w.r.t. a target state relation after compilation.

10 Concluding Remarks

We have analyzed a provably secure separation kernel using a compositional approach to information flow preserving refinement that is based on observer ignorance. In our abstract model, we have formalized a flat memory machine and axiomatized kernel properties that entail functional correctness. Once we refined the model to include caches and timing effects, we have shown information leakage for a simple key manager, i.e. observer ignorance is not preserved. Deploying two widely-adopted countermeasures such as cache coloring or a constant-time programming policy recovers ignorance preservation.

A possible extension of this work is to consider different attacker models. If extended to multiple attackers, constant time execution does not guarantee ignorance preservation individually for each attacker due to the presence of covert channels. However, it may be possible to reduce this case to a single, distributed attacker. In addition, one can handle access-driven attackers, i.e., attackers that cannot perform observations while suspended, by introducing a weak transition system that hides attacker-inactive steps.

It would also be interesting to extend the framework to handle other types of system features. For example, to handle cache flushes we would require constant time execution only for addresses that collide with cache entries that have not been previously cleaned. This approach can work if one can demonstrate that cache metadata is correctly reset after a flush, though unfortunately hardware vendors do not usually provide enough details about implementations to conclude this. To handle dynamic cache partitions at the kernel level, we should ensure the abstract model reflects any side channels that arise from the partitioning mechanism.

A final and more difficult problem concerns attackers potentially abusing low level "features" such as Rowhammer [31], mismatched cache attributes [27], speculative execution, or self-modifying code to induce changes in program behavior that are not visible at the abstract level where the effect of these features may not be reflected. We call such attacks *behavior morphing*. In such cases the machinery in this paper no longer works fully as intended. On the other hand, countermeasures against vulnerabilities like Rowhammer do exist [14] that suitably confine the effects of any behavior morphing and should be verifiable using techniques akin to the ones we present in this paper. We leave a proper treatment of ignorance-preserving refinement in the presence of behavior morphing for future work.

Acknowledgment. This work partially was supported by grants from the Swedish Foundation for Strategic Research and the Swedish Civil Contingencies Agency, and the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

1. seL4 Project. http://sel4.systems/. Accessed 21 Apr 2017
2. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
3. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 225–242. Springer, Heidelberg (2006). https://doi.org/10.1007/11967668_15

4. Alur, R., Černý, P., Zdancewic, S.: Preserving secrecy under refinement. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 107–118. Springer, Heidelberg (2006). https://doi.org/10.1007/11787006_10
5. Askarov, A., Chong, S.: Learning is change in knowledge: knowledge-based security for dynamic policies. In: Computer Security Foundations Symposium (CSF), pp. 308–322. IEEE (2012). https://doi.org/10.1109/CSF.2012.31
6. Askarov, A., Sabelfeld, A.: Gradual release: unifying declassification, encryption and key release policies. In: Symposium on Security and Privacy, pp. 207–221. IEEE (2007). https://doi.org/10.1109/SP.2007.22
7. Balliu, M., Dam, M., Guanciale, R.: Automating information flow analysis of low level code. In: Proceedings of the Conference on Computer and Communications Security, CCS 2014, pp. 1080–1091. ACM (2014). https://doi.org/10.1145/2660267.2660322
8. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: Workshop on Programming Languages and Analysis for Security, pp. 6:1–6:12. ACM (2011). https://doi.org/10.1145/2166956.2166962
9. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: IEEE 31st Computer Security Foundations Symposium (CSF), pp. 328–343, July 2018. https://doi.org/10.1109/CSF.2018.00031
10. Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1267–1279. ACM (2014)
11. Baumann, C., Dam, M., Guanciale, R., Nemati, H.: On compositional information flow aware refinement. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, 21–25 June 2021, pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00010
12. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. SIGPLAN Not. **39**(1), 14–25 (2004). https://doi.org/10.1145/982962.964003
13. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10492, pp. 260–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_16
14. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: Can't touch this: software-only mitigation against Rowhammer attacks targeting kernel memory. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 117–130 (2017)
15. Chong, S., Myers, A.C.: Security policies for downgrading. In: Conference on Computer and Communications Security, pp. 198–209. ACM (2004). https://doi.org/10.1145/1030083.1030110
16. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), vol. 2, pp. 945–952 (2009)
17. Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 648–664 (2016). https://doi.org/10.1145/2908080.2908100

18. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proceedings of the Conference on Computer and Communications Security, CCS 2013, pp. 223–234. ACM (2013)
19. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32004-3_20
20. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
21. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. J. Cryptogr. Eng. **8**(1), 1–27 (2018)
22. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy, pp. 11–20. IEEE (1982). https://doi.org/10.1109/SP.1982.10014
23. Graham-Cumming, J., Sanders, J.W.: On the refinement of non-interference. In: Proceedings Computer Security Foundations Workshop IV (CSFW), pp. 35–42 (1991). https://doi.org/10.1109/CSFW.1991.151567
24. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in JavaScript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 300–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_15
25. Gu, R., et al.: CertikOS: an extensible architecture for building certified concurrent OS kernels. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2016, pp. 653–669. USENIX Association, Berkeley (2016)
26. Guanciale, R., Balliu, M., Dam, M.: Inspectre: breaking and fixing microarchitectural vulnerabilities by formal analysis. In: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, CCS 2020, USA, 9–13 November 2020, pp. 1853–1869 (2020). https://doi.org/10.1145/3372297.3417246
27. Guanciale, R., Nemati, H., Baumann, C., Dam, M.: Cache storage channels: alias-driven attacks and verified countermeasures. In: Symposium on Security and Privacy, pp. 38–55. IEEE (2016). https://doi.org/10.1109/SP.2016.11
28. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: principled detection of speculative information flows. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, 18–21 May 2020, pp. 1–19 (2020). https://doi.org/10.1109/SP40000.2020.00011
29. Heiser, G., Klein, G., Murray, T.: Can we prove time protection? In: Proceedings of the Workshop on Hot Topics in Operating Systems, pp. 23–29 (2019)
30. Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. Trans. Comput. Syst. **10**(4), 338–359 (1992). https://doi.org/10.1145/138873.138876
31. Kim, Y., et al.: Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In: Proceeding of the 41st Annual International Symposium on Computer Architecture, ISCA 2014, Piscataway, NJ, USA, pp. 361–372. IEEE Press (2014)
32. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. arXiv e-prints, January 2018

33. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
34. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
35. McIver, A.K., Morgan, C.C.: *Sums and Lovers:* case studies in security, compositionality and refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 289–304. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_19
36. McLean, J.: The specification and modeling of computer security. Computer **23**(1), 9–16 (1990). https://doi.org/10.1109/2.48795
37. Meyden, R.: What, indeed, is intransitive noninterference? In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74835-9_16
38. Morgan, C.: The shadow knows: refinement and security in sequential programs. Sci. Comput. Program. **74**(8), 629–653 (2009). https://doi.org/10.1016/j.scico.2007.09.003
39. Murray, T.C., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: IEEE 29th Computer Security Foundations Symposium, (CSF), pp. 417–431 (2016). https://doi.org/10.1109/CSF.2016.36
40. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
41. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive **2002**, 169 (2002)
42. Rushby, J.: Noninterference, transitivity and channel-control security policies. Technical report, SRI International (1992)
43. Santen, T., Heisel, M., Pfitzmann, A.: Confidentiality-preserving refinement is compositional—sometimes. In: Gollmann, D., Karjoth, G., Waidner, M. (eds.) ESORICS 2002. LNCS, vol. 2502, pp. 194–211. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45853-0_12
44. Stefan, D., et al.: Eliminating cache-based timing attacks with instruction-based scheduling. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 718–735. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_40
45. Van der Meyden, R., Zhang, C.: Information flow in systems with schedulers, Part II: Refinement. Theor. Comput. Sci. **484**, 70–92 (2013). https://doi.org/10.1016/j.tcs.2013.01.002

Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY

Tabea Bordis[1,2](✉), Loek Cleophas[3,4], Alexander Kittelmann[1,2], Tobias Runge[1,2], Ina Schaefer[1,2], and Bruce W. Watson[4,5]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{tabea.bordis,alexander.kittelmann,tobias.runge,ina.schaefer}@kit.edu
[2] TU Braunschweig, Braunschweig, Germany
[3] TU Eindhoven, Eindhoven, The Netherlands
l.g.w.a.cleophas@tue.nl
[4] Stellenbosch University, Stellenbosch, South Africa
bwwatson@sun.ac.za
[5] School for Data-Science and Computational Thinking, Stellenbosch University, Stellenbosch, South Africa

Abstract. Deductive program verification is a post-hoc quality assurance technique following the design-by-contract paradigm where correctness of the program is proven only after it was written. Contrary, correctness-by-construction (CbC) is an incremental program construction technique. Starting with the functional specification, the program's correctness is guaranteed by application of a small set of refinement rules. Even though CbC is supposed to lead to code with a low defect rate and improve the traceability of errors, it is not widespread. One of the main reasons is insufficient tool support which we addressed with our tool CorC. CorC provides support for CbC-based program construction with the KeY program verifier as backend prover for checking correctness of refinement rule applications. However, CorC was limited to constructing single method bodies restricting its applicability. In this work, we introduce and discuss CorC 2.0, which extends CorC's programming model with objects as used in object-oriented programming. We integrate CorC into a development process that allows to use post-hoc verification and CbC interchangeably to construct correct programs, and scale the applicability of CbC on the architectural level in our tool extension ArchiCorC. We developed three object-oriented case studies and evaluated the verification effort and the usability of CorC in comparison to post-hoc verification.

1 Introduction

The amount of software in safety-critical systems increases, and, therefore, functional correctness of programs is an important concern. While most verification approaches rely on post-hoc verification [13,23,50,51], where a program is only

© Springer Nature Switzerland AG 2022
W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 80–104, 2022.
https://doi.org/10.1007/978-3-031-08166-8_5

verified after it is implemented, the stepwise *correctness-by-construction* development approach (CbC) as proposed by Dijkstra [20], Gries [21], or Kourie and Watson [29] offers an alternative approach. A behavioral specification in form of a pre- and postcondition pair is refined into code using a set of tractable *refinement rules*. To guarantee the correctness of the refinement steps, each rule defines specific side conditions for its applicability. As a result, when applying CbC compared to classical post-hoc verification, errors are more likely to be detected earlier in the design process [35].

Our long-term vision is to make CbC accessible for large-scale software development. CorC [43] is a tool based on the deductive program verifier KeY [3] that supports the development of single methods following the CbC paradigm as imagined by Dijkstra [20]. The program and its specification are separated into several Hoare triples, each triple consisting of a pre- and postcondition pair and a statement written in Java. These triples can typically be proven automatically [43] with KeY. Thus, CorC adds tool support for CbC-based development to the KeY ecosystem. So far CorC could only be used to develop single algorithms as methods, independent of classes or larger software systems. One major stepping stone towards our vision to extend the applicability of CbC-based development is a development process that uses post-hoc verification and CbC in concert to take advantage of both approaches. In this paper, we focus on integrating object-orientation and a roundtrip engineering approach into the new version of CorC, called CorC 2.0. CorC 2.0 covers the same object-orientation language concepts that KeY covers for post-hoc verification and enables the combination of classic post-hoc verification using KeY and CbC-based development to improve the development of correct Java programs.

We extend CorC by four features such that object-oriented Java programs can be created using CbC and integrated into existing Java projects.

Graphical View. We provide a graphical view to create classes with fields, class invariants, and methods.

Inheritance and Interfaces. We support constructive development with interfaces and inheritance using the Liskov principle.

Roundtrip Engineering. We implement a roundtrip engineering approach such that existing Java classes can easily be imported into CorC to show their correctness and afterwards exported back to the original project as verified Java code. Thereby, the developer can freely decide which parts of the software shall be constructed using CbC in CorC and which parts shall be verified with post-hoc verification (or even stay unverified if it is not a safety-critical part).

Change Tracking. We use a change tracking mechanism that simplifies ongoing development by marking the refinement steps that have to be re-verified due to changing contracts and implementations of methods or classes (e.g. method calls, inherited method contracts, changed interface specifications).

To evaluate our concept, we recreate three case studies containing multiple classes and methods and compare the verification effort in terms of needed proof steps and verification time to deductive verification with KeY. By doing so, we found that with CorC 2.0 we were able to prove a larger number of methods compared to post-hoc verification with KeY, and that CorC 2.0 also sometimes outperforms post-hoc verification with respect to verification effort. To further extend CbC-based development, we give an outlook on CbC-based development of component-based systems on the architectural level in our tool ArchiCorC and on CbC-based development of feature-oriented software product lines in our tool VarCorC.

2 Related Work

Besides of the CbC-based program construction approach proposed by Dijkstra [20] and Kourie and Watson [29], which we pursue in this work, there are other *refinement-based approaches* that guarantee the correctness of the program under development. In the Event-B framework [1], automata-based system descriptions are refined to concrete implementations. This approach is implemented in the Rodin platform [2]. In comparison to the CbC approach used here, the abstraction level is different. CorC uses specified source code instead of automata as main artifact. Morgan [36] and Back [8] also proposed related CbC approaches. Morgan's refinement calculus, which comprises a very large number of different refinement rules in comparison to the minimal set of refinement rules in CorC, is implemented in the tool ArcAngel [37]. Back et al. [6,7] developed the tool SOCOS. In comparison to CbC in CorC, SOCOS works with invariants in contrast to pre-/postcondition specifications as used in CorC.

Another field that emerged from the ideas of Gries [21] and Dijkstra [20] is *program synthesis*. Program synthesis is the task to *automatically* find a program that satisfies a formal specification provided by a developer. Foundational work has been proposed by Manna and Waldinger [33] and has been continued by many others. For example, Stickel et al. [48] propose a deductive approach that extracts a Fortran program from a user-given graphical specification by composing entries of subroutine libraries. Gulwani et al. [22] propose a component-based synthesis that generates and resolves so called synthesis constraints and apply their approach to bitvector programs. Heisel [25] uses a proof system that builds up a proof during program development. Polikarpova et al. [41] propose an approach to synthesize recursive programs from a specification in the form of a polymorphic refinement type. In contrast to program synthesis, CbC as we apply it does not automatically synthesize code from a specification. CbC is rather a development approach that is guided by a specification and guarantees correctness by proving that the side conditions that are introduced by the set of refinement rules are fulfilled. The developer therefore still has control over the program resulting from the approach while with program synthesis one of possibly many implementations that fulfill the specification is generated. Furthermore, program synthesis has limitations regarding scalability, as for example recursive programs including loops are hard to synthesize.

Some of the authors' recent work on trait-based CbC [44] proposes to replace meta-refinement rules of CbC that are outside of the programming language with trait-based program development and trait-based composition. Refinement in TraitCbC amounts to implementing an abstract method in a trait by a concrete method in another trait and composing those two traits to realize the abstract method by the concrete implementation. TraitCbC per se is parametric in the specification language, meaning that any trait-based language with a corresponding specification language and verification framework can be used to instantiate TraitCbC. In some of the authors' implementation [44], we use KeY [3] to establish the correctness of traits and trait composition. Abstract execution [47] in KeY allows verifying the correctness of methods with abstract program parts, which are specific by contracts. Abstract execution can also be used for refinement-based CbC where abstract program parts are incrementally refined to more concrete program parts. This allows for a fine-grained adaption of the granularity of refinement that ranges between single program statements (as in CorC) and whole methods (as in TraitCbC). The main difference in abstract execution is however that it extends the programming language with abstract program parts and, thus, allows to better reason about irregular termination (e.g., break/continue) of methods.

KeY [3] is a deductive program verifier for Java programs specified with the Java modeling language. KeY is the verification backend of CorC. Besides KeY, there are a number of tools for program specification and verification, such as: the language Eiffel [34] with the verifier AutoProof [53], the languages SPARK [9], Dafny [31], and Whiley [38], and the tools OpenJML [15], Frama-C [17], VCC [14], VeriFast [26], and VerCors [4]. All those are candidates for post-hoc verification tools to be compared with the CbC methodology, or they can serve as backends for guaranteeing the correctness of rule applications in refinement-based CbC, similar to the usage of KeY in CorC. In CorC, we decided to use KeY as backend because of previous familiarity and support from the KeY developer and user community.

3 Correctness-by-Construction in CorC

In this section, we introduce correct-by-construction software development in the previous version of CorC. In Sect. 3.1, correctness-by-construction in CorC is introduced by an example. In Sect. 3.2, we present the basic functionality of CorC with the graphical and textual editor.

3.1 Correctness-by-Construction

The correctness-by-construction program development approach starts with a Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$. An abstract program S is refined stepwise into a correct implementation by applying refinement rules. A refinement rule concretizes the Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$ to $\{\mathsf{P}'\}\ \mathsf{S}'\ \{\mathsf{Q}'\}$. For example, a loop, a conditional statement, or a sequence of statements could be introduced as S'. CbC by Kourie and Watson [29] offers a set of refinement rules that guarantee the correctness of the refined program if specific side conditions hold.

In Fig. 1, we show an example of a linear search algorithm that is created with the graphical CorC tool. Each node with a green border represents a Hoare triple and corresponds to the application of one refinement rule. We start with the program $\{\mathsf{P}\}\texttt{statement}\{\mathsf{Q}\}$ at the top, where `statement` is an abstract statement. The starting precondition $\mathsf{P} := \texttt{appears}(\mathtt{a}, \mathtt{x}, 0, \texttt{a.length})$ states that the element `x` appears in the array `a` inside the boundaries of the array. Here, `appears` is a predicate to shorten the specification. The postcondition $\mathsf{Q} := \texttt{modifiable}(\mathtt{i});\ \mathtt{a}[\mathtt{i}] = \mathtt{x}$ states that the element `x` is in the array at position `i`. We specify with the predicate `modifiable` that only `i` can be altered in the program. In CorC, the accessible variables are defined in a variables box, which is shown on the right side. The function has two parameters `a` and `x` and a local variable `i`. The global conditions box specifies conditions which are valid in every step of the program (i.e. invariants), and hence added implicitly to every Hoare triple.

The first refinement is the application of the composition rule [29], which splits the starting Hoare triple by triples $\{\mathsf{P}\}\texttt{statement1}\{\mathsf{M}\}$ and $\{\mathsf{M}\}\texttt{state-ment2}\{\mathsf{Q}\}$ with an intermediate condition M. The idea of the algorithm is that we traverse the array from back to front and stop, when the element `x` is found. The invariant of the program is $!\texttt{appears}(\mathtt{a}, \mathtt{x}, \mathtt{i} + 1, \texttt{a.length})$. If we have not yet stopped, we know that the element `x` does not appear in the end of the array that is already examined. We also use this invariant as intermediate condition M to establish this condition at the start of our loop. The first abstract statement `statement1` is now refined with a refinement rule for assignments. We refine `statement1` to $\mathtt{i} = \texttt{a.length} - 1;$ to start at the end of the array, and we verify that the Hoare triple $\{\mathsf{P}\}\mathtt{i} = \texttt{a.length} - 1;\{\mathsf{M}\}$ is fulfilled with the concrete instances for P and M. In the example, the green border indicates a successful proof. The second abstract statement `statement2` is refined to a repetition statement using the invariant as discussed above. The loop guard is $\mathtt{a}[\mathtt{i}] \neq \mathtt{x}$. As long as the element `x` is not found, the loop is repeated. Here, we have to prove four conditions. First, the invariant must be established before the first loop iteration. Second, the postcondition P must follow after the last loop iteration. Third, the preservation of the invariant is shown in the last refinement, where we introduce the loop body $\mathtt{i} = \mathtt{i} - 1;$ to iterate through the array. Fourth, the loop must terminate. For termination, a variant is used which decreases monotonically and is bounded from below.

3.2 CorC

CorC [43] is a hybrid textual and graphical IDE to develop correct software using the CbC process. CorC supports programmers to refine programs and to check the correct application of the refinements. A check is for example on the correctness of a Hoare triple, the initial validity of a loop invariant, or the termination of a loop. For each check, CorC prepares a proof goal which is verified by the program verifier KeY [3] integrated in its backend. To be compatible with KeY, CorC prepares proof goals where the code is written in Java syntax with specifications in Java Dynamic Logic (JDL) [3]. The extent of language

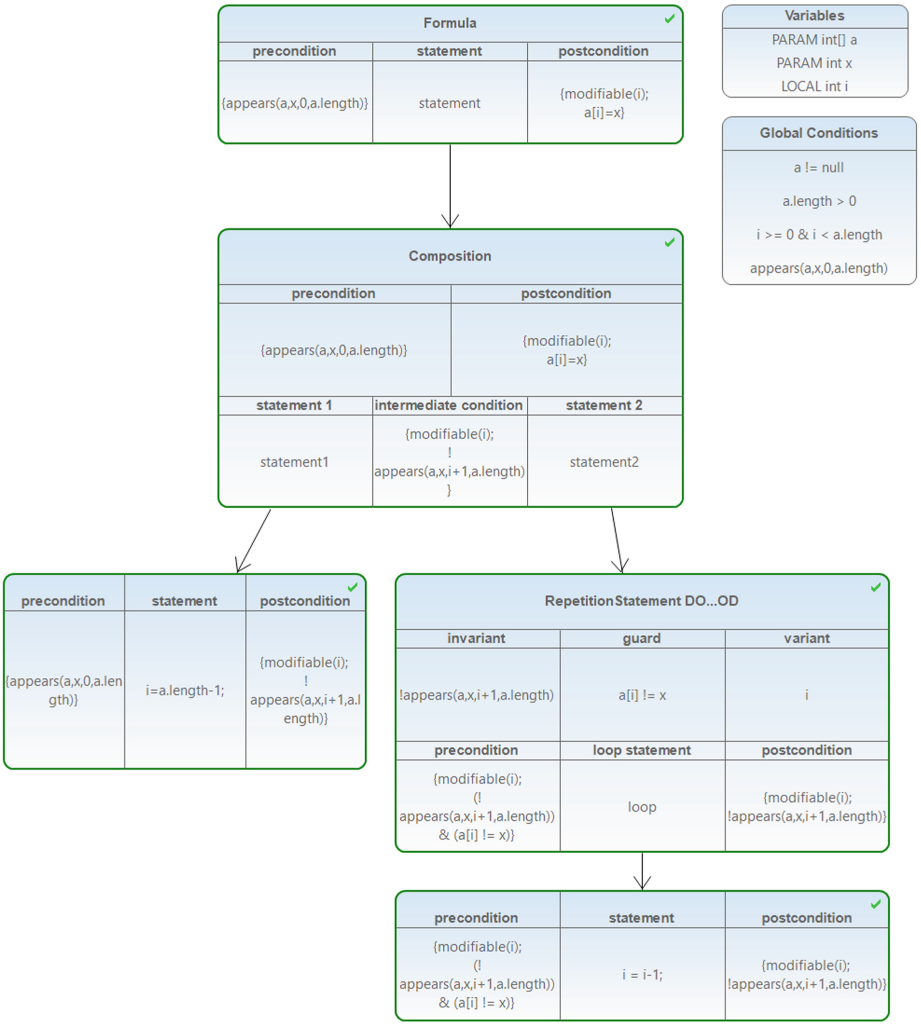


Fig. 1. Linear Search Algorithm Constructed in CorC

constructs covered in CorC is similar to the guarded command language [19] with statements for skip, assignment, function call, composition, selection, and repetition. The CorC IDE also offers a textual editor. The syntax of the textual editor is based on Java that is enriched with keywords for the application of refinements and additional specifications for loop invariants and intermediate assertions. Programs created in the textual editor can be transformed to the graphical editor and vice versa.

In comparison to the verification of complete Java programs in KeY, CorC splits the verification effort of a complete method into the verification of several refinement steps (e.g., checking the refinement of introducing: skip, assignment, function call, composition, selection, repetition, weakening precondition, and strengthening postcondition). In each step, a side condition, such as the establishment of a loop invariant before the first loop iteration, or the correctness of a Hoare triple, is verified. All proofs combined guarantee the correctness of the whole program. This split into several proofs can reduce the proof complexity and proof effort [54]. Thus, CorC can be used as a frontend for KeY that enables a correct-by-construction development process for the construction of individual algorithms.

4 Object-Oriented Development in CorC 2.0

Object-oriented programming is state-of-the-art in software engineering and supported by most modern programming languages. In the previous section, we described how single algorithms can be created using CorC. However, these algorithms are independent of any class structure which means that these single algorithms cannot access the same set of global fields like methods in a class in object-oriented programming can. Additionally, objects containing methods that have been created by CorC can only be created using laborious copy-and-paste workarounds. In other words, the current implementation of CorC can hardly be integrated into a software engineering process as it lacks a concept for modularization and ownership, as well as processes that enable the integration of CbC into a software development workflow. Therefore, in this section we present our concept for CbC-based object-oriented software development and the corresponding extension of CorC in the tool CorC 2.0. CorC 2.0 implements a roundtrip engineering from existing Java projects to CbC-based program development, which allows for a combination of post-hoc and CbC-based program development and verification.

4.1 Object-Oriented Concepts in CorC 2.0

Object-oriented programming is a common programming paradigm based around objects containing data fields and methods. In class-based languages, like Java, C++, C#, PHP, or Smalltalk, objects are instances of classes that have to be defined in advance. Classes are extensible templates for creating objects and provide initial values for fields and implementations of methods. Other paradigms that most object-oriented languages share are encapsulation, inheritance, polymorphism, and dynamic dispatch. As CorC already uses Java syntax in the refinement steps and KeY as deductive verification tool to verify the single refinement steps, we focus on object-orientation as realized in Java and how to combine this with CbC.

Classes. To support object-orientation in CorC 2.0, we introduce the construction of classes that hold methods implemented with CorC and fields that can be accessed by the methods contained in the respective class. The visibility of fields can be modified using the Java visibility modifiers `public`, `private`, `package`, and `protected`. Fields can also be defined as `static` or `final`. Besides fields that have been defined in the class, methods of that class can define a set of local variables including parameters and a return variable. Additionally, we add class invariants as a new specification type to our class definitions. Class invariants specify conditions that are fulfilled by the class, i.e. that are preserved by all methods of that class or re-established at the end of method execution. To guarantee that a method created with CorC fulfills the class invariants, they are automatically added to the pre- and postcondition of the starting Hoare triple when that method is constructed.

Method Calls. Methods can either be called inside of the same class, by an object instantiating the class, or directly by the class if the method is static. The implementation of that method can either be in CorC or in Java. For the verification of a method call, CorC supports inlining and contracting [28] (i.e., inserting its implementation or using its contract as defined in the CbC method call refinement rule). When contracting is used, it is assumed that the contract holds for that method, however, this is not specifically verified in this step.

Framing. Besides a pre- and a postcondition, a frame that contains all variables whose data can be modified is defined for each method. This information helps callers of the method to determine which parts of the state are not changed due to the call [12]. For formal verification of object-oriented programs, framing is important, because the caller implicitly knows which fields remain unchanged during the execution of a method [3,55]. Furthermore, framing is important for information hiding [30] and to avoid unwanted side effects [32]. In CorC, the frame of a method is automatically determined by traversing its refinement steps and collecting the variables that are on the left of an assignment. However, the frame can also be defined manually by the user. For the verification of a method with frame, it is checked whether all variables that are not included in the frame still have the same value as before the execution of that method.

Inheritance and Interfaces. Inheritance and interfaces are two important features in Java. While interfaces can be used as a layer of abstraction, inheritance can be used to create classes built upon existing classes to, for example, enable code reuse. For both, inheritance and interfaces, we check that the Liskov principle is fulfilled by the child class or the class that implements the interface. This means that class invariants that are defined in the parent class or the interface also have to be fulfilled by the class that extends or implements it. Furthermore, if a method is overridden in the child class or implemented from an interface it also has to fulfill the contract that has been defined for that method in the parent class or interface. We do not require an interface for every class.

Limitations. Besides methods, classes also contain constructors that are used to instantiate an object from a class. Even though constructors are very important in object-oriented programming, we do not support their creation and verification in CorC. However, an initialization method which creates a new instance

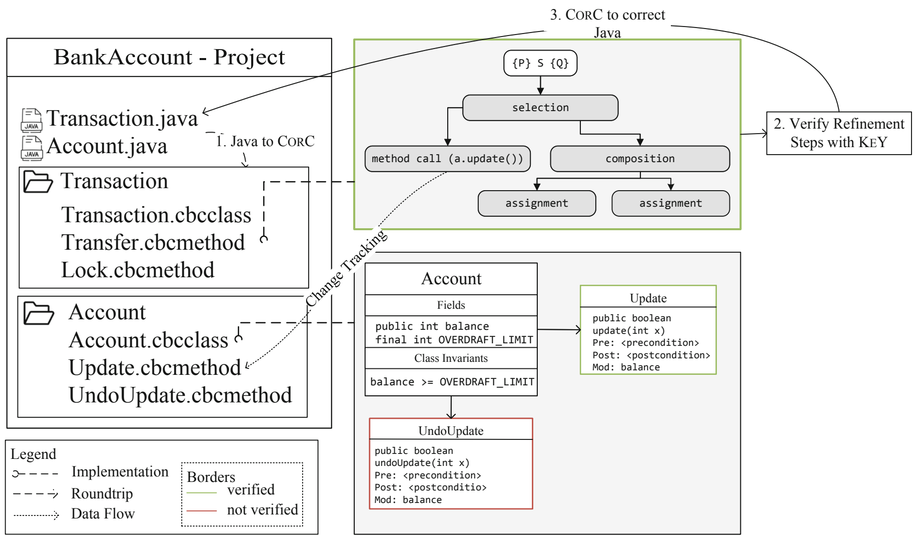


Fig. 2. Development Process in CorC 2.0

by calling a default constructor can be created and verified with CorC. The contracts of methods are limited to a pre- and postcondition pair including a frame. Exceptional behavior such as expecting a specific exception to be thrown cannot be expressed in the contracts, and hence we cannot reason about exceptional behavior.

4.2 Development Process in CorC 2.0

In Fig. 2, we give an overview of the project structure and the development process in CorC 2.0. We do this using an example of a Bank Account software system that consists of two classes. The class `Account` has two methods `update` and `undoUpdate` to manipulate the balance of the account. The class `Transaction` provides the method `transfer` which allows to transfer money from a source account to a destination account and the method `lock` which locks an account such that the balance cannot be changed anymore.

On the left side of Fig. 2, we show the project structure of the Bank Account system. There are two folders named by the two classes `Account` and `Transaction` that hold all files that have been created with CorC for each class. The cbcclass and cbcmethod files are representations of the Java classes and methods in CbC format which can be displayed, edited, and verified by CorC. Each folder contains a cbcclass file which is also named after the class and contains all information related to this class (i.e., class and method information). The implementation of `Account.cbcclass` is shown in the bottom center of Fig. 2 similar to a UML class diagram. There is one bigger box with the title Account that defines the field `balance` and the constant `OVERDRAFT_LIMIT` and a class invariant. If this class inherits from another class or implements an interface this would be defined using the Java keywords `extends` and `implements`. The methods `update` and `undoUpdate` are shown in two separate boxes that are connected to the Account class. They show the method signature and the contract consisting of precondition, postcondition, and framing. Furthermore, their border is either green or red to display their verification status.

Besides the cbcclass files, there is also a cbcmethod file per method in the class folders. The development of methods generally stays the same as before in CorC without object-orientation. The content of `Transfer.cbcmethod` is shown in the top center of Fig. 2. It shows the refinement steps that are used in CorC to construct method `transfer` starting from the starting Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$. Precondition P and postcondition Q are the same as the pre- and postcondition that are contained in `Transaction.cbcclass` for method `transfer`. The single refinement steps are created in CorC and verified with KeY as described in Sect. 3.

In the following, we describe some of the new core features of CorC 2.0, which are important for the integration of CbC into the software engineering development process.

Roundtrip Engineering. To simplify the integration of CorC 2.0 into existing Java software system development, we introduce a roundtrip engineering functionality. This roundtrip engineering process can be used for example for (1) implementing new methods using CbC, (2) guaranteeing the correctness of an already implemented method, or (3) to be able to better track the source of error when the developer fails to prove a certain method with post-hoc verification. Either way, the correct and changed implementation and specification can be integrated back into the original project. The roundtrip engineering is performed in three steps as displayed in Fig. 3. These three steps can be iteratively repeated to create an incremental development process.

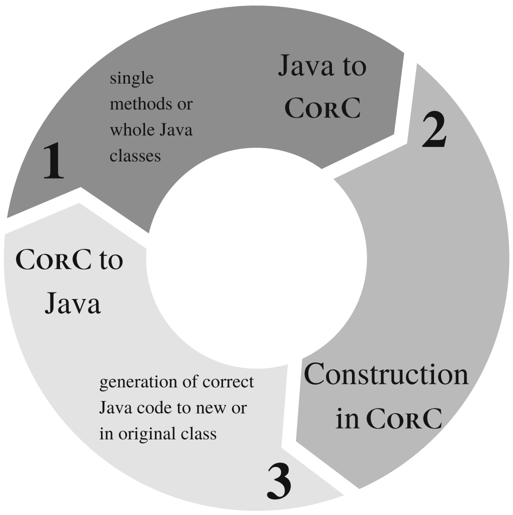

1
single methods or whole Java classes
Java to CorC
2
Construction in CorC
3
generation of correct Java code to new or in original class
CorC to Java

Fig. 3. Roundtrip Engineering Workflow

Step 1: If there are already Java classes that contain methods that need to be verified, the classes and methods can be converted into the cbcclasses and cbcmethods in a first step. During this step, the user can select the methods that shall be converted. The user can then complete missing specifications and annotations (e.g. intermediate conditions, loop invariants, variants, ...) in the cbcmethods. The cbcclasses and cbcmethods can also be created from scratch in CorC, which makes this first step optional. In that case, the user has to manually apply the refinement rules to construct the methods in Step 2.

Step 2: The refinement steps in the cbcmethods are verified. Method calls can either be verified by inlining or contracting (i.e. using either implementation or contract of the called method). If a method call is verified using its contract, the method can either be implemented in CorC or provided in Java specified with a JML contract. This allows the user to freely combine CorC with existing Java methods and post-hoc verification. However, in that step it is not verified whether a called method actually fulfills its contract. Consequently, not all methods need to be specified (when inlining is used) and methods that are specified do not necessarily have to be verified to be called and can be assumed to be correct.

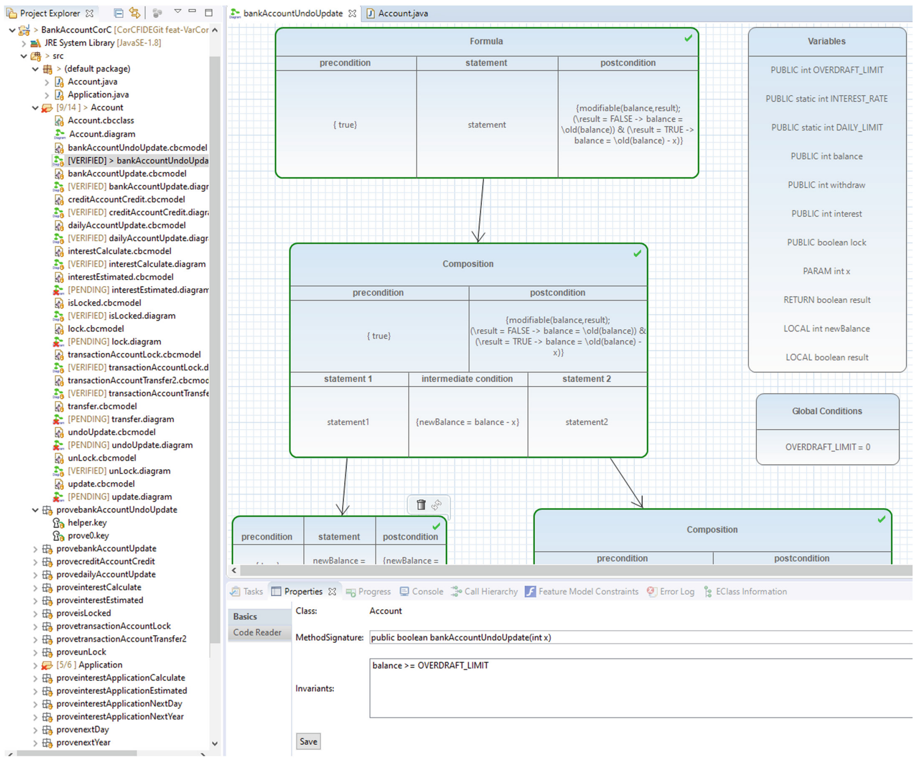
Step 3: CorC can generate correct Java code from the verified cbcmethods, either to a new Java class or back into the original Java class it has been imported from where it replaces the original implementation and contract.

Change Tracking. To further improve the usability of CorC, we introduce a notification system that keeps track of changes by setting already verified refinements to not verified. This is especially critical for methods using method calls, inheritance, or interfaces. In Fig. 2, the implementation of method `transfer` calls method `update` on `Account a`. Now, if there is a change in method `update` in class `Account`, the method call refinement in method `transfer` needs to be re-verified, as the old proof relies on a possibly outdated contract. The change tracking system prevents the developer from overlooking these changes. Additionally, it enables the direct navigation to the affected method (in our example to method `transfer`) for re-verification. In the background, the affected refinement rules are automatically set to not verified so that these refinement steps cannot mistakenly be assumed to be correct. Since CbC has a fine-grained structure with single refinement steps, not all refinement steps of a method have to be re-verified, but only those that are affected by the change. CorC 2.0 can better maintain the correctness of evolving software than its predecessor, since it is no longer possible to have refinements falsely marked as verified.

4.3 Implementation

CorC 2.0[1] is an open-source Eclipse plug-in supporting the development of object-oriented programs using CbC as described in this work. CorC captures the structure of a CbC program including the refinement rules through two meta-models, one for the class files and one for the methods, modeled using the Eclipse Modeling Framework[2]. The graphical editing framework Graphiti[3] is used to visualize the underlying meta-models. For the methods, we use a tree-like

[1] https://github.com/TUBS-ISF/CorC.
[2] https://eclipse.org/emf/.
[3] https://eclipse.org/graphiti/.


Project Explorer
> BankAccountCorC [CorCFIDEGit feat-VarCor
> JRE System Library [JavaSE-1.8]
> src
> (default package)
Account.java
Application.java
[9/14] > Account
Account.cbcclass
Account.diagram
bankAccountUndoUpdate.cbcmodel
[VERIFIED] > bankAccountUndoUpda
bankAccountUpdate.cbcmodel
[VERIFIED] bankAccountUpdate.diagr
creditAccountCredit.cbcmodel
[VERIFIED] creditAccountCredit.diagr
dailyAccountUpdate.cbcmodel
[VERIFIED] dailyAccountUpdate.diagr
interestCalculate.cbcmodel
[VERIFIED] interestCalculate.diagram
interestEstimated.cbcmodel
[PENDING] interestEstimated.diagram
isLocked.cbcmodel
[VERIFIED] isLocked.diagram
lock.cbcmodel
[PENDING] lock.diagram
transactionAccountLock.cbcmodel
[VERIFIED] transactionAccountLock.d
transactionAccountTransfer2.cbcmoc
[VERIFIED] transactionAccountTransfe
transfer.cbcmodel
[PENDING] transfer.diagram
undoUpdate.cbcmodel
[PENDING] undoUpdate.diagram
unLock.cbcmodel
[VERIFIED] unLock.diagram
update.cbcmodel
[PENDING] update.diagram
provebankAccountUndoUpdate
helper.key
prove0.key
provebankAccountUpdate
provecreditAccountCredit
provedailyAccountUpdate
proveinterestCalculate
proveinterestEstimated
proveisLocked
provetransactionAccountLock
provetransactionAccountTransfer2
proveunLock
[5/6] Application
proveinterestApplicationCalculate
proveinterestApplicationEstimated
proveinterestApplicationNextDay
proveinterestApplicationNextYear
provenextDay
provenextYear
bankAccountUndoUpdate
Account.java
Formula
precondition
statement
postcondition
{ true}
statement
{modifiable(balance,result); (\result = FALSE -> balance = \old(balance)) & (\result = TRUE -> balance = \old(balance) - x)}
Composition
precondition
postcondition
{ true}
{modifiable(balance,result); (\result = FALSE -> balance = \old(balance)) & (\result = TRUE -> balance = \old(balance) - x)}
statement 1
intermediate condition
statement 2
statement1
{newBalance = balance - x}
statement2
precondition
statement
postcondition
newBalance =
{newBalance =
Composition
precondition
postcondition
Variables
PUBLIC int OVERDRAFT_LIMIT
PUBLIC static int INTEREST_RATE
PUBLIC static int DAILY_LIMIT
PUBLIC int balance
PUBLIC int withdraw
PUBLIC int interest
PUBLIC boolean lock
PARAM int x
RETURN boolean result
LOCAL int newBalance
LOCAL boolean result
Global Conditions
OVERDRAFT_LIMIT = 0
Tasks
Properties
Progress
Console
Call Hierarchy
Feature Model Constraints
Error Log
EClass Information
Basics
Code Reader
Save

Class: Account

MethodSignature: public boolean bankAccountUndoUpdate(int x)

Invariants: balance >= OVERDRAFT_LIMIT

Fig. 4. Screenshot of CorC 2.0 with Method `update`

structure. The beginning of a method in CbC always consists of a Hoare triple, which can be refined until there are no more abstract statements. Thereby, the pre- and postconditions and other annotations are automatically passed on in each refinement step. Furthermore, the deductive verification tool KeY [3] is used to prove the correct usage of each refinement rule.

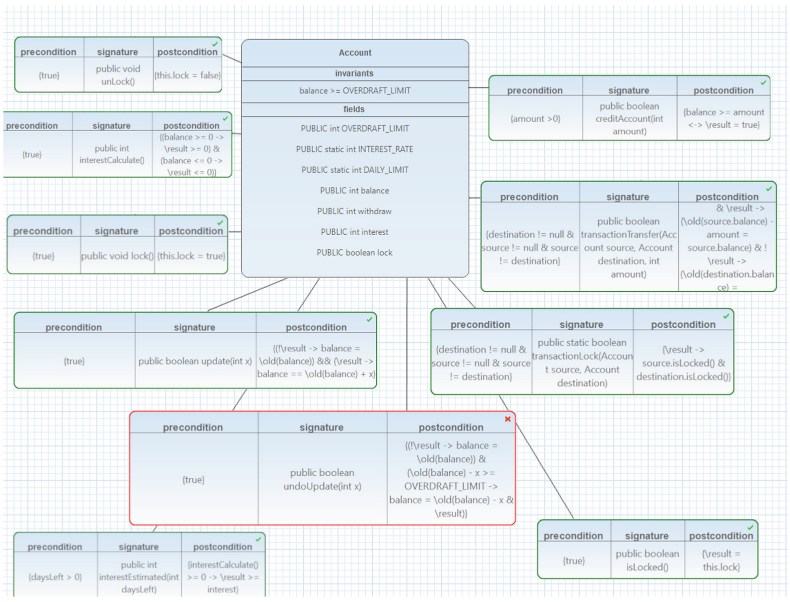
In Fig. 4, we show a screenshot of the graphical view to develop methods in CorC 2.0. On the left side, there is the project structure of the Bank Account case study which has been used as an example in the previous subsection. The Java-classes are in the default package. Then, there are folders named after the classes holding the information about the class and all methods in form of diagram and model files named after the distinct classes and methods. The diagram files (*<methodName/className>.diagram*) contain the graphical representation and the model files (*<methodName>.cbcmodel/<className>.cbcclass*) store the information about the methods and classes in the corresponding meta-model. The *prove<methodName>* folder stores generated proof files, which contain the side conditions for a refinement step which need to be verified. Each proof file is verified (semi-)automatically by KeY. For a better overview, in front of the folder name the proportion of verified methods is given and in front of the

$< methodName >$ *.diagram* files the verification state is given. In this context, *verified* means that all refinement steps of a method could be proven and *pending* means that at least one refinement step is still unverified.

In the center of Fig. 4, we can see the CorC-diagram of method `undoUpdate`. At the top of the diagram, we can see a formula component for our starting Hoare triple. Underneath, we see a refinement step using the composition refinement rule. That refinement rule splits the abstract statement into two abstract statements. Afterwards, these are further refined. We can right-click on these components to trigger a verification process. In the verification process, we translate the Hoare triple of the selected component into a proof file for the corresponding refinement rule in the format required by KeY. All components in the CbC construction tree are marked green in our example so that we can conclude that all refinement steps have been verified. If a refinement step could not be verified, the corresponding component is marked red which enhances the traceability of incorrectly defined refinement rules or inconsistencies in relation to the specifications. At the top right, we can see two boxes which hold the defined variables and global conditions.

At the bottom of Fig. 4, we can see the properties view with the currently active *Basics* tab. It shows further information about the method `undoUpdate`, such as the class it belongs to, its signature, and the class invariants it has to fulfill. The signature of the method can also be edited. To change any other information, the class file has to be opened. The other tab in the properties view is called *Code Reader* and displays the selected Java statements or specifications in the diagram in a bigger window with syntax highlighting. It enables better readability of especially long specifications and helps to modify them more easily without introducing syntax errors.

In Fig. 5, we show a screenshot of the class view in CorC 2.0. It displays the content of file *Account.diagram*. In the top center, we see a component for the class definition that looks similar to a UML class diagram. At the top, it displays the name of the class, and below that, it lists the class invariants and fields with their visibility, type, and name. Around the class component, there are several other components which are the methods that are implemented in this class. They show the signature and the contract of each method. Additionally, their verification status is displayed by the red and green borders. In this view, new methods, fields, and class invariants can be added and existing ones can be edited. Changed information, for example a changed precondition of a method or a changed type of a field, is directly available in all method diagrams since they directly reference this file. In the case of a changed precondition, the user is notified by the change tracking mechanism and the verification status is set to not verified as described in the previous subsection. For an easy navigation to the method diagrams, the user can double-click on a method component.



Account
invariants
balance >= OVERDRAFT_LIMIT
fields
PUBLIC int OVERDRAFT_LIMIT
PUBLIC static int INTEREST_RATE
PUBLIC static int DAILY_LIMIT
PUBLIC int balance
PUBLIC int withdraw
PUBLIC int interest
PUBLIC boolean lock

precondition	signature	postcondition
{true}	public void unLock()	{this.lock = false}

precondition	signature	postcondition
{true}	public int interestCalculate()	{(balance >= 0 -> \result >= 0) & (balance <= 0 -> \result <= 0)}

precondition	signature	postcondition
{true}	public void lock()	{this.lock = true}

precondition	signature	postcondition
{amount >0}	public boolean creditAccount(int amount)	{balance >= amount <-> \result = true}

precondition	signature	postcondition
{destination != null & source != null & source != destination}	public boolean transactionTransfer(Account source, Account destination, int amount)	& \result -> (\old(source.balance) - amount = source.balance) & ! \result -> (\old(destination.balance) =

precondition	signature	postcondition
{true}	public boolean update(int x)	{(!\result -> balance = \old(balance)) && (\result -> balance == \old(balance) + x}

precondition	signature	postcondition
{destination != null & source != null & source != destination}	public static boolean transactionLock(Account source, Account destination)	{\result -> source.isLocked() & destination.isLocked()}

precondition	signature	postcondition
{true}	public boolean undoUpdate(int x)	{(!\result -> balance = \old(balance)) & (\old(balance) - x >= OVERDRAFT_LIMIT -> balance = \old(balance) - x & \result)}

precondition	signature	postcondition
{daysLeft > 0}	public int interestEstimated(int daysLeft)	{interestCalculate() >= 0 -> \result >= interest}

precondition	signature	postcondition
{true}	public boolean isLocked()	{\result = this.lock}

Fig. 5. Screenshot of Class `Account` in CorC 2.0

5 Evaluation

In this section, we evaluate CorC 2.0 by comparing it with post-hoc verification. We use KeY for post-hoc verification, since KeY is also the integrated verifier in CorC, but KeY can be understood as synonym for post-hoc verification. To evaluate whether it is feasible to construct correct programs with CorC, we want to answer the following two research questions:

RQ1: How does the verification effort (w.r.t. execution time and proof steps) of verifying algorithms in CorC compare to post-hoc verification in KeY?

RQ2: How does the CorC development process assist in creating correct programs in comparison to the assistance of KeY for post-hoc verification?

The first research question is answered by creating three case studies with both CorC and KeY, and measuring the verification time and the number of verification steps. Each case study consists of several Java classes containing specified methods. Each method is created and verified with CorC. For the post-hoc verification approach, we verified the methods written in Java and specified with JML (precondition, postcondition, and loop invariants, but no further intermediate specification was given). For method calls, we used contracting to prove correctness, but inlining can be used as an alternative. We always used KeY as automatic verification tool. We measured the verification steps by the number of rule applications of KeY. The verification time was measured five times and the average was calculated. We do not consider the manual effort of

Table 1. Metrics of the case studies

Case Study	#Classes	#Methods	#Verified with CorC	#Verified with KeY
Bank Account [52]	2	10	10	9
Email [24]	2	12	12	8
Elevator [39]	5	18	18	17

writing additional specification for CorC. For the second research question, we qualitatively discuss the CorC tool by referring to two user studies conducted at TU Braunschweig. We also discuss the new features of CorC 2.0.

Case Studies. We have three cases studies that are implemented and verified with CorC and KeY. We decide to use the case studies *Bank Account* [52], *Email* [24], and *Elevator* [39] because they are implemented in an object-oriented fashion, such that we can evaluate the new feature of CorC 2.0. In Table 1, we show some metrics for the case studies. We have two to fives classes and 10 to 18 methods per case study. The size of the methods ranges from 1 to 20 lines of code.

5.1 RQ1 - Verification Time and Verification Steps

To answer the first research question, we verified all 40 methods in CorC. For post-hoc verification, we used the same pre-/postcondition specifications as in CorC. However, six algorithms could not be verified (cf. Table 2). The verification of methods with KeY failed, for example, in the steps where the loop invariant must be proven. In CorC, the verification of loops is split into several smaller proofs reducing the proof complexity. Another reason for a reduced proof complexity in CorC is that we introduce intermediate specifications, which guide the verifier. In contrast, applying post-hoc verification is more coarse-grained, as only precondition, postcondition, and loop invariants are specified. We observed that debugging verification problems that could not be verified automatically in post-hoc fashion, is more challenging than debugging the same problem constructed with CbC in CorC.

In Fig. 6, we show the average verification time measured in milliseconds, and in Fig. 7, we show the average verification steps for each case study, but only for the 34 methods which could be verified with CorC and KeY. The verification time ranges from 0.1 s for some methods to 16 s for the most complex methods. For 22 methods, the verification time with CorC is faster, for 12 methods the verification with KeY is faster. The largest differences are: `enterElevator` is 268% faster with CorC, `addWaitingPerson` is 271% faster with KeY. The average verification time for the *Email* case study shows that the verification is 23% faster with CorC, but on average the *Elevator* case study is 24% faster with KeY. For the number of proof steps, we got similar results. In 26 cases,

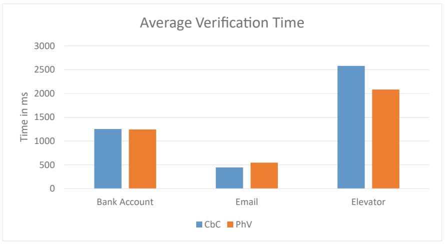


Fig. 6. Average Verification Time of the Case Studies with CbC and PhV

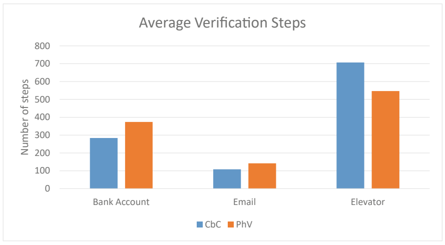


Fig. 7. Average Verification Steps of the Case Studies with CbC and PhV

CorC has fewer steps, and in 8 cases, KeY has fewer steps. The case studies *Bank Account* and *Email* are on average 32% and 31% smaller with CorC, and *Elevator* is 29% smaller with KeY. Overall, the results are of the same order of magnitude. The exact verification time and steps for each method are shown in Appendix A.

Discussion. Regarding the verification effort, no trend can be identified. Therefore, we cannot answer the research question positively that the verification with CbC is faster than with PhV in terms of verification time or verification steps. The verification time and the verification steps are similar for both approaches. However, we could verify more methods in total with CorC. The additional specifications in the form of intermediate annotations and the splitting into several smaller proofs facilitates the completion of proofs, but it does not significantly affect the verification effort. As we are promoting to use CbC and PhV in concert, a similar verification effort is beneficial. There is no discernible disadvantage in terms of effort in using one of the two approaches.

5.2 RQ2 - Usability of CorC

For the second research question, we conducted two user studies [42,45] with a total of 23 students from the TU Braunschweig. In both studies, a group of students was divided into half. One group created and verified a method with CorC and a second method with KeY. The second group created the same methods, but used the tools in reverse order. We measured the defects in the developed and verified methods and performed a questionnaire on the usability of CorC and KeY at the end of the user study. The first user study took place in person in 2019 [45], the second user study was online in 2021 [42]. In both user studies and with both tools, we had several correct implementations but which were not verified in the given time frame. A common problem was a too weak loop invariant. In the usability questionnaire, most participants preferred CorC over KeY due to the better and more fine-grained feedback when errors occurred during the verification. It was easier to detect and solve errors with CorC. A minority of participants preferred KeY because they were more familiar with the syntax of Java and JML. As we are now supporting roundtrip development with CorC 2.0, we believe that this statement is weakened. Users can now freely develop and verify programs with CorC or KeY and generate the program for the other approach automatically. Thus, the preferred tool can be used without restrictions.

During the construction of the three case studies (cf. RQ1), CorC's change tracking feature was valuable. When we verified a refinement step that called another method, it occurred that we had to change the contract of this called method. The change tracking feature then set the affected method calls in all CorC programs to not be verified. With this feature, we did not miss any open verification obligations. In summary, we can confirm that CorC assists in developing correct software. Additionally, with the new features of CorC 2.0, the implementation of object-oriented code is supported.

5.3 Threats to Validity

External Validity. The methods implemented in the three case studies have a size of 1 to 30 lines of code. These small methods reduce the generalizability for larger algorithmic problems. While CorC 2.0 extends the application field to object-orientation, we still consider CorC to be a tool for smaller, but challenging algorithmic problems. The generalizability of the user studies are limited as only 23 students participated, but due to the small number of participants, we were able to interview them in more detail. Nevertheless, the participants were no experts in verification. We classify the participants as junior developers.

Internal Validity. We wrote most of the specifications of the three case studies ourselves. Thus, there could still be defects in the specifications or implementations. However, we were able to verify all methods with CorC and most methods with KeY, which is a strong indication of correctness of the case studies. In particular, we checked the equality of the specifications for the two different approaches. The time frame of the user study was limited: the participants had only 30 min for each method implementation. With more time, we expect that more participants verify the assigned methods.

6 Beyond Monolithic Program Construction with CorC

The previous sections emphasize our ongoing vision to integrate the correctness-by-construction methodology into the realm of mainstream software development through sophisticated tool support. Besides extending CorC's programming model with advanced paradigms, such as object orientation (cf. Sect. 4), a natural follow-up is to develop concepts and tool support that make the correct-by-construction approach available *at scale*. Currently, we aim to address this by two further directions. First, VarCorC is a framework for CbC-based development of *variational* software. That is, instead of developing monolithic programs, the goal is to systematically construct a family of similar software programs following the CbC paradigm. Second, we study the role of correct-by-construction implementations in software architectures with ArchiCorC, where the main goal is to *bundle* CorC programs into *reusable software components*. We briefly present our vision for both lines of research in the following.

VarCorC: Correct Variational Software Construction

Software product lines [40] are increasingly used to lower the costs in producing custom-tailored software products, also including the domain of safety-critical software systems. VarCorC is an extension of CorC to create feature-oriented software product lines [11]. Software product lines are families of related programs sharing a common code base [18]. The common and varying parts of a product line are defined by features. In feature-oriented programming [5,10], features are implemented by feature modules. Similar to inheritance in object-oriented languages, in a feature module, methods can be added or overridden in a specific order defined by the product line. Overridden methods can use the original keyword to call the previous implementation of that method. The feature configuration then defines the explicit relationship between feature modules. For VarCorC, we extended CbC by two new *variability-aware refinement rules*, one for original calls and one for variational method calls. Since implementation of a method depends on the different features, method calls encompass variability in software product lines. Additionally, we integrated the concept of *contract composition* [51] to allow for a varying method contract per feature that can be composed along a feature configuration to form the contract of a method in a product of the product line.

ArchiCorC: Correct Software Architectures

The benefits of combining correctness-by-construction with *component-based software engineering* [16,46,49] are manifold. For instance, component-based architectures allow to establish a repository of correct-by-construction components. This is not only interesting for standard libraries, where implementations are accessed through explicit interfaces, but also for third-party developments that are easier to integrate into personal projects. Most importantly, modularization of correct implementations into components allows developers to think about how to *compose* software systems instead of how to program a monolithic software system from scratch. We argue that this is the foundation for building large and complicated systems that are based on the correctness-by-construction approach.

As an extension to CorC, we propose a framework and an open-source implementation named ArchiCorC [27] that connects UML-style component modeling, specification, and code generation. In more detail, ArchiCorC comprises four key ingredients. First, a *component and interface description language* is used to describe interconnections between provided and required interfaces of components, where interfaces comprise method signatures that are specified with

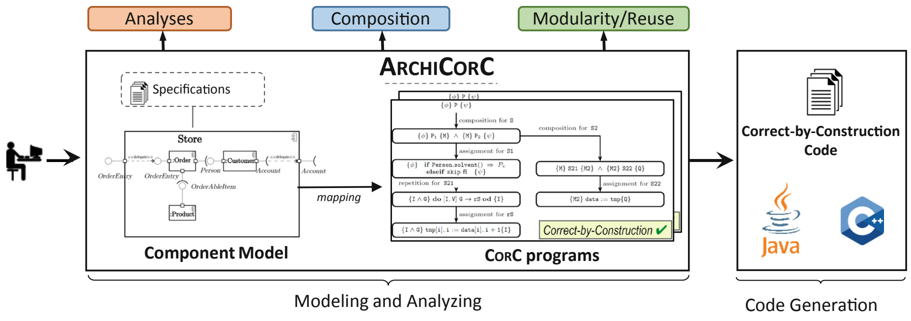


Fig. 8. Envisioned Workflow of the ArchiCorC Development Process

Hoare triples. Second, a *construction technique* aids developers with either refining method signatures of provided interfaces to correct implementations using CorC itself, or with mapping signatures to already existing CorC programs. Third, ArchiCorC integrates analyses and algorithms to check compatibility between components and to build composites to form larger components. Finally, ArchiCorC allows to generate code in a general-purpose programming language (e.g., Java).

In Fig. 8, we illustrate an envisioned development process using ArchiCorC. A developer starts by designing a high-level component model including required and provided interfaces and connections between them. Hierarchical composition allows to build *larger* components from a set of smaller ones. Method signatures of provided interfaces of atomic components (i.e., components that are not decomposed any further) must be mapped to CorC programs, which are assumed to be correct-by-construction. The component model can then be translated to a general-purpose programming language (e.g., Java or C++) and can be imported into other projects. This development process embodies the next milestone of our ongoing vision of correct-by-construction software development.

7 Conclusion

Our long-term vision is to make the correctness-by-construction (CbC) approach accessible for large-scale software development. The formal framework of CbC enables developers to start with a specification for safety-critical parts and algorithms. Then, the framework guides developers in deriving a provably correct implementation. A major stepping stone towards this vision is the development of tool support that allows to apply traditional software development and CbC in concert. In this work, we presented state-of-the-art tool support for this mission, namely the CorC 2.0 tool family, including a comprehensive overview of its current status and prospective directions.

In particular, we introduce CorC 2.0 (the successor of CorC), which combines CbC with object-oriented software development in Java. In CorC, single algorithms are developed completely independently of other programs, making the integration into software engineering processes impractical. As a new key feature, CorC 2.0 adds a class concept for structuring programs (i.e., methods) inspired by Java's object-oriented programming model. CorC 2.0 now supports a roundtrip engineering process that is important for applying CbC to existing software projects; existing Java classes comprising specified methods can be converted into CorC 2.0 projects and, after verified construction, they can be converted back to verified Java implementations. This strong improvement in tool support directly targets scalability concerns of the CbC approach, as it allows developers now to decide whether parts are (1) verified using post-hoc verification techniques, (2) implemented following the CbC approach, or (3) remain unverified.

In alignment with our vision, we believe that CorC 2.0 allows developers to address program verification of general-purpose programming languages in a new way. One outcome of our evaluation illustrates that CorC 2.0 can sometimes outperform post-hoc verification with respect to verification effort and success rate. Although specification effort can be higher, the benefits of additional specification together with the fine-grained refinement rules are easier debugging and better feedback in general. Our future direction with the CorC ecosystem is therefore a seamless integration into mainstream software development. One focal point is to identify and study possible synergies when applying post-hoc verification and CbC in concert. Another focal point is to conduct larger user studies, which provide important insights on how CbC is applied in practice and, consequently, influence the development of the CorC ecosystem in general. Only the recent advancements made in CorC 2.0 enable us to develop more complex case studies necessary to address these future directions.

Acknowledgement. We thank Maximilian Kodetzki from TU Braunschweig for implementing large parts of the new features for CorC 2.0.

A Appendix

Table 2. Verification Time and Verification Steps of All Methods

Method	#Steps CbC	#Steps PhV	Time in ms CbC	Time in ms PhV
undoUpdate	205	183	730,75	747,3
update	181	188	626,75	627,67
creditAccount	25	45	104,5	132,67
dailyUpdate	467	636	1990,5	2261
interestCalculate	422	Unclosed	1164	Unclosed
transactionLock	432	663	2253	1995,67
transactionTransfer	799	989	3556,75	3323,3
unLock	21	41	104,5	175,67
interestNextDay	233	381	1066,25	1154,67
interestNextYear	191	239	836,75	786,3
constructClient	50	68	207,75	292,3
createClient	1391	Unclosed	7108,5	Unclosed
getClientByAdress	1420	Unclosed	4509,75	Unclosed
getClientById	67	Unclosed	210,75	Unclosed
outgoing	67	61	212,75	200
resetClients	673	Unclosed	2450,75	Unclosed
constructEmail	36	42	111,25	156,33
createEmail	558	772	2591	2996,67
setEmailBody	38	47	110,25	185
setEmailFrom	40	51	104,5	192,33
setEmailSubject	38	47	113,25	170,67
setEmailTo	38	47	111,5	173,67
areDoorsOpen	64	41	229,25	141,67
buttonIsPressed	70	69	206	144
enterElevatorBase	805	1092	4829,5	4577
enterElevator	457	2315	2370	8710
pressButtonBase	87	94	317,25	529
pressButton	97	109	444,25	503,33
resetFloorButton	79	84	336,25	287
reverse	192	108	985,25	338,33
stopRequestedBase	878	1150	3179	4367,67
createEnvironment	1367	Unclosed	4375,5	Unclosed
isTopFloor	86	97	255,5	345,33
addWaitingPerson	4768	1041	16012,25	4320,33
callElevator	28	42	105,5	174,67
createFloor	346	1066	1641,75	4027
reset	29	42	106,25	173,67
createPerson	3835	1764	12323	6171,67
PersonenterElevator	163	134	432	393,67
PersonleaveElevator	30	44	105,25	174,67

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. 1st edn. (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book (2016)
4. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_5
5. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines (2013)
6. Back, R.J.: Invariant based programming: basic approach and teaching experiences. Formal Aspects Comput. **21**(3), 227–244 (2009). https://doi.org/10.1007/s00165-008-0070-y
7. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73770-4_4
8. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer Science & Business Media (2012). https://doi.org/10.1007/978-1-4612-1674-2
9. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education (2003)
10. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30**(6), 355–371 (2004)
11. Bordis, T., Runge, T., Schaefer, I.: Correctness-by-construction for feature-oriented software product lines. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 22–34 (2020)
12. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Softw. Eng. **21**(10), 785–798 (1995)
13. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_5
14. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
15. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35

16. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2010)
17. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
18. Czarnecki, K., Østerbye, K., Völter, M.: Generative programming. In: Hernández, J., Moreira, A. (eds.) ECOOP 2002. LNCS, vol. 2548, pp. 15–29. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36208-8_2
19. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
20. Dijkstra, E.W.: A Discipline of Programming. 1st edn. Prentice Hall PTR (1976)
21. Gries, D.: The Science of Programming. 1st edn. (1981)
22. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Component-based Synthesis Applied to Bitvector Programs
23. Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_4
24. Hall, R.J.: Fundamental nonmodularity in electronic mail. Autom. Softw. Eng. **12**(1), 41–79 (2005)
25. Heisel, M.: Formalizing and implementing Gries' program development method in dynamic logic. Sci. Comput. Program. **18**(1), 107–137 (1992)
26. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
27. Knüppel, A., Runge, T., Schaefer, I.: Scaling correctness-by-construction. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 187–207. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_10
28. Knüppel, A., Thüm, T., Padylla, C., Schaefer, I.: Scalability of deductive verification depends on method call treatment. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 159–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_15
29. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming (2012)
30. Leavens, G.T., Müller, P.: Information Hiding and Visibility in Interface Specifications, pp. 385–395 (2007)
31. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
32. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Trans. Program. Lang. Syst. **24**(5), 491–553 (2002)
33. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
34. Meyer, B.: Eiffel: a language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988)
35. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992)
36. Morgan, C.: Programming from Specifications. Prentice Hall (1998)
37. Oliveira, M., Cavalcanti, A., Woodcock, J.: ArcAngel: a tactic language for refinement. Form. Asp. Comput. **15**(1), 28–47 (2003)

38. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 238–248. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_13
39. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001)
40. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques (2005)
41. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. ACM SIGPLAN Not. **51**(6), 522–538 (2016)
42. Runge, T., Bordis, T., Thüm, T., Schaefer, I.: Teaching correctness-by-construction and post-hoc verification – the online experience. In: Ferreira, J.F., Mendes, A., Menghi, C. (eds.) FMTea 2021. LNCS, vol. 13122, pp. 101–116. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91550-6_8
43. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
44. Runge, T., Servetto, M., Potanin, A., Schaefer, I.: Traits for Correct-by-Construction Programming. To be published (2021)
45. Runge, T., Thüm, T., Cleophas, L., Schaefer, I., Watson, B.W.: Comparing correctness-by-construction with post-hoc verification—a qualitative user study. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) FM 2019. LNCS, vol. 12233, pp. 388–405. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54997-8_25
46. Sametinger, J.: Software Engineering with Reusable Components. Springer Science & Business Media (1997)
47. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 319–336. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_20
48. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 341–355. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58156-1_24
49. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Pearson Education (2002)
50. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 1–45 (2014)
51. Thüm, T., Knüppel, A., Krüger, S., Bolle, S., Schaefer, I.: Feature-oriented contract composition. J. Syst. Softw. **152**, 83–107 (2019)
52. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, p. 11–20. GPCE 2012, Association for Computing Machinery, NY (2012)
53. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_53

54. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_52
55. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic, and predicate abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)

Specifying the Boundary Between Unverified and Verified Code

David R. Cok[1](✉) and K. Rustan M. Leino[2]

[1] Safer Software, LLC, Rochester, NY, USA
david.r.cok@gmail.com
[2] Amazon Web Services, Seattle, WA, USA
leino@amazon.com

Abstract. This paper introduces a specification construct that is fitting when combining verified code with unverified code. The specification is a form of precondition that imposes proof obligations for both callers and callees. This precondition, `recommends`, blends in well with the parameter-validation conventions in Java and with the syntax and semantics of specification languages such as JML, ACSL, and Dafny.

1 Introduction

In a Utopian world, every line of software we run would be formally verified. That's not the world we live in. Even with the increasing industrial demand for and use of verified software, there will always be components of our software systems that are not verified. This may be an intentional business decision, just like a manufacturing company today makes a decision about the strength of plastic to use in a product, or it may reflect a transient situation where unverified software components are gradually being replaced by verified components.

Whatever the situation, there are issues to consider on the boundary between unverified and verified software. Indeed, by hardening this boundary, we may also be able to ease the transition to software with more verified components.

In this paper, we focus on the interface to a verified component. In particular, we want to address the question of how to allow calls into a verified component.

As usual in specification and verification, a method has a *precondition* that must hold at the time of a call. For example (the code examples are written in a blend of Dafny [12,13], Java [9], and the Java Modeling Language (JML) [6], but are intended to be language-agnostic), the method

```
method Example(x: int)
  requires 0 <= x < 10
```

declares as a precondition that parameter x must be in the range `0` to `10`. It is the *caller's* responsibility to establish the precondition (see e.g., [16,1,10]). In the situation we're focusing on in this paper, the code of the caller may not be verified, which means that the callee really has no guarantee that the precondition will hold on entry. Instead, we propose an alternative precondition, which we write as

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 105–128, 2022.
https://doi.org/10.1007/978-3-031-08166-8_6

```
method Example(x: int)
  recommends 0 <= x < 10
```

This form of precondition still documents the *intention* that the given condition hold on entry to the method. The difference from the standard `requires` precondition is that `recommends` shifts some responsibility from the caller to the callee. It is important to allow the callee to meet this responsibility using the natural patterns of programming that are used today, for example in Java. It is also important to let the construct pave the way to more verified code. Our proposal addresses both of these points.

In the next three sections, we review some points about parameter validation, method specifications, and the strictness of a static verifier. In Sect. 5, we give the precise details of the basic `recommends` construct, and in the subsequent three sections we extend the basic construct to some variations. In Sect. 9, we report on a prototype of `recommends` implemented in OpenJML for Java/JML and on possible extensions that incorporate `recommends` into Dafny.

2 Parameter Validation

Embracing the *fail-fast* philosophy, a common programming pattern in Java and other languages is for a method to validate its parameters on entry. This validation typically consists of checking some simple property that is expected to hold of a parameter and throwing an exception if the property does not hold.

For example, a method that expects a non-`null` reference to a `Cell` and an integer in the range `0` to `10` is written

```
method FailFastExample(c: Cell, x: int) {
  if c == null {
    throw ArgumentNullException();
  }
  if ¬(0 <= x < 10) {
    throw BadParameterException();
  }
  // ...
}
```

On entry, this method immediately rejects unsupported parameter values.

Any exception thrown by a parameter-validation check indicates a program error, a failure on behalf of the caller to use the method correctly. They are what Goodenough calls *client failures* [8]. Even in languages that allow such exceptions to be caught, there's usually not much of a recourse, other than letting a backstop clean up the program state before exiting the program gracefully, or in some cases obtaining new input to the method and trying again.

The common documentation style in Java and .NET is to be obsessively specific about which exception is thrown in response to which unsupported parameter values. While looking at the information associated with such an exception is useful when debugging a program crash, we maintain that the particular exception thrown is not so important to the program itself. Indeed, it is not a good idea for a program to read too much into the parameter-validation exceptions. For example, consider a caller that tries to use the callee's parameter validation to check the value of the parameters being passed in:

```
try {
  FailFastExample(c, 0);
} catch ArgumentNullException {
  // silly me, I must have called FailFastExample with c == null
  // ...
}
```

If `c` is `null` in this call to `FailFastExample`, then the exception handler will be invoked. But it's a mistaken view to think that this is the only situation under which the exception handler is invoked. It may be that the implementation of `FailFastExample` is buggy and passes in `null` to another method it calls, which would also result in the exception handler above being invoked. So, it is safer to consider all parameter-validation exceptions as catastrophic errors rather than trying to handle them in overly specific ways.

The explicit `if` statements in the implementation of `FailFastExample` above are not the only way to perform parameter validation. Let us describe two alternatives that are common in practice.

One alternative is for the code to perform an operation that is sure to result in an exception when a parameter has an unsupported value. For example, in a language like Java, which throws a `NullPointerException` if a `null` pointer is dereferenced at run time, the example method above can be written

```
method FailFastExample(c: Cell, x: int) {
  var y := c.data; // this throws a NullPointerException if c is null
  if ¬(0 <= x < 10) {
    throw BadParameterException();
  }
  // ...
}
```

By writing the code in this way, one saves the explicit check. But since the new check is implicit in the language semantics, it is easier to forget to include it at the right time in the method. Without a code comment like the one above, it may also be difficult for a human to determine which pointer dereferences are in effect parameter validations and which ones are expected to work. Finally, the exception thrown by the language semantics may be slightly different from one chosen in an explicit check (as the examples above show), but, as we argued, the particular exception thrown is not so important.

The second alternative to an explicit parameter-validation check is to delegate the check to another method. For example, if the `FailFastExample` method calls a companion method that is going to validate the parameters, then `FailFastExample` can leave the check for the companion method.

```
method FailFastExample(c: Cell, x: int) {
  // the following method starts by validating the parameters
  // c and x, so we delegate the checking to it
```

```
    ASimilarFailFastExample(c, x, 100, true);
    // ...
  }
```

To stick with the fail-fast philosophy, these alternatives are advisable only if they can be placed near the entry to the method, in order to avoid any state changes before the parameters have been validated. For example, if the first dereference of `c` does not happen until later in the method when the method has already started its intended job or if the call to the companion method doesn't always take place, then parameter validation must be done explicitly.

3 Method Specifications

In this paper, we write method specifications following the method signature. We have already seen a glimpse of the two kinds of preconditions we'll consider, `requires` and `recommends`. These say what is expected to hold on entry to the method. In addition, a method specification can relate the method's post-state to its pre-state, which is done using a combination of three *post-specification* clauses. These have the form

```
modifies Frame
ensures Post
throws Exc ensures EPost
```

A `modifies` clause describes which parts of the program state are allowed to be changed by the method. The details of how `Frame` is specified are not important to this paper. The only thing we need to know is that the absence of a `modifies` clause means the method is not allowed any visible change to the program state.[1]

An `ensures` clause describes what is expected to hold on *normal* exit from the method. In other words, if the method terminates normally (that is, terminates without throwing an exception), then `Post` relates the method's pre- and post-states. Since `Post` is a two-state predicate, we use the expression `old(E)` to denote the expression `E` evaluated in the method's pre-state.

A `throws` Exc `ensures` clause describes what is expected to hold on exit from the method, in the event that the method exits by throwing an `Exc` exception. The specification can include several `throws ensures` clauses, each possibly mentioning a different exception. For the purposes of this paper, we ignore Java's `throws` clauses. That is, we allow methods to throw any exception, as long as all `throws ensures` clauses are satisfied.

We also allow a clause

```
throws * ensures EPost
```

which describes what is expected to hold on exit in the event that the method exits by throwing an exception not covered by any other `throws ensures` clause in the method post-specifications.

[1] This coincides with the interpretation of `modifies` clauses in Dafny. In JML, frames are defined using `assignable` clauses, and an absent `assignable` clause means `assignable \everything`. For our discussion here, `assignable \nothing` is more convenient as a default.
.

For example,

```
method PostSpecifications(c: Cell, x: int)
  modifies c
  ensures c.data == old(c.data) + 1
  throws ArithmeticException ensures x == 0 && c.data == old(c.data)
```

declares that method `PostSpecifications` does not have an effect on any object except possibly `c` and that upon normal termination, `c.data` is `1` more than it had been in the method's pre-state. The specification also says that if the method happens to terminate with an `ArithmeticException`, then the parameter `x` was `0` (in other words, this specification clause says that `x == 0` is the only circumstance in which the method is allowed to throw an `ArithmeticException`) and the value of `c.data` is unchanged.

In this paper, it will be convenient to group post-specification clauses according to what holds in the pre-state. For a pre-state condition `Guard` and a list of post-specification clauses `PS`, we will write

```
when Guard
  PS
```

to express that post-specifications `PS` apply only if the condition `Guard` holds in the pre-state. If `Guard` does not hold in the pre-state, then `PS` is ignored.[2] The $*$ in any `throws * ensures` clause in `PS` denotes those exceptions not covered by other `throws ensures` clauses in `PS`.

4 Language Discipline Versus Verifier Discipline

When designing a static verifier for an existing programming language, it is common (and desirable) to enforce a stricter discipline than the language requires. As an example, the C and Java languages implement modulo semantics for operations on fixed-bit-width integers; for instance, addition of 32-bit `int`s silently overflows to negative values. However, as overflows frequently are unintended and mask bugs, the verification discipline might be stricter, at least by default, warning the implementer by reporting potential over- and underflows in fixed-bit-width operations as errors.

In a language like Java, a choice about what discipline to enforce also arises in those places where the Java language definition prescribes run-time exceptions to be thrown. For example, the effect of an assignment `y := c.data;` at run time is really

```
if c == null {
  throw NullPointerException();
}
y := c.data;
```

[2] JML uses an operator called `also` to combine specifications. It gives a nice way to express what we write with `when` in this paper. Nevertheless, we chose not to use the `also` operator in this paper, because `also` in essence uses `requires` clauses for the guards. By using the `when` notation in this paper, we avoid any confusion regarding `requires` and `recommends` in this context.

A static verifier that uses this loose language discipline of pointer dereference would have to deal with all of the control paths that the `throw` operation gives rise to. A respectable alternative is to instead enforce the stricter discipline of banning `null` from ever being dereferenced. So, if a program possibly dereferences `null`, then the verifier reports an error rather than analyzing any exceptional control paths. (You may think of this stricter discipline as the verifier's fail-fast alternative to the loose language discipline.)

For flexibility, a static verifier can let the user choose between the language discipline and the stricter discipline. For this purpose, the KeY tool [1] supports both a `ban` mode and an `allow` mode. In this paper, we will use the stricter discipline unless explicitly directed by the user to use the language discipline. The directive we propose is to write `//@ allow` at the end of a line, which will cause the verifier to use the language discipline for any construct on that source line that has a choice between the two.

For example, consider the following program snippet in a context where `z` may be `0` and `c` may be `null`:

```
try {
  x := 100 / z;
} catch ArithmeticException {
  y := c.data;
}
```

Under the stricter discipline, the verifier will detect and report the potential division by zero. No `ArithmeticException` is thrown, so the `catch` block and its dereference of `c` are not reachable. In contrast, by including `//@ allow`, the program snippet

```
try {
  x := 100 / z; //@ allow
} catch ArithmeticException {
  y := c.data;
}
```

directs the verifier to use the language discipline for the statement that assigns to `x`. Under this discipline, the attempt to evaluate `100 / z` when `z` is zero turns into a jump to the exception handler. No error is reported for the attempt to divide by zero, but the verifier detects and reports the potential `null`-dereference error in the `catch` block.

In its form above, `allow` uses the language discipline for all constructs where a choice exists on the given line. We also permit `allow` to take a list of exception names, with the effect of using the language discipline on that line only for the indicated exceptions.

We'll come back to `allow` directives in Sect. 6, where we show how the new `recommends` clauses benefit from the directives.

5 Recommends Clauses

With those introductory sections out of the way, we now define `recommends` clauses.

Just like the standard precondition declaration `requires P`, the declaration `recommends P` says that `P` is a precondition for the method. That is, the intention is that `P` hold on entry to the method. For this reason, the `recommends` condition is checked by the verifier at call sites. Of course, this only refers to the call sites that the verifier is aware of, so in the presence of unverified code, it can still happen that the method is called from a state where the precondition does not hold. But whenever the verifier encounters a call to a method with a `recommends` clause, it will check that the given condition holds at the call site.

For method *implementations*, there is a big difference between `requires` and `recommends`. To guard a method implementation from breaches of the precondition at call sites where the verifier has not been applied, one could imagine automatically compiling the `recommends` clause into a run-time check. For example, a method declaration

```
method M(u: U)
  recommends P(u)
{
  Body
}
```

could be compiled in the same way as

```
method M(u: U) {
  if ¬P(u) {
    throw PreconditionFailure();
  }
  Body
}
```

Although simple, this strawman solution falls short in at least a couple of ways.

One shortcoming is that this strawman can give rise to unnecessarily repeated run-time checks. For example, if (like we saw in Sect. 2) `Body` calls another method that also performs parameter validation, then some parameter-validation checks would be performed once in `M` and once in the other method.

A second shortcoming of this strawman is that the condition `P(u)` may not be compilable. Specifications (like those in JML and in Dafny) can contain specification-only features or ghost variables that are not available at run time. In these cases, it would be nice to allow the more abstract condition `P(u)` in the `recommends` clause as part of the method's specification, and to allow the implementation of the method to prescribe an alternate way to check `P(u)` in terms of compilable constructs.

For these reasons, we abandon the idea of automatically compiling `recommends` clauses. Instead, we let the programmer write the code for testing the `recommends` conditions manually, throwing parameter-validation exceptions if the conditions do not hold, and generate verification conditions that check the correctness of that manually authored code. That is, suppose method `M` is declared by

```
method M(u: U)
  recommends Pre
  modifies Frame
  ensures Post
  throws Exc ensures EPost
```

From the point of view of verifying the body of M, it is as if its specification had been written

```
method M(u: U)
  when Pre
    modifies Frame
    ensures Post
    throws Exc ensures EPost
  when ¬Pre
    ensures false
    throws PreconditionFailure ensures true
    throws ∗ ensures false
```

This specification says that, if Pre holds on entry to the method, then the method is bound by the given post-specification. If Pre does not hold on entry to the method, then the method body must arrange to throw a PreconditionFailure exception—it is not allowed to terminate any other way. Furthermore, if Pre does not hold on entry, then (since the second post-specification group does not have a modifies clause) the method is not allowed to modify anything.

Note that in this rewritten specification—which, remember, is from the point of view of the method implementation—there is no precondition. In particular, the method body is *not* allowed to assume the condition Pre on entry.

Here is an example with a method body:

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  if c == null || x < 0 || 10 <= x {
    throw PreconditionFailure();
  }
  c.data := c.data + x;
}
```

This method implementation meets its specification. The method body itself checks that the conditions stated in the recommends clauses hold. If they do not hold, the method modifies nothing and throws a PreconditionFailure exception. If the conditions do hold, the method body increments c.data as described by the post-specification clauses.

Two more points are worthy of attention in this example.

One point is to remember that the recommends conditions are not assumed on entry to the method body. If they were assumed, the “then” branch of the if statement would be unreachable code, and then the verifier would not detect if the method failed to throw the desired exception or if the “then” branch contained any other errors. That would be most regrettable, because the control flow through such a “then” branch is less likely to be thoroughly exercised by testing, so it is especially important that the verifier scrutinize such code.

The other point has to do with the well-definedness of conditions. To be meaningful, the conditions mentioned in specifications must themselves be free of errors. For example, what would a specification `c.data == 100 / x` mean if `x` could be `0` or `c` `null`? Although `recommends` clauses are not assumed on entry to the method's body, they can be assumed when checking the well-definedness of post-specifications. This is achieved by our transformation of `recommends` clauses into post-specifications grouped by `when` guards. For the `Increment` method above, this means that `c != null` can be assumed when checking the well-definedness of the postcondition `c.data == old(c.data) + x`, which avoids any `null`-dereference errors in this postcondition.

6 Using Recommends Clauses with Standard Patterns

Let us now explain how to use `allow` directives to achieve the programming patterns mentioned in Sect. 2. As we saw in that section, fail-fast parameter validation can be achieved by performing an operation that the language defines as triggering a run-time exception. For example, instead of testing the condition `c == null`, a program can attempt to read a field of `c`. In Java, that causes a `NullPointerException` to be thrown if `c == null` holds. As we discussed in Sect. 5, the verifier ordinarily reports an error if the program attempts to dereference a pointer that may be `null`. But by using an `allow` directive on a source line that performs such a dereference, the verifier will instead treat that operation as possibly throwing an exception. This is just what we need to encode this fail-fast pattern.

For example, the `Increment` method from the previous section can be written

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  var y := c.data; //@ allow
  if x < 0 || 10 <= x {
    throw PreconditionFailure();
  }
  c.data := y + x;
}
```

By including the `allow` directive, the verifier is satisfied that the first `recommends` clause is tested and properly handled. Actually, there is a difference between this `Increment` method and the one in the previous section, namely that if `c` is `null`, this one will at run time throw a `NullPointerException`, whereas, the way we wrote it, `Increment` in the previous section throws a `PreconditionFailure` exception. As we mentioned in Sect. 2, the particular exception thrown as part of parameter validation is not so important. But if you're worried about this, just think of `PreconditionFailure` as a supertype of all parameter-validation exceptions for now; we'll return to this issue in the next section.

In the other fail-fast pattern from Sect. 2, parameter validation is delegated to another method. To illustrate, suppose there is another method

```
method IncrementAndDecrement(c: Cell, x: int, z: int)
  recommends c != null
  recommends 0 <= x < 10 && 0 <= z < 10
  modifies c
  ensures c.data == old(c.data) + x - z
```

Since `Increment` is a special case of this more general method, it can be implemented by a call to `IncrementAndDecrement`. As we glean from its `recommends` clauses, `IncrementAndDecrement` has to do (a superset of) the parameter-validation checks that `Increment` has to do, so it seems everything `Increment` has to do can be done by a single call to `IncrementAndDecrement`.

If we implement `Increment` as we just discussed, then the verifier would complain. It would complain that the call to `IncrementAndDecrement` does not live up to the precondition of the callee—remember that `recommends` clauses are enforced as preconditions at call sites. It would also complain that `Increment` is not doing the parameter validation that its `recommends` clauses demand. We solve both of these problems by marking the call with the `allow` directive:

```
method Increment(c: Cell, x: int)
  recommends c != null
  recommends 0 <= x < 10
  modifies c
  ensures c.data == old(c.data) + x
{
  IncrementAndDecrement(c, x, 0); //@ allow
}
```

In our design of the `recommends` specifications, placing an `allow` directive on a call causes the verifier to consider the callee's `recommends` clauses as control flow that may throw exceptions. That is, the call above with the `allow` directive is treated as

```
if ¬(c != null && 0 <= x < 10 && 0 <= z < 10) {
  throw PreconditionFailure();
}
IncrementAndDecrement(c, x, 0);
```

This will satisfy the verifier as a correct implementation of `Increment`.

A detail remains. As for the first pattern above, we need to be careful with any code that pays specific attention to which parameter-validation exception is thrown. In the `allow`-rewrite of the call above, we wrote `throw PreconditionFailure()`, but at run time, `IncrementAndDecrement` may actually throw some other exception. To make this more exact, our semantics is not exactly the `throw PreconditionFailure()` we showed above; rather, it is to throw *some* parameter-validation exception. If you think of `PreconditionFailure` as the supertype of all parameter-validation exceptions, then it would be accurate to replace the `throw` above with a call

```
_ThrowPreconditionFailure();
```

where

```
method _ThrowPreconditionFailure()
  ensures false
  throws PreconditionFailure ensures true
  throws * ensures false
```

is a method invented by the verifier. Appropriately, this method is not too specific about which `PreconditionFailure` subtype the thrown exception has or how that exception is created.

7 Recommends Else

As we have defined the basic `recommends` clause, a method implementation is obligated to throw a `PreconditionFailure` if a recommended condition does not hold on entry. This is the *specification*, but an implementation is, as usual, allowed to take a more specific action. In particular, the exception thrown by the implementation may be an exception of any subtype of `PreconditionFailure`. This can be useful when debugging the stack trace from a parameter-validation crash.

Though our contention is that one should not put too much emphasis on the actual exception thrown, we are sympathetic to the practice that the documentation style in Java and .NET prescribes particular exceptions to be thrown in response to invalid parameters. To let a verifier check that method implementations follow the documented behavior, we extend the basic `recommends` clause with the ability to specify the use of more specific exceptions.

7.1 Exception Designations

To specify that violations of a `recommends` clause are to be countered by a designated exception, we allow an optional `else` suffix. For a condition `Pre` on the method's pre-state and an exception type `Exc`, the precondition

```
recommends Pre else Exc
```

expresses the intention that `Pre` hold on entry to the method, and obliges the implementation to throw an `Exc` exception if `Pre` does not hold. A basic `recommends` clause defaults to having the suffix `else PreconditionFailure`.

For example, the `Increment` method can be specified and implemented as follows:

```
method Increment(c: Cell, x: int)
  recommends c != null else NullPointerException
  recommends 0 <= x < 10 else IllegalArgumentException
  modifies c
  ensures c.data == old(c.data) + x
{
  var y := c.data; //@ allow
  if x < 0 || 10 <= x {
    throw IllegalArgumentException();
  }
```

```
    c.data := y + x;
  }
```

7.2 Multiple Recommends Clauses

The `recommends else` constructs bring up a question: What do these specifications mean if multiple recommended conditions do not hold? For example, what does the specification above say for a call `Increment(null, 12)`? One could adopt the design that `recommends` clauses are ordered and that the first condition to fail is the one whose `else` exception must be thrown. However, this would force specifications to be overly specific, because they would not give implementations a choice about the order. Indeed, insisting on such an order would even be stricter than the natural-language descriptions of methods in the Java and .NET standard libraries, which often do not state an order.[3]

So, we instead adopt the design that `recommends` clauses are unordered. If any recommended condition does not hold, then the implementation is obligated to throw *some* exception and must throw an exception that corresponds to one of the failing conditions.

For example, consider a specification

```
  recommends A else X
  recommends B else Y
  ensures Post
```

where `X` and `Y` denote disjoint exception types. Its meaning for a caller is, as before, that both `A` and `B` are required to hold. For an implementation, the meaning of the specification is[4]

```
  when A && B
    ensures Post
  when ¬A || ¬B
    ensures false
    throws X ensures ¬old(A)
    throws Y ensures ¬old(B)
    throws * ensures false
```

This says that if `A` and `B` are both true, then the implementation must live up to the given post-specification. If `A` and `B` are both false, then one of the exceptions `X` and `Y` must be thrown, but the choice between these two is up to the implementation. If only one of `A` and `B` is false, then the exception for that one must be thrown. (Recall that the `throws ensures` clause is read as "if the given exception is thrown, then the given condition holds").

[3] Some methods in Java 11, like `java.lang.System.arraycopy` (https://docs.oracle.com/javase/7/docs/api/java/lang/System.html), spell out the order, whereas many others, like `javax.crypto.Cipher.getInstance` (https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html), do not.

[4] Since the second `when` group has an empty `modifies` clause, `old(A)` is the same as `A`. Nevertheless, we will continue to write `old(A)` to emphasize that we're referring to the value of `A` on entry to the method.

Note that it is possible to write a specification that says recommended conditions must be checked in some particular order. For example, we can modify the previous example to specify that X should be thrown if A doesn't hold, regardless of whether or not B holds:

```
recommends A else X
recommends ¬A || B else Y
ensures Post
```

(With an implication operator, the second recommended condition can be equivalently written as `A ==> B`.) To see that this says what is intended, it is perhaps easiest to write out the implementation-side view of this specification, which is

```
when A && (¬A || B)
  ensures Post
when ¬A || ¬(¬A || B)
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures ¬old(¬A || B)
  throws * ensures false
```

After some logical simplifications and distributing `old` over connectives, this is equivalent to

```
when A && B
  ensures Post
when ¬A || ¬B
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures old(A) && ¬old(B)
  throws * ensures false
```

Here, we see that, in the event that either A or B is false, only X and Y are allowable outcomes of the implementation. And in that situation, exception Y is allowed to be thrown only if the recommended condition A did in fact hold and B did not.

In general, different `recommends else` clauses may mention the same exception. And if, like in Java, exception types form a subtyping hierarchy, then the exceptions in different `recommends else` clauses may overlap. To illustrate the interpretation of these, suppose X and Y are disjoint exception types and that X0 is a proper subtype of X. Then, the implementation-side view of

```
recommends A else X
recommends A0 else X0
recommends B0 else Y
recommends B1 else Y
requires C
modifies Frame
ensures Post
```

is

```
when A && A0 && B0 && B1
  requires C
  modifies Frame
  ensures Post
when ¬A || ¬A0 || ¬B0 || ¬B1
  ensures false
  throws X ⟍ X0 ensures ¬old(A)
  throws X0 ensures ¬old(A) || ¬old(A0)
  throws Y ensures ¬old(B0) || ¬old(B1)
  throws ∗ ensures false
```

where we have used the notation `X ⟍ X0` to denote any exception type that is a subtype of `X` but not a subtype of `X0`. Formulaically, for every exception `Z` mentioned in some `recommends else` clause, the interpretation has a clause

```
throws Z ⟍ Z0 ⟍ Z1 ⟍ ... ensures ¬old(P) || ¬old(Q) || ¬old(R) || ...
```

where `Z0`, `Z1`, ... are proper subtypes of `Z` mentioned in `else` designations and `P`, `Q`, `R`, ... are the conditions in `recommends` clauses whose `else` designation is a subtype of `Z` but not a subtype of any of `Z0`, `Z1`, ...

7.3 Well-definedness

As we touched on in Sect. 5, specifications have to be well-defined. Since we consider `recommends` clauses to be unordered, our design is also to check their well-definedness independently. In particular, this means that each `recommends` condition must be well-defined whether or not the other `recommends` conditions hold.

To illustrate, suppose we want to create an array containing the first `n` elements of a given array `a`. We would then write the specification

```
method CopyN(a: array<int>, n: int) returns (r: array<int>)
  recommends a != null else NullPointerException
  recommends 0 <= n else NegativeArraySize
  recommends a != null ==> n <= a.length else ArrayIndexOutOfBounds
  // ...
```

Note that the last `recommends` clause must include the "`a != null ==>`", as it may not presume the first `recommends` to hold. Also, an easy mistake to make is to instead write "`a != null &&`", but that would allow `ArrayIndexOutOfBounds` to be thrown in response to `a` being passed in as `null`.

8 Combining Recommends and Requires

We have motivated our work by the need to write specifications at the boundary between unverified and verified code. Our general guideline is to use `recommends` preconditions for methods that may be called by unverified code and to use standard `requires` preconditions for methods that are called only from call sites where those conditions are enforced statically by a verifier. But there are also some situations where a method may be specified by a combination of `recommends` and `requires`.

One example of such a situation is an auxiliary method, perhaps like `ASimilarFailFastExample` in Sect. 2, that is not callable from outside the verified module. In addition to the conditions for which this method promises to throw parameter-validation exceptions, there may be additional conditions that the auxiliary method wants to assume that callers establish. Since the auxiliary method is private to the verified module, all of its callers are verified, so these additional conditions can indeed be enforced at all call sites.

Another example of such a situation is when a precondition is complicated, which means the condition wouldn't be checked at run time anyway. In such a case, the caller is expected to establish the condition and there is no telling what might happen if the caller doesn't live up to this obligation. For example, a method that searches an array for a given element may expect the array to be sorted and may expect a given comparison operator to be transitive. Here, it makes sense to document the transitivity precondition and to let the method implementation be verified under the assumption that the precondition does hold.

To support combinations of `recommends` and `requires`, we need to be precise about the semantics. As before, for callers, there's no difference between the two kinds of preconditions, except that any `allow` directives on a call apply only to `recommends` clauses, never to `requires` clauses. For implementations, we consider `requires` conditions to be contingent on the `recommends` conditions being true. That is, an implementation is not allowed to assume `requires` conditions until after the `recommends` conditions have been checked.

To illustrate, for disjoint exception types `X` and `Y`, consider the following specification:

```
recommends A else X
recommends B else Y
requires C
requires D
ensures Post
```

For the method implementation, the expanded view of this specification is

```
when A && B
  requires C
  requires D
  ensures Post
when ¬A || ¬B
  ensures false
  throws X ensures ¬old(A)
  throws Y ensures ¬old(B)
  throws * ensures false
```

That is, the `requires` conditions `C` and `D` apply only if `A` and `B` hold on entry. So, effectively, the condition that an implementation may assume on entry is

```
A && B  ==>  C && D
```

where ==> denotes implication.

As follows from the expansion above, the well-definedness checks of the `requires` conditions are allowed to assume both `A` and `B`. For example, the array dereferences in the following specification are well-defined because of the `recommends` clauses:

```
method Search(a: array<int>, lo: int, hi: int, key: int)
     returns (r: Option<int>)
  recommends a != null
  recommends a != null ==> 0 <= lo <= hi <= a.Length
  requires forall i, j :: lo <= i < j < hi ==> a[i] <= a[j]
  // ...
```

9 Prototype Implementations

In this section, we discuss a prototype of `recommends` that we have built in the OpenJML tool for Java annotated with JML specifications. We also discuss possibilities of adding `recommends` specifications to the verification-aware language Dafny.

9.1 JML and OpenJML

Recommends Clauses. The Java Modeling Language (JML) [6] and its supporting tool OpenJML [4,5] implement a prototype version of the `recommends else` clause described in this paper. Here is an example[5]

```
//@   recommends v != null else NullPointerException;
//@   ensures \result == v.value;
//@ pure
public int get(/*@ nullable */Value v) {
  return v.value; // implicit allow because of the recommends
}
```

We discuss defaults for `allow` behavior below.

JML already has the capability to specify the behavior of exceptional control paths. The following code is the equivalent of the above example, but in the older style:

```
//@ public normal_behavior
//@   requires v != null;
//@   ensures \result == v.value;
//@ also public exceptional_behavior
//@   requires v == null;
//@   signals_only NullPointerException;
//@ pure
```

[5] The `nullable` keyword avoids the default situation in JML in which `Value` is implicitly a non-null type. We use it in this section, since the focus of our discussion is on using specifications for parameter validation.

```
public int get(/*@ nullable */Value v) {
  return v.value; // implicit allow because of
                  // the exceptional_behavior
}
```

Compared to using recommends else, this specification is verbose and repetitive; it is worse when there are many exception conditions to consider. Furthermore, it is easy during code maintenance to have the two preconditions become out of sync. The recommends else syntax is a more readable and understandable equivalent. Though the normal-termination effect of this program is simple, the exceptional-behavior specification is typical even of much more complex methods.

The method in the example above is without side-effects (is pure) in both the normal and exceptional behaviors. That is not always the case; when it is not, then separate frame conditions (notated in JML by assignable clauses) must be written:

```
//@ public normal_behavior
//@   requires v != null;
//@   assignable v.value;
//@   ensures v.value == i;
//@ also public exceptional_behavior
//@   requires v == null;
//@   assignable \nothing;
//@   signals_only NullPointerException;
public void set(/*@ nullable */Value v , int i) {
  v.value = i; // implicit allow
}
```

and correspondingly

```
//@   recommends v != null else NullPointerException;
//@   modifies v.value; // does not apply to the else case
//@   ensures v.value == i;
public void set(/*@ nullable */Value v, int i) {
  return v.value; // implicit allow
}
```

Our design of recommends clauses does not let an implementation have any side-effects until after the recommends conditions have been tested. In our experience in specifying Java programs, exceptional control flow almost never has any side-effects. Therefore, we expect that many current specifications of exceptional behavior can be simplified using recommends else clauses. Even in the simple examples in this section, we see that recommends clauses add readability and save several lines of specification. If the method being specified does allow side-effects in exceptional behavior, then the more lengthy normal_behavior/exceptional_behavior form must be used.

Allow Designations. OpenJML also implements the allow designation as described above, with a bit of enhancement. If a language construct might throw a particular subtype of RuntimeException then:

– if there is no allow designation that includes a supertype of that exception type and the verifier cannot establish that the offending condition never happens, then a verification warning is issued for that language construct on that line;
– otherwise, the verifier treats the language construct as if the exceptional control flow might happen, and continues on to verify any internal exception catches within the method or the method's throws clauses.

But OpenJML has these enhancements:

– If the language construct in question is nested inside a try-catch block for the relevant exception or if the method specification itself declares (in an explicit JML signals_only clause) that it might throw that exception, then that construct has an implicit allow.
– In order to turn off the implicit allow where needed, OpenJML has a corresponding forbid designation, which will cause the verifier to check that no exception can be thrown from that construct.
– The default exception if an allow or forbid has no exception stated is RuntimeException; a comma-separated list of exception names is also permitted.

This language feature solves a usability problem with JML that confused users. A user trying out JML might write this simple method and specification:

```
//@ ensures \result == c[i];
public int value(int[] c, int i) {
  return c[i];
}
```

Without a precondition on the value of i, OpenJML will issue a verification error that i might not be in the range of allowed indices for the array c. No problem for the user there. But then the user would go on to try some exception handling:

```
//@ ensures \result == c[i];
public int value(int[] c, int i) {
  try {
    return c[i];
  } catch (ArrayIndexOutOfBoundsException e) {
    // ...
  }
}
```

Now the user does not expect to see verification errors on c[i], but originally OpenJML would. With these proposed features, OpenJML takes the presence of the catch clause in this example as evidence of the user's intent that an exception should be allowed at c[i]. If that is not the desired behavior, an explicit forbid can be written. Even without the catch clause, the user can use an explicit allow to clearly document that an exception is allowed.

9.2 Dafny

Dafny is a programming language designed for verification [12,13]. Failures in Dafny are different in two major ways from what we have discussed for Java. Still, the situation of calling verified Dafny methods from unverified code in other languages can benefit from `recommends` specifications.

One major difference is that what in a language like Java or C# would be a run-time exception is typically a verification failure in Dafny. For example, writing `c.v` or `a[j]` in Dafny gives rise to proof obligations that `c` is not `null` and `j` is in range. The program is not valid—and no compiled code is emitted—unless such proof obligations can be validated by the static verifier. In other words, Dafny's language semantics already follows a strict discipline.

Another major difference is that Dafny does not have exceptions. In languages like Java and C#, exceptions are used for both recoverable and unrecoverable errors, whereas in Dafny, there is a different mechanism for each of these. In the rest of this section, we show how our `recommends` declarations could be added to Dafny.

Recommendation Failures as Unrecoverable Errors. The most natural way to guard verified Dafny code from unverified callers is to test preconditions and use an unrecoverable error in response to any detected violation. Unrecoverable errors are produced using Dafny's `expect` statement, which performs a run-time check of its given condition and halts program execution if the condition evaluates to `false`.[6]

For example, here is a method with the envisioned `recommends` clauses for Dafny:

```
method Double(a: array?<int>, i: int)
  recommends a != null
  recommends a != null ==> 0 <= i < a.Length
  modifies a
  ensures a[i] == 2 * old(a[i])
{
  expect a != null;
  expect 0 <= i < a.Length;
  var x := a[i];
  a[i] := x + x;
}
```

There are several things to note in this simple example.

The first is a reminder that callers written in Dafny are verified, so such callers would be verified to satisfy the `recommends` preconditions.

The second is that a method like this is more commonly written with Dafny's non-null type `array<int>`. However, if this method is at the boundary from unverified code written in a different language that does not support non-null reference types, then it's

[6] In other words, the `expect` statement trades a verification assumption for a run-time check. The run-time behavior of an `expect` statement is similar to an always-enabled `assert` statement in Java, but from the verification perspective, the `assert` condition is checked by the verifier whereas the `expect` condition is assumed by the verifier.

safer to declare the parameter to be of the nullable type `array?<int>` and to list the non-nullness as a precondition.

The third thing to note is that, without `else` suffixes, the `recommends` clauses may as well be ordered, in which case the `a != null` antecedent can be dropped in the second `recommends` clause.

The fourth is that the `expect` statements for this simple example are, except for the ordering, identical to the recommended conditions, which makes it tempting to design the `recommends` clauses to compile into `expect` statements automatically. However, the condition in an `expect` statement must be compilable, whereas preconditions in Dafny often use specification-only ("ghost") features. Therefore, imposing on method implementations to add tests for recommended conditions (as we have done everywhere else in this paper) gives more flexibility.

The fifth is that it's not possible in Dafny to replace the body of `Double` with

```
{
  var x := a[i]; //@ allow
  a[i] := x + x;
}
```

where the `allow` directive is intended to rely on the language's run-time checks to test and handle `a != null` and `0 <= i < a.Length`. This is because there are no run-time checks for these operations in Dafny, so turning off verification for the indicated source line would not properly implement the fail-fast parameter validation we're after.

The sixth and final thing to note is that `allow` directives still make sense on calls to methods with `recommends` clauses. This would allow delegating `recommends` checking to another method, as we have discussed earlier in the paper.

In the method implementation's view, a specification

```
recommends A
requires C
modifies Frame
ensures Post
```

expands to

```
when A
  requires C
  modifies Frame
  ensures Post
when ¬A
  ensures false
```

where, as earlier in the paper, we have used `when` groups as a way of explaining when various specification clauses apply. In the case where `A` does not hold on entry, the expanded specification shows that the method implementation is obliged to establish `false` upon normal termination, and the only way to do this is via the `expect` statement, which avoids normal termination by generating an unrecoverable error.

Recommendation Failures as Recoverable Errors. Another possible way of adding `recommends` clauses to the Dafny language is to tie them to the standard mechanism for reporting recoverable errors. For this, Dafny uses a return value of a *failure-compatible type*. For illustration, we will use the typical type

```
datatype Result<T> = Success(value: T) | Failure(error: string)
{
  function method IsFailure(): bool // ...
  function method PropagateFailure(): Result<T> // ...
  function method Extract(): T // ...
}
```

whose three shown members make it a failure-compatible type [14].

For a method that returns a failure-compatible type, we can oblige the implementation to return a failure if a recommended condition does not hold. For example, the specification of

```
method GetAny<T>(a: array?<T>) returns (r: Result<T>)
  recommends a != null && a.Length != 0
  ensures r.Success? ==>
          exists i :: 0 <= i < a.Length && r.value == a[i]
{
  if a == null {
    return Failure("array is null");
  }
  if a.Length == 0 {
    return Failure("array has no elements");
  }
  return Success(a[a.Length / 2]);
}
```

promises, in the event of success, to return an element of the array a. The `recommends` clause says that the implementation must return a failure if the recommended condition does not hold on entry. As written, the specification allows the method to return a failure in other situations, too.

In the method implementation's view, the specification

```
recommends A
requires C
modifies Frame
ensures Post
```

expands to

```
when A
  requires C
  modifies Frame
  ensures Post
when ¬A
  ensures r.IsFailure()
```

Dafny has special syntax that makes it easy to propagate failures and easy to use successful values. For a method `M` that returns a failure-compatible type, the statement

```
x :- M();
```

(note the operator `:-` instead of the usual assignment operator `:=`) immediately propagates any failure that `M` returns. If `M` returns success, then the successful value is extracted into `x`. The context using the `:-` statement must itself allow a failure-compatible type to be returned. (Dafny's `:-` operator and failure-compatible types are closely related to the `?` operator and built-in `Result` type in the Rust language [11].)

The use of the `allow` directive to delegate parameter validation is conveniently combined with the `:-` operator. For example, consider a method

```
method GetElement<T>(a: array<T>, i: nat) returns (r: Result<T>)
  recommends i < a.Length
  ensures r.Success? ==> r.value == a[i]
{
  if i < a.Length {
    return Success(a[i]);
  } else {
    return Failure("index out of bounds");
  }
}
```

We're supposing this method is called only from within Dafny code, since it takes a non-null array and a non-negative integer as parameters (and these types may not be available in the foreign language that otherwise may call the method). Even for this Dafny-only method, the `recommends` clause usefully allows the implementation of `GetAny` to delegate its responsibility to check the emptiness of the array:

```
method GetAny<T>(a: array?<T>) returns (r: Result<T>)
  recommends a != null && a.Length != 0
  ensures r.Success? ==>
          exists i :: 0 <= i < a.Length && r.value == a[i]
{
  if a == null {
    return Failure("array is null");
  }
  r :- GetElement(a, a.Length / 2); //@ allow
}
```

10 Related Work

There are several contract languages that provide run-time checking of preconditions. Among these are the programming languages Eiffel [16,7] and Spec# [2] and the modeling languages JML for Java [6,1,5] and ACSL for C [3]. The compiled preconditions provide fail-fast checking at run time, while still allowing the static verifiers for these languages to check the preconditions at call sites. The `requires otherwise` clauses in

Spec#, which inspired our `else` suffixes, also allow customization of which run-time exception is thrown in response to violated preconditions [15].

In these languages, the condition that is compiled into a run-time check is the condition given in the `requires` clause. This forgoes the flexibility of having ghost expressions in specifications and compilable counterparts in method implementations. It also misses the opportunity to save on some explicit checks using the common patterns we showed in Sect. 2.

These contract languages generally allow the expanded implementation-side view of our `recommends` clauses to be specified explicitly. This supports programmer-defined testing of the preconditions. Unfortunately, this long form of specifications is error-prone to write and hard to read. Moreover, with just the implementation-side view of the specification, there is nothing that enforces the intended preconditions at call sites.

The only way we see to reap the benefits of the two kinds of specifications is to provide different precondition interpretations for callers and callees. Our proposed `recommends` clauses achieve that. Used in conjunction with `allow` directives, it is also possible to choose between the interpretations at call sites.

The `recommends` capability helps address the case of unverified callers calling verified callees. A related problem, which we have not considered in this paper, is that of verified callers calling unverified methods (e.g., unverified libraries). We consider this to be a significantly different problem worth separate attention. Even with a mechanism for writing out-of-band specifications for unverified library methods, it's not easy to use run-time monitoring to detect whether or not callees follow the specifications. For example, if the precondition in such a specification is too weak, then the callee may cause irreparable damage before a run-time monitor has a chance to react. As another example, it is at best expensive to detect whether or not the callee modifies only what the specification says.

11 Concluding Remarks

We have proposed a new specification-language feature, `recommends`, that (like common `requires` preconditions) states what conditions are *intended* to hold on method entry, but (unlike `requires` preconditions) does not trust that these conditions actually hold. Not only is a method implementation not allowed to assume these preconditions on entry, but the `recommends` feature forces the method implementation to test them on entry. This makes `recommends` suitable in situations where the method may be called from unverified code. The additional `allow` directive equips the software developer with flexibility and precision to say which locations in a program allow exceptional behavior and which locations outright forbid it.

We illustrated the naturalness of our new notation by example, and our prototype in JML and OpenJML illustrates its feasibility. Supporting the claim that our proposal is language agnostic, we described two ways in which `recommends` clauses could fit into Dafny, a verification-aware language that treats failures in a significantly different way from Java.

We hope that the clarity of `recommends` declarations will help move the unverified-verified boundary in the direction of more verified software.

Acknowledgments. We thank Serdar Tasiran for suggesting the keyword `recommends` as a good description of the alternative precondition. We are grateful to the reviewers and Mattias Ulbrich for useful comments on a draft of this paper. Happy birthday, Reiner Hähnle, and thank you for your groundbreaking contributions to verifying software and the tooling that is taking us there.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book—From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Comm. ACM **54**(6), 81–91 (2011). https://doi.org/10.1145/1953122.1953145
3. Baudin, P., et al.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Comm. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
4. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
5. Cok, D.R.: JML and OpenJML for Java 16. In: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-Like Programs, FTfJP 2021, pp. 65–67. ACM (2021). https://doi.org/10.1145/3464971.3468417
6. Cok, D.R., Leavens, G.T., Ulbrich, M.: JML Reference Manual, 2nd edn. (2021). https://www.openjml.org/documentation/JML_Reference_Manual.pdf
7. ECMA International: Eiffel: Analysis, Design and Programming Language, 2nd edn., June 2006. Standard ECMA-367
8. Goodenough, J.B.: Structured exception handling. In: Graham, R.M., Harrison, M.A., Reynolds, J.C. (eds.) Conference Record of the Second ACM Symposium on Principles of Programming Languages, pp. 204–224. ACM, January 1975. https://doi.org/10.1145/512976.512997
9. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, Boston (1996)
10. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012). https://doi.org/10.1145/2187671.2187678. Article 16
11. Klabnik, S., Nichols, C.: The Rust Programming Language (2018). https://doc.rust-lang.org/book/
12. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
13. Leino, K.R.M.: Accessible software verification with Dafny. IEEE Softw. **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
14. Leino, K.R.M., Ford, R.L., Cok, D.R.: Dafny reference manual (2021). https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef
15. Leino, K.R.M., Schulte, W.: Exception safety for C#. In: Cuellar, J.R., Liu, Z. (eds.) SEFM 2004–Second International Conference on Software Engineering and Formal Methods, pp. 218–227. IEEE, September 2004. https://doi.org/10.1109/SEFM.2004.14
16. Meyer, B.: Object-oriented Software Construction. Series in Computer Science, Prentice-Hall International, Hoboken (1988)

Programming Legal Contracts
– A Beginners Guide to *Stipula* –

Silvia Crafa[1] and Cosimo Laneve[2(✉)]

[1] University of Padova, Padua, Italy
silvia.crafa@unipd.it
[2] University of Bologna, Bologna, Italy
cosimo.laneve@unibo.it

Abstract. We discuss the design principles of *Stipula*, a domain specific language that can assists lawyers in programming legal contracts through specific software patterns. The language is based on a small set of primitives, that precisely correspond to the distinctive elements of legal contracts, and that are amenable to be prototyped on both centralized or distributed systems. We also outline two formal techniques to reason about *Stipula* contracts: a type inference system that allows to derive types for fields, assets and contract's functions, and an analyzer of liquidity that pinpoints those contracts that do not freeze any asset forever.

1 Introduction

The legal field is one of the domains that are currently most influenced by the so-called digital revolution. A large number of legal texts, ranging from laws, regulations, administrative procedures, to contractual agreements, court judgements and jurisprudence, might considerably benefit from a sensible digitalisation. The advantages are not only in terms of efficiency, like speed-up and automatic execution of fully defined procedures, but also in terms of data organisation and transparency of processes. As a cons, computationally dealing with laws is difficult because of the complexity of the legal texts: human judgement is often required to interpret the natural language since it is, at the same time, very expressive and quite ambiguous.

In this article we focus on *legal contracts*, a specific subset of the legal field that define "those agreements which are intended to give rise to a binding legal relationship or to have some other legal effect" [12]. These agreements are basically *protocols* that regulate the *relationships* between parties in terms of permissions, obligations, prohibitions, escrows and securities. In turn, according to the principle of *freedom of form*, which is shared by the contractual law of modern legal systems, the agreements can be expressed by the parties using the language and medium they prefer, including a programming language. When a programming language is chosen, it is mandatory that the language be high level enough so that writing and inspecting a software contract do not require proficiency in computer science. In fact, only if the parties (which, in this domain, are usually lawyers, notaries, ordinary citizens, etc.) are fully aware of the computational effects of their code there may be a genuine agreement over the content of the contract, thus reducing or possibly eliminating applications to courts for either misinterpretations or misunderstandings.

Therefore we decided to work in close connection with lawyers to select few concise and intelligible primitives that have a precise correspondence with the distinctive elements of legal contracts. The resulting language, called *Stipula*, is a new domain-specific language with a formal operational semantics, so that the behaviour is fully specified and amenable to automatic verification. Its actual adoption by legal practitioners still requires a human-readable interface, such as an IDE support or a visual language interface, but we think that the design and the theory of this *legal calculus* [3] provide interesting insights on the application of programming languages to the legal field.

The complete definition of *Stipula* can be found in [7]; in this paper we give a gentle introduction to *Stipula*, by motivating its distinctive features with contractual elements taken from two paradigmatic legal contracts: a rental contract and a bet contract. We also discuss two analysis techniques that have been defined for *Stipula* and that can bee seen as useful tools that support a safe programming of legal contracts. First, since *Stipula* is untyped, we illustrate a type inference system that allows to automatically derive types for fields, assets and contracts' functions. This system is useful to type check the correctness of operations, thus preventing basic errors with contract's data and assets. Then we overview an analyzer that statically checks the presence of executions of the legal contract leaving assets frozen into the contract without being redeemable by any party (liquidity). We conclude the paper with a number of final remarks about the implementation of *Stipula* and a discussion of related work.

2 Legal Contracts' Elements as *Stipula* Building Blocks

Contractual agreements are generally written as combinations of distinctive elements, such as permissions, prohibitions, obligations, fungible and non fungible assets exchanges, and aleatory or real-world data retrieval. These elements are combined into common legal patterns that either establish new obligations, rights, powers and liabilities between the parties, or transfer rights (such as rights to property) from one party to another, often subject to specific conditions and by taking advantage of escrows and securities.

A first distinctive feature of a legal contract is the "*meeting of the minds*", *i.e.* the moment when, after the possible negotiation of the contractual content, the parties express consent on the terms of the agreement and the contract produces its legal effects. *Stipula* provides an ad-hoc primitive, called *agreement*, which marks that contracts' parties have reached a consensus on the contractual arrangement they want to create. As an example, consider a contract regulating a bike rental service: the following *Stipula* code

```
agreement (Lender, Borrower){
   Lender, Borrower: rentingTime, cost
   } ⇒ @Inactive
```

is meeting a `Lender` and a `Borrower` to agree on both the `rentingTime` and on its `cost`. After the agreement the contract starts and it goes into a state `@Inactive` that expresses that no rent will occur until the payment (some money has been transferred from `Borrower` to `Lender`). A variant of the contract might also including an `Authority` that is charged to monitor contextual constraints, such as obligations of diligent storage and care, or the obligations of using goods only as intended, taking care of litigations and dispute resolution. In this case, the agreement would be

```
agreement (Lender, Borrower, Authority){
   Lender, Borrower: rentingTime, cost
   } ⇒ @Inactive
```

expressing the fact that only the `Lender` and the `Borrower` agree on both `rentingTime` and `cost`, while the `Authority`, which also engage in the meeting of minds, is the pointer to a *third party* that will supervise `Lender` and `Borrower` behaviours.

A second distinctive feature of legal contracts is that the set of normative elements, namely permissions, prohibitions and obligations, usually changes over time according to the actions that have been done (or not). To model these changes, *Stipula* commits to a *state-machine programming* style, inspired by the state machine pattern that is supported by almost every programming language (with ad-hoc libraries and/or modules). For instance, in a bike rental, once the lender and the borrower have agreed on the rental period and cost, the lender is prohibited from preventing the borrower from paying for the service and (afterwards) using the bike. *Stipula* expresses this feature by letting the contract play a proactive role: assuming that the bike can be used if the borrower has a temporary access code, the contract stores the temporary code in a field, thus disallowing the lender to withdraw from the rental. The following code defines the *function* `offer` that can be invoked by `Lender` when the contract is in state `@Inactive` to send an access code to be used by the `Borrower`.

```
@Inactive Lender : offer(x) {
   x → code
   } ⇒ @Payment
```

Of course, the value of "`code`" is not disclosed to the `Borrower` before the payment for the service. In other terms, the above fragment is giving *permission* to the `Lender` to invoke `offer` in the state `@Inactive` and, if no further function is defined in `@Inactive`, the contract is *prohibiting* other parties to do any action at this stage. Once `code` has been received, the contract moves to a state `@Payment` where presumably the `Borrower` will pay (actually, he is allowed to pay) for the rental.

It is worth to notice that the foregoing code also highligths that `Lender` is trusting the contract to act as intermediary that can store relevant informations (such as, as we will see below, assets). In fact, `x → code` stores the value sent by the Lender into a contract's field called code that cannot be accessed outside the contract.

A further distinctive feature of legal contracts is the management of *assets*: currency is required for payments and escrows, tokens, both fungible and non-fungible, are useful to model securities and to provide a digital handle on a physical good. For instance, in the case of the bike rental, instead of a simple numeric code, a more innovatory IoT technique would be to rely on a unique token that grants access to the bike's smart lock. Moreover, in the traditional setting, the Borrower pays the Lender with a credit card before he can use the bike and the money transaction is only specified by the contract through a normative clause. Its occurrence is not guaranteed (in case of dispute, one party has to appeal to a court). On the other hand, *Stipula* admits digital legal contracts that automatically deal with assets transfers, so to remove intermediation even from the payments. In *Stipula* assets can be also temporarily retained by legal contracts, which may decide to redistribute them when particular conditions occur. To this aim, the language promotes an explicit management of assets by regarding them as first-class values with ad-hoc operations. For example, the function

```
@Payment Borrower : pay[h]
   (h == cost) {
      h ⊸ wallet
      code → Borrower
   } ⇒ @Using
```

is defining the payment of the rental by Borrower, which sends an asset h – the argument is in square brackets – to the contract. The function call has a precondition – operation h == cost – that checks whether the borrower pays the correct fee or not. The semantics of the operation h ⊸ wallet, which is an abbreviation for h ⊸ h, wallet, is that, after the execution, h is not owned by Borrower anymore and is taken by the contract that stores it in the *asset field* wallet. The design choice of explicitly marking asset movements with the ad hoc operator "⊸" (thus separating it from "→") promotes a safer, asset-aware, programming discipline that reduces the risk of the so-called double spending, the accidental loss or the locked-in assets. Notice that the contract does not immediately forward the payment to the Lender, rather it is retained for some time until the rental period is terminated (in this way, in case of disputes, neither the Borrower nor the Lender can access/use the asset while the dispute is in progress). Once the fee has been payed, the Borrower gets the access code to the bike and the contract transits into a @Using state.

There is a fourth distinctive feature of legal contracts: the *obligations*, namely operations that must be done, typically within a deadline, by some party. In *Stipula*, obligations are recast into commitments that are checked at a specific time limit and the corresponding programming abstraction is the *event* primitive. For example, the foregoing pay function may be refined by issuing an event that terminates the renting service when the time limit is reached. The code becomes

```
@Payment Borrower : pay[v]
   (v == cost) {
      v ⊸ wallet
      code → Borrower
      now + rentingTime ≫             //end-of-time usage
         @Using {
            "End_Reached" → Borrower
            wallet ⊸ Lender
         } ⇒ @End
   } ⇒ @Using
```

asserting that the bike can be used until the renting period terminates. The time limit is expressed by `now + rentingTime` and, at that moment, if the bike has not been already returned (the state of the contract is still `@Using`), a message of returning the bike is sent to the `Borrower` (`"End_Reached" → Borrower`) and the fee that was stored in `wallet` is delivered to the `Lender` (`wallet ⊸ Lender`). We remark that events are not triggered by any party: they are automatically executed when the time condition is met. Since the statements in the body of events will be executed in the future, we assume for simplicity that the event's body is outside of the scope of functions's parameters, both assets and non assets. A more complex alternative would be to save for future execution the *closure* of the event statements, that captures the local values of functions' parameters.

The foregoing codes do not address disputes, *e.g.* contentions because the bike is returned, or initially was, broken or damaged. These are common elements of legal contracts, that are usually assessed by means of *third party enforcements*, typically by a court. Disputes have a simple modelling in *Stipula* that does not require any new ad-hoc feature, and somehow mimic the behaviour of a court. In fact, when contract's violations cannot be fully checked by the software, such as the damage or misuse of the bike, or the renting of a broken bike, then a trusted third party, the `Authority`, is necessary to supervise the dispute and to provide a resolution mechanism. The code below illustrates the encoding of the off-chain monitoring and enforcement mechanism by means of an `Authority` (which must have been included in the agreement) in *Stipula*.

```
@Using Lender,Borrower : dispute(x) {
   x → _
   } ⇒ @Dispute

@Dispute Authority : verdict(x,y)
   (y >= 0 && y <= 1) {
      x → Lender, Borrower
      y*wallet ⊸ wallet, Lender
      wallet ⊸ Borrower
   } ⇒ @End
```

The function `dispute` may be invoked either by the `Lender` or by the `Borrower` and carries the reasons for kicking the dispute off (`x` is intended to be a string). Once the reasons are communicated to every party (we use the abbreviation "`_`" instead of writing three times the sending operation) the contract transits into a state `@Dispute` where the `Authority` will analyze the issue and emit a verdict.

legal contracts	***Stipula* contracts**
meeting of the minds	agreement primitive
permissions, prohibitions	state-aware programming
currency and tokens	asset-aware (linear) programming
obligations	event primitive
judicial enforcement	explicit Authority and ad-hoc pattern
exceptional behaviors	explicit Authority and ad-hoc pattern

Fig. 1. Correspondence between legal elements and *Stipula* features

This is performed by permitting in the state `@Dispute` only the invocation of the `verdict` function, that has two arguments: a string of motivations `x`, and a coefficient `y` that denotes the part of the wallet that will be delivered to `Lender` as reimbursement; the `Borrower` will get the remaining part. It is worth to spot this point: the statement `y*wallet ⊸ wallet, Lender` *takes* the `y` part of `wallet` (`y` is in [0..1]) and sends it to `Lender`; *at the same time* the `wallet` is reduced correspondingly. The remaining part is sent to `Borrower` with the statement `wallet ⊸ Borrower` (which is actually a shortening for `1*wallet ⊸ wallet, Borrower`) and the `wallet` is emptied.

There is a last distinctive element in legal contracts that deserves a comment: the management of *exceptional behaviours*, *i.e.* all those behaviours that cannot be anticipated due to the occurrence of unforeseeable and extraordinary events. As in the above case, *Stipula* does not use any new ad-hoc feature, rather a simple pattern is provided that defines a template:

```
~@End _ : block(x) {
   x → _
   } ⇒ @Exception

@Exception Authority : handle(x,y) //similar to verdict(x,y)
```

According to the above pattern, the function `block` may be invoked by any party (notation "_") provided the lifetime of the contract is *not terminated* (the contract is not in the state `End`). The management of the exception is similar to that of disputes and therefore omitted.

Figure 1 recaps the normative elements of a legal contract and the corresponding modellings in *Stipula*. Figure 2 uses coconnected boxes to highlight the correspondence between the normative elements of a standard bike rental contract and the corresponding editing in *Stipula*. In this case the *Stipula* code is a bit more complex than the one discussed above: `Borrower` pays the double of the fee in order to safeguard `Lender` from damages, late returns, etc. Accordingly, the termination of the rental requires the `Borrower` to call the function `end`, after which the `Lender` has to confirm the absence of damages by invoking `rentalOK`. Only this sequence of actions, which is enforced by the additional state `@Return`, allows the lender to be payed and the borrower to get back the money deposited as security.

It is worth to notice that a *Stipula* contract begins with the keyword `stipula` and define *assets* and *fields* that are used therein. We also observe that *Stipula* is untyped, to keep a simple syntax; however, a type inference system that allows one to derive types is discussed in Sect. 4. Finally, we notice that the code of Fig. 2 is also *liquid*: at the end of any contract execution, in the final state `@End`, the asset `wallet` is empty, *i.e.* `Bike_Rental` has no locked-in value (see the discussion in Sect. 5).

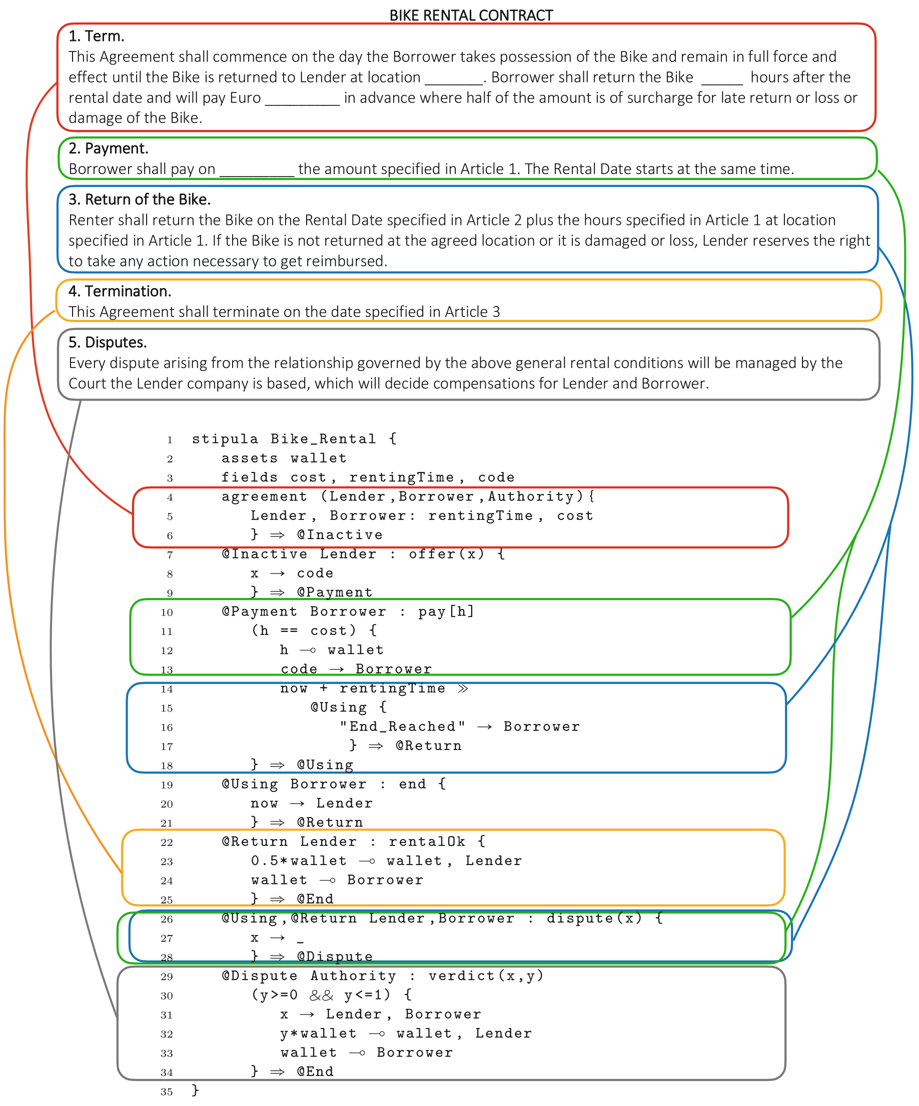
3 Example: The Bet Contract

An example for testing the expressivity of *Stipula* is a contract ruling a bet. This is a legal contract that contains an element of randomness (*alea*, such as a future, aleatory event, such as the winner of a football match, the delay of a flight, the future value of a company's stock) that is entirely independent of the will of the parties.

A digital encoding of a bet contract requires that the parties explicitly agree on the source of data that will determine the final value of the aleatory event – the `DataProvider` –, which is usually a specific online service, an accredited institution, or any trusted third party. It is also important that the digital contract defines precise time limits for accepting payments and for providing the actual value of the aleatory event. Indeed there can be a number of issues: the aleatory event does not happen, *e.g.* the football match has been cancelled, or the data provider fails to deliver the required value, *e.g.* the online service is down.

The *Stipula* code in Listing 1.1 corresponds to the case where `Better1` and `Better2` respectively place in `val1` and `val2` their bets, while the agreed `amount` of currency is stored in the contract's assets `wallet1` and `wallet2`[1]. Observe that both bets must be placed within an (agreed) time limit `t_before` (line 15), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 22, is used to ensure the contract termination even if the `DataProvider` fails to provide the expected data, through the call of the function `data`. When the function `data` is called and the first argument `x` is the `alea` of the bet, the betters are rewarded according to the result `y`. For simplicity we assume that the data-provider service gets the two bets when they lose.

[1] For simplicity, this code requires `Better1` to place its bet before `Better2`. It is easy to extend the code to let the two bets be placed in any order.



BIKE RENTAL CONTRACT

1. Term.
This Agreement shall commence on the day the Borrower takes possession of the Bike and remain in full force and effect until the Bike is returned to Lender at location ________. Borrower shall return the Bike ______ hours after the rental date and will pay Euro __________ in advance where half of the amount is of surcharge for late return or loss or damage of the Bike.

2. Payment.
Borrower shall pay on __________ the amount specified in Article 1. The Rental Date starts at the same time.

3. Return of the Bike.
Renter shall return the Bike on the Rental Date specified in Article 2 plus the hours specified in Article 1 at location specified in Article 1. If the Bike is not returned at the agreed location or it is damaged or loss, Lender reserves the right to take any action necessary to get reimbursed.

4. Termination.
This Agreement shall terminate on the date specified in Article 3

5. Disputes.
Every dispute arising from the relationship governed by the above general rental conditions will be managed by the Court the Lender company is based, which will decide compensations for Lender and Borrower.

```
stipula Bike_Rental {
   assets wallet
   fields cost, rentingTime, code
   agreement (Lender,Borrower,Authority){
      Lender, Borrower: rentingTime, cost
      } ⇒ @Inactive
   @Inactive Lender : offer(x) {
      x → code
      } ⇒ @Payment
   @Payment Borrower : pay[h]
      (h == cost) {
         h ⊸ wallet
         code → Borrower
         now + rentingTime ≫
            @Using {
               "End_Reached" → Borrower
                } ⇒ @Return
      } ⇒ @Using
   @Using Borrower : end {
      now → Lender
      } ⇒ @Return
   @Return Lender : rentalOk {
      0.5*wallet ⊸ wallet, Lender
      wallet ⊸ Borrower
      } ⇒ @End
   @Using,@Return Lender,Borrower : dispute(x) {
      x → _
      } ⇒ @Dispute
   @Dispute Authority : verdict(x,y)
      (y>=0 && y<=1) {
         x → Lender, Borrower
         y*wallet ⊸ wallet, Lender
         wallet ⊸ Borrower
      } ⇒ @End
}
```

Fig. 2. A standard Bike Rental contract and its modelling in *Stipula*

```
stipula Bet {
  assets wallet1, wallet2
  fields val1, val2, source, alea, amount, t_before, t_after

  agreement (Better1, Better2, DataProvider){
     DataProvider, Better1, Better2 : source, alea, t_after
     Better1, Better2 : amount, t_before
  } ⇒ @Init

  @Init Better1 : place_bet(x)[h]
```

```
    (h == amount){
        h ⊸ wallet1
        x → val1
        t_before ≫ @First { wallet1 ⊸ Better1 } ⇒ @Fail
    } ⇒ @First

  @First Better2: place_bet(x)[h]
    (h == amount){
        h ⊸ wallet2
        x → val2
        t_after ≫ @Run {
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2 } ⇒ @Fail
  } ⇒ @Run

  @Run DataProvider : data(x,y)[]
    (x==alea){
        if (y==val1 && y==val2){         // Better1 and Better2 win
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2
        } else if (y==val1 && y!=val2){  // The winner is Better1
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1
        } else if (y!=val1 && y==val2){  // The winner is Better2
            wallet1 ⊸ Better2
            wallet2 ⊸ Better2
        } else {                         // No winner
            wallet1 ⊸ DataProvider
            wallet2 ⊸ DataProvider
        }
   } ⇒ @End
}
```

Listing 1.1. The contract for a bet

Compared to the Bike Rental in Sect. 2, the role of the `DataProvider` here is less pivotal than that of the `Authority`. While it is expected that `Authority` will play its part, `DataProvider` is much less than a peer of the contract. It is sufficient that it is an independent party that is entitled to call the contract's function to supply the expected external data that will extract from `source`. In case `DataProvider` behaves incorrectly, *e.g.* it supplies an incorrect value through the function `data`, the betters can appeal against the data provider since they agreed upon the data emitted by the `source`. As usual, any dispute that might render the contract voidable or invalid, *e.g.* one better knew the result of the match in advance, can be handled by including an `Authority`, according to the pattern illustrated in the Bike Rental example.

4 Type Inference in *Stipula*

Stipula is type-free: types have been dropped because there is no type annotation in standard legal contracts and therefore they may be initially obscure to unskilled users, such as lawyers. On the other hand, a lightweight and well designed typed syntax is acknowledged as an effective support to produce quality software and to enhance code comprehension. Therefore, we postpone the choice of a suitable typed syntax to the study of an appropriate programming interface that help legal practitioners to program in *Stipula*. Nevertheless the language comes with a type inference system that allows one to derive types of assets, fields and functions' arguments, so to statically prevent basic programming errors. In this section we discuss the main design principles of the system.

Stipula has the following *primitive types*

$$T ::= \quad \texttt{real} \quad | \quad \texttt{bool} \quad | \quad \texttt{string} \quad | \quad \texttt{time} \quad | \quad \texttt{asset}$$

that mirror the set of values of the language: real numbers, booleans, strings, time values, and assets. What is exactly a time value will be specified by the concrete implementation (either over a centralized system or a distributed platform such a s a blockchain), but in general *Stipula* admits both *absolute time* values (as the date `"2022/1/1:00:15"T`) and *relative time* expressions, like `now + 3`, standing for 3 days from now or after that at least 3 blocks have been appended to the underlying blockchain. Values of type asset can be *divisible* resources (e.g. (crypto)currencies) or *indivisible* assets (e.g. smart keys, or NFT tokens). In particular, divisible assets correspond to positive real numbers (therefore we admit a subsumption rule from assets to real numbers), while indivisible ones must be considered as a whole and can be either empty asset (`0`) or a full asset (e.g. the constants `key1234` and `nft123`).

The inference system of *Stipula* is almost standard: it associates pairwise different type variables to the names of a program and parses the code by collecting constraints. At the end of the parsing process, the constraints are solved by means of a unification technique and the type variables are replaced by the resulting values (see [10] for details of the technique). Here we just discuss the most relevant rules, that are based on the following notation

- *type terms* α, α', $\cdots$, which are either *type variables* X, Y, Z, $\cdots$, or primitive types;
- *environments* Γ that maps fields and non-asset functions' arguments to type variables, and Δ maps assets and assets functions' arguments to type variables. The notation $\Gamma[\mathtt{x} \mapsto X]$, resp. $\Delta[\mathtt{h} \mapsto V]$, stands for either the update or the extension of the environment, depending on whether `x`, resp. `h`, belongs to the domain of the environment.
- *constraints* Υ, Υ', $\cdots$, which are conjunctions of equations $\alpha = \alpha'$;
- *judgments* $\Gamma, \Delta \vdash E : \alpha, \Upsilon$ for expressions E and $\Gamma, \Delta \vdash S : \Upsilon$ for statements S.

The simplest rule of the inference system is the typing of a value κ:

$$\frac{\kappa \in \mathtt{T}}{\Gamma, \Delta \vdash \kappa : \mathtt{T}, \mathit{true}}$$

That is, assuming that the constant κ belongs to `T` we derive that κ has type `T` without any constraint (the term *true*) in *every* environments Γ and Δ. A simple rule that generates constraints is the assignment of a value to a field:

$$\frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Upsilon' = (\Gamma(\mathtt{x}) = \alpha) \wedge \Upsilon}{\Gamma, \Delta \vdash E \rightarrow \mathtt{x} : \Upsilon'}$$

As usual, statements $E \rightarrow \mathtt{x}$ have no type: the typing system returns a constraint imposing that the typing of E is equal to the type of $\mathtt{x}$, *i.e.,* $\Gamma(\mathtt{x}) = \alpha$. For example, the typing of the assignment $\mathtt{"hello"} \rightarrow \mathtt{x}$ in the environments Γ, Δ returns the constraint $\Gamma(\mathtt{x}) = \mathtt{string}$.

The following rule is the typing of the asset transfer:

$$\frac{\Gamma, \Delta \vdash E : \alpha, \Upsilon \qquad \Upsilon' = (\alpha = \mathtt{real}) \wedge (\Delta(\mathtt{h}), \Delta(\mathtt{h}') = \mathtt{asset}) \wedge \Upsilon}{\Gamma, \Delta \vdash E \multimap \mathtt{h}, \mathtt{h}' : \Upsilon'}$$

The rule defines the type of the *withdraw* of the value of E from the asset $\mathtt{h}$ and the corresponding *addition* to $\mathtt{h}'$. Therefore, the expression E must have type $\mathtt{real}$, since it corresponds to a quantity to be withdrawn form the asset $\mathtt{h}$ and added to the asset $\mathtt{h}'$. An additional Subsumption rule is introduced to promote assets to be real, so that assets, when used within expressions (*e.g.* $\mathtt{h} \multimap \mathtt{h}, \mathtt{wallet}$), are considered as reals. We also remark that the type system does not check the amount of assets that are withdrawn, but the operational semantics of *Stipula* prevents the (unsafe) execution of the transfer operation whenever $\mathtt{h}$ does not own enough assets, *e.g.* $2 * \mathtt{h} \multimap \mathtt{h}, \mathtt{wallet}$.

Given a contract's function $\mathtt{F}_i$, the judgment $\Gamma, \Delta \vdash \mathtt{F}_i : \Gamma_i, \Delta_i, \Upsilon_i$ collects the constraints Υ_i generated from the typing of the function body, and the type environments Γ_i, Δ_i that associate fresh type variables to the parameters names:

$$\frac{\begin{array}{c}\overline{Y}, \overline{V} \textit{ fresh} \qquad \Gamma' = \Gamma[\overline{\mathtt{y}} \mapsto \overline{Y}] \qquad \Delta' = \Delta[\overline{\mathtt{k}} \mapsto \overline{V}] \\ \Gamma', \Delta' \vdash E : \alpha, \Upsilon \qquad \Gamma', \Delta' \vdash S : \Upsilon' \qquad \Gamma, \Delta \vdash W : \Upsilon'' \\ \Upsilon''' = (\alpha = \mathtt{bool}) \wedge \Upsilon \wedge \Upsilon' \wedge \Upsilon''\end{array}}{\Gamma, \Delta \vdash @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}](E)\,\{\, S\, W \,\} \Rightarrow @\mathtt{Q}' : [\overline{\mathtt{y}} \mapsto \overline{Y}], [\overline{\mathtt{k}} \mapsto \overline{V}], \Upsilon'''}$$

Finally, the typing of a *Stipula* contract is given in the following rule, where $\vdash \mathtt{G}$ stands for the syntactic check that the agreement $\mathtt{G}$ is well formed, and $\Upsilon \Vdash \sigma$ means that the type variable substitution σ satisfies the constraints Υ:

$$\frac{\begin{array}{c}\overline{X}, \overline{Z} \textit{ fresh} \qquad \vdash \mathtt{G} \qquad \Gamma = [\overline{\mathtt{x}} \mapsto \overline{X}] \quad \Delta = [\overline{\mathtt{h}} \mapsto \overline{Z}] \\ \Big(\Gamma, \Delta \vdash \mathtt{F}_i : \Gamma_i, \Delta_i, \Upsilon_i\Big)^{i \in 1..n} \qquad \bigwedge_{i \in 1..n} \Upsilon_i \Vdash \sigma\end{array}}{\vdash\ \mathtt{stipula}\ \mathtt{C}\,\{\,\mathtt{assets}\ \overline{\mathtt{h}}\ \ \mathtt{fields}\ \overline{\mathtt{x}}\ \ \mathtt{G}\ \ \mathtt{F}_1 \cdots \mathtt{F}_n\ \}\ : [\,\sigma(\Gamma, \Gamma_1 \cdots \Gamma_n), \sigma(\Delta\Delta_1 \cdots \Delta_n)\,]}$$

In the rule fresh type variables are associated to contract's fields and functions' parameters, both assets and non assets. These associations are recorded in the type environments $\Gamma, \Gamma_1 \cdots \Gamma_n$ and $\Delta\Delta_1 \cdots \Delta_n$ (all contract's names are assumed to be different). Then the type of the contract is obtained by applying the substitution σ to these environments. Whenever the inference system is not able to derive a ground type for a contract name, that is $\sigma(X)$ is a type variable, it

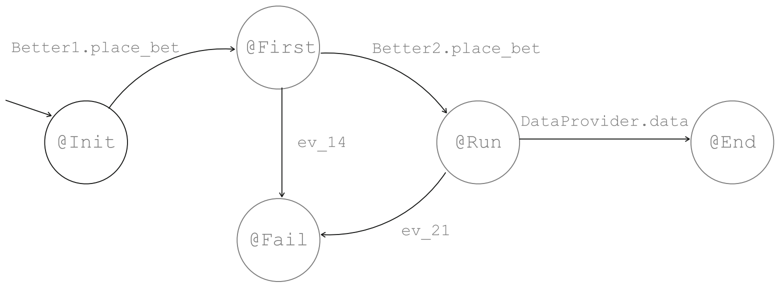


Fig. 3. The finite-state automata of the Bet contract

means that there are no type constraints for that name. In particular, as regards assets, the type system only collects assets type identities and the type variable can be safely instantiated to the ground type `asset`.

5 An Analyzer of Liquidity

Liquidity is a major security property of every program managing assets because it guarantees that assets are never frozen forever inside contracts [2]. In particular,

liquidity: a *Stipula* contract is *liquid* if, whenever an asset becomes not-0, then there is a continuation that has a state where *every asset* is 0.

According to the definition, liquidity reduces to the analysis that a state is *reachable*, which is not trivial in Solidity because functions have guards that may disable invocations and events that may prevent invocations. What we discuss in this paper is a technique for pruning the space of analysis of reachability: our technique returns witnesses to be checked by an off-the-shelf reachability tool. For simplicity sake, in this section, we consider the sublanguage where statements $\mathtt{E} \multimap \mathtt{h}, \mathtt{A}$ and $\mathtt{E} \multimap \mathtt{h}, \mathtt{h}'$ have the shape $c{*}\mathtt{h} \multimap \mathtt{h}, \mathtt{A}$ and $c{*}\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$, respectively (every example in this paper matches this constraint). We also restrict our analysis to contracts such that computations do not pass through the same state twice. That is, consider the underlying finite state automaton of the contract where states are those specified by the contract and transitions are either functions or events – Fig. 3 reports the finite state automaton of the Bet contract in Sect. 3. We are restricting to contracts where the underlying finite state automata have no cycle (the technique where this limitation is drop requires technicalities that are out of the scope of this paper).

The liquidity analyzer of a *Stipula* contract has three phases:

1. the liquidity effects of each transition of the automaton is statically computed by calculating (an over-approximation of) the assets and the asset parameters

of every function and event. More precisely, using a *type system*, we define the *liquidity label* of every function `Q A.f Q`$'$: $\Xi \to \Xi'$ and every event `Q ev_i Q`$'$: $\Xi \to \Xi'$, where the initial environment Ξ associates contract's assets with symbolic names, while the final environments Ξ' records the effect of the execution of the corresponding bodies;
2. then we compute the liquidity effects of *computations* (*i.e.* sequences of transitions) of the automaton, by suitably merging the final environment of a transition with the initial environment of the next transition;
3. finally, we consider all the functions and events that either updates the assets or carry asset parameters and check whether (*i*) all asset parameters are emptied by functions' executions, and (*ii*) for every function/event modifying an asset, there is a continuation that empties all the assets of the contract. We remark the role of asset parameters: if a contract function is called by passing an asset parameter, like an amount of currency (as in the function `pay` of the `Bike_Rental` contract), that amount is no more available to the caller because of the linear semantics of assets. Therefore it is essential that the parameter is drained by the function and the currency is moved into a contract asset (*cif.* the instruction `h` $\multimap$ `wallet` in line 12 of the `Bike_Rental` contract in Fig. 2) or sent to a party, otherwise that currency is frozen and the program is not liquid. The condition (*ii*) above requires to trace the asset movements performed by computations and to verify that contract assets (*cif.* `wallet`) have been emptied.

Below we detail few critical aspects of the analysis. The liquidity type system returns, for every transition of the automaton, an over-approximation of the balances of the assets that expresses whether an asset is empty – notation $\mathbb{0}$ – or not empty – notation ∞. The values $\mathbb{0}$ and ∞ are called *liquidity values*. We use the following notation:

– *liquidity expressions* $\mathbb{e}$, are defined as follows, where ξ, ξ', $\cdots$ range over (symbolic) liquidity names:

$$\mathbb{e} ::= \; \mathbb{0} \;\mid\; \infty \;\mid\; \xi \;\mid\; \mathbb{e} \sqcup \mathbb{e} \;\mid\; \mathbb{e} \sqcap \mathbb{e}.$$

They are ordered as $\mathbb{0} < \infty$ and $0 \leqslant \xi$ and $\xi \leqslant \infty$; the operations $\sqcup$ and $\sqcap$ respectively return the maximum and the minimum value of the two arguments.
– *environments* Ξ, which map contract's assets and asset parameters to liquidity expressions.
– *liquidity labels* $t : \Xi \to \Xi'$ where t is either `Q A.f Q`$'$ (a function) or `Q ev_i Q`$'$ (an event) and $\Xi \to \Xi'$ records the liquidity effects of fully executing the body of the transition t.
– *judgments* $\Xi \vdash E : \mathbb{e}$ for expressions, $\Xi \vdash S : \Xi'$ for statements and $\Xi \vdash$ @`Q A:f(`$\overline{\mathtt{x}}$`)[`$\overline{\mathtt{h}'}$`]` $(E)\{\,S\,W\,\} \Rightarrow$ @`Q`$'$: $\mathcal{L}$ for function definitions, where $\mathcal{L}$ is a set of liquidity labels.

As regards expressions, we consider only constants (because of the restriction on the shape of asset expressions). In particular, the two rules

$$\varXi \vdash 0 : \mathbb{0} \qquad \frac{\kappa \neq 0}{\varXi \vdash \kappa : \infty}$$

assert that every constant has liquidity value ∞ but for the constant 0.

As regard statements, we have two rules for asset movements:

$$\frac{\mathbb{e} = \varXi(\mathtt{h}) \sqcup \varXi(\mathtt{h}')}{\varXi \vdash \mathtt{h} \multimap \mathtt{h}' : \varXi[\mathtt{h} \mapsto \mathbb{0}, \mathtt{h}' \mapsto \mathbb{e}]} \qquad \frac{c \neq 1 \quad \varXi \vdash c : \mathbb{e} \quad \mathbb{e}' = (\mathbb{e} \sqcap \varXi(\mathtt{h})) \sqcup \varXi(\mathtt{h}')}{\varXi \vdash c * \mathtt{h} \multimap \mathtt{h}, \mathtt{h}' : \varXi[\mathtt{h}' \mapsto \mathbb{e}']}$$

According to the rule on the left, the final asset environment of $\mathtt{h} \multimap \mathtt{h}'$ (which is an abbreviation for $\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$) has $\mathtt{h}$ that is emptied and $\mathtt{h}'$ that gathers the value of $\mathtt{h}$, henceforth the liquidity expression $\varXi(\mathtt{h}) \sqcup \varXi(\mathtt{h}')$. Notice that, when both $\mathtt{h}$ and $\mathtt{h}'$ are $\mathbb{0}$, the overall result is $\mathbb{0}$. In the rule on the right, the asset $\mathtt{h}$ is decreased by an amount that is moved to $\mathtt{h}'$. Since c is not 1, the static analysis can only safely assume that the asset $\mathtt{h}$ is not emptied by this operation (if it was not empty before). Therefore, after the withdraw, the liquidity value of $\mathtt{h}$ has not changed. On the other hand, the asset $\mathtt{h}'$ is increased of some amount if $\mathtt{h}$ has a non zero liquidity value, henceforth the expression $(\mathbb{e} \sqcap \varXi(\mathtt{h})) \sqcup \varXi(\mathtt{h}')$. In particular, when both $\varXi(\mathtt{h})$ and $\varXi(\mathtt{h}')$ are $\mathbb{0}$, the overall result is $\mathbb{0}$.

The rule for conditionals is

$$\frac{\varXi \vdash S : \varXi' \qquad \varXi \vdash S' : \varXi''}{\varXi \vdash \mathtt{if}\ (E)\ \{\ S\ \}\ \mathtt{else}\ \{\ S'\ \} : \varXi' \sqcup \varXi''}$$

where the operation $\sqcup$ on environments is defined pointwise: $(\varXi' \sqcup \varXi'')(\mathtt{h}) = \varXi'(\mathtt{h}) \sqcup \varXi''(\mathtt{h})$. That is, the liquidity analyzer over-approximates the final environments of $\mathtt{if}\ (E)\ \{\ S\ \}\ \mathtt{else}\ \{\ S'\ \}$ by taking the maximum values between the results of parsing S (that corresponds to a true value of E) and those of S' (that corresponds to a false value of E). The expression E is overlooked by the analyzer.

The rule for *Stipula* contracts collects the *liquidity labels* that describe the liquidity effects of each contract's function; each function assumes injective environments that just associate contracts' assets with symbolic names:

$$\frac{\overline{\chi}, \overline{\xi}\ \mathit{fresh} \quad \left([\overline{\mathtt{h}} \mapsto \overline{\xi}] \vdash \mathtt{F}_i : \mathcal{L}_i\right)^{i \in 1..n}}{\vdash \mathtt{stipula}\ \mathtt{C}\ \{\ \mathtt{assets}\ \overline{\mathtt{h}}\ \mathtt{fields}\ \overline{\mathtt{x}}\ \mathtt{G}\ \mathtt{F}_1 \cdots \mathtt{F}_n\ \} : \bigcup_{i \in 1..n} \mathcal{L}_i}$$

In turn, the rule for function definitions is:

$$\frac{W = \left(E_i \gg @\mathtt{Q}_i\{\ S_i\ \} \Rightarrow @\mathtt{Q}_i'\right)^{i \in I} \quad \varXi[\overline{\mathtt{h}'} \mapsto \overline{\infty}] \vdash S : \varXi' \qquad \left(\varXi \vdash S_i : \varXi_i'\right)^{i \in I}}{\varXi \vdash @\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{x}'})[\overline{\mathtt{h}'}](E)\{\ S\ W\} \Rightarrow @\mathtt{Q}' : \begin{array}{l}\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}' : \varXi[\overline{\mathtt{h}'} \mapsto \overline{\infty}] \to \varXi' \\ \left(\mathtt{Q}_i\ \mathtt{ev\_}i\ \mathtt{Q}_i' :\ \varXi \to \varXi_i'\ \right)^{i \in I}\end{array}}$$

This rule produces a set $\mathcal{L}$ of *liquidity labels* associated to transitions of the finite state automaton. The main label is that of the function, saying that the transition named $\mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q'}$ has liquidity effects $\Xi[\overline{\mathtt{h'}} \mapsto \overline{\infty}] \to \Xi'$. As explained above, Ξ just associates contract's assets, with symbolic names. The analysis of the function's body S additionally assumes that the function parameters $\overline{\mathtt{h'}}$ are bound to ∞, because they may be any value. The liquidity effects of S are recorded by the environment Ξ'. In particular, if $\Xi'(\mathtt{h'}) = \infty$, where $\mathtt{h'}$ is an asset parameter, *i.e.* $\mathtt{h'} \notin \mathit{dom}(\Xi)$, then the asset $\mathtt{h'}$ has not been emptied by S. Therefore the asset is frozen into the parameter and the contract is not liquid. The premises also verifies the liquidity effects of events' bodies S_i, but in this case the initial environments are not extended with function parameters because the syntax of *Stipula* imposes that events are out of their scope. The set I in the rule is intended to be the set of (the initial) code lines of the events scheduled by the function. For example, the liquidity types of the Bet contract are ($\Xi = [\ \mathtt{wallet1} \mapsto \xi_1, \mathtt{wallet2} \mapsto \xi_2\ ]$):

$$\begin{array}{ll}
\mathtt{Init\ \ Better1.place\_bet\ \ First} : & \Xi[\mathtt{h} \mapsto \infty] \to \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\mathtt{First\ \ Better2.place\_bet\ \ Run} : & \Xi[\mathtt{h} \mapsto \infty] \to \Xi[\mathtt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\mathtt{Run\ \ DataProvider.data\ \ End} : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}] \\
\mathtt{First\ \ ev\_14\ \ Fail} : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}] \\
\mathtt{Run\ \ ev\_21\ \ Fail} : & \Xi \to \Xi[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}]
\end{array}$$

To calculate the effects that a computation has on the assets' balances we use abstract computations. An *abstract computation* is a finite sequences of labelled transitions $\varphi = \{t_i : \Xi_i \to \Xi'_i\}^{i \in 1..n}$. We define the liquidity type of an abstract computation φ, noted $\mathtt{L}_\varphi$, by merging the final environments of a transition with the initial environments of the next one. In particular, let $\overline{\mathtt{h}}$ be the assets of the contract and $\Xi|_{\overline{\mathtt{h}}}$ be the environment Ξ restricted to the domain $\overline{\mathtt{h}}$. Then

$$\mathtt{L}_\varphi = \ \Xi_1^{(b)}|_{\overline{\mathtt{h}}} \to \Xi_n^{(e)}|_{\overline{\mathtt{h}}}$$

where $\Xi_1^{(b)}$ and $\Xi_n^{(e)}$ ("b" stays for *begin*, "e" stays for *end*) are defined as follows

$$\Xi_1^{(b)} = \Xi_1 \qquad \Xi_{i+1}^{(b)} = \Xi_{i+1}\{^{\Xi_i^{(e)}(\overline{\mathtt{h}})}/_{\overline{\xi}}\} \qquad \Xi_i^{(e)} = \Xi'_i\{^{\Xi_i^{(b)}(\overline{\mathtt{h}})}/_{\overline{\xi}}\}\ .$$

For example, consider the Bet contract computation

$$\begin{aligned}\varphi = {} & \mathtt{Init\ \ Better1.place\_bet\ \ First}\ ;\ \mathtt{First\ \ Better2.place\_bet\ \ Run}\ ; \\ & \mathtt{Run\ \ DataProvider.data\ \ End}\end{aligned}$$

Then $\mathtt{L}_\varphi = \Xi_1^{(b)}|_{\{\mathtt{wallet1},\mathtt{wallet2}\}} \to \Xi_3^{(e)}|_{\{\mathtt{wallet1},\mathtt{wallet2}\}}$ where

$$\begin{aligned}
\Xi_1^{(b)} &= \Xi[\mathtt{h} \mapsto \infty] \\
\Xi_1^{(e)} &= \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_2^{(b)} &= \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_2^{(e)} &= \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_3^{(b)} &= \Xi[\mathtt{wallet1} \mapsto \xi_1 \sqcup \infty, \mathtt{wallet2} \mapsto \xi_2 \sqcup \infty, \mathtt{h} \mapsto \mathbb{0}] \\
\Xi_3^{(e)} &= \Xi[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}, \mathtt{h} \mapsto \mathbb{0}]
\end{aligned}$$

The last phase of the liquidity analysis amounts to checking that (*i*) the execution of every function empties every asset parameter, and (*ii*) for every function or event modifying an asset field, there is a continuation (which is a computation) that empties all the assets. We notice that, as regards (*ii*), we cannot restrict to a local analysis (as in (*i*)) but we have to consider computations because assets may become 0 in several steps. For example, for the Bet contract, there are two problematic functions: `Init Better1.place_bet First` (that updates `wallet1`) and `First Better2.place_bet Run` (that updates `wallet2`). Our technique, by using liquidity types of computations, returns all the computations that start at `First` and at `Run` that empty the assets `wallet1` and `wallet2`. In particular, for `First`, it returns

```
First Better2.place_bet Run ; Run DataProvider.data End
First ev_14 Fail
First Better2.place_bet Run ; Run Ev_21 Fail
```

(the reader is invited to verify that, in the final environments are $[\mathtt{wallet1} \mapsto \mathbb{0}, \mathtt{wallet2} \mapsto \mathbb{0}]$). For `Run` we have

```
Run DataProvider.data End
Run ev_21 Fail .
```

Provided that, at least one of the computations starting at `First` and of those starting at `Run` can be actually executed (as we anticipated, our analysis needs to be complemented by a reachability analysis), the Bet contract is liquid. Actually, it turns out that this is the case because every foregoing computation can be performed.

6 Conclusions

We have presented *Stipula*, a simple domain-specific language featuring a distilled number of operations that enable the formalisation of the main elements of juridical acts, such as permissions, prohibitions, and obligations. A number of related projects [8,11,13] have put forward legal markup languages, to wrap logic and other contextual information around traditional legal prose, and providing templates for common contracts that can be customized by setting template's parameters with appropriate values. In *Stipula*, rather than software templates, it is possible to define specific programming patterns that can be used to encode the building blocks that can be used to describe, analyse and execute (thus enforce) legal agreements (see the Fig. 1).

This is similar to what has been done in [9] where the authors have defined a set of combinators expressing financial and insurance contracts, together with a denotational semantics and algebraic properties that says what such contracts are worth. These ideas have been implemented by the Marlowe and Findel languages [1,4], which are (small) domain specific languages featuring constructs like participants, tokens, currency and timeouts to wait until a certain condition becomes true (similarly to *Stipula*).

We remark that legal contracts are more general and expressive than financial contracts. Accordingly, languages like Marlowe and Findel are built around a fixed set of contract's *combinators*, and they can be implemented using an interpreter, that is a single program that handles any financial contract by evaluating its (most external) combinator. The case of *Stipula* is more complex: agreement, assets, events, named states and named functions are programming primitives rather than combinators. Therefore each *Stipula* contract must be implemented, actually compiled, into a suitable running software, and the parties must collaborate by invoking the contract's functions to make the contract progress.

Being a principled high-level language, *Stipula* is implementation-agnostic, and does not commit to any architecture. In [7] we provided a detailed discussion about the implementation of the main elements of *Stipula* on top of either a centralized Java application or a distributed system such as a blockchain. In particular, *Stipula* might actually be implemented in terms of smart contracts written in Solidity or Obsidian [5,6], which is based on state-oriented programming and explicit management of typed linear assets. This would bring in the advantages of a public and decentralized blockchain platform. However, we think that *Stipula*'s software/digital contracts are more general and encompass smart contracts: they provide benefits in terms of automatic execution and enforcement of contractual conditions, traceability, and outcome certainty even without using a blockchain. Their implementation might be more flexible, allowing a suitable level of privacy, reversibility and intermediation. Additionally, the intrinsic open nature of legal contracts is another challenge for blockchain-based smart contracts, that can hardly deal with the off-chain world: external data can enter the blockchain only through oracles, which are problematic in many senses, and the dynamic change of behaviour conflicts with the rigidity of smart contracts definition. On the other hand, we have shown that *Stipula* contracts may take advantage of an explicit Authority party and suitable programming patterns to flexibly deal with the exceptional behaviors occurring in the external context.

Overall, we think that *Stipula* provides a programming model that is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about legal contracts and for understanding their basic principles. In our mind *Stipula*, and its toolset of formal methods, is the backbone of a framework where addressing and studying other, more complex features that are drawn from juridical acts.

References

1. Cardano Documentation (2020). https://docs.cardano.org/
2. Bartoletti, M., Zunino, R.: Verifying liquidity of bitcoin contracts. In: Nielson, F., Sands, D. (eds.) POST 2019. LNCS, vol. 11426, pp. 222–247. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17138-4_10
3. Basu, S., Mohan, A., Grimmelmann, J., Foster, N.: Legal calculi. Technical report, ProLaLa 2022 ProLaLa Programming Languages and the Law (2022). https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi
4. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: secure derivative contracts for ethereum. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 453–467. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_28
5. Coblenz, M.J., Aldrich, J., Myers, B.A., Sunshine, J.: Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian. In: Proceedings of ACM Programming Language, vol. 4, no. OOPSLA, pp. 132:1–132:28 (2020)
6. Coblenz, M.J., et al.: Obsidian: typestate and assets for safer blockchain programming. ACM Trans. Program. Lang. Syst. **42**(3), 14:1–14:82 (2020)
7. Crafa, S., Laneve, C., Sartor, G.: Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, October 2021
8. Lexon Foundation. Lexon Home Page (2019). http://www.lexon.tech
9. Peyton Jones, S.L., Eber, J.-M., Seward, J.: Composing contracts: an adventure in financial engineering, functional pearl. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), Montreal, Canada, 18–21 September 2000, pp. 280–292. ACM (2000)
10. Pierce, B.C.: Types and Programming Languages. The MIT Press, Cambridge (2002)
11. Open Source Contributors. The Accord Project (2018). https://accordproject.org
12. Research Group on EC Private Law (Acquis Group) Study Group on a European Civil Code. Principles, Definitions and Model Rules of European Private Law: Draft Common Frame of Reference (DCFR), Outline Edition. Sellier (2009)
13. Wright, A., Roon, D., ConsenSys, A.G.: OpenLaw web site (2019). https://www.openlaw.io

Towards a Modular and Variability-Aware Aerodynamic Simulator

Ferruccio Damiani[1], Michael Lienhardt[2(✉)], Bruno Maugars[2], and Bertrand Michel[2]

[1] University of Turin, Turin, Italy
ferruccio.damiani@unito.it
[2] ONERA, Palaiseau, France
{michael.lienhardt,bruno.maugars,bertrand.michel}@onera.fr

Abstract. Computational Fluid Dynamics (CFD) consists of numerically solving the fluid dynamics equations and has become a major tool in designing and evaluating any physical structures, like airplane, rotors, or even nuclear plants, where the flow of a fluid can be a critical efficiency or security aspect of these structures. Our first contribution is a brief review of the core characteristics a CFD solver should have (based on two common functionalities they usually provide) and the state of the art of CFD tools. Indeed, research on this field principally focuses on specific numerical or computation methods, software architecture is rarely discussed. Moreover, to the best of our knowledge, all CFD tools have major structural flaws that limit their capacities to integrate new methods and take advantage of new hardware. Our second contribution is a new approach that aims to solve these flaws. We exploit formal methods (namely, *order-sorted algebra* and *Delta-Oriented Programming*) to build a flexible CFD framework in which new methods can be added as modules. By exploiting *dataflow automatic generation*, our approach adds no runtime overhead. We implemented our approach and tested it on a simple example.

1 Introduction

Over the past 30 years, aerodynamic numerical simulation tools (also called CFD tools) [8,28,41,48] has been largely used and become essential for the development, sizing and maintenance of products manufactured in the aeronautics sector, like airplanes, turbines, etc. These tools calculate the flow properties and the mechanical stress (like a wind shock, a drag or a lift) applied on the manufactured products, and this information is used by the designers of the different products to guide them in their tasks (e.g., development, sizing or maintenance). Aerodynamics is described by the Navier-Stokes (NS) equations [12] which have

The authors of this paper are listed in alphabetical order. This work was partially supported by the SONICE project, granted by the French Directorate General for Civil Aviation (DGAC).

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 147–172, 2022.
https://doi.org/10.1007/978-3-031-08166-8_8

no known analytic solution [38], and so, CFD tools are structured around two approximations: first, they use equations that approximate NS; second, they use physical configurations that approximate the volume in which the fluid flows. Many equations and many meshes have been designed over the years, each of them having a specific usage: some are more suited to some specific physical conditions (e.g., supersonic speed), some trade-off precision for efficiency. But even for very efficient equations and meshes, on realistic usecases these computations are very memory and computation intensive and require massive and efficient hardware.

Consequently, CFD tools face two design challenges that seem in opposition. On one hand, they must be flexible enough to support a large catalogue of NS approximations and meshes, so they can be used to analyse different manufactured products. Moreover, since the research on NS approximation and on meshes is very active, the CFD tools must regularly be updated to integrate these new results, which also requires flexibility. On the other hand, these tools must be very specific and close to the hardware in order to be as efficient as possible: a simple cache-miss in a loop could have desastrous effect on the computation time and make the tool useless in practice. Moreover, *heterogeneous computations* (i.e., distributing the computation on different kind of computation units, like CPU and GPU) must also be fine-tuned, to avoid costly transfert of control and data between the different computation units. Finally, even though CFD tools must be fine-tuned to take as much advantage of the hardware as possible, the workstations on which these tools run are all different and in constant evolution, with the rise of new hardware technologies (GPU, TPU, VE [19], etc.).

In this paper, on the occasion of Reiner Hähnle's 60th birthday, we present our ongoing research on an approach for solving the apparent incompatibility between the requirements of flexibility and fine-tuning of a CFD tool. This approach is based on *Delta-oriented Programming* [14,15,26,53] and on an ad-hoc code generation to enable flexibility and fine-tuning, respectively. Reiner Hähnle, as part his academic work, provided outstanding contributions in the development of formal methods and tools for supporting rigorous software engineering approaches – see, e.g., the KeY tool [2] and the ABS modelling language [24]. Notably, his research has always looked at practical applications – see, e.g., the EU FP7 project HATS (Highly Adaptable and Trustworthy Software using Formal Models) [22], the EU FP7 coordinating action Eternals (Trustworthy Eternal Systems via Evolving Software, Data and Knowledge) [23], the EU FP7 project Envisage (Engineering Virtualized Services) [25], and the DB Netz AG project FormbaR (Formal modelling and analysis of Railroad operations) [33]. We therefore believe that the research activity reported in this work fully falls in Reiner Hahnle's research interests.

Outline. Section 2 briefly outlines Computational Fluid Dynamics (CFD) and its challenges. Section 3 describes the characteristics of the current approaches, focusing in particular on the elsA tool [8]. Section 4 introduces our approach, and Sects. 5 and 6 focus on the data model and the variable operators aspects of the approach, respectively. Section 7 present our initial results. Finally, Sect. 8 concludes the paper and provides and outlook on future work.

2 Computational Fluid Dynamics Challenges

This Section presents the different characteristics required of a CFD tool, and the challenges in implementing them. We structure this Section in two parts: first, we discuss the *functional characteristics* of such a tool, i.e., what are the functionalities expected by the user; second, we discuss its *computational characteristics*, i.e., how to perform the actual computation on a given hardware.

2.1 Functional Characteristics

Equilibrium State Computation. The main functionality of a CFD tool consists of computing an *equilibrium state* [50] of a given physical configuration [51,52]. This state usually consists of the temperature, the velocity and the pressure of the fluid, at every point of the volume considered in the given physical configuration.

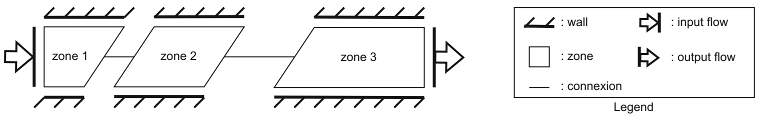


Fig. 1. CFD Physical Configuration Example

Consider for instance the configuration depicted in Fig. 1. This configuration corresponds to a simple 2D tube: the space where the fluid can flow is modelled by the three connected zones `zone 1`, `zone 2` and `zone 3` bounded by walls on the top and on the bottom, with an input gas flow on the left and a possible exit flow on the right. Depending on the fluid property, the input and output flow conditions, the friction on the walls, the computation could return an equilibrium state that can greatly vary.

The principle of the equilibrium computation is quite simple: the different zones in the configuration are implemented with meshes storing default values (2D meshes in our example in Fig. 1), and the constraints given by the different *boundary conditions* (BC) in the configuration (the input flows, the output flows and the wall in our example) are iteratively propagated in the meshes, thus changing the stored values until the value modifications derived from these constraints are negligible.

Numerical Optimization. Another essential functionality integrated in most CFD tool is called *Numerical Optimization*. Any realistic simulation depends on many parameters, e.g., the shape of a plane wing, that can have a great influence on some *objective functions* that should be minimized, e.g., the plane drag. A first approach to minimize these objective functions is to perform a large number of simulations, each of them with a different parameter configuration. This approach is not very practical however, due to size of the configuration space to explore.

An interesting alternative approach is to compute the derivative of the objective functions: since the zeros of these derivatives correspond to local minima and maxima of the objective functions, it is enough to perform the computation on these configurations, thus greatly reducing the search space. There exist two main approaches to compute these derivatives. The first one introduces a perturbation to the problem inputs and computes theirs influences on the result of the objective functions (this approach is called *forward* or *linearised* mode). It is also possible to introduce a perturbation to the result of the objective functions and study their influences on the inputs (this approach is called *backard* or *adjoint* mode). The two modes of computing this derivate have different properties: the forward/linearized mode is more efficient when number of objective functions is greater than the number of parameters, while the backward/adjoint mode is more efficient in the opposite case. In practice, the number of parameters is several order of magnitude bigger that the number of objective functions, and so the backward/adjoint mode is the most efficient. However, the backward/adjoint mode add huge constraints in term of software architecture because all the derivatives must be propagated and computed backwards [34].

Kenway et al. in [34] give an in-depth discussion on the different methods to compute the derivatives necessary to solve the numerical optimization problem, and give clear motivations why *discrete adjoint* approaches must in general be preferred over *continuous direct* approaches. Additionally, they compare different implementations for computing such a discrete adjoint, and a code generation technique called *Automatic differentiation* (AD) gives the best result, in term of memory usage, speed and accuracy.

AD thus is a very powerful technique, but it has one important limitation. It is a source to source transformation techniques that generates from the implementation of a function a code computing the derivative of that implementation. And in order to produce correct code, AD expects that the input implementation follows a simple workflow pattern.

Hence it is important to structure a CFD tool in a way so that the computation it performed is expressed as a simple workflow that matches the AD restrictions.

Functional Variability. While the computation of a physical configuration's equilibrium state is always a fixpoint loop, one of the main difficulty in designing a CFD solver is to manage the fact that the content of that loop has a very large number of variants, and that this number is always increasing. This very large variability has three causes, two of which we already introduced:

1. **The approximation method.** As previously discussed, many approximation methods for NS have been and are still being designed [4,29,32,36], each of them having their own advantages and disadvantages, e.g., are more suited to specific physical configurations, to specific data computation, etc.
   Additionally, many of these methods use constants (modelling some physical properties) that must be set by the user. Finally, some of these methods are designed in a way so they are incompatible with other variable elements, e.g., some physical configurations.

2. **The physical configuration.** Each configuration is unique and requires a tailored computation. First as previously discussed, the BCs describe the constraints on the flow of the fluid going through the space modelled by the zones, and each of them has a specific implementation. Then, the flow follows the links between the zone, and so the topology of the physical configuration has a direct impact on the computation.
   Of course it is possible to design (as it has already been done in the past) a unique spaghetti code that can manage all possible physical configurations. However such a code would be unmaintainable and highly inefficient. Indeed, such a centralized code needs to have direct access to all the arrays and matrices during the physical simulation, which causes important latency in accessing the memory for physical configuration of regular size. This issue is discussed in more detail in Sect. 2.2.
3. **The user requested data.** The user can request the computation of some specific data (e.g., an objective function), which must be included in the fixpoint loop. The computation of this data may require internally the computation of some other temporary information on the flow of the fluid, which adds a layer of complexity in the construction of the fixpoint loop. Moreover, in some case the precision of the requested data can be configured, which in turns may require to change how the temporary information are computed.

Runtime Checkup. Finally, it could be very useful to be able to insert monitoring and controlling capabilities at key points in a CFD computation. Indeed, such computations can take a very long time. So, it could first be very useful to be able to regularly store a snapshot of the current computation so in case of *hardware failure* we might not lose hours or even days of computation. Moreover, *convergence* of the fixpoint loop is not guaranteed in many cases: monitoring and controlling capabilities could be very useful to detect when the computation is not converging and to update some of the solver's options in order to solve the problem, or stop the computation if no solution can be found.

2.2 Computational Challenges

Like many other HPC applications, CFD is in general very memory and computation intensive, and so it is very important to use as efficiently as possible the available hardware.

Distribution. The first difficulty in using efficiently the hardware is data locality [35,46,58]. Indeed, in many cases the meshes of a physical configuration count several millions or even billions of points, and standard *SMP* memories (that can be accessed uniformly by all the CPU in a workstation) cannot scale to such sizes: the latency in accessing the memory becomes too big. The *NUMA* memory design (which stands for *Non Uniform Memory Access*) solves this scaling issue by structuring the memory in *nodes*, each one having a guaranteed good latency with one CPU. Consequently, it is important to partition the meshes in chunks that can be stored in one node of memory, and to partition the computation so that each part of the computation is performed on the CPU close to the data it manipulates.

Communication. Some functions, called *stencil functions* [9,30], compute a value on a node of a mesh by looking at the neighbours of that node (similarly to the *convolution* operation in Artificial Intelligence). However, since the mesh is distributed, some neighbours are not locally available on a CPU: it can be necessary before running a stencil function to fetch the value of these neighbours from the NUMA node that hosts them.

Computation Reordering. A well known method to optimize catch access is to reorder some computation, so that those that use the same data are executed together. This optimization is implemented in most compilers, but can only be applied on a fine-tuned program: any CFD tool with some flexibility will not be optimized by the compiler. Hence it is important to find a way for a CFD tool to perform this optimization itself.

Heterogeneous Hardware. Now-a-days, workstations have several kind of processing units (PU), e.g., CPU and GPU. Moreover, several means of communication (with different properties) exist between these PUs, e.g., PCIe and NVLink. Since some computation are more efficient on some hardware (e.g., a GPU handles well repetitive computation over large sets of data), and some cannot be performed on them (e.g., GPUs do not have function pointers), it is important to design a distribution plan that put the computation on suited PUs, while taking in account the latency of data transfer.

Variable Hardware. The final difficulty is to be able to manage the fact that a CFD tool will be executed on several workstation, each of them with its own hardware. Hence, the hardware itself becomes variable in this context, and the distribution plan discussed in the previous paragraph must be generated and tailored for workstation running the tool.

Figure 2 illustrates the shape of the a possible distribution plan of the physical configuration of Fig. 1. In this example, we consider an hardware architecture with two NUMA nodes, the first one hosting `zone 1` and `zone 2` and the second one hosting `zone 3` of our physical configuration example of Fig. 1. The first NUMA node is linked to a dual core, the first core having two threads while the second having only one. The second NUMA node is linked to a highly parallel architecture, like a GPU. The arrows between NUMA nodes and caches represent the different communications that are necessary for the computation of the equilibrium states.

Communication between a NUMA node and the local caches is quick and all the data stored in the node must at one point of the computation be sent in the cache. But it is still better to avoid useless transfer between the NUMA node and the cache. Communication between NUMA nodes is slower, and should be

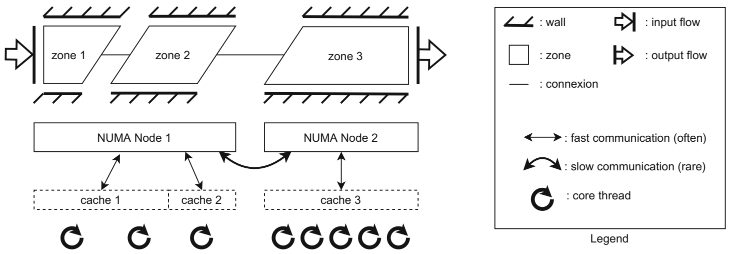


Fig. 2. CFD intended Running Architecture

perform only when necessary. In opposite to the communication between NUMA nodes and caches however, the data exchanged between NUMA nodes is far less than the whole content of the nodes.

3 State of the Art

In this Section, we give a brief presentation of the main CFD tools, and then focus on elsA [8] which has been developed at ONERA for the last 20 years. Many mature tools (like elsA) have been developed in research centre but in close relation to industry. Consequently, while documentations on how to use these tools are freely available, in-depth description of their internals and how they deal with the challenges presented in Sect. 2 are (to the best of our knowledge) not published. This is the case of FUN3D [5] (developed at NASA), TRACE [48] and Flucs [39] (developed at DLR), and elsA [8] (developed at ONERA).

On the other hand, less mature but more documented open source CFD tools are now available. SU^2 [49] and OpenFOAM [31] are developed in pure C++, and largely advertise the use of classes and inheritance to: *i*) structure their code in modules; *ii*) reuse the code in different part of their toolchain; and *iii*) use uniform APIs to have more generic code. However, having modular and generic code is not enough to capture all the flexibility expected from a CFD tool, and all the management of the user requested data, of the input physical configuration and of the hardware must be implemented directly by the user. Moreover, these tools do not implement numerical optimization and can only run on CPUs (due to the language limitation).

pyFR [57] is a tool implemented in python and uses the many libraries available in this language to perform quite well. While the `sympy` library is used to provide an abstract DSL in which the user can write his mathematical formula, the orchestration of these formula (how and when they are executed) must be written in python by the user. Then, at runtime, when a mathematical formula must be executed, pyFR translates it into C or CUDA code (for an execution on CPU or on GPU), and compiles and runs the code. This tool does not implement numerical optimization. Devito [40] is similar to pyFR. It also uses the `sympy` library to express mathematical formula in python, but uses its own DSL to orchestrate them. That way, the whole computation (the formula and the orchestration) can be translated into C code and run in parallel. However, since GPUs cannot perform orchestration, Devito does not run on GPU for now. While pyFR and Devito have very interesting approaches, these tools suffer from the same main issues as SU^2 and OpenFOAM: the orchestration of the mathematical formula (which includes the management of the user requested data, of the input physical configuration and of the hardware) must be implemented directly by the user.

3.1 elsA's Approach

elsA is a mature CFD tool and solves, at least partially, the gap between flexibility and fine-tuning. Its solution relies on its 3 parts structure which, for historical and performance reasons, are all implemented in a different language. First, similarly to pyFR and Devito, elsA distinguishes between mathematical functions and orchestration: mathematical functions are called *operators* in elsA and are implemented in Fortran 90 [45][1] which is a very efficient language for physic simulation and that is simple enough to support automatic differentiation; orchestration is implemented in python, but in the opposite of pyFR and Devito, it is handled by elsA directly and requires almost no setup by the user. The third part of elsA, called *HPC layer*, handles the hardware, in particular the management of the distribution of the computation, and is implemented in C++17 [55].

The transition between flexibility and fine-tuning is handled by an initial analysis of all the inputs by the orchestrator, which produces a plan of which function to execute, in what order and on which hardware. This initial analysis is structured in 3 steps, and once it completes, the actual computation, i.e., the execution of the generated plan, starts.

Step 1: Loading the Inputs. First, elsA loads the different inputs:

- It queries the available MPI [47] library for the NUMA and CPU structure (GPU are not supported by elsA). elsA at this stage assumes for simplicity that the only kind of computational units in the hardware architecture are identical CPUs. This hypothesis ensures that the cache sizes of all CPUs are the same.
- The physical configuration is loaded from a CGNS file [51,52], which is a standard format for CFD physical configuration. This file format stores among other data the topological structure of the physical configuration with all the BC setups, which makes it de facto one of the main standards to store this part of the configuration space.
- The user requested data is also loaded from the CGNS file. They are specified in special entries, distinct from the ones describing the topology of the physical configuration. These entries inform elsA about which data to compute

[1] Mathematical DSLs like `sympy` that could be translated in efficient code did not yet exist when elsA was already mature.

and on which zone to compute it. Currently, the set of data the user can request is fixed (this set is specified by an enum in the elsA API).

– The approximation method options are loaded from a elsA-specific python file. There is no well structured and clear management of the options in elsA. In order to manage the very large number of option, an initial dependency mechanism has been implemented, based on an ad-hoc usage of python dictionaries. But this system is difficult to maintain and cannot express all the dependencies and conflicts the options actually have, and currently, it only performs basic checks while many configuration errors are detected during computation.

Step 2: Managing Hardware Flexibility. Once the setup has been loaded, elsA uses its homogeneous hardware hypothesis to uniformly distribute the physical configuration over the CPUs. This is done by splitting the zones into subzones and distributing them over the NUMA nodes, so that every subzones are hosted on one unique NUMA node, and that all this data is equally distributed between the available CPUs. Additionally, elsA reorders the information within each subzone so that all data that should be access together are contiguous in memory, thus avoiding useless cache misses.

Note that information about the structure of the data distribution is kept by elsA's HPC layer which is responsible of managing the distribution of the computation. That way, it is able to give in parameter to each executing operator the data hosted in the local NUMA node.

Step 3: Managing Functional Flexibility. In order to manage the approximation method options and the user requested data, elsA uses an ad-hoc and powerful architecture that generates the list of all the operator to execute for each CPU on the workstation. This architecture, called *Factory*, is illustrated in Fig. 3.

The factory is structured in two parts. The first part is a hard-coded registration of all the operators available in elsA with: *i*) their dependencies (i.e., when some input data is computed by another operator); and *ii*) how they are triggered or disabled by the different inputs. The dependencies and triggers are mainly implemented with simple `if` conditions and result in a rather large spaghetti code. This part takes in input the approximation method options and the user requested data, and produces an initial, non optimized list of operators to execute. The second part of the factory cleans and restructure the list of operator to make it more efficient by applying standard optimization techniques, like operator reordering. Moreover, this part also insert communication operators in the list to ensure that the local data is consistent with the data on other CPUs.

The result of the Factory's computation (on the right of Fig. 3) is a list of operators and communications to be executed for a given subzone: the factory is executed on every CPU hosting data, to compute what this CPU needs to do. Moreover, this list has an essential property for the numerical optimization capabilities of elsA: it can be efficiently differentiated. Indeed, since all the operators (implemented in Fortran) can be automatically differentiated, so is a simple sequence of these operators.

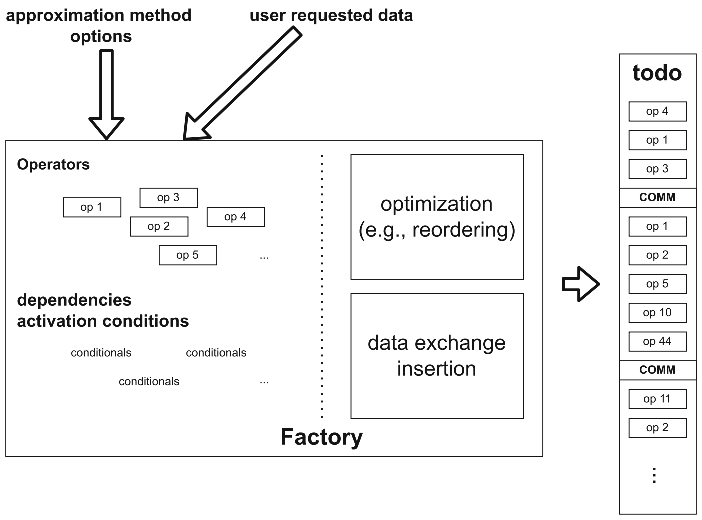


Fig. 3. elsA Factory

3.2 elsA's Approach Limitations

While elsA and its workflow is used in production to solve complex industrial usecases, it should be clear now that it has strong limitations both in its management of the hardware and functional flexibility.

Hardware Flexibility. The first limitation of elsA is its uniform CPU architecture hypothesis. Heterogeneous architectures involving different kind of computational units are getting ubiquitous [1,18], and this hypothesis is simply no longer realistic.

Functional Flexibility. In this context, elsA suffers from 4 main issues. The first one concerns the approximation method options. elsA currently contains more that 2000 of these interdependent options, without any validation tool: when the user is lucky, an erroneous configuration makes the factory fail and gets an error message almost instantaneously; however for many erroneous configurations, the factory can generate a plan, and the user needs to wait the result of the computation to see that something went wrong, without knowing where.

The second one is the difficulty to maintain the specification of the dependencies between operators and their activation conditions. Simple conditionals do not scale to manage hundreds of operators and thousands of options.

The third one concerns numerical optimization. While the operator list generated by the factory can be differentiated, many operators and communications in that list are not relevant for the derivation of many objective functions, and so the automatic differentiation implemented in elsA performs many useless computation and should be optimized.

Finally, elsA is too restrictive in its specification of user requested data. Indeed, as previously stated, elsA only provides a fixed list of possible data to compute to the user. This limitation make it so that every time a user wants to analyse some new data, or some new interesting objective functions is designed, the user needs to ask the elsA team to implement the computation of that specific data, even if all the operators necessary to compute it are already implemented. This could cost a lot time and effort to the user and the elsA team, and having a more generic approach to user request could significantly reduce this cost.

4 Our Approach

As stated in the introduction, the goal of our approach is to bridge the gap between the requirements of flexibility and fine-tuning in a CFD tool. This step is necessary to answer the different challenges described in Sect. 2. To achieve this, our approach follows the 3 parts structure of elsA, with a complete redesign of the factory. Indeed, structuring the computation in elementary operators is necessary to manage flexible hardware without cluttering the computation code with concurrency concerns; the factory's automatic generation of a plan is necessary to be able to seamlessly use the tool with different configurations; and the HPC layer is necessary for the management of the actual computation.

Hence, the main novelty of our approach is a new factory, whose architecture is presented in Fig. 4. This architecture is structured in 4 parts: one for the *graph generator* and one for each of the factory's input.

The Graph Generator. The core idea of our approach is to replace the spaghetti code in the factory with a clear notion of dependencies between operators. That way, the generation of the plan simply corresponds to a dependency resolution, which results in a *Directed Acyclic Graph* of operators instead of a list. Incidentally, this graph is actually exactly what is needed to avoid the problem elsA has with numerical optimization: since the dependencies between operators are explicit, we can identify the operators that are necessary for the computation of a specific objective function, and only derivate them.

We implement the notion of dependency by requiring the developer to specify the semantics of the inputs and outputs of each operators. Indeed, while the actual inputs and outputs of the operators are usually arrays of double, the semantics of the contained values, e.g., the fluid density or the gradient of the temperature, is specific to each operator. This specification is similar to type annotations, except that an operator can have several outputs. Moreover, some operators require special care, like the gradient operator because it can compute the gradient of any value. its specification corresponds to a type of the form $\alpha \rightarrow$

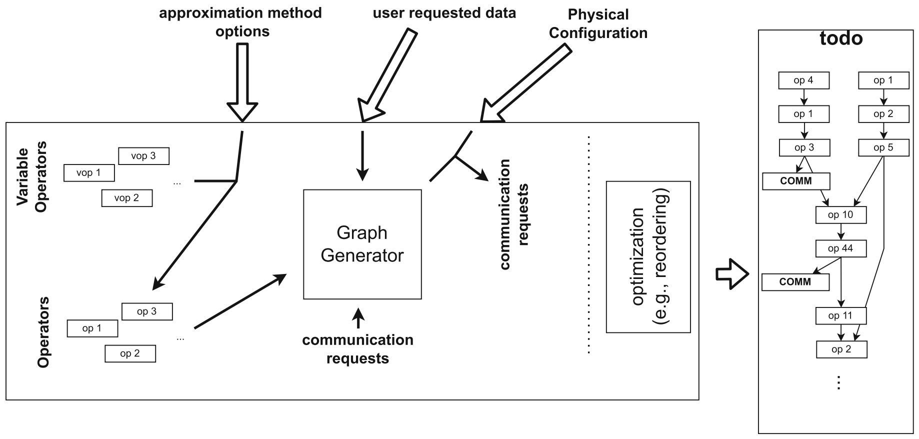


Fig. 4. New Variability Management

`grad`(α). For simplicity and genericity, we thus use *terms* for our specifications, and will give more details on our usage of terms and our implementation in Sect. 5.

Managing the Options and the Operators. Selecting or not options changes the implementation of related operators. Moreover, depending on which implementation is used, the inputs and outputs of the operator may vary.

To deal with this variability, we use *Software Product Lines* (SPLs) [3,13,53,56], and more precisely *Delta-Oriented Programming* (DOP) [14,15,53] to make the specification of the operators variable w.r.t. the options selected by the user. We will detail this part in Sect. 6, but for now it is enough to know that applying a specific set P of selected options on a *variable operator specification* returns the specification of this operator's implementation for P.

Managing the User Requested Data. Our approach uses the same terms for the User Requested Data and for the operator specifications. That way, such a request can be considered like any other dependency by our graph generator.

Managing the Physical Configuration. The physical configuration is used by our graph generator to identify the communications that need to be inserted in the graph, and where. Indeed, during computation, stencil operators require fetching specific values from the neighbours of the zone on which the stencil is being executed. To illustrate how this requirement is managed in our approach, let first state that similarly to elsA, we have one graph generator per PU, that generates the graph to execute on that PU. Let now consider a specific PU, and note G the graph generator for that PU, and if Z is a zone, then *neigh*(Z) is the set of all neighbours of Z. Upon inserting a stencil operator working on a zone Z in the graph, G also adds the corresponding `receive` operators with all the zones in *neigh*(Z) and sends a *communication request* to the graph generators managing the zones in *neigh*(Z). Upon the reception of a communication request, G adds a corresponding `send` operator in the graph: that operator depends on the requested data, and so the dependency analysis will ensure that this data is computed before being sent.

The two following Sections will go more in-depth into two aspects of this new variability management: Sect. 5 discusses the usage and implementation of terms in our approach; and Sect. 6 details the notions of SPLs and DOP, and discusses the implementation of the variable operators.

5 Data Model

As described in Sect. 4, we use terms to specify the inputs and outputs of our operators. Additionally, we use *order-sorted Algebra* [16,44] to specify which terms are valid inputs and outputs. This combination is particularly suited to our needs: terms offer a very flexible structure to specify the data exchanged between operators, and such flexibility is necessary when considering the maintainability and future evolutions of the tool; on the other hand, order-sorted Algebra is used to ensure that the user gives at least a sensible specification to his operators[2].

Finally, terms support efficient *pattern matching* (a subcase of term unification [17] where one term is ground), which is a functionality required by the graph generator: solving a dependency corresponds to finding an operators that has one output matching that dependency.

5.1 Implementation

While order-sorted algebra has been implemented in various formal specification tools [11,21], to the best of our knowledge no existing implementation can be used in our approach. Indeed, our approach requires the order-sorted algebra implementation to provide the following three elements:

- a simple syntax to specify the inputs and outputs of an operator;
- a pattern matching API that can be used by an external graph generator;
- a mean to integrate the syntax in external transformation function used to generate operator specifications.

Consequently, we implemented our own library, and choose python for the implementation language. Python is a popular language in which to embed Domain Specific Languages (DSLs) due to its very flexible syntax and its readiness to support C and C++ libraries [6,27,59]. Moreover, embedding a DSL in an existing language allows for its seamless integration with other functionalities available in the language. That way, all three requirements we listed are satisfied: we have a DSL for the syntax requirement; and its integration in python

[2] The flexibility of terms and ease to specify algebra was also a key element in the development of our approach: many trials and errors went into the design of a term structure that captures the necessary features of a CFD data.

answers to last two requirements. In particular, it allows for the integration of this library with our other library implementing Delta-Oriented Programming and presented in Sect. 6.

First we designed the following DSL to specify a signature:

$\mathit{sig} ::= \texttt{declare\_sig}(\overline{\mathit{srt}}, \texttt{order} = (\overline{\mathit{od}}))$	Signature Declaration
$\mathit{srt} ::= \mathit{id} = (\overline{\mathit{ct}})$	Sort Declaration
$\mathit{ct} \;\; ::= (\mathit{id}, \overline{\mathit{id}})$	Constructor Declaration
$\mathit{od} \; ::= (\mathit{id}, \mathit{id})$	Order Declaration

As usual, $\overline{X}$ denotes a possibly empty finite sequence of elements X and $[X]$ denotes that the element X is optional. This DSL does not follow standard signature declaration like in Maude [11] because of the limitation of Python syntax. A signature is declared using the `declare_sig` function, and is composed by the declaration of a list of sort, plus some partial ordering between sorts. A sort declaration *srt* first gives a name *id* to the sort, and introduces the set of constructors of this sort. A constructor declaration *ct* is a tuple of names *id* where the first one is the name of the constructor, and the others are the sorts of the different parameters of the constructor. Finally, an order declaration *od* simply gives a order relation between two sorts.

Example 1. A simple signature for natural numbers can be described as follow:

```
sig_nat = declare_sig(
 nat = (
  ("zero",),
  ("succ", "nat")
))
```

Here, the signature `sig_nat` contains one sort called `nat` and two constructors: `zero` of sort `nat`, with no parameter; and `succ` of sort `nat`, with one parameter of sort `nat`.

Once a signature has been defined, it is first possible to extend it by calling: the method `add_sort(`*id*`)` which adds a new sort named *id* to the signature; or the method `add_constructor(`id`,`id`,`$\overline{\mathit{id}}$`)` where the first parameter is the name of the constructor's sort, the second parameter is the name of the constructor, and the other parameters are the names of the sort of the constructor's parameters.

It is also possible to create terms. The method `fresh_variable(`*id*`)` returns a fresh variable of sort *id*. Structured term construction uses the Python lookup API to make term constructors directly available as fields or methods of a signature. For instance in the context of Example 1, the expression `sig_nat.succ( sig_nat.zero)` corresponds to 1.

Finally, pattern matching is available with the method `match`. This method returns `None` if the pattern matching fails, or a substitution that can be applied on a term.

Example 2. To illustrate the term construction and pattern matching of our library, let consider the following python code:

```
1 plus_one = sig_nat.succ(sig_nat.fresh_variable("nat"))
2 two = sig_nat.succ(sig_nat.succ(sig_nat.zero))
3 subst = sig_nat.match(plus_one, two)
4 if(subst is not None):
5  assert(subst(plus_one) == two)
```

Line 1 creates a nat term called `plus_one` containing a fresh variable. Line 2 creates a term called `two` corresponding to the number 2. Line 3 matches `plus_one` against `two` and stores the result in `subst`. By construction, the pattern matching succeeds, and the result is the substitution mapping the variable to `sig_nat.succ (sig_nat.zero)` (which corresponds to the number 1). In line 5 we check that the computed substitution is correct, by ensuring that applying it on `plus_one` does return the term `two`.

5.2 Application to CFD

In a simple setting, CFD data can be specified with a triplet. The first component corresponds to the a *value* stored in the data, like `Density` or `grad(Momentum)`. The second is a location, i.e., on which element of a mesh that value is placed; possible locations on a 3D mesh are `vertex`, `edge`, `face` or `cell`. The third component is the id of the zone (i.e., the mesh) where the data lives.

The following code excerpt presents a part of the signature we designed:

```
1 cfd_sig = declare_sig(
2  data = ( ("data", "value", "location", "zone_id"), ),
3  value = (
4   ("Density",), ("Energy",), ("Momentum",),
5   ("grad", "value"),
6  ),
7  location = ( ("cell",), ("face",), ("edge",), ("vertex",)
     ),
8  zone_id = ( ("zero",), ("succ", "zone_id") ),
9 )
```

We model an data with the `data` constructor (of sort `data`), declared in Line 2. This data takes three parameters, respectively of sort `value`, `location` and `zone_id`. A value can either be base values like `Density Energy` or `Momentum`, or structured ones like the `grad` of another value. As previously discussed, we have four constructors for locations, and `zone_id` are modelled like natural numbers.

The following code excerpt illustrates our signature by specifying the input and output data of the gradient operator.

```
vzone = cfd_sig.fresh_variable("zone_id")
vvalue = cfd_sig.fresh_variable("value")

gradient_input = cfd_sig.data(vvalue, cfd_sig.cell, vzone)
gradient_output = cfd_sig.data(
  cfd_sig.grad(vvalue), cfd_sig.cell, vzone)
```

We first declare two variables, one for the zone of the input data of the operator, and one for its input value. Then line 4 states that any data whose location is `cell` is a valid input to the gradient operator. Line 6 on the other hand states that the output data of the gradient operator is also on `cell`, on the same zone as the input data, and its value is the `grad` of the input value.

6 Variable Operators

As described in Sect. 4, we use SPL [3,13,53,56] and DOP [14,15,53] to manage both: the relationship between the approximation method options; and how the operators' implementation and specification are affected by the selection of these options.

SPL corresponds to the concept of managing a collection of similar software artefacts that are characterized by the set of *features* they implements. The selection of a set of feature is called a *product*, and the artefact corresponding to that product is called *the product's variant*. One key aspect of an SPL is the explicit specifications of its features' dependencies and incompatibilities. For instance, firefox can be compiled with the gtk or aqua graphics library, but not both at the same time: these two features are incompatible. *Feature Models* [13,56] are a standard way to declare the relationship between features.

DOP is a *transformative* approach to implement SPLs, i.e., a product's variant can be obtained by applying the set of transformations (called *delta*) corresponding to the product on a initial artefact. DOP structures an SPL in 4 parts: a *feature model* gives the features of the SPL and their relationship; an *initial artefact* gives the starting point for the generation of all variants of the SPL; a global *set of deltas* lists all the transformations that can be applied during the computation of a product's variant; and *configuration knowledge* maps every delta to the set of products that activate its execution, and also states in which order delta must be applied. The activation set of a delta is usually specified with a Boolean formula over the features of the SPL.

In our approach, we wrap every operator specification in a DOP product line, where deltas can: add new inputs and outputs to the initial specification; and change the link to the actual implementation of the operator. Moreover, all these SPL share a common Feature Model that lists all the available options of the CFD tool and their relationship. That way, we have a clear way to ensure that the options selected by the user are correct or issue a message stating which relationship is being broken before any computation happens.

While DOP has been implemented for several types of artefacts [10,37,54], to the best of our knowledge, no existing implementation can be used in our approach. Indeed, while the framework presented in [54] is generic enough to express DOP product lines over operator specifications, it has two major drawbacks: *i*) the amount of implementation to use this framework is disproportionate compared to the simple structure of an operator specification; and *ii*) it only considers product lines in isolation and thus cannot share a common feature model between different SPLs.

6.1 Implementation

Our implementation of DOP follows the same principles of our implementation of terms and is structured in two DSLs: one for the Feature Model and one for the definition of DOP product lines.

First, we designed the following Feature Model DSL, based on existing representation [13,56]:

$\mathit{fd} ::= \texttt{FD}(\mathit{id}, \overline{\mathit{fatt}}, \overline{\mathit{fg}}, [\mathit{ctc}])$	Feature Diagram
$\mathit{fg} ::= \mathit{fop}(\overline{\mathit{fd}})$	Feature Group
$\mathit{fop} ::= \texttt{FDAnd} \mid \texttt{FDAny} \mid \texttt{FDOr} \mid \texttt{FDXor} \mid \ldots$	Feature Group Operations
$\mathit{fatt} ::= \texttt{Att}(\mathit{id}, \text{domain})$	Feature Attribute
$\mathit{ctc} ::= \mathit{id} \mid \text{Pred}(\overline{\mathit{id}}) \mid \texttt{And}(\overline{\mathit{ctc}}) \mid \ldots$	Cross Tree Constraint

As usual, $\overline{X}$ denotes a possibly empty finite sequence of elements X and $[X]$ denotes that the element X is optional. This DSL fits Python syntax and describes a feature diagram with attributes and cross-tree constraints. A Feature diagram *fd* declares a feature *id* with possible associated attributes $\overline{\mathit{fatt}}$, can have sub-trees identified by a set of feature diagram groups $\overline{\mathit{fg}}$ and may have a cross-tree constraint *ctc* linking features and attributes declared in its sub-trees. A Feature diagram group *fg* gives a constraints *fop* on a set of feature diagrams $\overline{\mathit{fd}}$: `FDAnd` means that all diagrams must be selected; `FDAny` means that all diagrams are optional; `FDOr` means that at least one diagram must be selected; and `FDXor` means that exactly one diagram must be selected. Attributes *fatt* have a name *id* and a domain, which is left unspecified in this grammar (in elsA, it is expected that most of these attributes would be floats or arrays of floats). Finally, cross-tree constraints *ctc* are generic SAT constraints over feature names *id*, extended with domain specific predicates (e.g., float comparison).

Second, we implemented a very simple API to declare product lines and add deltas to it. The following line declares a new product line named `spl`, with configuration space `fm` (e.g., a feature diagram as discussed previously) and core product `core`:

```
  spl = SPL(fm, core)
```

Declaring a delta to the product line `spl` is done as follows:

```
@spl.delta(ac)
def delta(variant, product):
 code
```

The annotation `@spl.delta(ac)` registers the following function as a delta of `spl`, with the activation condition `ac` that follows the cross-tree constraint syntax. The function itself can have any name, but must have two parameters: the first one `variant` is the variant that is transformed by the delta; and the second one `product` is the product that may contain information necessary for the application of the deltas (e.g., the value of specific attributes). The transformation `code` performed by the delta is arbitrary python code. In particular, like in [54] transformation functions or methods must be provided to be able to construct a variant.

6.2 Application to CFD

Using the expressiveness available in feature models and in our python implementation in particular, we designed an initial feature model corresponding to a small part of the expected variability of a CFD tool. An except of that part is given in Fig. 5. In particular, the `physical_model` option which represent one aspect of the approximation method's variability is already quite large, and we didn't develop the `eos` subtree which too has many variation on the model of gas will be used in the fixpoint computation.

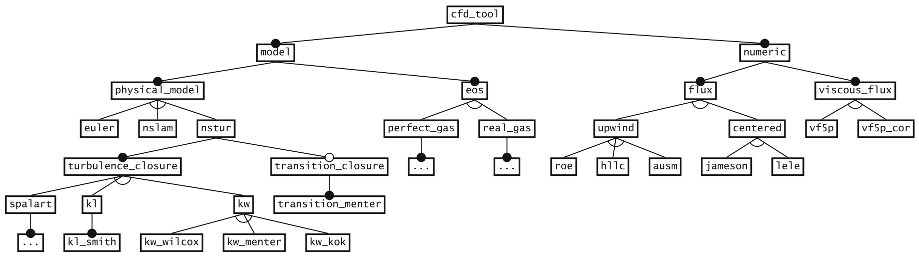


Fig. 5. Excerpt of our Feature Model

We illustrate our implementation of this feature model in Fig. 6, with the implementation of the feature `turbulence_closure`. Note that like in Fig. 5, the three dots corresponds to a set of large subtrees.

```
fm_turbulence_closure = FD("turbulence_closure",
 FDXor(
  FD("spalart", FDAnd(...)),
  FD("kl", FDAnd(FD("kl_smith"))),
  FD("kw",
   FDXor(FD("kw_wilcox"), FD("kw_menter"), FD("kw_kok"))
)))
```

Fig. 6. Implementation of the Feature `turbulence_closure`

Concerning the implementation of our variable operator specifications (VOSs), we implemented three core transformations on such specifications: the `add_input` method adds an input to the specification; the `add_output` method adds an output to it; and the `set_implementation` method states which implementation (given by the name of the implementing file) of the operator must be used.

We illustrate these methods in Fig. 7, which presents the VOS of the `FxcUpwindMeanFlow` operator. Line 1 declares the VO. Lines 3–7 states that the corresponding operator has two inputs and one outputs when the option

```
FxcUpwindMeanFlow = spl(fm, Operator)

@FxcUpwindMeanFlow.delta("upwind")
def fxc_upwind_construct_op(op, product):
 op.add_input(cfd_sig.conservatives(subsystem_term), cfd_sig.
    cell, vzone)
 op.add_input(cfd_sig.primitives(subsystem_term), cfd_sig.
    cell, vzone)
 op.add_output(cfd_sig.FxcUpwindMeanFlow, cfd_sig.cell, vzone
    )


@FxcUpwindMeanFlow.delta(And("perfect_gas", "roe"))
def fxc_upwind_mean_flow_perfect_gas_roe_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/roe"

@FxcUpwindMeanFlow.delta(And("perfect_gas", "hllc"))
def fxc_upwind_mean_flow_perfect_gas_hllc_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/hllc"

@FxcUpwindMeanFlow.delta(And("perfect_gas", "ausm"))
def fxc_upwind_mean_flow_perfect_gas_ausm_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/ausm"
```

Fig. 7. Implementation of the `FxcUpwindMeanFlow` variable operator

`"upwind"` is selected. The fact that the operator has no input or outputs when `"upwind"` is not selected encodes the fact that this operator is no used in these cases. The actual implementation of the operator to use is state in the other deltas of the `FxcUpwindMeanFlow` product line. For simplicity, we only give the deltas related to the `"perfect_gas"` option, which all depend on which subfeature of `"upwind"` is selected.

7 Initial Results

To test our approach, we integrated it into a core running prototype that could run code on a single processing unit, either CPU, GPU or VE. We applied this prototype to a simple common usecase: the 2D NACA 0012 [42,43]. This usecase is a simple 2D physical configuration modelling an airplane wing in a flow of air. The physical configuration is given in Fig. 8: on the left, we have a input air flow modelling the plane going forward, in the middle, we have a wall modelling the cross section of the wing, and on the right we have the output flow. The zone where the air can flow is a disc, so not to introduce artifacts in the air flow caused by artificial angles.

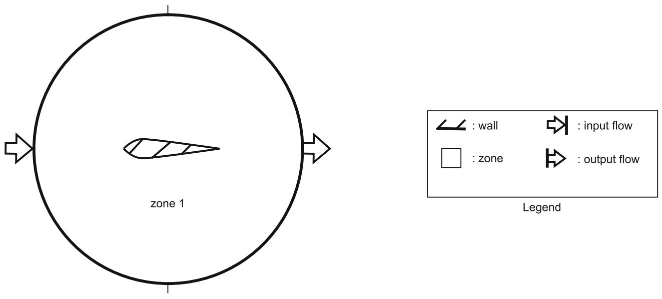


Fig. 8. Topology of the 2D NACA 0012 Usecase

The results of our study are shown in Figs. 9 and 10. Figure 9 gives three convergence criteria of the fixpoint loop obtained by running three different configuration of our prototype: once configuring it to execute on a CPU, one on a GPU and the last one on a VE. It might not be obvious to see, but in this picture there are actually three red lines, three blue lines and three cyan lines, corresponding to the three criteria of the three runs of the prototype: three runs of our prototype, even if running on different hardware, are indistinguishable between each other.

Figure 10 shows the actual result of the equilibrium state computation done by our prototype. In particular, the picture on the left shows the equilibrium air density on a scale from blue (not dense) to red (dense); and the picture of the right shows the equilibrium air speed, with arrows to show direction, and colour to show speed (blue being slow and red being quick).

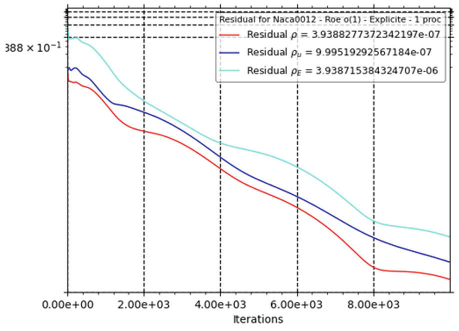


Fig. 9. Convergence of our prototype CPU/GPU/VE (Color figure online)

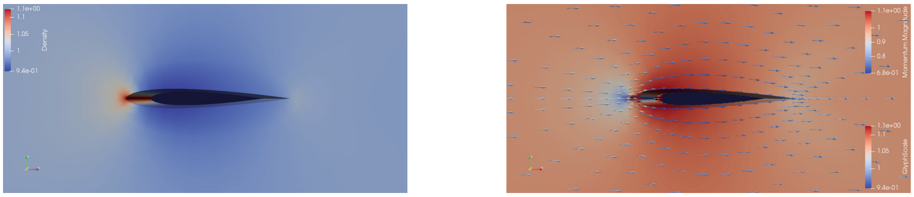


Fig. 10. Density on Vertex and Momentum on Vertex (Color figure online)

8 Conclusion and Future Work

This work presents an study into the requirements of CFD tools, some limitations of the current tools available, elsA in particular. It then provided with an approach to solve some of these limitations. Similarly to several existing tools, this approach structures a CFD tool in three parts that distinguishes between: *i*) the operators that implement all the core mathematical function used in any computation; *ii*) the orchestrator that assemble the operators into a complete dataflow that computes the required data; and *iii*) the HPC layer which manages the distribution and concurrency during the computation of the dataflow. The novelty of our approach lies in the definition of the orchestrator part, which is based on tools originating from formal methods: *terms* and *order-sorted Algebra* are used to specify the inputs and outputs of the available operators, and pattern matching (a subcase of unification) is then used to identify dependencies between operators and generate the dataflow; *Delta-Oriented Programming* is used to model the variability of the available operators, i.e., depending on the tool configuration, the inputs and outputs of the operators may change, which means that the dependencies between operators and thus the generated dataflow may change. Based on this approach, we implemented a prototype and tested it.

Going from a prototype to a useable tool still requires a lot of work. We need to integrate hardware distribution, and in particular integrate the possibility to manage heterogeneous hardware. A promising approach would be to use the standard API of `hwloc` [7,20] that gives the topology of the hardware, with the characteristics of the different memory and processing nodes.

Another issue to tackle is memory allocation. Indeed, array of the right size must be allocated to host the data computed by the operators in the dataflow generated by our orchestrator. Because the dataflow can vary, so does the memory allocation. However, memory is allocated by hand in our prototype, and known approaches for memory allocation are not satisfactory: in pyFR and Devito, the memory is managed by the user directly; on the other hand, elsA does not have a general framework to model data and relies to enumerations to list all the possible data it can handle.

Another interesting aspect of this work is the similarities of the problem of generating the graph of operators and the problem of type inhabitance: as hinted in Sect. 4, the term modelling the data to compute is similar to a type, and from that point of view our generated graph is an expression that have that type.

We will investigate this relationship further and possibly see if interesting result could be applied to our prototype. Moreover, this approach seems to integrate well with product lines. Indeed classic approach for product line definition is to add, remove or replace well identified code elements, but it is very difficult to have an expression always computing the same data in all variants, using however different methods to obtain it depending on the selected options.

References

1. Agosta, G., Fornaciari, W., Massari, G., Pupykina, A., Reghenzani, F., Zanella, M.: Managing heterogeneous resources in HPC systems. In: Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM 2018, pp. 7–12. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3183767.3183769
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY. Lecture Notes in Computer Science, vol. 12345. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-64354-6
3. Apel, S., Kästner, C., Lengauer, C.: FEATUREHOUSE: language-independent, automated software composition. In: Proceedings of 31st International Conference on Software Engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, pp. 221–231. IEEE (2009). https://doi.org/10.1109/ICSE.2009.5070523
4. Aupoix, B., Spalart, P.: Extensions of the Spalart-Allmaras turbulence model to account for wall roughness. Int. J. Heat Fluid Flow **24**(4), 454–462 (2003). https://doi.org/10.1016/S0142-727X(03)00043-2. Selected Papers from the Fifth International Conference on Engineering Turbulence Modelling and Measurements
5. Biedron, R.T., et al.: FUN3D manual: 13.7. National Aeronautics and Space Administration, Langley Research Center (2020)
6. Bourgeois, K., Robert, S., Limet, S., Essayan, V.: GeoSkelSL: a Python high-level DSL for parallel computing in geosciences. In: Shi, Y., et al. (eds.) ICCS 2018. LNCS, vol. 10862, pp. 839–845. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93713-7_83
7. Broquedis, F., et al.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. IEEE, Pisa, February 2010. https://doi.org/10.1109/PDP.2010.67
8. Cambier, L., Heib, S., Plot, S.: The Onera elsA CFD software: input from research and feedback from industry. Mech. Ind. **14**(3), 159–174 (2013). https://doi.org/10.1051/meca/2013056
9. Ciżnicki, M., Kurowski, K., eglarz, J.W.: Energy and performance improvements in stencil computations on multi-node HPC systems with different network and communication topologies. Future Gener. Comput. Syst. **115**, 45–58 (2021). https://doi.org/10.1016/j.future.2020.08.018
10. Clarke, D., et al.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_13

11. Clavel, M., et al.: All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
12. Constantin, P., Foias, C.: Navier-Stokes Equations. University of Chicago Press (1988)
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28630-1_17
14. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M.: A unified and formal programming model for deltas and traits. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 424–441. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_25
15. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., Paolini, L.: Variability modules for java-like languages. In: Mousavi, M., Schobbens, P. (eds.) SPLC 2021: 25th ACM International Systems and Software Product Line Conference, Leicester, UK, 6–11 September 2021, vol. A, pp. 1–12. ACM (2021). https://doi.org/10.1145/3461001.3471143
16. Dick, A.J.J., Watson, P.: Order-sorted term rewriting. Comput. J. **34**(1), 16–19 (1991). https://doi.org/10.1093/comjnl/34.1.16
17. Fay, M.: First-order unification in an equational theory. Technical report 78-5-002, University of California at Santa Cruz (1978)
18. Flich, J., et al.: Exploring manycore architectures for next-generation HPC systems through the mango approach. Microprocess. Microsyst. **61**, 154–170 (2018). https://doi.org/10.1016/j.micpro.2018.05.011
19. Focht, E.: VEO and PyVEO: vector engine offloading for the NEC SX-Aurora tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., Focht, E., Kobayashi, H. (eds.) Sustained Simulation Performance 2018 and 2019, pp. 95–109. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39181-2_9
20. Goglin, B.: Towards the structural modeling of the topology of next-generation heterogeneous cluster nodes with hwloc. Research report, Inria, November 2016. https://hal.inria.fr/hal-01400264
21. Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., Winkler, T.: An introduction to OBJ 3. In: Kaplan, S., Jouannaud, J.-P. (eds.) CTRS 1987. LNCS, vol. 308, pp. 258–263. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-19242-5_22
22. Hähnle, R.: HATS: highly adaptable and trustworthy software using formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 3–8. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_2
23. Hähnle, R.: Task forces in the EternalS coordination action. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 20–22. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_6
24. Hähnle, R.: The abstract behavioral specification language: a tutorial introduction. In: Giachino, E., Hähnle, R., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2012. LNCS, vol. 7866, pp. 1–37. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40615-7_1
25. Hähnle, R., Johnsen, E.B.: Designing resource-aware cloud applications. Computer **48**(6), 72–75 (2015). https://doi.org/10.1109/MC.2015.172
26. Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_4

27. Han, Z., Devarajegowda, K., Werner, M., Ecker, W.: Towards a python-based one language ecosystem for embedded systems automation. In: 2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), pp. 1–7 (2019). https://doi.org/10.1109/NORCHIP.2019.8906949
28. He, P., Mader, C.A., Martins, J.R.R.A., Maki, K.J.: DAFoam: an open-source adjoint framework for multidisciplinary design optimization with OpenFOAM. AIAA J. **58**(3), 1304–1319 (2020). https://doi.org/10.2514/1.J058853
29. Hink, R., Hannemann, V., Eggers, T.: Extension of the Spalart-Allmaras one-equation turbulence model for effusion cooling problems. In: Deutscher Luft - und Raumfahrtkongress 2013, September 2013. https://elib.dlr.de/84638/
30. Hoefler, T., Schneider, T.: Optimization principles for collective neighborhood communications. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–10. IEEE (2012). https://doi.org/10.1109/SC.2012.86
31. Jasak, H.: OpenFOAM: open source CFD in research and industry. Int. J. Naval Archit. Ocean Eng. **1**(2), 89–94 (2009). https://doi.org/10.2478/IJNAOE-2013-0011
32. Jung, Y.S., Baeder, J.: $\gamma - \overline{re_{\theta t}}$ Spalart–Allmaras with crossflow transition model using Hamiltonian–strand approach. J. Aircr. **56**(3), 1040–1055 (2019). https://doi.org/10.2514/1.C035149
33. Kamburjan, E., Hähnle, R.: Deductive verification of railway operations. In: Fantechi, A., Lecomte, T., Romanovsky, A.B. (eds.) RSSRail 2017. LNCS, vol. 10598, pp. 131–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68499-4_9
34. Kenway, G.K., Mader, C.A., He, P., Martins, J.R.: Effective adjoint approaches for computational fluid dynamics. Prog. Aerosp. Sci. **110**, 100542 (2019). https://doi.org/10.1016/j.paerosci.2019.05.002
35. Khaleghzadeh, H., Manumachu, R.R., Lastovetsky, A.: A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms. IEEE Trans. Parallel Distrib. Syst. **29**(10), 2176–2190 (2018). https://doi.org/10.1109/TPDS.2018.2827055
36. Knopp, T., Eisfeld, B., Calvo, J.B.: A new extension for k-ω turbulence models to account for wall roughness. Int. J. Heat Fluid Flow **30**(1), 54–65 (2009). https://doi.org/10.1016/j.ijheatfluidflow.2008.09.009
37. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: DeltaJ 1.5: delta-oriented programming for Java. In: International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ 2014, pp. 63–74 (2014). https://doi.org/10.1145/2647508.2647512
38. Ladyzhenskaya, O.A.: Sixth problem of the millennium: Navier-stokes equations, existence and smoothness. Russ. Math. Surv. **58**(2), 251–286 (2003). https://doi.org/10.1070/rm2003v058n02abeh000610
39. Leicht, T., Jägersküpper, J., Vollmer, D., Schwöppe, A., Hartmann, R., Fiedler, J., Schlauch, T.: DLR-project digital-X - next generation CFD solver 'flucs'. In: Deutscher Luft-und Raumfahrtkongress 2016, February 2016. https://elib.dlr.de/111205/
40. Louboutin, M., et al.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geosci. Model Dev. **12**(3), 1165–1187 (2019). https://doi.org/10.5194/gmd-12-1165-2019

41. Mader, C.A., Kenway, G.K.W., Yildirim, A., Martins, J.R.R.A.: ADflow–an open-source computational fluid dynamics solver for aerodynamic and multidisciplinary optimization. J. Aerosp. Inf. Syst. (2020). https://doi.org/10.2514/1.I010796
42. McAlister, K.W., Carr, L.W., McCroskey, W.J.: Dynamic stall experiments on the NACA 0012 airfoil. Technical report, NASA (1978)
43. McCroskey, W.: A critical assessment of wind tunnel results for the NACA 0012 airfoil. Technical report, National Aeronautics And Space Administration Moffett Field Ca Ames ... (1987)
44. Meseguer, J., Goguen, J.A., Smolka, G.: Order-sorted unification. J. Symb. Comput. **8**(4), 383–413 (1989). https://doi.org/10.1016/S0747-7171(89)80036-7
45. Metcalf, M., Reid, J.K.: Fortran 90/95 Explained. Oxford University Press, Inc. (1999)
46. Mofrad, M.H., Melhem, R., Ahmad, Y., Hammoud, M.: Graphite: a NUMA-aware HPC system for graph analytics based on a new MPI * X parallelism model. Proc. VLDB Endow. 13(6), 783–797 (2020). https://doi.org/10.14778/3380750.3380751
47. Nielsen, F.: Introduction to MPI: the message passing interface. In: Nielsen, F. (ed.) Introduction to HPC with MPI for Data Science. Undergraduate Topics in Computer Science, pp. 21–62. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-21903-5_2
48. NRC: Trace v5.0 theory manual - field equations, solution methods, and physical models. Technical report, United States Nuclear Regulartory Commission (2012)
49. Palacios, F., et al.: Stanford university unstructured (SU^2): An open-source integrated computational environment for multi-physics simulation and design. In: 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition 2013, January 2013. https://doi.org/10.2514/6.2013-287
50. Perraud, J., Durant, A.: Stability-based mach zero to four longitudinal transition prediction criterion. J. Spacecr. Rock. **53**(4), 730–742 (2016). https://doi.org/10.2514/1.A33475
51. Poinot, M., Rumsey, C.L.: Seven keys for practical understanding and use of CGNS. American Institute of Aeronautics and Astronautics (2018). https://doi.org/10.2514/6.2018-1503
52. Poirier, D., Allmaras, S., McCarthy, D., Smith, M., Enomoto, F.: The CGNS system. American Institute of Aeronautics and Astronautics (1998). https://doi.org/10.2514/6.1998-3007
53. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_6
54. Seidl, C., Schaefer, I., Aßmann, U.: DeltaEcore - a model-based delta language generation framework. In: Fill, H., Karagiannis, D., Reimer, U. (eds.) Modellierung 2014, 19–21 March 2014, Wien, Österreich. LNI, vol. P-225, pp. 81–96. GI (2014). https://dl.gi.de/20.500.12116/17067
55. Stroustrup, B.: A Tour of C++. Addison-Wesley Professional (2018)
56. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: an extensible framework for feature-oriented software development. Sci. Comput. Program. **79**, 70–85 (2014). https://doi.org/10.1016/j.scico.2012.06.002. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010)

57. Witherden, F., Farrington, A., Vincent, P.: PyFR: an open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. Comput. Phys. Commun. **185**(11), 3028–3040 (2014). https://doi.org/10.1016/j.cpc.2014.07.011
58. Young, V., Jaleel, A., Bolotin, E., Ebrahimi, E., Nellans, D., Villa, O.: Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 339–351 (2018). https://doi.org/10.1109/MICRO.2018.00035
59. Zhang, N., Driscoll, M., Markley, C., Williams, S., Basu, P., Fox, A.: Snowflake: a lightweight portable stencil DSL. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 795–804 (2017). https://doi.org/10.1109/IPDPSW.2017.89

Reasoning About Active Objects: A Sound and Complete Assertional Proof Method

Frank de Boer[1,2] and Stijn de Gouw[2,3](✉)

[1] Leiden Institute for Advanced Computer Science, Leiden, The Netherlands
f.s.de.boer@cwi.nl
[2] Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
stijn.degouw@ou.nl
[3] Open University, Heerlen, The Netherlands

Abstract. We present a novel assertional proof method for reasoning about global invariant properties of active objects in the context of the Abstract Behavioral Specification (ABS) language. The main result of this paper is a formal justification of the proof method which establishes both its soundness and completeness with respect to a formally defined operational trace semantics.

1 Introduction

Active objects provide a powerful conceptual model of distributed systems (see [6] for a survey of active object languages). Active objects support a "programming to interfaces" discipline by a strict encapsulation of their local state and communication via asynchronous method calls. A fundamental problem in reasoning about active objects is the decoupling of the sender and receiver in asynchronous method calls. This decoupling gives rise to a complex sequence of events consisting of the call, reception of the message, execution of the method, and, finally, asynchronous communication of the return value.

In this paper we generalize *Floyd's inductive assertions method* to the verification of global *user-defined* invariant properties of concurrent systems consisting of active objects, as described by the Abstract Behavioral Specification (ABS) language [18]. We focus on active (or concurrent) objects which only interact via asynchronous method calls and futures for returning values. The methods are executed by an object one at a time in a co-routine manner.

The validation of a user-defined global invariant requires the use of *program annotations* which associate with each class of the given program a *class invariant* and with each occurrence of a substatement of a method body a precondition[1]. Class invariants specify the *scheduling points* which in ABS are described

[1] Since we assume that every method body terminates with a return statement, the postcondition of a statement coincides with the precondition of the statement that follows, e.g., the postcondition of the method body itself is the precondition of the return statement.

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 173–192, 2022.
https://doi.org/10.1007/978-3-031-08166-8_9

by explicit statements for suspending the execution of a method. The validation of an annotated program consists of the validation of the local preconditions by means of corresponding verification conditions. The validation of the precondition of a method call, method body, and a suspension statement involves the global invariant, which provides information about the actual parameters and the return values, and the class invariant, which provides information about the local state. The following main question arises in the validation of the precondition of a method. *When exactly is the precondition of a method supposed to hold?*

There are two options: on call site, or when the method is selected for execution. The first option introduces the need for an interference freedom test to ensure that the precondition still holds when the method is executed. Such a test in general greatly increases the number of verification conditions. Therefore we have investigated in this paper the second option, using global invariants to describe user-defined properties of *histories* (also called *traces*) which record the interactions via method calls and futures between active objects.

The main result of this paper is a formal justification of the proof method which establishes both its soundness and (relative) completeness with respect to a formally defined operational trace semantics.

Related Work. A first proof system for asynchronous method calls and futures has been published in [8]. It is based on a local *interference freedom test*, as initially proposed for reasoning about shared-variable concurrency in [22]. This test ensures that the preconditions of await statements are not invalidated by the execution of other method invocations. In general, such tests lead to a combinatorial explosion of the number of verification conditions. In this paper we showed how to avoid such a test, and obtain a compositional method for the verification of the individual methods of a class. Further, the global invariant in [8] is used to reason about futures in terms of shared variables, exploiting that futures are assigned only once. Instead, the proof system of this paper is based on the compositional semantics described in [9], where both method calls and returning values via futures is recorded by a global communication history. The absence of an interference freedom test and the uniform explicit treatment of both communication mechanisms in the proof system allows to reduce the verification of ABS programs to the sequential verification of the individual method bodies. In this paper we further exploit the underlying compositional semantics in a novel approach to proving completeness.

Ahrendt et al. introduce a formal semantics and proof system for Creol in [2,3]. Creol is a predecessor to the ABS language used in this paper and, like ABS, Creol supports aysnchronously communicating concurrent objects. Their proof system is based on complex histories which record seven kinds of events and proof obligations with complex well-formed conditions on histories. Completeness of the proof system is not considered. Zaharieva-Stojanovski [23] contribute an approach to multi-threaded verification based on reasoning about class invariants and ownership. Blom et al. [5] presents a verification technique based on permission-based separation logic and involves several additional verification conditions concerning well-formedness/consistency. Kamburjan and Chen [19] present a top-down static analysis method for a similar langauge, but without cooperative scheduling and for a restricted class of properties (in particular, session types to specify protocols). Completeness is not considered. Din et al. [14–16] present a semantics and a proof system for reasoning about asynchronously communicating objects and futures. However, completeness is only established in [15] for active objects which execute their methods in a run-to-completion mode, i.e., without cooperative scheduling. Their general approach is based on the abstraction from the behavior of the environment by the specification of the local behavior and state of an individual object in terms of the *non-deterministic* selection of its methods, as recorded by its local history. Global properties are only supported indirectly in a bottom-up manner, by taking the conjunction of all local class invariants together with complex well-formedness conditions which ensure compatibility of the local histories. Consequently, only when composing the local histories, the local state properties of the objects can be inferred. In contrast, in our novel approach user-defined global properties ensure in a top-down manner compatibility of the local histories and allow to infer local state properties in the verification of a single class instance. Further, in [20], it is argued that a bottom-up approach limits severely the kind of global properties one can prove.

We prove completeness relative to all valid assertions of the underlying logic, as introduced in [7]. The completeness proof presented in this paper generalizes that of the seminal proof system for Communicating Sequential Processes (CSP), as described in [4], to active objects. It boils down to showing that a program annotated with so-called *reachability predicates* satisfies the *verification conditions*. These reachability predicates are semantically defined in terms of an operational *trace semantics*.

In our completeness proof, we distinguish two kinds of reachability predicates: one, the global invariant, which expresses that a given global history can be generated by a global computation, and one which expresses that a given local state of an active object can be reached by a global computation. As a particular case of the latter, we have class invariants which express that a given local state of an active object can be reached as a *scheduling* point in a global computation. The crucial semantic property in the completeness proof for the validation of the verification conditions is that *local computations of an object are globally indistinguishable when they give rise to the same local trace (or history).* In [9] we showed that for a compositional semantics of active objects we only need to record for each object its outgoing calls and the future completions generated by its own method invocations. In order to obtain by *projection* for each object its *local view of the global history*, which is not affected by the behavior of other objects (e.g., the calls of other objects), we also record in this paper the initial scheduling for execution of the invoked methods of an object. This concept of a global history allows for a clear separation of concerns between the *external* behavior of an object and its *internal co-routine* execution of its method invocations, and as such allows for an elegant generalization of the completeness proof for CSP [4].

Plan of the Paper. In the following section we introduce the basic modeling concepts of ABS and the proof methodology. We introduce the proof method itself incrementally in terms of the following sublanguages. Section 3 describes the proof method and its formal justification for the sublanguage of ABS which only features asynchronous calls to methods which are executed in a run-to-completion mode without return values. Section 4 then extends this basic language with a statement for suspending a method execution, and resuming it when the associated Boolean condition holds. Finally, Sect. 5 introduces futures. Section 6 some possible further research directions.

2 Preliminaries

We focus in this paper on the main OO-imperative *control structures* of ABS [10]: A program specifies a number of classes and each class defines a number of methods. The method bodies always end with a return statement and are constructed by the usual sequential control structures from basic assignments (including object creation), asynchronous method calls, and suspension statements (on Boolean guards and futures). Actors in this (Turing-complete) language are (concurrently) executing objects that only interact by *asynchronous method calls*, i.e., ABS actors do not share state. Each asynchronous method call immediately returns a *future*, which contains a unique reference to its return value (methods are always specified by a return type, the return type `Unit` is used to specify that no value is returned). The caller continues executing and the call is stored in the FIFO queue of the callee. After the method is scheduled for execution and terminated, the return value is computed and stored in the future uniquely associated with the method invocation. Futures support two operations. First, the return value of a future f can be retrieved by `f.get`, which blocks execution in the actor until the future is resolved (i.e., until the called method has computed the return value). Second, a future f can be polled whether it has been resolved by the statement `await f?`, which suspends the process, if the future has not been resolved (i.e. the condition is false), and resumes another suspended process (if any) or starts executing the next call from its FIFO queue (if any). This is a form of cooperative scheduling where the scheduling points are statically identified in the code by the `await`-statements. A Boolean `await`-statement checks a local Boolean condition, instead of a future.

We next introduce the following basic semantic notions. By V, with typical element v, we denote the set of all possible values (abstracting from typing information). By $O \subseteq V$ we denote the (infinite) set of object identities, with typical element o. For a given program P we assume a set O_C of instances of class C such that the sets O_C, for each class C of P, form a partitioning of O. The state-space $\Sigma(C)$, with typical element σ, of a class C assigns values (of the correct type) to its instance variables including the distinguished variable `this` which denotes the object id. The state-space $\Sigma(C.m)$ associated with a method m of a class C consists of pairs (σ, τ), where $\sigma \in \Sigma(C)$ and τ assigns values to its local variables (including its formal parameters). For notational convenience, when it is clear from the context to which class C the method m belongs, we omit C and write $\Sigma(m)$. As a particular case we denote by $\Sigma(main)$ the state-space of the main statement of a program which assigns value to its variables. By $\mathsf{Val}(e)(\sigma, \tau)$ we denote the value of the (side-effect free) expression e of the programming language (w.r.t. the variable assignments σ and τ). By Ω, with typical element ω, we denote the set of global environments which assign values to the global variables (which are assumed not to appear in a program). Updates of an assignment a of values to variables we denote $a[x := v]$.

By Θ we denote the set of *histories* of a system of concurrent objects, with typical element θ. In general, these histories record asynchronous method calls, method selections and future completions. The distinguished global variable h denotes the global history. In the following sections we define histories formally. For each object o, its local history is denoted by θ_o (defined in the following section).

As it has become common [13,17], we use the extensional approach (a shallow embedding) for our specification language: we do not fix a specific syntax for assertions, they are merely defined extensionally as functions from states to truth values. We denote assertions by $p, q \ldots$. We distinguish the following classes of assertions. For a method m of a class C, we denote by $\mathsf{Pred}(m)$ the (local) assertions of m. The truth of such a predicate p is defined with respect to a triple σ, τ, and ω, and denoted by $\sigma, \tau, \omega \models p$. Local predicates only provide a local view on the global history h: $\sigma, \tau, \omega \models p$ iff $\sigma, \tau, \omega' \models p$, whenever $\omega(\mathsf{h})_o = \omega'(\mathsf{h})_o$ and $\sigma(\mathsf{this}) = o$. As a particular case, $\mathsf{Pred}(main)$ denotes the set of assertions of the main statement, which also only have a local view (of the root object) on the global history h. By $\mathsf{Pred}(C)$ we denote the set of assertions of class C which do not refer to local variables. The truth of such a predicate p is defined with respect to a pair σ and ω, and denoted by $\sigma, \omega \models p$. As above, such predicates only provide a local view on the global history h: $\sigma, \omega \models p$ iff $\sigma, \omega' \models p$, whenever $\omega(\mathsf{h})_o = \omega'(\mathsf{h})_o$ and $\sigma(\mathsf{this}) = o$. Finally, by $\mathsf{Pred}(P)$ we denote the set of global assertions of a program P which only refer to the global variables. The truth of such a predicate p is defined with respect to a global environment which is denoted by $\omega \models p$. For notational convenience, we extend the truth definition of *any* predicate p to triples σ, τ, and ω, e.g., $\sigma, \tau, \omega \models p$ iff $\sigma, \omega \models p$, for $p \in \mathsf{Pred}(C)$. An assertion p is valid, denoted by $\models p$, iff $\sigma, \tau, \omega \models p$, for all σ, ω, and τ.

An *annotation* $A(P)$ of a program P consists of a global invariant $A(I) \in \mathsf{Pred}(P)$, and associates with each class C a class invariant $A(C) \in \mathsf{Pred}(C)$, and with each occurrence of a substatement S of a method m or the main statement it associates a precondition $A(S)$ in $\mathsf{Pred}(m)$ or $\mathsf{Pred}(main)$. We denote by $\models A(P)$ that the annotation $A(P)$ is valid, which will be defined formally in the following section. For two annotations $A(P)$ and $A'(P)$ of a program P, we denote by $\models A(P) \rightarrow A'(P)$ that $\models A(S) \rightarrow A'(S)$ for any each occurrence of a substatement S of a method m or the main statement (in P).

In the following sections we focus on the *verification conditions* for the basic statements which involve asynchronous method calls, object creation, returning a value, and await statements (the verification conditions for composite statements like choice, sequential composition, etc., are standard). To reason about updates of instance and local variables, and the global history h, we assume substitutions $p[e/x]$ which denote the result of replacing every occurrence of the variable x by the side-effect free expression e. As a special case, the substitution $p[\mathsf{new}/x]$ describes the effect of assigning to the variable x a new future or object id (its definition depends on the actual syntax used, see for example [12]). We assume that these substitutions satisfy the basic semantic property that if $p[e/x]$ (respectively $p[\mathsf{new}/x]$) holds then p holds *after* the assignment $x := e$ ($x := \mathsf{new}$). We denote by $\models_{\mathrm{vc}} A(P)$ that all the verification conditions of $A(P)$ hold. By $\vdash A(P)$ we denote that $\models_{\mathrm{vc}} A'(P)$, for some annotation $A'(P)$ of P such that $\models A'(P) \rightarrow A(P)$. We have the standard notions of soundness, i.e., $\vdash A(P)$ implies $\models A(P)$, and completeness, i.e., $\models A(P)$ implies $\vdash A(P)$.

3 Reasoning About Asynchronous Calls

In this section we focus on pure asynchronous method calls in the context of a basic run-to-completion mode of method execution, i.e., no cooperative scheduling or futures.

Semantics. A basic history for pure asynchronous method calls is a (finite) sequence θ of asynchronous method calls (including calls to constructor methods) and method selections. Semantically an asynchronous method call is of the form $o \mapsto o'.m(\tau)$, where o denotes the caller, o' denotes the callee, m denotes the (constructor) method called, and τ denotes the local environment which assigns the actual parameters to the formal parameters of m.

A method selection $o.m(\tau)$ indicates that the object o has selected a call to the method m for execution in the local environment τ. Let $\theta!o$ denote the subsequence of θ of calls to o, and $\theta?o$ denote the subsequence of θ of the method selections of o. We restrict to histories θ such that for every object o, each call appears before the corresponding selection. A call in $\theta!o$ is *pending* if there does not exist a corresponding method selection in $\theta?o$.

The local history of an object o is obtained by the projection θ_o from a global history θ:

- $(o \mapsto o'.m(\tau) \cdot \theta)_o = o'.m(\tau) \cdot \theta_o$
- $(o' \mapsto o''.m(\tau) \cdot \theta)_o = \theta_o$, if $o \neq o'$
- $(o.m(\tau) \cdot \theta)_o = m(\tau) \cdot \theta_o$
- $(o'.m(\tau) \cdot \theta)_o = \theta_o$, if $o \neq o'$

This projection thus records all the method calls generated by the object as caller and all its selected method invocations. It thus abstracts from incoming calls (generated by other objects).

A *global configuration* (of a given program) consists of a set Γ of object configurations and a global history θ. An object configuration is a tuple of the form $\langle \sigma, p \rangle$, where σ assigns values to the instance variables of the object $\sigma(\mathsf{this})$, p either denotes the active process (τ, S) which executes statement S with respect to the local environment τ or p denotes the empty process nil which indicates a scheduling point[2]. We assume that for each object o there is at most one object configuration $(\sigma, (\tau, S)) \in \Gamma$ such that $o = \sigma(\mathsf{this})$. An initial global configuration consists of a single object configuration $(\sigma, (\tau, main))$, where *main* denotes the main statement. The initial (or root) object is denoted by $\sigma(\mathsf{this})$. However, we assume that "this" does not appear in the main statement (it will be used to identify in the history the communications of the root object).

The semantics is formally described by transitions $(\Gamma, \theta) \rightarrow (\Gamma', \theta')$. In this semantics both asynchronous method calls and their initial selection for execution by the callee are recorded by the global history, as described in Table 1. We will focus on the concurrency and cooperative scheduling aspects of the language and omit transition rules for constructs such as loops, conditional statements, etc. as these are standard and can be adapted to our setting in a straightforward manner. Here and in the sequel, $\mathsf{Val}(e_0!m(\bar{e}))(\sigma, \tau)$ denotes the call $\sigma(\mathsf{this}) \mapsto o.m(\tau')$, where $o = \mathsf{Val}(e_0)(\sigma, \tau)$ and τ' is the local environment which assigns the values of the actual parameters $\bar{e}$ (with respect to σ and τ) to the formal parameters of m. Further, $\mathsf{Sched}(\theta, o)$ denotes the method selection $o.m(\tau)$ which corresponds to the *first* pending call $o' \mapsto o.m(\tau)$ in $\theta!o$.

Table 1. Semantics of pure asynchronous method calls and method selections

$(\Gamma \cup \{(\sigma, (\tau, e_o!m(\bar{e}); S'))\}, \theta) \rightarrow (\Gamma \cup \{(\sigma, (\tau, S'))\}, \theta \cdot \alpha))$ where $\alpha = \mathsf{Val}(e_0!m(\bar{e}))(\sigma, \tau))$
$(\Gamma \cup \{(\sigma, \mathsf{nil})\}, \theta) \rightarrow (\Gamma \cup \{(\sigma, (\tau, S))\}, \theta \cdot \alpha)$ where $\alpha = \sigma(\mathsf{this}).m(\tau) = \mathsf{Sched}(\theta, \sigma(\mathsf{this}))$and $S = Body(m)$

The above transition relation then allows the definition of a "big-step" semantics $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ which indicates the reachability of the global configuration (Γ, θ) from the initial configuration which consists of the empty history ϵ and (σ_0, τ_0) as the initial state of the root object.

We have the following basic property of the semantics which states that two local configurations of an object are globally indistinguishable when generated by the same local history. For notational convenience, we denote by $\Gamma(o)$ the object configuration of the object o in Γ and by $\Gamma[o := \gamma]$ the result of updating the configuration of o in Γ by the object configuration γ.

Lemma 1 (Substitutivity). *For any computations $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ and $(\sigma_0, \tau_0) \Rightarrow (\Gamma', \theta')$ such that $\theta_o = \theta'_o$ there exists a computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma[o := \Gamma'(o)], \theta)$.*

This lemma follows in a straightforward manner from the determination of the local behavior of an object by its local history (the sequential control flow structures used for describing the behavior of methods being deterministic).

[2] Note that p is also used to denote assertions. From the context however it is clear what is meant.

Given the above semantics we next define the semantics of annotated programs.

Definition 1 (Semantics annotations). *A set Γ of object configurations satisfies an annotation $A(P)$ with respect to a global environment ω, denoted by $\Gamma, \omega \models A(P)$, if*

- $\omega \models A(I)$,
- $\sigma, \omega \models A(C)$, *for all* $(\sigma, (\tau, \mathsf{nil})) \in \Gamma$, *where* $\sigma(\mathsf{this})$ *is an instance of class* C,
- $\sigma, \tau, \omega \models A(S)$, *for all* $(\sigma, (\tau, S)) \in \Gamma$.

We can now define formally $\models A(P)$*: for every computation* $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ *and global environment* ω *such that* $\sigma_0, \tau_0, \omega[\mathsf{h} := \epsilon] \models A(main)$ *we have* $\Gamma, \omega[\mathsf{h} := \theta] \models A(P)$.

Verification Conditions

The following verification conditions (VC's) formalize the validation of asynchronous method calls/returns and method selections.

Asynchronous Method Call. For a statement $S \equiv e_o!m(\bar{e}); S'$ we have the VC:

$$\models (A(I) \wedge A(S)) \rightarrow (A(I) \wedge A(S'))[\mathsf{h} \cdot e_0!m(\bar{e})/\mathsf{h}]$$

Given that the global invariant (denoted by $A(I)$) and the precondition of $e_o!m(\bar{e})$ hold, this VC guarantees that the invariant and the post-condition of $e_o!m(\bar{e})$ hold after updating the global history with the call $e_o!m(\bar{e})$.

Object Creation. For a statement $S \equiv x := \mathsf{new}\ C(\bar{e}); S'$ we have the VC:

$$\models (A(I) \wedge A(S)) \rightarrow (A(I) \wedge A(S'))[\mathsf{h} \cdot x!C(\bar{e})/\mathsf{h}][\mathsf{new}/x]$$

This VC disentangles the statement $x := \mathsf{new}\ C(\bar{e})$ into the object creation $x := \mathsf{new}$ followed by the asynchronous call $x!C(\bar{e})$ in which the newly created object appears as the callee of the (call to) the constructor method. Note that as such the constructor method can also be used as the so-called run method, which defines the behavior of an active object.

Method Selection. Let $\mathsf{Val}(m(\bar{e}))(\sigma, \tau)$ denote the method selection $\sigma(\mathsf{this}).m(\tau')$, where τ' is the local environment which assigns the values of the actual parameters $\bar{e}$ (with respect to σ and τ) to the formal parameters of m.

For a method m of a class C (excluding the constructor method) with body S we have the VC:

$$\models (A(I) \wedge A(C) \wedge m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \rightarrow (A(S) \wedge A(I))[\mathsf{h} \cdot m(\bar{u})/\mathsf{h}]$$

where $\bar{u}$ are the formal parameters of m. Here, $\sigma, \tau, \omega \models m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})$ iff $\mathsf{Val}(m(\bar{u}))(\sigma, \tau) = o.m(\tau) = \mathsf{Sched}(\theta, o)$, where $\theta = \omega(\mathsf{h})$ and $o = \sigma(\mathsf{this})$. Note that $m(\bar{u})$ thus corresponds to the first pending call to a method of o in $\omega(\mathsf{h})$ (as defined above), and that this VC thus abstracts from the actual call site and guarantees that the precondition $A(S)$ and the global invariant $A(I)$ hold when the method is scheduled.

Selection of the constructor method requires the following adaptation:

$$\models (A(I) \wedge \mathit{Init}(C) \wedge C(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \rightarrow (A(S) \wedge A(I))[\mathsf{h} \cdot C(\bar{u})/\mathsf{h}]$$

where $\mathit{Init}(C) \in \mathsf{Pred}(C)$ describes the initial values of the instance variables of class C and it additionally states that the local history of the object this as recorded by the global history h is empty.

Return. For a return statement $S \equiv \mathsf{return}$ (of a method in class C) we have the following VC:

$$\models (A(I) \wedge A(S)) \rightarrow A(C)$$

Given that the global invariant (denoted by $A(I)$) and the precondition of the return statement hold, this VC establishes the class invariant $A(C)$ after termination of the method body.

Initialization. Finally, the following VC initially establishes the global invariant.

$$\models (A(\mathit{main}) \wedge \mathsf{h} = \epsilon) \rightarrow A(I)$$

As explained in the introduction the main contribution of this paper is a novel sound and complete proof theory for the ABS language. But to provide a general idea of how to prove correctness of ABS programs by means of user-defined global invariants, we discuss below the toy example in Listing 1.1. The class Toy defines two methods m1 and m2. The number of method selections is recorded by the instance variables c1 and c2, respectively. The main statement creates an instance of class Toy and invokes the m1 method.

Listing 1.1. Class Toy in ABS

```
class Toy {
    Int c1, c2 ;
    Toy() { c1=0 ; c2=0 ; return}
    Unit m1() { c1 = c1+1; return}
    Unit m2() { c2 = c2+1; return}
 }

Main {
    toy = new Toy();
    toy!m1()
  }
```

Let $\#z.$m1 and $\#z.$m2 denote the number of *calls* of the methods m1 and m2 of the object (the callee) denoted by the variable z, as recorded by the global history h, and #m1$\downarrow$ z and #m2$\downarrow$ z denote the number of *selections* of the methods m1 and m2 by the object denoted by the variable z (again, as recorded by the global history h). We will show how to validate the global invariant $A(I)$

$$A(I) \equiv \forall z : \mathrm{Toy}(\#m1 \downarrow z \leq \#z.m1 \leq 1 \wedge \#m2 \downarrow z = \#z.m2 = 0)$$

where the universal quantification ranges over all created instances of class Toy (by a call of its constructor method), as recorded by the global history. We thus abstract from the uniqueness of the created instance of class Toy. Note that for any program and method we have that for any object the number of times it has selected the method is smaller or equal to the number of calls of the method of this object. This general (global) invariant is in fact provable in our proof method, as we show in this particular case.

To prove the above global invariant, we introduce a local class invariant

$$A(\text{Toy}) \equiv \#m1 \downarrow \mathsf{this} = c1 \leq 1 \wedge \#m2 \downarrow \mathsf{this} = c2 = 0$$

which states, among others, that c1 and c2 equal the number of selections of the method m1 and m2, respectively, of the object this. Note that both #m1↓ this and #m2↓ this, as defined above, implicitly refer to the global history h. However, since the method selections of an object as recorded by the global history correspond with those recorded by its local history, $A(\text{Toy})$ is in fact a local assertion (as defined in Sect. 2).

It is worthwhile to observe here that, in the standard bottom-up approach as used in [15], class invariants have to take into account *arbitrary* environments, which results in the following *weakening*

$$\#m1{\downarrow}\ \mathsf{this} = c1 \wedge \#m2{\downarrow}\ \mathsf{this} = c2$$

of the invariant of class Toy. Specific information about the values of c1 and c2 then can only be derived when combining the local invariants (in our case, combining the local invariant of class Toy with the local invariant of the main statement). In general, this complicates reasoning both at the local and global level.

Let S denote the body of the constructor method (of class Toy) and $A(S)$ denote its precondition #m1↓ this = 0∧ #m2↓ this = 0. The verification condition

$$\models (A(I) \wedge \mathit{Init}(\text{Toy}) \wedge \text{Toy}() = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \rightarrow (A(S) \wedge A(I))[\mathsf{h} \cdot \text{Toy}()/\mathsf{h}]$$

corresponding to the selection of the constructor method Toy validates this precondition $A(S)$. It holds trivially because *Init*(Toy) includes (by definition) the information that the local history of the object denoted by this is empty and the global history update with the selection of the constructor method does not affect the number of selections of the method m1 and m2. Further, since the global invariant $A(I)$ does not refer to the constructor method, the global history update also does not affect the global invariant. A subtle point here is that it also does not affect the scope of the quantification (such as in $A(I)$, over Toy objects) because it ranges over objects which have been *already* generated by a call of their constructor method. Note that such calls do affect the scope of the quantification which thus has to be accounted for in the verification condition of the object creation statement toy=new Toy() (because of space limitations omitted here).

Given this precondition $A(S)$ of the constructor method it is straightforward to validate as precondition of the return statement of the constructor method the class invariant $A(\mathrm{Toy})$ itself. This further trivializes the verification condition for the return statement of the constructor method (which should validate the class invariant).

Let S now denote the body $c1 = c1+1; \mathsf{return}$ of method m1 and $A(S)$ denote its precondition $\#\mathrm{m1} \downarrow \mathsf{this} = 1 \wedge \#\mathrm{m2} \downarrow \mathsf{this} = 0 \wedge \mathrm{c1} = \mathrm{c2} = 0$. As another example, we show how to validate the verification condition

$$\models (A(I) \wedge A(\mathrm{Toy}) \wedge \mathrm{m1}() = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \to (A(S) \wedge A(I))[\mathsf{h} \cdot \mathrm{m1}()/\mathsf{h}]$$

which corresponds to the selection of the m1 method (by the object this). To validate that $A(S)$ holds under the global history update (with the selection of m1), we first observe that $\#\mathrm{m2} \downarrow \mathsf{this}$ is not affected by this update and $A(\mathrm{Toy})$ already implies $\#\mathrm{m2} \downarrow \mathsf{this} = \mathrm{c2} = 0$. Next observe that $\#\mathrm{m1} \downarrow \mathsf{this} = 1$ under the global history update boils down to $\#\mathrm{m1} \downarrow \mathsf{this} + 1 = 1$, where $\#\mathrm{m1}\downarrow \mathsf{this}$ is thus evaluated with respect to the "old" history. From the global invariant $A(I)$ and $\mathrm{m1}() = \mathsf{Sched}(\mathsf{h}, \mathsf{this})$) we infer that $\#\mathrm{m1}\downarrow \mathsf{this} < \#\mathsf{this}.\mathrm{m1}{=}1$, that is, $\#\mathrm{m1}\downarrow \mathsf{this} = 0$. From the class invariant $A(\mathrm{Toy})$ we then derive that $\#\mathrm{m1}\downarrow \mathsf{this} = \mathrm{c1} = 0$. Further, to prove that $A(I)$ holds under the global history update, it suffices to observe that, as already argued above, $\#\mathrm{m1}\downarrow \mathsf{this} + 1 \leq \#\mathsf{this}.\mathrm{m1}$ and that $\#\mathsf{this}.\mathrm{m1}\leq 1\wedge$ $\#\mathrm{m2}\downarrow \mathsf{this} = \#\mathsf{this}.\mathrm{m2}{=}0$ is not affected by the global history update. Neither is $\#\mathrm{m1}\downarrow z \leq \#z.\mathrm{m1}\leq 1\wedge$ $\#\mathrm{m2}\downarrow z = \#\mathrm{z}.\mathrm{m2} = 0$, where z refers to a (created) instance of class Toy different from (the object denoted by) this.

As a last example we discuss the validation of the verification condition

$$\models (A(I) \wedge A(S)) \to (A(I)[\mathsf{h} \cdot \mathrm{toy!m1}/\mathsf{h}])$$

where S denotes the call toy!m1 in the main statement and $A(S)$ denotes its precondition $\#\mathrm{m1}\uparrow\mathsf{this}{=}0\wedge\#\mathrm{m2}\uparrow\mathsf{this}{=}0$, where $\#\mathrm{m1}\uparrow \mathsf{this}$ and $\#\mathrm{m2}\uparrow \mathsf{this}$ denote the number of (outgoing) calls of m1 and m2 by the root object. We thus use the variable this to denote the root object executing the main statement. Note that $\#\mathrm{m1}\uparrow \mathsf{this}$ and $\#\mathrm{m2}\uparrow \mathsf{this}$ both refer to the number of calls of the methods m1 and m2 by the root object, as recorded by the global history. However, since these method calls by the root object correspond with those recorded by its local history, $A(S)$ is in fact a local assertion. To validate that the global invariant $A(I)$ holds under the global history update with the call of m1 by the root object, we need to validate that $\#\mathrm{toy}.\mathrm{m1} = 0$ holds *before* the update (that is, in the "old" history). This requires a formalization of the absence of other objects calling m1. We can do so by strengthening the global invariant $A(I)$ with the information that the methods m1 and m2 are (only) called by the root object. It is easy to check that this additional information preserves the validation of the above verification conditions since the corresponding global history updates do not affect it (they do not involve any calls to the methods m1 and m2).

We conclude the discussion of this toy example with the observation that our proof method supports a more expressive notion of modularity than the one supported by the bottom-up approach. In fact, we can derive global invariants *compositionally* by, roughly, a conjunction of the *behavioral interfaces* of the constituent classes. Such a behavioral interface (of a class) describes for each instance of the class its local history as a projection of the global history *extended* with the *incoming* calls of its methods. Like a pre/postcondition provides a state-based contract of a method, such a behavioral interface provides a history-based contract stating the expected interaction with the environment in terms of both incoming and outgoing method calls. Validation of behavioral interfaces as global invariants allows to resolve the non-determinism in the selection of the methods.

Soundness and Completeness. We have the following soundness theorem.

Theorem 1. *If* $\vdash A(P)$ *then* $\models A(P)$.

Proof. We have to show for every computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ and global environment ω such that $\sigma_0, \tau_0, \omega[\mathsf{h} := \epsilon] \models A(main)$, then $\Gamma, \omega[\mathsf{h} := \theta] \models A(P)$. The proof proceeds by a straightforward induction on the length of the computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$. We treat the case of the selection of a method. Let (induction hypothesis) $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ and ω be such that $\Gamma, \omega[\mathsf{h} := \theta] \models A(P)$. Further, let $(\sigma, \mathsf{nil}) \in \Gamma$ and $\alpha = \sigma(\mathsf{this}).m(\tau) = \mathsf{Sched}(\theta, \sigma(\mathsf{this}))$, where $\sigma(\mathsf{this})$ is an instance of class C. By the induction hypothesis we have that $\omega[\mathsf{h} := \theta] \models A(I)$ and $\sigma, \omega[\mathsf{h} := \theta] \models A(C)$. Further, $\alpha = \mathsf{Sched}(\theta, \sigma(\mathsf{this}))$ logically amounts to $\sigma, \tau, \omega[\mathsf{h} := \theta] \models m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this}))$, where $\bar{u}$ are the formal parameters of m. So we derive from the VC

$$\models (A(I) \wedge A(C) \wedge m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \to (A(S) \wedge A(I))[\mathsf{h} \cdot m(\bar{u})/\mathsf{h}]$$

that $\sigma, \tau, \omega[\mathsf{h} := \theta] \models (A(S) \wedge A(I))[\mathsf{h} \cdot m(\bar{u})/\mathsf{h}]$, that is, $\sigma, \tau, \omega[\mathsf{h} := \theta \cdot \alpha] \models A(S) \wedge A(I)$. Further, since $\theta_{o'} = (\theta \cdot \alpha)_{o'}$, for $o' \neq \sigma(\mathsf{this})$, we have for any local assertion p (of a method or specifying a class invariant) that $\sigma', \tau', \omega[\mathsf{h} := \theta] \models p$ iff $\sigma', \tau', \omega[\mathsf{h} := \theta \cdot \alpha] \models p$. In words, the update of the global history does not affect the validity of the local assertions of the other objects. We conclude that $\Gamma', \omega[\mathsf{h} := \theta \cdot \alpha] \models A(P)$, where $(\Gamma, \theta) \to (\Gamma', \theta')$ results from executing the method selection α.

Next we discuss completeness. We introduce for each variable x of the main statement a fresh global variable x' which will be used to store the initial value of x. For any global environment ω we denote by $\omega(main)$ the pair (σ, τ) such that $\sigma(x) = \omega(x')$, for every instance variable x appearing in $main$, and $\tau(x) = \omega(x')$, for every local variable x appearing in $main$.

Definition 2 (Reachability). *We construct an annotation $A'(P)$ which associates with each statement S occurring in a method m the following reachability assertions:*

> $\sigma, \tau, \omega \models A'(S)$ *iff there exists a computation* $\omega(main) \Rightarrow (\Gamma, \theta)$ *such that* $\langle \sigma, (\tau, S) \rangle \in \Gamma$ *and* $\theta_o = \omega(\mathsf{h})_o$, *where* $o = \sigma(\mathsf{this})$.

Further, for each class C we introduce the following reachability assertion $A'(C) \in \mathsf{Pred}(C)$ which characterizes the scheduling points:

$\sigma, \tau, \omega \models A'(C)$ iff there exists a computation $\omega(main) \Rightarrow (\Gamma, \theta)$ such that $\langle\sigma, \mathsf{nil}\rangle \in \Gamma$ and $\theta_o = \omega(\mathsf{h})_o$, where $o = \sigma(\mathsf{this})$.

Finally, we have the following global invariant $A'(I) \in \mathsf{Pred}(P)$ which characterizes the reachability of the global history.

$\sigma, \tau, \omega \models A'(I)$ iff there exists a computation $\omega(main) \Rightarrow (\Gamma, \omega(\mathsf{h}))$.

Next we have to prove the validity of the VC's for the above reachability assertions. We treat the following main cases.

Lemma 2 (Asynchronous method call). *For a statement $S \equiv e_o!m(\bar{e}); S'$ we have*

$$\models (A'(I) \wedge A'(S)) \rightarrow (A'(I) \wedge A'(S'))[\mathsf{h} \cdot e_0!m(\bar{e})/\mathsf{h}]$$

Proof. Let $\sigma, \tau, \omega \models A'(I) \wedge A'(S)$. Further, let $\omega(\mathsf{h}) = \theta$ and $o = \sigma(\mathsf{this})$. By the above definition of $A'(I)$ there exists a computation $\omega(main) \Rightarrow (\Gamma, \theta)$ (*), and by the above definition of $A'(S)$, there exists a computation $\omega(main) \Rightarrow (\Gamma', \theta')$ (**), such that $(\sigma, (\tau, S)) \in \Gamma'$ and $\theta_o = \theta'_o$. We have to show that $\sigma, \tau, \omega[\mathsf{h} := \theta \cdot \alpha] \models A'(I) \wedge A'(S)$, where $\alpha = \mathsf{Val}(e_0!m(\bar{e}))(\sigma, \tau)$. By Lemma 1 we can replace the computation of o in (*) by that of o in (**), which gives us a computation $\omega(main) \Rightarrow (\Gamma[o := \Gamma'(o)], \theta)$. Executing the call $e_o!m(\bar{e})$ (according to the above transition for asynchronous method calls) then gives rise to a computation $\omega(main) \Rightarrow (\Gamma[o := (\sigma, (\tau, S'))], \theta \cdot \alpha)$, from which we conclude $\sigma, \tau, \omega[\mathsf{h} := \theta \cdot \alpha] \models A'(I) \wedge A'(S')$.

Lemma 3 (Method selection). *For a method m of a class C (excluding the constructor method) with body S we have*

$$\models (A'(I) \wedge A'(C) \wedge m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})) \rightarrow (A'(S) \wedge A'(I))[\mathsf{h} \cdot m(\bar{u})/\mathsf{h}]$$

where $\bar{u}$ are the formal parameters of m.

Proof. Let $\sigma, \tau, \omega \models A'(I) \wedge A'(C) \wedge m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})$. We have to show that $\sigma, \tau, \omega[\mathsf{h} := \theta \cdot \alpha] \models A'(S) \wedge A'(I)$, where $\alpha = Val(m(\bar{u}))(\sigma, \tau)$. Let $\omega(\mathsf{h}) = \theta$ and $o = \sigma(\mathsf{this})$. As in the proof of the above lemma, we obtain a computation $\omega(main) \Rightarrow (\Gamma[o := (\sigma, \mathsf{nil})], \theta)$. Since $\sigma, \tau, \omega \models m(\bar{u}) = \mathsf{Sched}(\mathsf{h}, \mathsf{this})$, according to the above transition for method selection, the object o can select the method m for execution in the local environment τ, which gives rise to a computation $\omega(main) \Rightarrow (\Gamma[o := (\sigma, (\tau, S))], \theta \cdot \alpha)$, from which we conclude $\sigma, \tau, \omega[\mathsf{h} := \theta \cdot \alpha] \models A'(I) \wedge A'(S)$.

Theorem 2 (Completeness). *We have that $\models A(P)$ implies $\vdash A(P)$.*

Proof. Let $\models A(P)$. Since the VC's of $A'(P)$ are valid, we have $\models_{\mathrm{vc}} A'(P)$. Let p be the precondition $A(main)$ with all its instance and local variables replaced by the global variables introduced above to store the initial state of the root object. Let $p[\epsilon/\mathsf{h}] \wedge A'(P)$ (ϵ denotes the empty history) denote the annotation obtained by adding to each assertion of $A'(P)$ the invariant assertion $p[\epsilon/\mathsf{h}]$. It is straightforward to check that $\models_{\mathrm{vc}} A'(P)$ implies $\models_{\mathrm{vc}} p[\epsilon/\mathsf{h}] \wedge A'(P)$. From

$\models A(P)$ it follows that $\models (p[\epsilon/\mathsf{h}] \wedge A'(P)) \rightarrow A(P)$, and so $\vdash A(P)$. For example, let $\sigma, \tau, \omega \models p[\epsilon/\mathsf{h}] \wedge A'(S)$, for some statement S appearing in method m. Let $\omega(main) = (\sigma_0, \tau_0)$. By definition of $p[\epsilon/\mathsf{h}]$ and $\omega(main)$, it follows that $\sigma, \tau, \omega \models p[\epsilon/\mathsf{h}]$ iff $\sigma_0, \tau_0, \omega[\mathsf{h} := \epsilon] \models A(main)$, By definition of $A'(S)$, there exists a computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$, such that $(\sigma, (\tau, S)) \in \Gamma$ (and $\theta_o = \omega(\mathsf{h})_o$). So by definition of $\models A(P)$ (Definition 1) we obtain $\sigma, \tau, \omega[\mathsf{h} := \theta] \models A(S)$, from which in turn we derive $\sigma, \tau, \omega \models A(S)$ (because $\theta_o = \omega(\mathsf{h})_o$ and $A(S) \in \mathsf{Pred}(m)$).

4 Reasoning About Cooperative Scheduling

In this section we extend the language with statements which await on Boolean conditions[3]. In order to reason about local interference between the different processes of an object we need to distinguish between different method invocations. We will do so by associating with each method invocation[4] a unique future of type $\mathsf{Fut}\langle\mathsf{Unit}\rangle$, that is, a future with no return value. These futures are generated by method calls $x := e_0!m(x, \bar{e})$, where x is a future variable of type $\mathsf{Fut}\langle\mathsf{Unit}\rangle$ which is passed as parameter to the corresponding method invocation. Each method definition is therefore extended with a formal parameter d (for "destiny") of type $\mathsf{Fut}\langle\mathsf{Unit}\rangle$). In the following section we will study their actual use at the programming level. Note that the futures in this section are only used to identify different method invocations. In the next section we show how futures can be used as a mechanism for communicating asynchronously return values.

Given the above identification scheme we can introduce a local scheduling history s as a sequence of futures. Such a sequence indicates when which process is scheduled for execution. We add to each class the instance variable s and specify the semantics and VC's of programs which are augmented with the following updates of s: Every method body starts with the update $\mathsf{s} := \mathsf{s} \cdot \mathsf{d}$ and every Boolean await statement is followed by the update $\mathsf{s} := \mathsf{s} \cdot \mathsf{d}$. Note that thus s is only updated when a method invocation has been selected for execution, in other words, release of control is *not* recorded.

Semantics. An object configuration (σ, p, Q) additionally includes a set Q of suspended processes. We present the following transitions for the await statement and the scheduling of a suspended process (Table 2).

Boolean await statements thus introduce *non-determinism* in the local behavior of an object. However, the substitutivity Lemma 1 still holds because of the compositionality result in [9]. The next lemma further states that the local scheduling history of an object together with its local view of the global history fully determine its local behavior (in the context of the standard deterministic sequential programming statements).

Lemma 4 (Determinism). *Let $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ be a computation such that $\Gamma(o)$ denotes a scheduling point (i.e., its active process is* nil*). Further, let*

[3] For technical convenience only, we assume that such await statements do not occur in constructor methods.

[4] Excluding calls to constructor methods.

Table 2. Semantics Boolean await statement and process selection

$(\Gamma \cup \{(\sigma, p, Q)\}, \theta) \to (\Gamma \cup \{(\sigma, \mathsf{nil}, Q \cup \{p\})\}, \theta))$ where $p = (\tau, \mathsf{await}\ b; S')$
$(\Gamma \cup \{(\sigma, \mathsf{nil}, Q \cup \{p\})\}, \theta) \to (\Gamma \cup \{(\sigma, (\tau, S'), Q)\}, \theta)$ where $p = (\tau, \mathsf{await}\ b; S')$and $Val(b)(\sigma, \tau) = true$

$(\sigma_0, \tau_0) \Rightarrow (\Gamma', \theta')$ *be a computation such that* θ'_o *and* $\sigma'(\mathsf{s})$ *are prefixes of* θ_o *and* $\sigma(\mathsf{s})$*, respectively, where* σ *and* σ' *denote the state of* o *in* $\Gamma(o)$ *and* $\Gamma'(o)$*, respectively. Then there exist a decomposition* $(\sigma_0, \tau_0) \Rightarrow (\Gamma'', \theta'') \to^* (\Gamma, \theta)$ *of* $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ *such that* $\Gamma''(o) = \Gamma'(o)$*.*

Proof. The proof proceeds by induction on the length of the computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma', \theta')$ and a case analysis of the transition relation.

Verification Conditions. In the verification conditions discussed below, for each instance variable x of a class (excluding the variable this), we use a corresponding global variable x' which will be used to store the *old* value of x at a previous scheduling point. Similarly, the global variable h' will be used to store the old value of the global history h at a previous scheduling point. Global variables are used only in the verification conditions, and do not appear in the program code. For technical convenience only, we restrict the notion of validity of assertions to assignments σ and ω such that $\omega(\mathsf{s}')$ and $\omega(\mathsf{h}')$ are prefixes of $\sigma(\mathsf{s})$ and $\omega(\mathsf{h})$, respectively.

We have the following adaptation of the VC for method calls which caters for the creation of a new future uniquely identifying the call.

Asynchronous Method Call. For a statement $S \equiv x := e_0!m(x, \bar{e}); S'$ we have the VC:

$$\models (A(I) \wedge A(S)) \to (A(I) \wedge A(S'))[\mathsf{h} \cdot e_0!m(x, \bar{e})/\mathsf{h}][\mathsf{new}/x]$$

Boolean Await Statement. For a statement $S \equiv \mathsf{await}\ b; S'$ we have the VC's:

$$\models A(S) \to A(C)$$

which ensures that the class invariant $A(C)$ holds before execution of the await statement, and

$$\models (A(S)[\boldsymbol{x}'/\boldsymbol{x}] \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}' \wedge b) \to A(S')$$

The assertion $A(S)[\boldsymbol{x}'/\boldsymbol{x}]$ expresses that $A(S)$ holds for the "old" values of the instance variables and the global history, which are represented by the global "freeze" variables $\boldsymbol{x}'$ (which do not appear in $A(S')$). The condition $\mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}'$ expresses that the method invocation identified by the destiny variable d has not been scheduled for execution since s', i.e., it does not appear in the suffix $\mathsf{s} \setminus \mathsf{s}'$ of s determined by its prefix s' (note that by the notion of validity we have that s' is a prefix of s).

It is worthwhile to observe that an alternative approach to the verification of Boolean await statements uses a *local* identification scheme for method invocations which generates such identifications upon the selection of a method (and thus requires an adaptation of the corresponding VC instead of the above adaptation of the VC for method calls). However, this approach is not compatible with the general use of futures described in the next section which requires a global correspondence between futures and method invocations.

Soundness and Completeness. We extend the semantics of annotated programs (Definition 1) to the new semantics by simply abstracting from the set of suspended processes. The following theorem generalizes Theorem 1 to programs P of the language extended with Boolean await statements.

Theorem 3 (Soundness). *Let* $\models_{\mathrm{vc}} A(P)$*. For every computation* $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ *and global environment* ω *such that* $\sigma_0, \tau_0, \omega[\mathsf{h} := \epsilon] \models A(main)$ *we then have that* $\Gamma, \omega[\mathsf{h} := \theta] \models A(P)$*.*

Proof. The proof proceeds by induction on the length of the computation: Let (induction hypothesis) $(\sigma_0, \tau_0) \Rightarrow (\Gamma, \theta)$ and ω be such that $\Gamma, \omega[\mathsf{h} := \theta] \models A(P)$. We treat the main case that there exists a scheduling point $(\sigma, \mathsf{nil}, Q) \in \Gamma$ such that $Val(b)(\sigma, \tau') = true$, for some process $(\tau', S) \in Q$, where S denotes a statement $\mathsf{await}\ b; S'$. Let $\omega' = \omega[\mathsf{h} := \theta]$. Since $\Gamma, \omega' \models A(P)$, we have $\sigma, \omega' \models A(C)$ (assuming that $\sigma(\mathsf{this})$ is an instance of class C), and so $\sigma, \tau', \omega' \models A(C) \wedge b$. Next, let $(\sigma_0, \tau_0) \Rightarrow (\Gamma', \theta') \rightarrow^* (\Gamma, \theta)$ be a decomposition of the given computation such that for some σ' and Q' we have $(\sigma', (\tau', S), Q') \in \Gamma'$ and $\tau'(\mathsf{d})$ does not appear in the suffix of $\sigma(\mathsf{s})$ determined $\sigma'(\mathsf{s})$ (i.e., the process (τ', S) has not been scheduled in the computation $(\Gamma', \theta') \rightarrow^* (\Gamma, \theta)$). Applying the induction hypothesis to the computation $(\sigma_0, \tau_0) \Rightarrow (\Gamma', \theta')$ we obtain $\sigma', \tau', \omega[\mathsf{h} := \theta'] \models A(S)$. Let ω'' be obtained from ω' be assigning $\sigma'(x)$ to the global freeze variable x', for each instance variable x, and assigning θ' to the freeze variable h'. We then derive that $\sigma, \tau', \omega'' \models A(S)[\boldsymbol{x'}/\boldsymbol{x}]$. Since the freeze variables are assumed not to occur in $A(P)$, it follows that $\sigma, \tau', \omega' \models A(C) \wedge b$ implies $\sigma, \tau', \omega'' \models A(C) \wedge b$. Summarizing, we have obtained that $\sigma, \tau', \omega'' \models A(S)[\boldsymbol{x'}/\boldsymbol{x}] \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \backslash \mathsf{s}' \wedge b$. From

$$\models (A(S)[\boldsymbol{x'}/\boldsymbol{x}] \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}' \wedge b) \rightarrow A(S')$$

then we derive $\sigma, \tau', \omega'' \models A(S')$. Finally, we conclude that $\sigma, \tau', \omega' \models A(S')$ (since the freeze variables are assumed not to occur in $A(P)$).

We next discuss completeness. Definition 2 of the reachability assertions is extended to programs P containing Boolean await statements by simply abstracting from the set of suspended processes, i.e., $\sigma, \tau, \omega \models A'(S)$ iff there exists a computation $\omega(main) \Rightarrow (\Gamma, \theta)$ such that $\langle \sigma, (\tau, S), Q \rangle \in \Gamma$, *for some* Q, and $\theta_o = \omega(\mathsf{h})_o$, where $o = \sigma(\mathsf{this})$.

As in the previous section, in order to prove that $\vdash A'(P)$ we have to validate the associated VC's. We consider the following main case (the other cases are treated as described previously).

Lemma 5 (Awaiting a Boolean condition). *For a statement* $S \equiv \mathsf{await}\ b; S'$ *we have*

$$\models (A'(S)[\boldsymbol{x'}/\boldsymbol{x}] \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}' \wedge b) \rightarrow A'(S')$$

Proof. Let

$$\sigma, \tau, \omega \models A'(S)[\boldsymbol{x'}/\boldsymbol{x}] \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}' \wedge b$$

Further, let $o = \sigma(\mathsf{this})$. By definition of $A(C)$ there exists a computation $\omega(\mathit{main}) \Rightarrow (\Gamma, \theta)$ (*) such that $\langle \sigma, \mathsf{nil}, Q \rangle \in \Gamma$, for some set Q of suspended processes, and $\theta_o = \omega(\mathsf{h})_o$. Let σ' be such that $\sigma'(x) = \omega(x')$, for every instance variable x, and $\sigma'(\mathsf{this}) = o$. It follows that $\sigma', \tau, \omega \models A'(S)$, so there exists a computation $\omega(\mathit{main}) \Rightarrow (\Gamma', \theta')$ (**) such that $\langle \sigma', (\tau, S), Q' \rangle \in \Gamma'$, for some set Q' of suspended processes, and $\theta'_o = \omega(\mathsf{h}')_o$. By the validity of assertions we have that $\omega(\mathsf{h}')$ is a prefix of $\omega(\mathsf{h})$, and $\omega(\mathsf{s}')$ is a prefix of $\sigma(\mathsf{s})$. It follows that θ'_o is a prefix of θ_o, and $\sigma'(\mathsf{s}) = \omega(\mathsf{s}')$ is a prefix of $\sigma(\mathsf{s})$. By Lemma 4 then we infer that the computation of o in (**) is a prefix of the computation of o in (*). Next observe that $\tau(\mathsf{d})$ has not been scheduled since (because $\sigma, \tau, \omega \models \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}'$), so we have that $(\tau, S) \in Q$. Thus, we can schedule the process (τ, S) in Γ because $\sigma, \tau, \omega \models b$, from which we obtain by executing the await statement (according to the above transition) that $\sigma, \tau, \omega \models A'(S')$.

Finally, it is straightforward to check that the proof of the completeness Theorem 2 carries over to the language with Boolean await statements.

5 Reasoning About Futures

Finally, we consider reasoning about futures. We introduce return statements $\mathsf{return}\ e$, where e is a side-effect basic expression of some type T. Such a statement returns a value of type $\mathsf{Fut}\langle T \rangle$ which denotes a reference to a value of type T. As in the previous section a future is generated by a call $x := e_o!m(x, \bar{e})$ and passed as parameter to the method invocation. The statement $\mathsf{await}\ x?$ and the assignment $y := x.\mathsf{get}$, respectively, suspend and block, as long as the future denoted by the variable x has not been resolved.

Semantics. We extend global histories with future completions of the form $o \mapsto f!v$ which indicate that object o has completed future f with value v, and future queries $o \mapsto f?v$ which indicate that object o read value v stored by future f. For $f = \mathit{Val}(y)(\sigma, \tau)$ we define $\mathit{Val}(y.\mathsf{get})(\sigma, \tau, \omega) = v$, if $o \mapsto f!v$, for some object o, appears in $\omega(\mathsf{h})$, and $\mathit{Val}(y.\mathsf{get})(\sigma, \tau, \omega) = \perp$, otherwise.

Further, programs are augmented with updates $\mathsf{s} := \mathsf{s} \cdot \mathsf{d}$ to the local scheduling history, as described in the previous section.

Table 3 presents the transitions for returning/getting a return value[5], and awaiting a future. For the Substitutivity Lemma 1 to hold, it is not difficult to see that we only need to observe for each object o and future f the *first* occurrence of a query $o \mapsto f?v$. Therefore, we define $\theta \cdot o \mapsto f?v = \theta$, in case $o \mapsto f?v$ already appears in θ.

[5] In the transition for the get operation we assume without loss of generality that the variable x is a local variable.

Table 3. Semantics return/get operation and awaiting return values

$(\Gamma \cup \{(\sigma,(\tau,\mathsf{return}\ e;S,Q)\},\theta) \rightarrow (\Gamma \cup \{(\sigma,(\tau,S),Q\},\theta \cdot \sigma(\mathsf{this}) \mapsto f!v))$ where $f=\tau(\mathsf{d})$and $v=\ Val(e)(\sigma,\tau)$
$(\Gamma\cup\{(\sigma,(\tau,x:=y.\mathsf{get};S),Q\},\theta) \rightarrow (\Gamma\cup\{(\sigma,(\tau[x:=v],S),Q)\},\theta\cdot\sigma(\mathsf{this}) \mapsto f?v)$ where $v=\ Val(y.\mathsf{get})(\sigma,\tau,\omega) \neq\perp$
$(\Gamma \cup \{(\sigma,\mathsf{nil},Q \cup \{p\})\},\theta) \rightarrow (\Gamma \cup \{(\sigma,(\tau,S'),Q)\},\theta \cdot \sigma(\mathsf{this}) \mapsto f?v)$ where $p=(\tau,\mathsf{await}\ x?;S')$and $v=\ Val(y.\mathsf{get})(\sigma,\tau,\omega) \neq\perp$

Verification Conditions. We have the following verification conditions which closely reflect the above semantics.

Returning a Value. For a return statement $S \equiv \mathsf{return}\ e$ (of a method in class C) we have the following VC:

$$\models (A(I) \wedge A(S)) \rightarrow (A(I) \wedge A(C))[\mathsf{h} \cdot \mathsf{this} \mapsto \mathsf{d}!e/\mathsf{h}]$$

Given that the global invariant $A(I)$ and the precondition of $\mathsf{return}\ e$ hold, this VC guarantees that the global invariant $A(I)$ and the class invariant $A(C)$ hold after updating the global history .

Getting a Return Value. For a statement $S \equiv x := y.\mathsf{get}; S'$ we have the VC:

$$\models (A(I) \wedge A(S) \wedge z = y.\mathsf{get} \wedge z \neq\perp) \rightarrow A(S')[z/x][\mathsf{h} \cdot \mathsf{this} \mapsto y?z/\mathsf{h}]$$

Awaiting a Future. For a statement $S \equiv \mathsf{await}\ y?; S'$ we have the VC's (here z is a fresh global variable):

$$\models A(S) \rightarrow A(C)$$

and

$$\models (A(S)[\boldsymbol{x}'/\boldsymbol{x}] \wedge A(I) \wedge A(C) \wedge \mathsf{d} \notin \mathsf{s} \setminus \mathsf{s}' \wedge z = y.\mathsf{get} \wedge z \neq\perp) \rightarrow A(S')[\mathsf{h} \cdot \mathsf{this} \mapsto y?z/\mathsf{h}]$$

The global invariant $A(I)$ here is used to import information about the future queried.

It is not difficult to see that the semantics of the language extended with futures satisfies both the above properties of (global) substitutivity and (local) determinism. Consequently, soundness and completeness proofs follow the same pattern as that of the previous sections.

6 Future Work

A major future effort concerns the implementation of the proof method in the KeY theorem prover [1], and the application of this implementation to ABS programs, e.g., the ABS model of railway operations described in [21]. This involves the integration of a suitable assertion language, e.g., as described in

[12]. We refer to [11] for how to extend the completeness result by a logical formulation of the reachability assertions defined in this paper.

Currently, we are also working on a formalisation of the soundness and completeness proof in a theorem prover, like Coq or Isabelle/HOL. To the best of our knowledge such a formalization of the soundness and completeness proofs would be the first for a concurrent language.

Acknowledgement. We thank Olaf Owe for some insightful discussions.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. Lecture Notes in Computer Science, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Ahrendt, W., Dylla, M.: A verification system for distributed objects with asynchronous method calls. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 387–406. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_20
3. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. **77**(12), 1289–1309 (2012). https://doi.org/10.1016/j.scico.2010.08.003
4. Apt, K.R.: Formal justification of a proof system for communicating sequential processes. J. ACM **30**(1), 197–216 (1983). https://doi.org/10.1145/322358.322372
5. Blom, S., Huisman, M., Mihelcic, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014). https://doi.org/10.1016/j.scico.2014.03.013
6. De Boer, F., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017). http://doi.acm.org/10.1145/3122848, https://doi.org/10.1145/3122848
7. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1), 70–90 (1978). https://doi.org/10.1137/0207005
8. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
9. Boer, F.S., Gouw, S.: Compositional semantics for concurrent object groups in ABS. In: Müller, P., Schaefer, I. (eds.) Principled Software Development, pp. 87–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98047-8_6
10. de Boer, F.S., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
11. de Gouw, S., de Boer, F., Ahrendt, W., Bubel, R.: Weak arithmetic completeness of object-oriented first-order assertion networks. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 207–219. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35843-2_19
12. de Gouw, S., de Boer, F., Ahrendt, W., Bubel, R.: Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic. Softw. Syst. Model. **15**(4), 1117–1140 (2014). https://doi.org/10.1007/s10270-014-0446-9

13. de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
14. Din, C.C., Hähnle, R., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. In: Schmidt, R.A., Nalon, C. (eds.) TABLEAUX 2017. LNCS (LNAI), vol. 10501, pp. 22–43. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66902-1_2
15. Crystal Chang Din and Olaf Owe: A sound and complete reasoning system for asynchronous communication with shared futures. J. Log. Algebr. Meth. Program. **83**(5–6), 360–383 (2014)
16. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2014). https://doi.org/10.1007/s00165-014-0322-y
17. Haslbeck, M.P.L., Nipkow, T.: Hoare logics for time bounds - a study in meta theory. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part I. LNCS, vol. 10805, pp. 155–171. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_9
18. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
19. Kamburjan, E., Chen, T.-C.: Stateful behavioral types for active objects. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 214–235. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_13
20. Kamburjan, E., Din, C.C., Hähnle, R., Johnsen, E.B.: Asynchronous cooperative contracts for cooperative scheduling. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 48–66. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_3
21. Kamburjan, E., Hähnle, R., Schön, S.: Formal modeling and analysis of railway operations with active objects. Sci. Comput. Program. **166**, 167–193 (2018)
22. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
23. Zaharieva-Stojanovski, M., Huisman, M.: Verifying class invariants in concurrent programs. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 230–245. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_16

Improving Automatic Complexity Analysis of Integer Programs

Jürgen Giesl(✉), Nils Lommen, Marcel Hark, and Fabian Meyer

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
{giesl,lommen,marcel.hark,fabian.meyer}@cs.rwth-aachen.de

Abstract. In [16], we developed an approach for automatic complexity analysis of integer programs, based on an alternating modular inference of upper runtime and size bounds for program parts. In this paper, we show how recent techniques to improve automated termination analysis of integer programs (like the generation of multiphase-linear ranking functions and control-flow refinement) can be integrated into our approach for the inference of runtime bounds. The power of the resulting approach is demonstrated by an extensive experimental evaluation with our new re-implementation of the corresponding tool KoAT.

1 Introduction

There are many techniques and tools for automated complexity analysis of programs, e.g., [2–6,8–10,16,17,22–24,29,32,33,40,43,46,47]. Most of them infer variants of (mostly linear) polynomial ranking functions (see, e.g., [15,44]) which are then combined to get a runtime bound for the overall program. However, approaches based on linear ranking functions are incomplete for termination and thus also for complexity analysis. For example, consider the loop from [12,38] in Fig. 1, which terminates, but does not admit a linear ranking function. Its runtime is linear in the initial values of x and y, if they are positive initially. The reason is that if $y > 0$, then x grows first but it is decreased with the same "speed" once y has become negative.

```
while $x > 0$ do
    $x \leftarrow x + y$
    $y \leftarrow y - 1$
```

Fig. 1. Loop without Linear Ranking Function

Recently so-called multiphase-linear ranking functions have gained interest (see, e.g., [12,13,38,50]). For loops as in Fig. 1, ranking functions of this form detect that the program has two phases: first y is decremented until it is negative. Afterwards, x is decremented until it is negative and the loop terminates. In [12], it is shown that the existence of a multiphase-linear ranking function for a loop implies linear runtime complexity. In the current paper, we embed multiphase-linear ranking functions into our modular approach for complexity analysis of

Funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL.

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 193–228, 2022.
https://doi.org/10.1007/978-3-031-08166-8_10

```
while $x < 0$ do
    if $y < z$ then
        $y \leftarrow y - x$
    else
        $x \leftarrow x + 1$
```

Fig. 2. Original Loop

```
while $x < 0 \land y < z$ do
    $y \leftarrow y - x$
while $x < 0 \land y \geq z$ do
    $x \leftarrow x + 1$
```

Fig. 3. After Control-Flow Refinement

integer programs from [16]. In contrast to [12], we infer multiphase-linear ranking functions for parts of the program and combine the so-obtained bounds to an overall runtime bound. In this way, we obtain a powerful technique which is able to infer finite runtime bounds for programs that contain loops such as Fig. 1.

Moreover, different forms of control-flow refinement were used to improve the automatic termination and complexity analysis of programs further, see, e.g., [20,22]. The basic idea is to gain "more information" on the values of variables to sort out certain paths in the program. For example, the control-flow refinement technique from [20] detects that the programs in Fig. 2 and Fig. 3 are equivalent. Clearly, the program in Fig. 3 is easier to analyze as the two consecutive loops do not interfere with each other: x and z are constants in its first loop, while y and z are constants in its second loop. We show how to integrate the technique for control-flow refinement from [20] into our modular analysis in a non-trivial way. This increases the power of our approach further.

Structure: We first recapitulate our approach from [16] in Sect. 2. Afterwards, we adapt it to multiphase-linear ranking functions in Sect. 3. In Sect. 4, we discuss how to incorporate control-flow refinement from [20] into our analysis. We provide an extensive experimental evaluation of our corresponding new version of the tool KoAT [42] and compare it with existing tools in Sect. 5. Finally, we discuss related work and conclude (Sect. 6). All proofs can be found in Appendix A.

2 Preliminaries

In this section we recapitulate our approach for complexity analysis from [16]. We first introduce *constraints*, which are used in the guards of programs.

Definition 1 (Constraints). *Let $\mathcal{V}$ be a set of variables. The set of* constraints *$\mathcal{C}(\mathcal{V})$ over $\mathcal{V}$ is the smallest set containing $e_1 \leq e_2$ for all polynomials $e_1, e_2 \in \mathbb{Z}[\mathcal{V}]$ and $c_1 \land c_2$ for all $c_1, c_2 \in \mathcal{C}(\mathcal{V})$.*

In addition to "$\leq$", we also use relations like "$>$" and "$=$", which can be simulated by constraints (e.g., $e_1 > e_2$ is equivalent to $e_2 + 1 \leq e_1$ when regarding integers).

Now we define the notion of integer programs which we use in this paper. Instead of **while** loops as in Fig. 1, 2 and 3, we use a formalism based on transitions (which of course also allows us to represent **while** programs easily).

Definition 2 (Integer Program). *An* integer program *$\mathcal{P}$ over a set of variables $\mathcal{V}$ is a tuple $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ of*

- *a finite set of* program variables $\mathcal{PV} \subseteq \mathcal{V}$*,*
- *a finite set of* locations $\mathcal{L}$ *with a distinguished* initial location $\ell_0 \in \mathcal{L}$*, and*
- *a finite set of* transitions $\mathcal{T}$*. A transition is a tuple* $(\ell, \tau, \eta, \ell')$ *consisting of*
  1. *the* start location $\ell \in \mathcal{L}$ *and the* target location $\ell' \in \mathcal{L} \setminus \{\ell_0\}$*,*
  2. *the* guard $\tau \in \mathcal{C}(\mathcal{V})$ *of* t*, and*
  3. *the* update function $\eta \colon \mathcal{PV} \to \mathbb{Z}[\mathcal{V}]$ *of* t*, mapping every program variable to an update polynomial.*

We call $\mathcal{TV} = \mathcal{V} \setminus \mathcal{PV}$ *the set of* temporary variables*.*

Note that the initial location has *no* incoming transitions. The transitions $(\ell_0, ...)$ whose start location is ℓ_0 are called *initial* transitions.

Thus, integer programs contain two kinds of non-determinism. Non-deterministic branching is realized by multiple transitions with the same start location whose guards are non-exclusive. Non-deterministic sampling is modeled by temporary variables (which can be restricted in the guard of a transition). Temporary variables are not updated in the program. Intuitively, these variables are set by an adversary trying to "sabotage" the program in order to obtain long runtimes.

Example 3. Consider the integer program in Fig. 4 over the program variables $\mathcal{PV} = \{x, y, z\}$, the locations $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, and the transitions $\mathcal{T} = \{t_0, t_1, t_2, t_3\}$. In Fig. 4, we omitted trivial guards, i.e., $\tau = \texttt{true}$, and trivial updates, i.e., updates of the form $\eta(v) = v$. This integer program corresponds to two nested loops: the inner loop is given by t_2, the outer loop by t_1 and t_3. Transition t_0 just forwards the input values. If $z > 0$, t_1 sets x and y to $z - 1$. Then, t_2 decrements y by 1 and updates x to $x+y$ repeatedly as long as $x > 0$ (i.e., it corresponds to the loop in Fig. 1). Transition t_3 decrements z by 1 and leads back to the starting point of the outer loop.

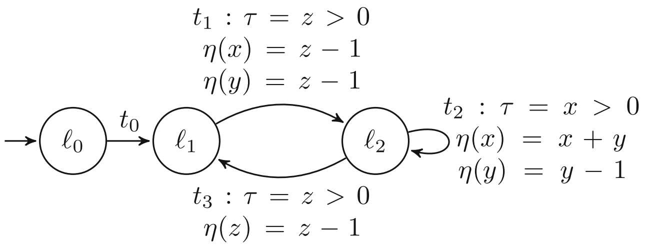


Fig. 4. Integer Program with Nested Loops

Note that t_2 and t_3 correspond to a non-deterministic branching as their guards are non-exclusive. If t_0 had the update $\eta(x) = u$ and the guard $u > 0$, then this would correspond to a non-deterministic sampling of a positive value.

From now on, we fix an integer program $\mathcal{P}$ over the variables $\mathcal{V}$. A mapping $\sigma : \mathcal{V} \to \mathbb{Z}$ is called a *state* and Σ denotes the set of all states. We also apply states to arithmetic expressions e and constraints c, where the number $\sigma(e)$ resp. the Boolean value $\sigma(c)$ results from e resp. c by replacing each variable v by $\sigma(v)$.

Definition 4 (Evaluation of Integer Programs). *A* configuration *is an element of* $\mathcal{L} \times \Sigma$*. For two configurations* (ℓ, σ) *and* (ℓ', σ')*, and a transition* $t = (\ell_t, \tau, \eta, \ell'_t) \in \mathcal{T}$*,* $(\ell, \sigma) \to_t (\ell', \sigma')$ *is an* evaluation *step by* t *if*

- $\ell = \ell_t$ *and* $\ell' = \ell'_t$*,*
- $\sigma(\tau) = \texttt{true}$*, and*
- *for every program variable* $v \in \mathcal{PV}$ *we have* $\sigma(\eta(v)) = \sigma'(v)$*.*

We denote the union of all relations $\to_t$ *for* $t \in \mathcal{T}$ *by* $\to_{\mathcal{T}}$*. Whenever it is clear from the context, we omit the transition* t *resp. the set* $\mathcal{T}$ *in the index. We also abbreviate* $(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \cdots \to_{t_k} (\ell_k, \sigma_k)$ *by* $(\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)$*.*

Example 5. For the integer program in Fig. 4, when denoting program states σ as tuples $(\sigma(x), \sigma(y), \sigma(z)) \in \mathbb{Z}^3$, we have $(\ell_0, (0,0,2)) \to_{t_0} (\ell_1, (0,0,2)) \to_{t_1} (\ell_2, (1,1,2)) \to_{t_2} (\ell_2, (2,0,2)) \to_{t_3} (\ell_1, (2,0,1))$.

For an integer program, the (worst-case) runtime complexity w.r.t. an initial state σ_0 is defined to be the length of the longest evaluation starting in σ_0.

Definition 6 (Runtime Complexity). *The (worst-case)* runtime complexity *of* $\mathcal{P}$ *is the function* $\text{rc} : \Sigma \to \overline{\mathbb{N}}$ *with* $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$ *and* $\text{rc}(\sigma_0) = \sup\{k \in \mathbb{N} \mid \ell_k \in \mathcal{L}, \sigma_k \in \Sigma, (\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)\}$ *for all* $\sigma_0 \in \Sigma$*.*

As in [16], our approach combines bounds for program parts. We restrict ourselves to bounds that represent weakly monotonically increasing functions. Such bounds have the advantage that they can easily be "composed", i.e., if f and g are both weakly monotonically increasing upper bounds, then so is $f \circ g$.

Definition 7 (Bounds). *The set of* bounds $\mathcal{B}$ *is the smallest set with* $\overline{\mathbb{N}} \subseteq \mathcal{B}$*,* $\mathcal{PV} \subseteq \mathcal{B}$*,* $b_1 + b_2 \in \mathcal{B}$*,* $b_1 \cdot b_2 \in \mathcal{B}$*, and* $k^b \in \mathcal{B}$ *for all* $k \in \mathbb{N}$ *and* $b, b_1, b_2 \in \mathcal{B}$*.*

A bound which is only constructed from $\mathbb{N}$*,* $\mathcal{PV}$*,* $+$*, and* $\cdot$ *is called* polynomial*. A polynomial bound of degree at most* 1 *is called* linear*.*

For any $\sigma \in \Sigma$, $|\sigma|$ denotes the state with $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$. Clearly, a bound $b \in \mathcal{B}$ induces a weakly monotonic function on states by mapping any $\sigma \in \Sigma$ to $|\sigma|(b) \in \overline{\mathbb{N}}$. Then, $|\sigma| \leq |\sigma'|$ implies $|\sigma|(b) \leq |\sigma'|(b)$. As usual, we compare functions pointwise, i.e., $|\sigma| \leq |\sigma'|$ means that $|\sigma|(v) \leq |\sigma'|(v)$ for all $v \in \mathcal{V}$.

Example 8. For $\mathcal{PV} = \{x, y\}$, we have ω, x^2, $x+y$, $2^{x^2+y} \in \mathcal{B}$. Here, x^2 and $x+y$ are polynomial bounds and $x+y$ is linear. Consider the state σ with $\sigma(x) = 1$ and $\sigma(y) = -2$. Then, $|\sigma|(x+y) = |1| + |-2| = 3$.

To over-approximate the runtime complexity, we now introduce the concepts of runtime and size bounds. A runtime bound for a transition $t \in \mathcal{T}$ over-approximates the maximal number of occurrences of that transition in any evaluation starting with the initial state $\sigma_0 \in \Sigma$. Here, $\to^* \circ \to_t$ denotes the relation describing arbitrary many evaluation steps followed by a step with transition t.

Definition 9 (Runtime Bound). *The function* $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ *is a* runtime bound *if for all* $t \in \mathcal{T}$ *and all states* $\sigma_0 \in \Sigma$ *we have*

$$|\sigma_0|(\mathcal{RB}(t)) \geq \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0)\,(\to^* \circ \to_t)^k\,(\ell, \sigma)\}.$$

Note that we require the runtime bound to only contain *program variables* since the values of temporary variables are "set by the adversary".

Example 10. For the program in Fig. 4, the technique from [16] obtains the following runtime bound. Trivially, $\mathcal{RB}(t_0) = 1$, as t_0 can only be applied once in any evaluation. Since the outer loop is only executed if $z > 0$ and every iteration of the outer loop decreases z by 1, we get $\mathcal{RB}(t_1) = \mathcal{RB}(t_3) = z$, i.e., these transitions can occur at most $|z_0|$ times, if z has the value $z_0 \in \mathbb{Z}$ initially. However, the implementation of [16] in the original version of the tool KoAT cannot infer a finite runtime bound for t_2 since this transition does not admit a linear ranking function, i.e., a linear function which decreases by at least one and is bounded from below for each iteration of the loop. Intuitively, the reason is that x is bounded, but it does not decrease in every iteration. In contrast, y decreases in every iteration, but it is not bounded. In Sect. 3, we will show how to improve our approach for complexity analysis such that it obtains a finite runtime bound for transitions like t_2 (see Example 21).

The following corollary shows that every runtime bound $\mathcal{RB}$ directly yields an upper bound for the program's runtime complexity: Instead of over-approximating the runtime complexity of the full program at once, one can compute runtime bounds for each transition separately and simply add these bounds.

Corollary 11 (Over-Approximating rc). *Let $\mathcal{RB}$ be a runtime bound. Then for all states $\sigma_0 \in \Sigma$ we have $|\sigma_0| \left(\sum_{t \in \mathcal{T}} \mathcal{RB}(t)\right) \geq \mathrm{rc}(\sigma_0)$.*

The framework in [16] performs a *modular* analysis of the program, i.e., parts of the program are analyzed as standalone programs and the results are then lifted to contribute to the overall analysis. For example, for the integer program in Fig. 4, the inner loop t_2 is analyzed separately in order to compute its runtime bound. But to lift a local runtime bound of t_2 to a runtime bound of t_2 in the full program, one has to take into account that the values of the variables when executing t_2 are not the input values of the program, but the values that the variables have after an execution of the previous transition t_1.

So to compute the runtime bound of a transition t', our approach considers all transitions t that can occur directly before t' in evaluations and it needs size bounds $\mathcal{SB}(t, v)$ to over-approximate the absolute values that the variables $v \in \mathcal{PV}$ may have *after* these "previous" transitions t. (This intuition will later be formalized in Theorem 20). Here, we call $\mathcal{RV} = \mathcal{T} \times \mathcal{PV}$ the set of *result variables.*

Definition 12 (Size Bound). *The function $\mathcal{SB} : \mathcal{RV} \to \mathcal{B}$ is a* size bound *if for all $(t, v) \in \mathcal{RV}$ and all states $\sigma_0 \in \Sigma$ we have*

$$|\sigma_0| \left(\mathcal{SB}(t, v)\right) \geq \sup\{|\sigma(v)| \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0) \left(\to^* \circ \to_t\right) (\ell, \sigma)\}.$$

Example 13. Consider again the program in Fig. 4. Here, $\mathcal{SB}(t_0, v) = v$ for $v \in \{x, y, z\}$, because t_0 does not change any variable. So if $(\ell_0, \sigma_0) \to_{t_0} (\ell_1, \sigma_1)$

then $|\sigma_0|\,(\mathcal{SB}(t_0,v)) = |\sigma_0|\,(v) = |\sigma_1|\,(v)$. Moreover, $\mathcal{SB}(t_1,z) = \mathcal{SB}(t_2,z) = \mathcal{SB}(t_3,z) = z$ as z is never increased in the program. For the computation of $\mathcal{SB}(t_1,x)$ and $\mathcal{SB}(t_1,y)$, the approach of [16] sums up the values of $\mathcal{SB}(t_0,z)$ and $\mathcal{SB}(t_3,z)$ (since t_0 and t_3 are the only transitions that can occur directly before t_1) and uses this as the "incoming size" of z. Hence, it obtains $\mathcal{SB}(t_1,x) = \mathcal{SB}(t_1,y) = z + z = 2 \cdot z$. The approach of [16] cannot compute finite size bounds for (t_2,x), (t_2,y), (t_3,x), and (t_3,y), since it needs a runtime bound for t_2 to over-approximate how often the "previous" transition t_2 may have been executed. In contrast, our results from Sect. 3 will enable the computation of finite size bounds for all result variables of this program, see Example 21.

So size bounds on previous transitions are needed to compute runtime bounds, and similarly, runtime bounds are needed to compute size bounds. The algorithm for the computation of size bounds in [16] is not needed to understand the techniques presented in the current paper and thus, we use it as a black box.

3 Runtime Bounds by Multiphase Ranking Functions

The approach for computing runtime bounds in [16] relies on polynomial ranking functions (see, e.g., [15,44]). In this section, we extend this approach to so-called multiphase-linear ranking functions (MΦRFs) (see, e.g., [12,13,38,50]). Our experiments in Sect. 5 demonstrate that this improves its power significantly.

In [12] it was already shown how to obtain a runtime bound from an MΦRF for a full integer program. We now adapt this result to our modular approach which allows for the computation of MΦRFs for parts of the program (Theorem 20).

3.1 Multiphase-linear Ranking Functions

As mentioned, the idea of ranking functions is to construct a function which decreases by at least one in every evaluation step when a specific transition is applied. Moreover, the ranking function has to be non-negative before we apply a transition. Thus, if the function becomes negative, then the program terminates.

An MΦRF extends this idea and uses a ranking function f_i for every "phase" $1 \leq i \leq d$ of a program. When the phases 1 to $i-1$ are finished, the functions $f_1, \ldots, f_{i-1}$ remain negative and decreasing, but now the function f_i becomes decreasing as well. If all functions are negative, then the program terminates.

Definition 14 corresponds to so-called nested MΦRFs from [12,38]. Here, the sum of f_{i-1} and f_i must be larger than the updated function f_i for all i. We set f_0 to 0. Then $f_0 + f_1 = f_1$ must be decreasing with each update. If f_1 becomes negative, then $f_1 + f_2 < f_2$ and thus, f_2 has to be decreasing with every update, and so on until f_d becomes decreasing. The program eventually terminates, since f_d must be non-negative whenever the program can be executed further. We restrict ourselves to such "nested" MΦRFs, as they are particularly easy to automate (i.e., one does not have to consider the mapping of evaluation steps to the different phases). As usual, we use an SMT solver to search for MΦRFs automatically.

In contrast to [12,38], we define MΦRFs for sub-programs $\mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ which is crucial for our modular approach (see Theorem 20). Let $\mathbb{Z}[\mathcal{PV}]_{\text{lin}}$ denote the set of linear polynomials (i.e., of degree at most 1) over $\mathbb{Z}$ in the variables $\mathcal{PV}$.

Definition 14 (MΦRFs for Sub-Programs). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ and $d \geq 1$. A tuple $f = (f_1, \ldots, f_d)$ of functions $f_1, \ldots, f_d : \mathcal{L} \to \mathbb{Z}[\mathcal{PV}]_{\text{lin}}$ is an MΦRF* of depth *d for $\mathcal{T}'_>$ and $\mathcal{T}'$ if for all evaluation steps $(\ell, \sigma) \to_t (\ell', \sigma')$:*

(a) If $t \in \mathcal{T}'_>$, then we have $\sigma\left(f_{i-1}(\ell)\right) + \sigma\left(f_i(\ell)\right) \geq \sigma'\left(f_i(\ell')\right) + 1$ for all $1 \leq i \leq d$ and $\sigma\left(f_d(\ell)\right) \geq 0$.
(b) If $t \in \mathcal{T}' \setminus \mathcal{T}'_>$, then we have $\sigma\left(f_i(\ell)\right) \geq \sigma'\left(f_i(\ell')\right)$ for all $1 \leq i \leq d$.

Here, we set $f_0(\ell) = 0$ for all $\ell \in \mathcal{L}$. We say that $\mathcal{T}' \setminus \mathcal{T}'_>$ is the set of non-increasing *transitions and $\mathcal{T}'_>$ is the set of* decreasing *transitions of the MΦRF f.*

The definitions of MΦRFs and of linear ranking functions coincide in the special case of a single phase (i.e., if $d = 1$). Note that for $d > 1$, the requirement for decreasing transitions in (a) does not imply the requirement for non-increasing transitions in (b). The reason is that for decreasing transitions, f_i may increase in the beginning (if f_{i-1} is large enough), because eventually f_{i-1} will become negative. In contrast, for non-increasing transitions, (b) prohibits any increase of f_i, since the MΦRF does not represent any bound on the number of applications of these non-increasing transitions. Thus, we cannot replace (b) by $\sigma\left(f_{i-1}(\ell)\right) + \sigma\left(f_i(\ell)\right) \geq \sigma'\left(f_i(\ell')\right)$, because then such transitions might make f_i arbitrarily large if their repeated application does not change a positive f_{i-1}.

Example 15. Consider again the integer program in Fig. 4 and let $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$. (See Algorithm 1 for our heuristic to choose $\mathcal{T}'_>$ and $\mathcal{T}'$.) An execution of the loop $\mathcal{T}'_> = \{t_2\}$ has two phases: In the first phase, both x and y are positive. In every iteration, x increases until y is 0. The second phase starts when y is negative. This phase ends when x is negative, since then the guard $x > 0$ is not satisfied anymore. We now show that the tuple (f_1, f_2) is an MΦRF for $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$ where $f_1(\ell_1) = f_1(\ell_2) = y + 1$ and $f_2(\ell_1) = f_2(\ell_2) = x$.

Since t_2 has the update function η with $\eta(x) = x + y$ and $\eta(y) = y - 1$, for any evaluation step $(\ell_2, \sigma) \to_{t_2} (\ell_2, \sigma')$, we have $\sigma'(x) = \sigma(\eta(x)) = \sigma(x) + \sigma(y)$ and $\sigma'(y) = \sigma(\eta(y)) = \sigma(y) - 1$. Hence, $\sigma\left(f_0(\ell_2)\right) + \sigma\left(f_1(\ell_2)\right) = 0 + \sigma\left(y + 1\right) = \sigma\left(y\right) + 1 = \sigma'\left(y + 1\right) + 1 = \sigma'\left(f_1(\ell_2)\right) + 1$ and $\sigma\left(f_1(\ell_2)\right) + \sigma\left(f_2(\ell_2)\right) = \sigma\left(y + 1\right) + \sigma\left(x\right) = \sigma\left(x\right) + \sigma\left(y\right) + 1 = \sigma'\left(x\right) + 1 = \sigma'\left(f_2(\ell_2)\right) + 1$. Moreover, due to the guard $x > 0$, $\sigma\left(x > 0\right) = \texttt{true}$ implies $\sigma\left(f_2(\ell_2)\right) = \sigma\left(x\right) \geq 0$. Note that neither $y + 1$ (as y is not bounded) nor x (as x might increase) are ranking functions for t_2.

Similarly, since the update function η of t_3 does not modify x and y, for every evaluation step $(\ell_2, \sigma) \to_{t_3} (\ell_1, \sigma')$, we have $\sigma'(x) = \sigma(\eta(x)) = \sigma(x)$ and $\sigma'(y) = \sigma(\eta(y)) = \sigma(y)$. Hence, $\sigma\left(f_1(\ell_2)\right) = \sigma\left(y + 1\right) = \sigma\left(y\right) + 1 = \sigma'\left(y + 1\right) = \sigma'\left(f_1(\ell_1)\right)$ and $\sigma\left(f_2(\ell_2)\right) = \sigma\left(x\right) = \sigma'\left(x\right) = \sigma'\left(f_2(\ell_1)\right)$.

3.2 Computing Runtime Bounds

We now show how to compute runtime bounds using MΦRFs. As in [16], for a sub-program $\mathcal{T}'$, the *entry transitions of a location* ℓ are all transitions outside $\mathcal{T}'$ which reach ℓ. The *entry locations of* $\mathcal{T}'$ are all locations where an evaluation of the sub-program $\mathcal{T}'$ can begin. Finally, the *entry transitions of* $\mathcal{T}'$ are all entry transitions to entry locations of $\mathcal{T}'$.

Definition 16 (Entry Transitions and Entry Locations). *Let* $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$. *We define the set of* entry transitions of $\ell \in \mathcal{L}$ *as* $\mathcal{T}_\ell = \{t \mid t = (\ell', \tau, \eta, \ell) \wedge t \in \mathcal{T} \setminus \mathcal{T}'\}$. *The set of* entry locations *is* $\mathcal{E}_{\mathcal{T}'} = \{\ell_{in} \mid \mathcal{T}_{\ell_{in}} \neq \varnothing \wedge \exists \ell' : (\ell_{in}, \tau, \eta, \ell') \in \mathcal{T}'\}$. *Finally, the* entry transitions of $\mathcal{T}'$ *are* $\mathcal{ET}_{\mathcal{T}'} = \bigcup_{\ell \in \mathcal{E}_{\mathcal{T}'}} \mathcal{T}_\ell$.

Example 17. Again, consider the integer program in Fig. 4 and $\mathcal{T}' = \{t_2, t_3\}$. Then we have $\mathcal{T}_{\ell_2} = \{t_1\}$, $\mathcal{E}_{\mathcal{T}'} = \{\ell_2\}$, and $\mathcal{ET}_{\mathcal{T}'} = \{t_1\}$.

In [12, Lemma 6], the authors considered programs consisting of a single looping transition and showed that an MΦRF for the loop yields a linear bound on the possible number of its executions. We now generalize their lemma to our modular setting where we regard sub-programs $\mathcal{T}'$ instead of the full program $\mathcal{T}$.[1] The sub-program $\mathcal{T}'$ may contain arbitrary many transitions and loops.

For a start configuration (ℓ, σ) where ℓ is an entry location of $\mathcal{T}'$ and an MΦRF $f = (f_1, \ldots, f_d)$ for $\mathcal{T}'_>$ and $\mathcal{T}'$, Lemma 18 gives a bound $\beta \in \mathbb{N}$ which ensures that whenever there is an evaluation of $\mathcal{T}'$ that begins with (ℓ, σ) and where transitions from $\mathcal{T}'_>$ are applied at least β times, then all ranking functions in f have become negative. As f is an MΦRF (and thus, in every application of a transition from $\mathcal{T}'_>$, some f_i must be decreasing and non-negative), this implies that in any evaluation of $\mathcal{T}'$ starting in (ℓ, σ), transitions from $\mathcal{T}'_>$ can be applied *at most* β times. Since the bound β depends *linearly* on the values $\sigma(f_1(\ell)), \ldots, \sigma(f_d(\ell))$ of the ranking functions in the start configuration (ℓ, σ) and since all ranking functions f_i are linear as well, this means that we have inferred a linear bound on the number of applications of transitions from $\mathcal{T}'_>$. However, this is only a *local* bound w.r.t. the values of the variables at the start of the sub-program $\mathcal{T}'$. We lift these local bounds to global runtime bounds for the full program in Theorem 20. See Appendix A for the proofs of both Lemma 18 and Theorem 20.

Lemma 18 (Local Runtime Bound for Sub-Program). *Let* $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$, $\ell \in \mathcal{E}_{\mathcal{T}'}$, $\sigma \in \Sigma$, *and let* $f = (f_1, \ldots, f_d)$ *be an MΦRF for* $\mathcal{T}'_>$ *and* $\mathcal{T}'$. *For all* $1 \leq i \leq d$, *we define the constants* $\gamma_i \in \mathbb{Q}$ *and* $\beta \in \mathbb{N}$ *with* $\gamma_i, \beta > 0$:

- $\gamma_1 = 1$ *and* $\gamma_i = 2 + \frac{\gamma_{i-1}}{i-1} + \frac{1}{(i-1)!}$ *for* $i > 1$
- $\beta = 1 + d! \cdot \gamma_d \cdot \max\{0, \sigma(f_1(\ell)), \ldots, \sigma(f_d(\ell))\}$

Then for any evaluation $(\ell, \sigma) \, (\to^*_{\mathcal{T}' \setminus \mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})^n \, (\ell', \sigma')$ *with* $n \geq \beta$ *and any* $1 \leq i \leq d$, *we have* $\sigma'(f_i(\ell')) < 0$.

[1] So in the special case where $\mathcal{T}'_> = \mathcal{T}'$ and $\mathcal{T}'$ is a singleton, our Lemma 18 corresponds to [12, Lemma 6] for nested MΦRFs.

Note that the constants γ_i do not depend on the program or the MΦRF, and the factor $d! \cdot \gamma_d$ only depends on the depth d.

Example 19. Reconsider the MΦRF $= (f_1, f_2)$ that we found for $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$ in Example 15. The constants of Lemma 18 are $\gamma_1 = 1$ and $\gamma_2 = 2 + \frac{1}{1} + \frac{1}{1} = 4$. Thus, when $\mathcal{T}'$ is interpreted as a standalone program, then transition t_2 can be executed at most $\beta = 1 + 2! \cdot \gamma_2 \cdot \max\{0, \sigma\left(f_1(\ell_2)\right), \sigma\left(f_2(\ell_2)\right)\} = 1 + 8 \cdot \max\left\{0, \sigma\left(y+1\right), \sigma\left(x\right)\right\}$ many times when starting in $\sigma \in \Sigma$.

Lemma 18 yields the runtime bound

$$1 + d! \cdot \gamma_d \ \cdot \ \max\{0, f_1(\ell), \ldots, f_d(\ell)\} \tag{1}$$

for the transitions $\mathcal{T}'_>$ in the standalone program consisting of the transitions $\mathcal{T}'$. However, (1) is not yet a bound from $\mathcal{B}$, because it contains "max" and because the polynomials $f_i(\ell)$ may have negative coefficients. To transform polynomials into (weakly monotonically increasing) bounds, we replace their coefficients by their absolute values (and denote this transformation by $\lceil \cdot \rceil$). So for example we have $\lceil -x + 2 \rceil = |-1| \cdot x + |2| = x + 2$. Moreover, to remove "max", we replace it by addition. In this way, we obtain the bound

$$\beta_\ell \ = \ 1 + d! \cdot \gamma_d \ \cdot \ (\lceil f_1(\ell) \rceil + \ldots + \lceil f_d(\ell) \rceil).$$

In an evaluation of the full program, we enter a sub-program $\mathcal{T}'$ by an entry transition $t \in \mathcal{T}_\ell$ to an entry location $\ell \in \mathcal{E}_{\mathcal{T}'}$. As explained in Sect. 2, to lift the local runtime bound β_ℓ for $\mathcal{T}'_>$ to a global bound, we have to instantiate the variables in β_ℓ by (over-approximations of) the values that the variables have when reaching the sub-program $\mathcal{T}'$, i.e., after the transition t. To this end, we use the size bound $\mathcal{SB}(t, v)$ which over-approximates the largest absolute value of v after the transition t. We also use the shorthand notation $\mathcal{SB}(t, \cdot) : \mathcal{PV} \to \mathcal{B}$, where $\mathcal{SB}(t, \cdot)(v)$ is defined to be $\mathcal{SB}(t, v)$ and for every arithmetic expression b, $\mathcal{SB}(t, \cdot)(b)$ results from b by replacing each variable v in b by $\mathcal{SB}(t, v)$. Hence, $\mathcal{SB}(t, \cdot)(\beta_\ell)$ is a (global) bound on the number of applications of transitions from $\mathcal{T}'_>$ if $\mathcal{T}'$ is entered once via the entry transition t. Here, weak monotonic increase of β_ℓ ensures that the over-approximation of the variables v in β_ℓ by $\mathcal{SB}(t, v)$ indeed leads to an over-approximation of $\mathcal{T}'_>$'s runtime.

However, for every entry transition t we also have to take into account how often the sub-program $\mathcal{T}'$ may be entered via t. We can over-approximate this value by $\mathcal{RB}(t)$. This leads to Theorem 20 which generalizes a result from [16] to MΦRFs. The analysis starts with a runtime bound $\mathcal{RB}$ and a size bound $\mathcal{SB}$ which map all transitions resp. result variables to ω, except for the transitions t which do not occur in cycles of *mathcalT*, where $\mathcal{RB}(t) = 1$. Afterwards, $\mathcal{RB}$ and $\mathcal{SB}$ are refined repeatedly. Instead of using a single ranking function for the refinement of $\mathcal{RB}$ as in [16], Theorem 20 now allows us to replace $\mathcal{RB}$ by a refined bound $\mathcal{RB}'$ based on an MΦRF.

Theorem 20 (Refining Runtime Bounds Based on MΦRFs). *Let $\mathcal{RB}$ be a runtime bound, $\mathcal{SB}$ a size bound, and $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ does not* *contain any initial transitions. Let $f = (f_1, \ldots, f_d)$ be an MΦRF for $\mathcal{T}'_>$ and $\mathcal{T}'$. For any entry location $\ell \in \mathcal{E}_{\mathcal{T}'}$ we define $\beta_\ell = 1 + d! \cdot \gamma_d \cdot (\lceil f_1(\ell) \rceil + \ldots + \lceil f_d(\ell) \rceil)$, where γ_d is as in Lemma 18. Then $\mathcal{RB}'$ is also a runtime bound, where we define $\mathcal{RB}'$ by $\mathcal{RB}'(t) = \mathcal{RB}(t)$ for all $t \notin \mathcal{T}'_>$ and*

$$\mathcal{RB}'(t_>) = \sum\nolimits_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum\nolimits_{t \in \mathcal{T}_\ell} \mathcal{RB}(t) \cdot \mathcal{SB}(t, \cdot)\,(\beta_\ell) \quad \textit{for all } t_> \in \mathcal{T}'_>.$$

Example 21. We use Theorem 20 to compute a runtime bound for t_2 in Fig. 4. In Example 17, we showed that $\mathcal{E}_{\{t_2, t_3\}} = \{\ell_2\}$ and $\mathcal{T}_{\ell_2} = \{t_1\}$. Thus, we obtain

$$\mathcal{RB}(t_2) = \mathcal{RB}(t_1) \cdot \mathcal{SB}(t_1, \cdot)\,(\beta_{\ell_2}).$$

Using our calculations from Example 19 we have $\beta_{\ell_2} = 1 + 2! \cdot \gamma_2 \cdot (\lceil f_1(\ell_2) \rceil + \lceil f_2(\ell_2) \rceil) = 1 + 8 \cdot (y + 1 + x) = 8 \cdot x + 8 \cdot y + 9$.

We use the runtime bound $\mathcal{RB}(t_1) = z$ and the size bounds $\mathcal{SB}(t_1, x) = \mathcal{SB}(t_1, y) = 2 \cdot z$ from Example 10 and Example 13 and get $\mathcal{RB}(t_2) = \mathcal{RB}(t_1) \cdot (8 \cdot \mathcal{SB}(t_1, x) + 8 \cdot \mathcal{SB}(t_1, y) + 9) = z \cdot (8 \cdot 2 \cdot z + 8 \cdot 2 \cdot z + 9) = 32 \cdot z^2 + 9 \cdot z$.

By Corollary 11 and Example 10, the runtime complexity of the program in Fig. 4 is at most $\sum_{j=0}^{3} \mathcal{RB}(t_j) = 1 + z + 32 \cdot z^2 + 9 \cdot z + z = 32 \cdot z^2 + 11 \cdot z + 1$, resp. $\mathrm{rc}(\sigma_0) \leq 32 \cdot |\sigma_0(z)|^2 + 11 \cdot |\sigma_0(z)| + 1$, i.e., the program's runtime complexity is at most quadratic in the initial absolute value of z. Thus, in contrast to [16], we can now infer a finite bound on the runtime complexity of this program.

3.3 Complete Algorithm

Based on Theorem 20, in Algorithm 1 we present our complete algorithm which improves the approach for complexity analysis of integer programs from [16] by using MΦRFs to infer runtime bounds. As mentioned in Sect. 2, the computation of size bounds from [16] is used as a black box. We just take the alternating repeated improvement of runtime and size bounds into account. So in particular, size bounds are updated when runtime bounds have been updated (Lines 12 and 14).

First, we preprocess the program (Line 1) by eliminating unreachable locations and transitions with unsatisfiable guards, and infer program invariants using the Apron library [34]. In addition, we remove variables which clearly do not have an impact on the termination behavior of the program. Then, we set all runtime bounds for transitions outside of cycles to 1, and all other bounds to ω initially (Line 2).

For the computation of an MΦRF, the considered subset $\mathcal{T}'$ has to be chosen heuristically. We begin with regarding a strongly connected component[2] (SCC) $\widetilde{\mathcal{T}}$ of the program graph in Line 3. Then we try to generate an MΦRF, and choose $\mathcal{T}'$ to consist of a maximal subset of $\widetilde{\mathcal{T}}$ where all transitions are non-increasing

[2] As usual, a graph is *strongly connected* if there is a path from every node to every other node. A *strongly connected component* is a maximal strongly connected subgraph.

```
Input: An integer program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$
1  Preprocess $\mathcal{P}$
2  Create an initial runtime bound $\mathcal{RB}$ and an initial size bound $\mathcal{SB}$ and set $d \leftarrow 1$
3  forall SCCs $\widetilde{\mathcal{T}}$ without initial transitions of $\mathcal{P}$ in topological order do
4      repeat
5          forall $t_> \in \widetilde{\mathcal{T}}$ with $\mathcal{RB}(t_>) = \omega$ do
6              repeat
7                  Search for an M$\Phi$RF with depth $d$ for a maximal subset $\mathcal{T}' \subseteq \widetilde{\mathcal{T}}$
                     that has a subset $\mathcal{T}'_> \subseteq \mathcal{T}'$ with $t_> \in \mathcal{T}'_>$
                     such that all transitions in $\mathcal{T}'_>$ are decreasing
                     and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing
8                  $d \leftarrow d + 1$
9              until M$\Phi$RF was found or $d > mdepth$
10             if M$\Phi$RF was found then
11                 Update $\mathcal{RB}(t)$ for all $t \in \mathcal{T}'_>$ using Theorem 20
12         Update all size bounds for transitions in $\widetilde{\mathcal{T}}$ and reset $d \leftarrow 1$
13     until No runtime or size bound improved
14     Update all size bounds for outgoing transitions of $\widetilde{\mathcal{T}}$.
Output: Runtime Bound $\mathcal{RB}$ and Size Bound $\mathcal{SB}$
```

Algorithm 1: Inferring Global Runtime and Size Bounds

or decreasing (and at least one of the unbounded transitions $t_>$ is decreasing). So for the program in Fig. 4, we would start with $\widetilde{\mathcal{T}} = \{t_1, t_2, t_3\}$, but when trying to generate an MΦRF for $\mathcal{T}'_> = \{t_2\}$, we can only make t_2 decreasing and t_3 non-increasing. For that reason, we then set $\mathcal{T}'$ to $\{t_2, t_3\}$.

We treat the SCCs in topological order such that improved bounds for previous transitions are already available when handling the next SCC. If an MΦRF was found, we update the runtime bound for all $t \in \mathcal{T}'_>$ using Theorem 20 (Line 11). If we do not find any MΦRF of the given depth that makes $t_>$ decreasing, we increase the depth and continue the search until we reach a fixed *mdepth*. We abort the computation of runtime bounds if no bound has been improved. Here, we use a heuristic which compares polynomial bounds by their degrees.

Finally, let us elaborate on the choice of *mdepth*. For example, if *mdepth* is 1, then we just compute linear ranking functions. If *mdepth* is infinity, then we cannot guarantee that our algorithm always terminates. For certain classes of programs, it is possible to give a bound on *mdepth* such that if there is an MΦRF for the program, then there is also one of depth *mdepth* [11,12,50]. However, it is open whether such a bound is computable for general integer programs. As the amount of time the SMT solver needs to find an MΦRF increases with the depth, we decided to use 5 as a fixed maximal depth, which performed well in our examples. Still, we provide the option for the user to change this bound.

4 Improving Bounds by Control-Flow Refinement

Now we present another technique to improve the automated complexity analysis of integer programs, so-called *control-flow refinement*. The idea is to transform a program $\mathcal{P}$ into a new program $\mathcal{P}'$ which is "easier" to analyze. Of course, we ensure that the runtime complexity of $\mathcal{P}'$ is *at least* the runtime complexity of $\mathcal{P}$. Then it is sound to analyze upper runtime bounds for $\mathcal{P}'$ instead of $\mathcal{P}$.

Our approach is based on the *partial evaluation* technique of [20]. For termination analysis, [20] shows how to use partial evaluation of *constrained Horn clauses* locally on every SCC of the program graph. But for complexity analysis, [20] only discusses *global* partial evaluation as a preprocessing step for complexity analysis. In Sect. 4.1, we formalize the partial evaluation technique of [20] such that it operates directly on SCCs of integer programs and prove that it is sound for complexity analysis. Afterwards, we improve its locality further in Sect. 4.2 such that partial evaluation is only applied *on-demand* on those transitions of an integer program where our current runtime bounds are "not yet good enough". Our experimental evaluation in Sect. 5 shows that our local partial evaluation techniques of Sect. 4.1 and Sect. 4.2 lead to a significantly stronger tool than when performing partial evaluation only globally as a preprocessing step.

As indicated in Sect. 1, the loop in Fig. 2 can be transformed into two consecutive loops (see Fig. 3). The first loop in Fig. 3 covers the case $x < 0 \wedge y < z$ and the second one covers the case $x < 0 \wedge y \geq z$. These cases correspond to the conjunction of the loop guard with the conditions of the two branches of the **if**-instruction. Here, partial evaluation detects that these cases occur *after* each other, i.e., if $y \geq z$, then the case $y < z$ does not occur again afterwards. In Example 22, we illustrate how our algorithm for partial evaluation performs this transformation. In the refined program of Fig. 3, it is easy to see that the runtime complexity is at most linear. Thus, the original loop has at most linear runtime complexity as well. Note that our tool KoAT can also infer a linear bound for both programs corresponding to Fig. 2 and 3 *without* control-flow refinement. In fact, for these examples it suffices to use just linear ranking functions, i.e., Theorem 20 with MΦRFs of depth $d = 1$. Still, we illustrate partial evaluation using this small example to ease readability. We will discuss the relationship between MΦRFs and partial evaluation at the end of Sect. 4.2, where we also show examples to demonstrate that these techniques do not subsume each other (see Example 26 and 27).

```
    **Input:** A program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and a non-trivial SCC $\mathcal{T}_{SCC} \subseteq \mathcal{T}$
 1  $\mathcal{L}_1 \leftarrow \{\langle \ell', \mathtt{true} \rangle \mid \ell' \in \mathcal{E}_{\mathcal{T}_{SCC}}\}$
 2  $\mathcal{T}_{res} \leftarrow \{(\ell, \tau, \eta, \langle \ell', \mathtt{true} \rangle) \mid \ell' \in \mathcal{E}_{\mathcal{T}_{SCC}} \wedge (\ell, \tau, \eta, \ell') \in \mathcal{T} \setminus \mathcal{T}_{SCC}\}$
 3  $\mathcal{L}_0 \leftarrow \varnothing$, $\mathcal{L}_{done} \leftarrow \varnothing$
 4  repeat
 5      $\mathcal{L}_0 \leftarrow \mathcal{L}_1$, $\mathcal{L}_1 \leftarrow \varnothing$
 6      forall $\langle \ell, \varphi \rangle \in \mathcal{L}_0$ do
 7          forall $(\ell, \tau, \eta, \ell') \in \mathcal{T}_{SCC}$ do
 8              Compute $\varphi_{new}$ from $\varphi$, $\tau$, and $\eta$ such that $\models (\varphi \wedge \tau) \to \eta(\varphi_{new})$
 9              if $\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle \notin \mathcal{L}_{done}$ then
10                  $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \cup \{\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle\}$
11              $\mathcal{T}_{res} \leftarrow \mathcal{T}_{res} \cup \{(\langle \ell, \varphi \rangle, \varphi \wedge \tau, \eta, \langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle)\}$
12          forall $(\ell, \tau, \eta, \ell') \in \mathcal{T} \setminus \mathcal{T}_{SCC}$ do
13              $\mathcal{T}_{res} \leftarrow \mathcal{T}_{res} \cup \{(\langle \ell, \varphi \rangle, \varphi \wedge \tau, \eta, \ell')\}$
14          $\mathcal{L}_{done} \leftarrow \mathcal{L}_{done} \cup \{\langle \ell, \varphi \rangle\}$
15  until $\mathcal{L}_1 = \varnothing$
    **Output:** $\mathcal{P}' = (\mathcal{PV}, (\mathcal{L} \setminus \{\ell \mid \ell$ occurs as source or target in $\mathcal{T}_{SCC}\}) \cup \mathcal{L}_{done}, \ell_0,$
                $(\mathcal{T} \setminus \{(\ell, \tau, \eta, \ell') \mid \ell$ or $\ell'$ occurs as source or target in $\mathcal{T}_{SCC}\}) \cup \mathcal{T}_{res})$
```

Algorithm 2: Partial Evaluation for an SCC

4.1 SCC-Based Partial Evaluation

We now formalize the partial evaluation of [20] as an SCC-based refinement technique for integer programs in Algorithm 2 and show its correctness for complexity analysis in Theorem 24. The intuitive idea of Algorithm 2 is to refine a non-trivial[3] SCC $\mathcal{T}_{SCC}$ of an integer program into multiple SCCs by considering "abstract" evaluations which do not operate on concrete states but on sets of states. These sets of states are characterized by constraints, i.e., a constraint φ stands for all states σ with $\sigma(\varphi) = \mathtt{true}$. To this end, we label every location ℓ in the SCC by a constraint $\varphi \in \mathcal{C}(\mathcal{PV})$ which describes (a superset of) those states σ which can occur in this location. So all reachable configurations with the location ℓ have the form (ℓ, σ) such that $\sigma(\varphi) = \mathtt{true}$. We begin with labeling the entry locations of $\mathcal{T}_{SCC}$ by the constraint $\mathtt{true}$. The constraints for the other locations in the SCC are obtained by considering how the updates of the transitions affect the constraints of their source locations and their guards. The pairs of locations and constraints then become the new locations in the refined program.

Since locations can be reached by different paths, the same location may get different constraints, i.e., partial evaluation can transform a former location ℓ into several new locations $\langle \ell, \varphi_1 \rangle, \ldots, \langle \ell, \varphi_n \rangle$. So the constraints are not necessarily invariants that hold for *all* evaluations that reach a location ℓ but instead of "widening" (or "generalizing") constraints when a location can be reached by different states, we perform a case analysis and split up a location ℓ according to the different sets of states that may reach ℓ.

After labeling every entry location ℓ of $\mathcal{T}_{SCC}$ by the constraint $\mathtt{true}$ in Line 1 of Algorithm 2, we modify the entry transitions to ℓ such that they now reach the new location $\langle \ell, \mathtt{true} \rangle$ instead (Line 2). The sets $\mathcal{L}_0$ and $\mathcal{L}_1$ (the new locations

[3] As usual, an SCC is *non-trivial* if it contains at least one transition.

whose outgoing transitions need to be processed) and $\mathcal{L}_{done}$ (the new locations whose outgoing transitions were already processed) are used for bookkeeping. We then apply partial evaluation in Lines 4 to 15 until there are no new locations with transitions to be processed anymore (see Line 15).

In each iteration of the outer loop in Line 4, the transitions of the current new locations in $\mathcal{L}_1$ are processed. To this end, $\mathcal{L}_0$ is set to $\mathcal{L}_1$ and $\mathcal{L}_1$ is set to $\varnothing$ in Line 5. During the handling of the locations in $\mathcal{L}_0$, we might create new locations and these will be stored in $\mathcal{L}_1$ again.

We handle all locations $\langle \ell, \varphi \rangle$ in $\mathcal{L}_0$ (Line 6) by using all outgoing transitions $(\ell, \tau, \eta, \ell')$. We first consider those transitions which are part of the considered SCC (Line 7), whereas the transitions which leave the SCC are handled in Line 12.

The actual partial evaluation step is in Line 8. Given a new location $\langle \ell, \varphi \rangle$ and a transition $t = (\ell, \tau, \eta, \ell')$, we compute a constraint φ_{new} which over-approximates the set of states that can result from those states that satisfy the constraint φ and the guard τ of the transition when applying the update η. More precisely, φ_{new} has to satisfy $\models (\varphi \wedge \tau) \to \eta(\varphi_{new})$, i.e., $(\varphi \wedge \tau) \to \eta(\varphi_{new})$ is a tautology. For example, if $\varphi = (x = 0)$, $\tau = \texttt{true}$, and $\eta(x) = x - 1$, we derive $\varphi_{new} = (x = -1)$. However, if we now created the new location $\langle \ell', \varphi_{new} \rangle$, this might lead to non-termination of our algorithm. The reason is that if ℓ' is within a loop, then whenever one reaches ℓ' again, one might obtain a new constraint. In this way, one would create infinitely many new locations $\langle \ell', \varphi_1 \rangle, \langle \ell', \varphi_2 \rangle, \ldots$. For instance, if in our example the transition with the update $\eta(x) = x - 1$ is a self-loop, then we would derive further new locations with the constraints $x = -2$, $x = -3$, etc.

To ensure that every former location ℓ' only gives rise to finitely many new locations $\langle \ell', \varphi \rangle$, we perform *property-based abstraction* as in [20,28]: For every location ℓ' we use a finite so-called abstraction layer $\alpha_{\ell'} \subseteq \{e_1 \leq e_2 \mid e_1, e_2 \in \mathbb{Z}[\mathcal{PV}]\}$. So $\alpha_{\ell'}$ is a finite set of atomic constraints (i.e., of polynomial inequations). Then $\alpha_{\ell'}$ is extended to a function on constraints such that $\alpha_{\ell'}(\varphi_{new}) = \varphi'_{new}$ where φ'_{new} is a conjunction of inequations from $\alpha_{\ell'}$ and $\models \varphi_{new} \to \varphi'_{new}$. This guarantees that partial evaluation terminates, but it can lead to an exponential blow-up, since for every location ℓ' there can now be $2^{|\alpha_{\ell'}|}$ many possible constraints. In our example, instead of the infinitely many inequations $x = 0, x = -1, x = -2, \ldots$ the abstraction layer might just contain the inequation $x \leq 0$. Then we would only obtain the new location with the constraint $x \leq 0$.

Afterwards, in Lines 9 and 10 we add the new location $\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle$ to $\mathcal{L}_1$ if it was not processed before. Moreover, the transition $(\ell, \tau, \eta, \ell')$ which we used for the refinement must now get the new location as its target (Line 11) and $\langle \ell, \varphi \rangle$ as its source. In addition, we extend the transition's guard τ by φ.

Finally, we also have to process the transitions $(\ell, \tau, \eta, \ell')$ which leave the SCC. Thus, we replace the source transition ℓ by $\langle \ell, \varphi \rangle$ and again extend the guard τ of the transition by the constraint φ in Lines 12 and 13. Since we have now processed all outgoing transitions of $\langle \ell, \varphi \rangle$ we can add it to $\mathcal{L}_{done}$ in Line 14.

In the end, we output the program where the considered SCC and all transitions in or out of this SCC were refined (and thus, have to be removed from the original program). We now illustrate Algorithm 2 using the program from Fig. 2.

Example 22. Figure 5 represents the program from Fig. 2 in our formalism for integer programs. Here, we used an explicit location ℓ_3 for the end of the program to illustrate how Algorithm 2 handles transitions which leave the SCC.

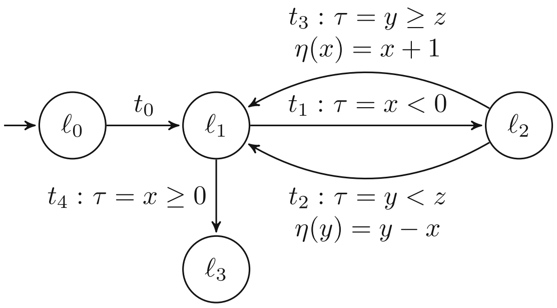


Fig. 5. Integer Program Corresponding to Fig. 2

We apply Algorithm 2 to the program in Fig. 5 and refine the SCC $\mathcal{T}_{SCC} = \{t_1, t_2, t_3\}$. The entry location is $\mathcal{E}_{\mathcal{T}_{SCC}} = \{\ell_1\}$. To increase readability, let τ_i be the guard and η_i be the update of transition t_i for all $0 \leq i \leq 3$.

For the abstraction layers, we choose[4] $\alpha_{\ell_1} = \alpha_{\ell_2} = \{x < 0, y \geq z\}$. It is not necessary to define abstraction layers for ℓ_0 and ℓ_3, as they are not part of the SCC. So for any constraint φ_{new} and $i \in \{1, 2\}$, $\alpha_{\ell_i}(\varphi_{new})$ can only be a conjunction of the inequations in α_{ℓ_i} (i.e., $\alpha_{\ell_i}(\varphi_{new})$ is $\texttt{true}$, $x < 0$, $y \geq z$, or $x < 0 \wedge y \geq z$, where $\texttt{true}$ corresponds to the empty conjunction).

Since t_0 is the only entry transition to the entry location ℓ_1, we initialize $\mathcal{T}_{res}$ with $\{(\ell_0, \tau_0, \eta_0, \langle \ell_1, \texttt{true} \rangle)\}$ and $\mathcal{L}_1$ with $\{\langle \ell_1, \texttt{true} \rangle\}$.

In the first iteration $\mathcal{L}_0$ only consists of $\langle \ell_1, \texttt{true} \rangle$. We have two possible transitions which we can apply in ℓ_1: $t_1 = (\ell_1, \tau_1, \eta_1, \ell_2) \in \mathcal{T}_{SCC}$ or $t_4 = (\ell_1, \tau_4, \eta_4, \ell_3) \in \mathcal{T} \setminus \mathcal{T}_{SCC}$. We start with transition t_1. Since the update η_1 is the identity, from the guard $\tau_1 = (x < 0)$ we obtain the resulting constraint $\varphi_{new} = (x < 0)$. We apply the abstraction layer and get $\alpha_{\ell_2}(x < 0) = (x < 0)$ because $\models (x < 0) \to (x < 0)$. Now we add the new transition

$$(\langle \ell_1, \texttt{true} \rangle, \texttt{true} \wedge \tau_1, \eta_1, \langle \ell_2, x < 0 \rangle)$$

to $\mathcal{T}_{res}$ and $\langle \ell_2, x < 0 \rangle$ to $\mathcal{L}_1$. For transition $t_4 = (\ell_1, \tau_4, \eta_4, \ell_3)$, we update its source location and get the resulting transition

$$(\langle \ell_1, \texttt{true} \rangle, \texttt{true} \wedge \tau_4, \eta_4, \ell_3)$$

in $\mathcal{T}_{res}$. We add $\langle \ell_1, \texttt{true} \rangle$ to $\mathcal{L}_{done}$. Now $\mathcal{L}_1$ consists of $\langle \ell_2, x < 0 \rangle$. There are two transitions t_2 and t_3 which can be applied in ℓ_2. For t_2, from the previous constraint $x < 0$ and the guard $\tau_2 = (y < z)$ we can infer that after the update $\eta_2(y) = y - x$ we have $x < 0 \wedge y < z - x$. As the abstraction layer α_{ℓ_1} consists of

[4] In [20], different heuristics are presented to choose such abstraction layers. In our implementation, we use these heuristics as a black box.

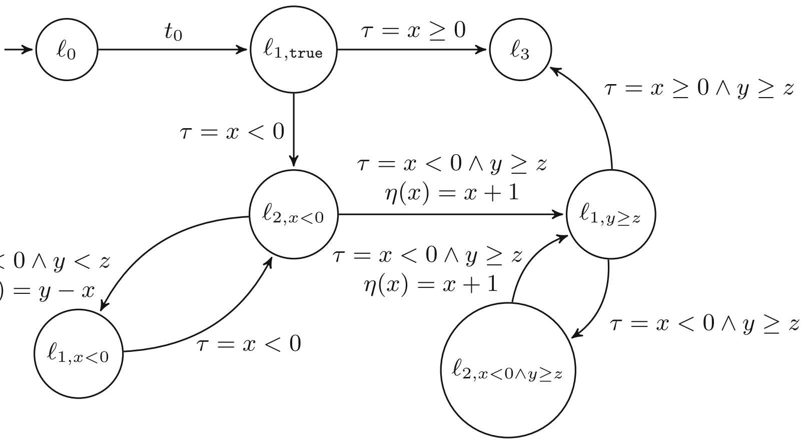


Fig. 6. Applying Partial Evaluation to Fig. 5

$x < 0$ and $y \geq z$, we have $\alpha_{\ell_1}(x < 0 \land y < z - x) = x < 0$, since $\not\models (x < 0 \land y < z - x) \to (y \geq z)$. Thus, we add the new transition

$$(\langle \ell_2, x < 0\rangle, x < 0 \land \tau_2, \eta_2, \langle \ell_1, x < 0\rangle)$$

to $\mathcal{T}_{res}$ and $\langle \ell_1, x < 0\rangle$ to the set $\mathcal{L}_1$. Similarly, for t_3, from $x < 0$ and the guard $\tau_3 = (y \geq z)$ we infer that after $\eta_3(x) = x + 1$ we have $x < 1 \land y \geq z$. Here, $\alpha_{\ell_1}(x < 1 \land y \geq z) = y \geq z$, since $\not\models (x < 1 \land y \geq z) \to (x < 0)$. Hence, we add

$$(\langle \ell_2, x < 0\rangle, x < 0 \land \tau_3, \eta_3, \langle \ell_1, y \geq z\rangle)$$

to $\mathcal{T}_{res}$ and $\langle \ell_1, y \geq z\rangle$ to $\mathcal{L}_1$. So $\mathcal{L}_1$ now consists of $\langle \ell_1, x < 0\rangle$ and $\langle \ell_1, y \geq z\rangle$. For $\langle \ell_1, x < 0\rangle$, in the same way as before we obtain the following new transitions:

$$(\langle \ell_1, x < 0\rangle, x < 0 \land \tau_1, \eta_1, \langle \ell_2, x < 0\rangle)$$
$$(\langle \ell_1, x < 0\rangle, x < 0 \land \tau_4, \eta_4, \ell_3)$$

Note that the guard $x < 0 \land \tau_4$ of the last transition is unsatisfiable. For that reason, we always remove transitions with unsatisfiable guard after partial evaluation was applied. For $\langle \ell_1, y \geq z\rangle$, we obtain the following new transitions:

$$(\langle \ell_1, y \geq z\rangle, y \geq z \land \tau_1, \eta_1, \langle \ell_2, x < 0 \land y \geq z\rangle)$$
$$(\langle \ell_1, y \geq z\rangle, y \geq z \land \tau_4, \eta_4, \ell_3)$$

Thus, $\mathcal{L}_1$ now consists of the new location $\langle \ell_2, x < 0 \land y \geq z\rangle$. For this location, we finally get the following new transitions:

$$(\langle \ell_2, x < 0 \land y \geq z\rangle, x < 0 \land y \geq z \land \tau_2, \eta_2, \langle \ell_1, x < 0 \land y \geq z\rangle)$$
$$(\langle \ell_2, x < 0 \land y \geq z\rangle, x < 0 \land y \geq z \land \tau_3, \eta_3, \langle \ell_1, y \geq z\rangle)$$

Since the guard $x < 0 \wedge y \geq z \wedge \tau_2$ of the penultimate transition is again unsatisfiable, it will be removed. For that reason, then the location $\langle \ell_1, x < 0 \wedge y \geq z \rangle$ will be unreachable and will also be removed.

Figure 6 shows the refined integer program where we wrote $\ell_{i,\varphi}$ instead of $\langle \ell_i, \varphi \rangle$ for readability. Moreover, transitions with unsatisfiable guard or unreachable locations were removed. The first SCC with the locations $\langle \ell_2, x < 0 \rangle$ and $\langle \ell_1, x < 0 \rangle$ is applied *before* the second SCC with the locations $\langle \ell_1, y \geq z \rangle$ and $\langle \ell_2, x < 0 \wedge y \geq z \rangle$. So we have detected that these two SCCs occur *after* each other. Indeed, the integer program in Fig. 6 corresponds to the one in Fig. 3.

Algorithm 2 is sound because partial evaluation transforms a program $\mathcal{P}$ into an *equivalent* program $\mathcal{P}'$. Therefore, it does not change the runtime.

Definition 23 (Equivalence of Programs). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$ be integer programs over $\mathcal{V}$. $\mathcal{P}$ and $\mathcal{P}'$ are* equivalent *iff the following holds for all states $\sigma_0 \in \Sigma$: There is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{k} (\ell, \sigma)$ for some $\sigma \in \Sigma$, some $k \in \mathbb{N}$, and some $\ell \in \mathcal{L}$ iff there is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^{k} (\ell', \sigma)$ for the same $\sigma \in \Sigma$ and $k \in \mathbb{N}$, and some location ℓ'.*

Theorem 24 (Soundness of Partial Evaluation in Algorithm 2). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be an integer program and let $\mathcal{T}_{SCC} \subseteq \mathcal{T}$ be a non-trivial SCC of the program graph. Let $\mathcal{P}'$ be the integer program resulting from applying Algorithm 2 to $\mathcal{P}$ and $\mathcal{T}_{SCC}$. Then $\mathcal{P}$ and $\mathcal{P}'$ are equivalent.*

4.2 Sub-SCC-Based Partial Evaluation

As control-flow refinement may lead to an exponential blow-up of the program, we now present an algorithm where we heuristically minimize the strongly connected part of the program on which we apply partial evaluation (Algorithm 3) and we discuss how to integrate it into our approach for complexity analysis. Our experiments in Sect. 5 show that such a sub-SCC-based partial evaluation leads to significantly shorter runtimes than the SCC-based partial evaluation of Algorithm 2.

The idea of Algorithm 3 is to find a minimal cycle of the program graph containing the transitions $\mathcal{T}_{cfr}$ whose runtime bound we aim to improve by partial evaluation. On the one hand, in this way we minimize the input set $\mathcal{T}_{SCC}$ for the partial evaluation algorithm. On the other hand, we keep enough of the original program's control flow such that partial evaluation can produce useful results.

Our local control-flow refinement technique in Algorithm 3 consists of three parts. In the first loop in Lines 2 to 8, we find a minimal cycle $\mathcal{T}_t$ for each transition t from $\mathcal{T}_{cfr}$. Afterwards, $\mathcal{T}_t$ is extended by all transitions which are parallel to some transition in $\mathcal{T}_t$ in Line 5. Otherwise, we would not be able to correctly insert the refined program afterwards. We add a fresh initial location ℓ_{new}, take all entry transitions to the previously computed cycle and extend $\mathcal{T}_t$ by new corresponding entry transitions which start in ℓ_{new} instead (Lines 6 and 7). We collect all these programs in a set $\mathcal{S}$, where the programs have ℓ_{new} as their initial location.

```
Input: A program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and a non-empty subset $\mathcal{T}_{cfr}$ of a
       non-trivial SCC from $\mathcal{T}$.
1  $\mathcal{S} \leftarrow \varnothing$
2  forall $t = (\ell, \tau, \eta, \ell') \in \mathcal{T}_{cfr}$ do
3      $\mathcal{T}_t \leftarrow$ a shortest path from $\ell'$ to $\ell$
4      $\mathcal{T}_t \leftarrow \mathcal{T}_t \cup \{t\}$
5      $\mathcal{T}_t \leftarrow \mathcal{T}_t \cup \{(\hat{\ell}, \_, \_, \hat{\ell}') \in \mathcal{T} \mid (\hat{\ell}, \_, \_, \hat{\ell}') \in \mathcal{T}_t\}$
6      forall entry transitions $(\overline{\ell}, \overline{\tau}, \overline{\eta}, \overline{\ell}') \in \mathcal{ET}_{\mathcal{T}_t}$ do
7          Add transition $(\ell_{new}, \overline{\tau}, \overline{\eta}, \overline{\ell}')$ to $\mathcal{T}_t$.
8      $\mathcal{S} \leftarrow \mathcal{S} \cup \{(\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}_t)\}$
9  repeat
10     if there exist $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}')$ and $\mathcal{P}'' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}'')$ with
        $\mathcal{P}', \mathcal{P}'' \in \mathcal{S}$, $\mathcal{P}' \neq \mathcal{P}''$, and a location $\ell \neq \ell_{new}$ occurs in both $\mathcal{T}'$ and $\mathcal{T}''$
        then
11         $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{\mathcal{P}', \mathcal{P}''\}) \cup \{(\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}' \cup \mathcal{T}'')\}$
12 until $\mathcal{S}$ does not change anymore
13 forall $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}') \in \mathcal{S}$ do
14     $\mathcal{P}'' = (\mathcal{PV}, \mathcal{L}'', \ell_{new}, \mathcal{T}'') \leftarrow$ apply Alg. 2 to $\mathcal{P}'$ and the single non-trivial
        SCC $\mathcal{T}_{SCC}$ in $\mathcal{T}'$
15     Extend the transitions $\mathcal{T}$ of $\mathcal{P}$ by the transitions $\mathcal{T}''$.
16     forall entry transitions $t = (\ell, \tau, \eta, \ell') \in \mathcal{ET}_{\mathcal{T}'}$ do
17         Replace $t$ by $(\ell, \tau, \eta, \langle \ell', \mathtt{true} \rangle)$ in $\mathcal{P}$.
18     forall outgoing transitions $t = (\ell, \tau, \eta, \ell') \in \mathcal{ET}_{\mathcal{T} \setminus \mathcal{T}'}$ do
19         Replace $t$ by $(\langle \ell, \varphi \rangle, \tau, \eta, \ell')$ in $\mathcal{P}$ for all $\langle \ell, \varphi \rangle \in \mathcal{L}''$.
20 Remove unreachable locations and transitions, and transitions with
    unsatisfiable guard.
Output: Refined program $\mathcal{P}$
```

Algorithm 3: Partial Evaluation for a Subset of an SCC

So for our example from Fig. 5 and $\mathcal{T}_{cfr} = \{t_3\}$, $\mathcal{S}$ only contains one program with locations $\ell_1, \ell_2, \ell_{new}$, transitions t_1, t_2, t_3, and a transition from ℓ_{new} to ℓ_1.

As the next step, in the second loop in Lines 9 to 12, we merge those programs which share a location other than ℓ_{new}. Again, this allows us to correctly insert the refined program afterwards (see the proof of Theorem 25).

The last loop in Lines 13 to 19 performs partial evaluation on each strongly connected part of the programs in $\mathcal{S}$, and inserts the refined programs into the original one by redirecting the entry and the outgoing transitions. Here, an outgoing transition is simply an entry transition of the complement.

At the end of Algorithm 3, one should remove unreachable locations and transitions, as well as transitions with unsatisfiable guard. This is needed, because the refined transitions $\mathcal{T}'$ are simply added to the old transitions $\mathcal{T}$, and entry and outgoing transitions are redirected. So the previous transitions might become unreachable.

Instead of implementing Algorithm 2 ourselves, our complexity analyzer KoAT calls the implementation of [20] in the tool iRankFinder [19] as a backend for partial evaluation.[5] So in particular, we rely on iRankFinder's heuristics to compute the abstraction layers $\alpha_{\ell'}$ and the new constraints φ_{new} resp. $\alpha_{\ell'}(\varphi_{new})$ in Algorithm 2.

So in our example, partial evaluation on the program in $\mathcal{S}$ would result in a program like the one in Fig. 6, but instead of the transition from ℓ_0 to $\ell_{1,\texttt{true}}$ there would be a transition from ℓ_{new} to $\ell_{1,\texttt{true}}$. Moreover, the location ℓ_3 and the transitions to ℓ_3 would be missing. The redirection of the entry and the outgoing transitions would finally create the program from Fig. 6.

The advantage of our technique in contrast to the naïve approach (i.e., applying partial evaluation on the full program as a preprocessing step) and also to the SCC-based approach in Algorithm 2, is that Algorithm 3 allows us to apply partial evaluation "on-demand" just on those transitions where our bounds are still "improvable". Thus, to integrate partial evaluation into our overall approach, Algorithm 1 is modified such that after the treatment of an SCC $\widetilde{\mathcal{T}}$ in Lines 5 to 12, we let $\mathcal{T}_{cfr}$ consists of all transitions $t \in \widetilde{\mathcal{T}}$ where $\mathcal{RB}(t)$ is not linear (and not constant). So this is our heuristic to detect transitions with "improvable" bounds. If $\mathcal{T}_{cfr} \neq \varnothing$, then we call Algorithm 3 to perform partial evaluation and afterwards we execute Lines 5 to 12 of Algorithm 1 once more for the SCC that results from refining $\widetilde{\mathcal{T}}$.

Theorem 25 (Soundness of Partial Evaluation in Algorithm 3). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be a program and $\mathcal{T}_{cfr} \subseteq \mathcal{T}$ a non-empty set of transitions from some non-trivial SCC. Then $\mathcal{P}$ and the program computed by Algorithm 3 are equivalent.*

Both MΦRFs and control-flow refinement detect "phases" of the program. An MΦRF represents these phases via different ranking functions, whereas control-flow refinement makes these phases explicit by modifying the program, e.g., by splitting an SCC into several new ones as in Example 22. Example 26 and Example 27 show that there are programs where one of the techniques allows us to infer a finite bound on the runtime complexity while the other one does not. This is also demonstrated by our experiments with different configurations of KoAT in Sect. 5.

Example 26. For the program corresponding to the loop in Fig. 1 we can only infer a finite runtime bound if we search for MΦRFs of *at least* depth 2. In contrast, control-flow refinement via partial evaluation does not help here, because it does not change the loop. The used MΦRF (f_1, f_2) with $f_1(\ell_1) = f_1(\ell_2) = y + 1$ and $f_2(\ell_1) = f_2(\ell_2) = x$ (see Example 15) corresponds *implicitly* to the case analysis $y \geq 0$ resp. $y < 0$. However, this case analysis is not detected by Algorithm 2, because $y < 0$ only holds after $|y_0| + 1$ executions of this loop if we have $y = y_0$ initially. Thus, this cannot be inferred when evaluating the loop partially for a finite number of times (as this number depends on the initial values of the

[5] To ensure the *equivalence* of the transformed program according to Definition 23, we call iRankFinder with a flag to prevent the "chaining" of transitions. This ensures that partial evaluation does not change the lengths of evaluations.

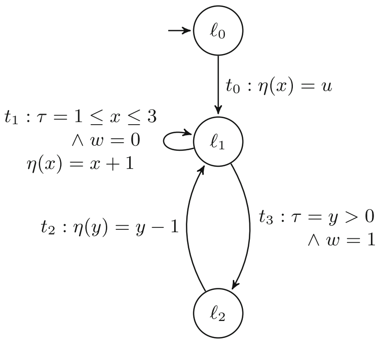


Fig. 7. Original Program

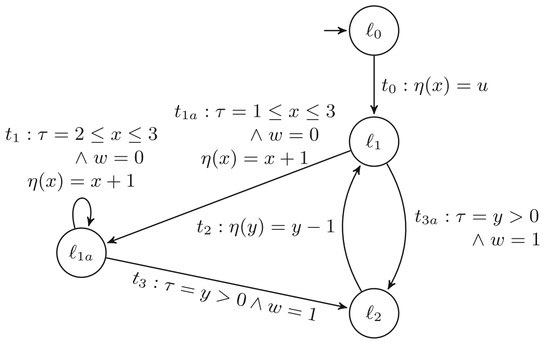


Fig. 8. Result of Algorithm 3 with $\mathcal{T}_{cfr} = \{t_1\}$

variables). As Fig. 1 does not admit a linear ranking function, this means that we fail to infer a finite runtime bound if we only use linear ranking functions and control-flow refinement. The same argument explains why we cannot infer a finite runtime bound for our running example in Fig. 4 (which contains the loop in Fig. 1) with only linear ranking functions and control-flow refinement. For this example, we again need MΦRFs of at least depth 2 (see Example 21). So control-flow refinement via partial evaluation does not subsume MΦRFs.

Example 27. Now we show an example where MΦRFs are not strong enough to infer a finite runtime bound, whereas this is possible using just linear ranking functions (i.e., MΦRFs of depth 1) if we apply partial evaluation before. Moreover, it illustrates Algorithm 3 which only performs partial evaluation on a subset of an SCC.

Consider the program in Fig. 7 where $\mathcal{PV} = \{x, y\}$ are the program variables and $\mathcal{TV} = \{u, w\}$ are the temporary variables. It has two independent components (the self-loop t_1 at location ℓ_1 and the cycle of t_2 and t_3 between ℓ_1 and ℓ_2) which do not influence each other, since t_1 operates only on the variable x and the cycle of t_2 and t_3 depends only on y. The choice which component is evaluated is non-deterministic since it depends on the value of the temporary variable w. Since the value of x is between 1 and 3 in the self-loop, t_1 is only evaluated at most 3 times. Similarly, t_2 and t_3 are each executed at most y times. Hence, the runtime complexity of the program is at most $1 + 3 + 2 \cdot y = 4 + 2 \cdot y$.

However, our approach does not find a finite runtime bound when using only MΦRFs *without* control-flow refinement. To make the transition t_1 in the self-loop decreasing, we need an MΦRF f where the variable x occurs in at least one function f_i of the MΦRF. So $f_i(\ell_1)$ contains x and thus, β_{ℓ_1} (as defined in Theorem 20) contains x as well. When constructing the global bound $\mathcal{RB}(t_1)$ by Theorem 20, we have to instantiate x in β_{ℓ_1} by $\mathcal{SB}(t_0, x)$, i.e., by the size-bound for x of the entry transition t_0. Since x is set to an arbitrary integer value u non-deterministically, its size is unbounded, i.e., $\mathcal{SB}(t_0, x) = \omega$. Thus, Theorem 20 yields $\mathcal{RB}(t_1) = \omega$. The alternative solution of turning t_0 into a non-initial transition and adding it to the subset $\mathcal{T}'$ in Theorem 20 does not work either. Since the value of x after t_0 is an arbitrary integer, t_0 violates the requirement of being non-increasing for every MΦRF where $f_i(\ell_1)$ contains x.

In this example, only the self-loop t_1 is problematic for the computation of runtime bounds. We can directly infer a linear runtime bound for all other transitions, using just linear ranking functions. Thus, when applying control-flow refinement via partial evaluation, according to our heuristic we call Algorithm 3 on just $\mathcal{T}_{cfr} = \{t_1\}$. The result of Algorithm 3 is presented in Fig. 8. Since partial evaluation is restricted to the problematic transition t_1, the other transitions t_2 and t_3 in the SCC remain unaffected, which avoids a too large increase of the program.

As before, in the program of Fig. 8 we infer linear runtime bounds for $t_0, t_2, t_3,$ and t_{3a} using linear ranking functions. To obtain linear bounds for t_{1a} and t_1, we can now use the following MΦRF f of depth 1 for the subset $\mathcal{T}' = \{t_1, t_{1a}, t_3, t_{3a}\}$ and the decreasing transition $\mathcal{T}'_{>} = \{t_{1a}\}$ resp. $\mathcal{T}'_{>} = \{t_1\}$:

$$f(\ell_1) = 3 \qquad f(\ell_{1a}) = 3 - x \qquad f(\ell_2) = 0$$

Thus, while this example cannot be solved by MΦRFs, we can indeed infer linear runtime bounds when using control-flow refinement and just linear ranking functions. Hence, MΦRFs do not subsume control-flow refinement.

5 Evaluation

As mentioned, we implemented both Algorithm 3 and the refined version of Algorithm 1 which calls Algorithm 3 in a new re-implementation of our tool KoAT which is written in OCaml. To find MΦRFs, it uses the SMT Solver Z3 [41] and it uses the tool iRankFinder [19] for the implementation of Algorithm 2 to perform partial evaluation.

To distinguish our re-implementation of KoAT from the original version of the tool from [16], let KoAT1 refer to the tool from [16] and let KoAT2 refer to our new re-implementation. We now evaluate KoAT2 in comparison to the main other state-of-the-art tools for complexity analysis of integer programs: CoFloCo [22,23], KoAT1 [16], Loopus [46], and MaxCore [6]. Moreover, we also evaluate the performance of KoAT1 and KoAT2 when control-flow refinement using iRankFinder [19] is performed on the complete program as a preprocessing step. We do not compare with RaML [33], as it does not support programs whose complexity depends on (possibly negative) integers (see [45]). We also do not compare with PUBS [2], because as stated in [20] by one of the authors of PUBS, CoFloCo is stronger than PUBS. Note that MaxCore is a tool chain which preprocesses the input program and then passes it to either CoFloCo or PUBS for the computation of the bound. As the authors' evaluation in [6] shows that MaxCore with CoFloCo as a backend is substantially stronger than with PUBS as a backend, we only consider the former configuration and refer to it as "MaxCore".

For our evaluation, we use the two sets for complexity analysis of integer programs from the *Termination Problems Data Base (TPDB)* [48] that are used in the annual *Termination and Complexity Competition (TermComp)* [31]: Complexity_ITS (CITS), consisting of integer transition systems, and Complexity_C_Integer (CINT), consisting of C programs with only integer variables. The integers in both CITS and CINT are interpreted as mathematical integers (i.e., without overflows).

Both Loopus and MaxCore only accept C programs as in CINT as input. While it is easily possible to transform the input format of CINT to the input format of CITS automatically, the other direction is not so straightforward. Hence, we compare with Loopus and MaxCore only on the benchmarks from the CINT collection. Our tool KoAT2 is evaluated in 7 different configurations to make the effects of both control-flow refinement and MΦRFs explicit:

1. KoAT2 denotes the configuration which uses Algorithm 1 with maximal depth *mdepth* set to 1, i.e., we only compute linear ranking functions.
2. CFR + KoAT2 first preprocesses the complete program by performing control-flow refinement using iRankFinder. Afterwards, the refined program is analyzed with KoAT2 where $mdepth = 1$.
3. KoAT2 + CFRSCC is the configuration where control-flow refinement is applied to SCCs according to Algorithm 2 and $mdepth = 1$.
4. KoAT2 + CFR uses Algorithm 3 instead to apply control-flow refinement on sub-SCCs and has $mdepth = 1$.
5. KoAT2 + MΦRF5 applies Algorithm 1 with maximal depth $mdepth = 5$, i.e., we use MΦRFs with up to 5 components, but no control-flow refinement.
6. KoAT2 + MΦRF5 + CFRSCC applies control-flow refinement to SCCs (Algorithm 2) and uses $mdepth = 5$.
7. KoAT2 + MΦRF5 + CFR uses sub-SCC control-flow refinement (Algorithm 3) and MΦRFs with maximal depth $mdepth = 5$.

Furthermore, we evaluate the tool KoAT1 in two configurations: KoAT1 corresponds to the standalone version, whereas for CFR + KoAT1, the complete program is first preprocessed using control-flow refinement via the tool iRankFinder before analyzing the resulting program with KoAT1. The second configuration was also used in the evaluation of iRankFinder in [20].

We compare the runtime bounds computed by the tools asymptotically as functions which depend on the largest initial absolute value n of all program variables. All tools were run inside an Ubuntu Docker container on a machine with an AMD Ryzen 7 3700X octa-core CPU and 32 GB of RAM. The benchmarks were evaluated in parallel such that at most 8 containers were running at once, each limited to 1.9 CPU cores. In particular, the runtimes of the tools include the times to start and remove the container. As in *TermComp*, we applied a timeout of 5 min for every program. See [42] for a binary and the source code of our tool KoAT2, a Docker image, web interfaces to test our implementation, and full details on all our experiments in the evaluation.

5.1 Evaluation on Complexity_ITS

The set CITS consists of 781 integer programs, where at most 564 of them *might* have finite runtime (since the tool LoAT [26,27] proves unbounded runtime com-

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + MΦRF5 + CFR	131	255	101	13	6	506	4.31	26.50
KoAT2+MΦRF5+CFRSCC	131	255	102	12	6	506	6.00	27.47
KoAT2 + CFR	131	245	101	10	6	493	5.00	21.81
KoAT2 + CFRSCC	131	245	101	9	6	492	6.37	23.45
KoAT2 + MΦRF5	126	235	100	13	6	480	2.19	13.32
KoAT1	132	214	104	14	5	469	0.65	9.38
CFR + KoAT1	128	231	93	10	5	467	5.54	40.81
CFR + KoAT2	130	231	93	6	6	466	10.44	43.05
CoFloCo	126	231	95	9	0	461	3.44	18.40
KoAT2	126	218	97	10	6	457	2.29	9.45

Fig. 9. Evaluation on Complexity_ITS

plexity for 217 examples). The results of our experiments on this set can be found in Fig. 9. So for example, there are $131 + 255 = 386$ programs where KoAT2 + MΦRF5 + CFR can show that $\text{rc}(\sigma_0) \in \mathcal{O}(n)$ holds for all initial states σ_0 where $|\sigma_0(v)| \leq n$ for all $v \in \mathcal{PV}$. For 131 of these programs, KoAT2 + MΦRF5 + CFR can even show that $\text{rc}(\sigma_0) \in \mathcal{O}(1)$, i.e., their runtime complexity is constant. In Fig. 9, "$< \infty$" is the number of examples where a finite bound on the runtime complexity could be computed by the respective tool within the time limit. "AVG$^+$(s)" is the average runtime of the tool on successful runs in seconds, i.e., where the tool proved a finite time bound before reaching the timeout, whereas "AVG(s)" is the average runtime of the tool on all runs including timeouts.

KoAT2 without MΦRFs and control-flow refinement infers a finite bound for 457 of the 781 examples, while CoFloCo solves 461 and KoAT1 solves 469 examples. In contrast to KoAT2, both CoFloCo and KoAT1 always apply some form of control-flow refinement. However, KoAT1's control-flow refinement is weaker than the one in Sect. 4, since it only performs loop unrolling via "chaining" to combine subsequent transitions. Indeed, when adding control-flow refinement as a preprocessing technique (in CFR + KoAT1 and CFR + KoAT2), the tools are almost equally powerful.

However, for efficiency it is much better to integrate control-flow refinement into KoAT2 as in Algorithm 2 or Algorithm 3 (KoAT2 + CFRSCC resp. KoAT2 + CFR) than to use it as a preprocessing step (CFR + KoAT2). This integration reduces the number of timeouts and therefore increases power. The corresponding configurations already make KoAT2 stronger than all previous tools on this benchmark. Nevertheless, while control-flow improves power substantially, it increases the resulting runtimes. The reason is that partial evaluation can lead to an exponential blow-up of the program. Moreover, we have to analyze parts of the program twice: we first analyze parts where we do not find a linear or a constant bound. Then, we apply control-flow refinement and afterwards, we analyze them again.

If instead of using control-flow refinement, the maximum depth of MΦRFs is increased from 1 to 5, KoAT2 can compute a finite runtime bound for 480 examples. As explained in Sect. 3, MΦRFs are a proper extension of classical

```
while $x > 0$ do
    $x \leftarrow x + y$
    $y \leftarrow y + z$
    $z \leftarrow z - 1$
```

Fig. 10. Loop With Three Phases

$$\begin{tikzcd} {} \arrow[r] & \ell_0 \arrow[r, "t_0"] & \ell_1 \arrow[loop right] \end{tikzcd}$$

$$\begin{aligned} t_1 : \tau &= x > 0 \\ \eta(x) &= x + y \\ \eta(y) &= y + z \\ \eta(z) &= z - 1 \end{aligned}$$

Fig. 11. Integer Program

linear ranking functions as used in KoAT1, for example. Thus, CoFloCo, KoAT1, and KoAT2 + CFR fail to compute a finite bound on the runtime complexity of our running example in Fig. 4, while KoAT2 + MΦRF5 succeeds on this example. In particular, this shows that KoAT2 + CFR does *not* subsume KoAT2 + MΦRF5 but the two techniques presented in Sect. 3 and 4 can have orthogonal effects and combining them leads to an even more powerful tool. Indeed, KoAT2 + MΦRF5 + CFR proves a finite bound for more examples than KoAT2 + MΦRF5 and KoAT2 + CFR, in total 506. The configuration KoAT2 + MΦRF5 + CFRSCC has approximately the same power, but a slightly higher runtime.

5.2 Evaluation on Complexity_C_Integer

The benchmark suite CINT consists of 484 C programs, where 366 of them *might* have finite runtime (since iRankFinder can show non-termination of 118 examples). To apply KoAT1 and KoAT2 on these benchmarks, one has to translate the C programs into integer programs as in Definition 2. To this end, we use the tool llvm2kittel [21] which performs this translation by using an intermediate representation of C programs as LLVM bytecode [37], obtained from the Clang compiler frontend [18]. The output of this transformation is then analyzed by KoAT1 and KoAT2.

The results of our evaluation on CINT can be found in Fig. 12. Here, Loopus solved 239 benchmarks, KoAT2 solved 281, KoAT1 solved 285, and CoFloCo solved 288 out of the 484 examples. Additionally, both MaxCore and KoAT2 + MΦRF5 solve 310 examples and KoAT2 + CFRSCC solves 320 examples. This makes KoAT2 the strongest tool on both benchmark sets. Applying partial evaluation on sub-SCCs instead of SCCs improves the average runtime of successful runs, without reducing the number of solved examples. When enabling both control-flow refinement and multiphase-linear ranking functions then KoAT2 is even stronger, as KoAT2 + MΦRF5 + CFR solves 328 examples. Moreover, it is faster than the equally powerful configuration KoAT2 + MΦRF5 + CFRSCC.

In contrast to KoAT1 and CoFloCo, MaxCore also proves a linear runtime bound for our example in Fig. 1, as it detects that y is eventually negative. However, when generalizing Fig. 1 to three phases as in [12] (see Fig. 10 and 11), KoAT2 with MΦRFs can infer the finite bound $27 \cdot x + 27 \cdot y + 27 \cdot z + 56$ on the runtime by using the MΦRF $(z + 1, y + 1, x)$, whereas the other tools fail. Moreover, KoAT2 with MΦRFs is the only tool that proves a finite time bound for the program in Fig. 4. To evaluate Loopus and MaxCore on this example, we translated it into C. While these tools failed, KoAT2 also succeeded on the integer

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + MΦRF5 + CFR	24	228	65	11	0	328	4.77	16.40
KoAT2+MΦRF5+CFRSCC	24	228	65	11	0	328	5.72	16.53
KoAT2 + CFR	25	216	68	11	0	320	5.14	11.67
KoAT2 + CFRSCC	28	216	66	10	0	320	6.00	11.93
MaxCore	23	214	66	7	0	310	1.94	5.24
KoAT2 + MΦRF5	23	204	71	12	0	310	2.11	5.16
CFR + KoAT2	27	200	70	2	1	300	11.26	19.92
CFR + KoAT1	29	187	74	7	0	297	5.34	12.64
CoFloCo	22	195	66	5	0	288	0.81	2.95
KoAT1	25	168	74	12	6	285	2.36	2.97
KoAT2	23	176	70	12	0	281	2.05	2.76
Loopus	17	169	49	4	0	239	0.84	0.72

Fig. 12. Evaluation on Complexity_C_Integer

program that was obtained by applying llvm2kittel to the translated program. This shows the robustness of MΦRFs for programs consisting of several phases.

For the example in Fig. 7 which demonstrates that MΦRFs do not subsume control-flow refinement (Example 27), KoAT2 with its control-flow refinement technique of Sect. 4 infers a linear runtime bound whereas KoAT1 fails, since its loop unrolling technique is a substantially weaker form of control-flow refinement. Besides our tool, only MaxCore was able to infer a finite runtime bound for the C version of this program, where however this bound was quadratic instead of linear.

To sum up, both multiphase-linear ranking functions and control-flow refinement lead to significant improvements. Combining the two techniques, our tool KoAT2 outperforms *all* existing state-of-the-art tools on both benchmark sets.

6 Related Work and Conclusion

Related Work. As mentioned in Sect. 1, many other techniques for automated complexity analysis of integer programs have been developed. The approach in [8] uses lexicographic combinations of linear ranking functions and Ehrhart polynomials to over-approximate the runtime complexity of integer programs. In [46], difference logic is used to analyze C programs. The works in [2–4,22,23] over-approximate so-called cost relations which are closely related to recurrence relations. In [6], a tool chain is presented which uses conditional termination proofs as in [14] to guide the inference of complexity bounds via cost relations by a complexity analyzer in the backend. Based on tools for complexity analysis of integer programs, there also exist approaches to analyze complexity for full programming languages like Java [24,40]. In this way, they complement successful tools for functional verification of Java programs like [1]. Other approaches use the potential method from amortized analysis or type systems to analyze the complexity of C (see, e.g., [17]) or ML programs [32,33]. An approach to verify whether a given resource bound for a program is valid is presented in [47].

While all of these works focus on over-approximating the worst-case runtime complexity of programs, there is also work on the inference of lower bounds on the worst-case runtime complexity, see, e.g., [7,27,49]. Moreover, our tool KoAT also offers the possibility to analyze the expected runtime complexity of probabilistic integer programs, because we also transferred the approach from [16] to probabilistic integer programs [39] and we also integrated decision procedures for the termination and complexity of restricted classes of probabilistic programs in KoAT [30]. See [35] for an overview on runtime analysis for probabilistic programs.

A fundamentally different concept to integer programs are so-called term rewrite systems. These systems model recursion and algebraic data structures, but they do not have any built-in data type for integers. There is also a wealth of techniques and tools to analyze the runtime complexity of term rewrite systems automatically (see [9,10,25,29,43], for example).

Multiphase-linear ranking functions are studied in [12,13,38,50], but these works mainly focus on termination instead of complexity analysis. Moreover, [12] shows how to obtain a linear bound on the runtime complexity of a program with a *single* MΦRF, while we developed a technique to combine MΦRFs on program parts to obtain bounds on the runtime complexity of the full program.

Using control-flow refinement for inferring runtime bounds is studied in [20, 22]. Here, [22] focuses on cost relations, while we embed the approach of [20] into our analysis of integer programs, where we do not apply this method globally but only locally on parts where we do not yet have a linear runtime bound.

Conclusion. In this paper, we showed how to adapt the approach for the computation of runtime and size bounds for integer programs from [16] to multiphase-linear ranking functions and to the use of control-flow refinement. As shown by our experimental evaluation, due to these new improvements, the resulting implementation in our new version of the tool KoAT outperforms the other existing tools for complexity analysis of integer programs.

KoAT's source code, a binary, and a Docker image are available at https://aprove-developers.github.io/ComplexityMprfCfr/. This web site also provides details on our experiments and *web interfaces* to run KoAT directly online.

Acknowledgments. This paper is dedicated to Reiner Hähnle whose ground-breaking results on functional verification and symbolic execution of Java programs with the KeY tool [1], on automatic resource analysis [22], and on its combination with deductive verification (e.g., [5]) were a major inspiration for us. Reiner's work motivated us to develop and improve KoAT such that it can be used as a backend for complexity analysis of languages like Java [24].

We are indebted to Samir Genaim and Jesús J. Doménech for their help and advice with integrating multiphase-linear ranking functions and partial evaluation into our approach, and for providing us with a suitable version of iRankFinder which we could use in KoAT's backend. Moreover, we are grateful to Albert Rubio and Enrique Martín-Martín for providing us with a static binary of MaxCore, to Antonio Flores-Montoya and Florian Zuleger for their help in running CoFloCo and Loopus for our experiments, and to Florian Frohn for help and advice.

A Proofs

A.1 Proof of Lemma 18

We first present lemmas which give an upper and a lower bound for sums of powers. These lemmas will be needed in the proof of Lemma 18.

Lemma 28 (Upper Bound for Sums of Powers). *For any $i \geq 2$ and $k \geq 1$ we have $\sum_{j=1}^{k-1} j^{i-2} \leq \frac{k^{i-1}}{i-1}$.*

Proof. We have $\sum_{j=1}^{k-1} j^{i-2} \leq \sum_{j=1}^{k-1} \int_j^{j+1} x^{i-2}\,dx \leq \int_0^k x^{i-2}\,dx = \frac{k^{i-1}}{i-1}$. □

For the lower bound, we use the summation formula of Euler (see, e.g., [36]).

Lemma 29 (Summation Formula of Euler). *We define the periodic function $H : \mathbb{R} \to \mathbb{R}$ as $H(x) = x - \lfloor x \rfloor - \frac{1}{2}$ if $x \in \mathbb{R} \setminus \mathbb{Z}$ and as $H(x) = 0$ if $x \in \mathbb{Z}$. Note that $H(x)$ is bounded by $-\frac{1}{2}$ and $\frac{1}{2}$. Then for any continuously differentiable function $f : [1, n] \to \mathbb{C}$ with $n \in \mathbb{N}$, we have $\sum_{j=1}^{k} f(j) = \int_1^k f(x)\,dx + \frac{1}{2} \cdot (f(1) + f(k)) + \int_1^k H(x) \cdot f'(x)\,dx$.*

This then leads to the following result.

Lemma 30 (Lower Bound for Sums of Powers). *For any $i \geq 2$ and $k \geq 1$ we have $\sum_{j=1}^{k-1} j^{i-1} \geq \frac{k^i}{i} - k^{i-1}$.*

Proof. Consider $f(x) = x^i$ with the derivative $f'(x) = i \cdot x^{i-1}$. We get

$$\begin{aligned}
&\sum\nolimits_{j=1}^{k} j^i \\
&= \int_1^k x^i\,dx + \tfrac{1}{2} \cdot (1 + k^i) + \int_1^k H(x) \cdot i \cdot x^{i-1}\,dx && \text{(by Lemma 29)} \\
&= \tfrac{k^{i+1}}{i+1} - \tfrac{1}{i+1} + \tfrac{1}{2} \cdot (1 + k^i) + \int_1^k H(x) \cdot i \cdot x^{i-1}\,dx \\
&= \tfrac{k^{i+1}}{i+1} + R && \text{(for } R = -\tfrac{1}{i+1} + \tfrac{1}{2} \cdot (1 + k^i) + \textstyle\int_1^k H(x) \cdot i \cdot x^{i-1}\,dx)
\end{aligned}$$

Since $|H(x)| \leq \frac{1}{2}$, we have $\left|\int_1^k H(x) \cdot i \cdot x^{i-1}\,dx\right| \leq \frac{1}{2} \cdot \left|\int_1^k i \cdot x^{i-1}\,dx\right| = \frac{1}{2} \cdot i \cdot \left|\frac{k^i}{i} - \frac{1}{i}\right| = \frac{k^i - 1}{2}$. Thus, we obtain

$$-\tfrac{1}{i+1} + \tfrac{1}{2} \cdot (1 + k^i) + \tfrac{k^i - 1}{2} \geq R \geq -\tfrac{1}{i+1} + \tfrac{1}{2} \cdot (1 + k^i) - \tfrac{k^i - 1}{2}$$

or, equivalently $-\frac{1}{i+1} + k^i \geq R \geq -\frac{1}{i+1} + 1$. This implies $k^i > R > 0$. Hence, we get $\sum_{j=1}^{k} j^i = \frac{k^{i+1}}{i+1} + R \geq \frac{k^{i+1}}{i+1}$ and thus, $\sum_{j=1}^{k-1} j^i = \sum_{j=1}^{k} j^i - k^i \geq \frac{k^{i+1}}{i+1} - k^i$. With the index shift $i \to i - 1$ we finally obtain the lower bound $\sum_{j=1}^{k-1} j^{i-1} \geq \frac{k^i}{i} - k^{i-1}$. □

Proof of Lemma 18. To ease notation, in this proof ℓ_0 does not denote the initial location of the program $\mathcal{T}$, but an arbitrary location from $\mathcal{L}$. Then we can write (ℓ_0, σ_0) instead of (ℓ, σ), (ℓ_n, σ_n) instead of (ℓ', σ'), and consider an evaluation

$$(\ell_0, \sigma_0)\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,(\ell_1, \sigma_1)\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,\ldots\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,(\ell_n, \sigma_n).$$

Let $M = \max\{0, \sigma_0\left(f_1(\ell_0)\right), \ldots, \sigma_0\left(f_d(\ell_0)\right)\}$. We first prove that for all $1 \le i \le d$ and all $0 \le k \le n$, we have

$$\sigma_k(f_i(\ell_k)) \;\le\; -k \text{ if M} = 0 \quad \text{and} \quad \sigma_k(f_i(\ell_k)) \;\le\; \gamma_i{\cdot}M{\cdot}k^{i-1} - \tfrac{k^i}{i!} \text{ if } M > 0. \quad (2)$$

The proof is done by induction on i. So in the base case, we have $i = 1$. Since $\gamma_1 = 1$, we have to show that $\sigma_k\left(f_1(\ell_k)\right) \le M \cdot k^0 - \frac{k^1}{1!} = M - k$.

For all $0 \le j \le k-1$, the step from (ℓ_j, σ_j) to $(\ell_{j+1}, \sigma_{j+1})$ corresponds to the evaluation of transitions from $\mathcal{T}' \setminus \mathcal{T}'_>$ followed by a transition from $\mathcal{T}'_>$, i.e., we have $(\ell_j, \sigma_j) \to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} (\ell'_j, \sigma'_j) \to_{\mathcal{T}'_>} (\ell_{j+1}, \sigma_{j+1})$ for some configuration (ℓ'_j, σ'_j). Since f is an MΦRF and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing, we obtain $\sigma_j(f_1(\ell_j)) \ge \sigma'_j(f_1(\ell'_j))$. Moreover, since the transitions in $\mathcal{T}'_>$ are decreasing, we have $\sigma'_j(f_0(\ell'_j)) + \sigma'_j(f_1(\ell'_j)) = \sigma'_j(f_1(\ell'_j)) \ge \sigma_{j+1}(f_1(\ell_{j+1})) + 1$. So together, this implies $\sigma_j(f_1(\ell_j)) \ge \sigma_{j+1}(f_1(\ell_{j+1})) + 1$ and thus, $\sigma_0\left(f_1(\ell_0)\right) \ge \sigma_1\left(f_1(\ell_1)\right) + 1 \ge \ldots \ge \sigma_k\left(f_1(\ell_k)\right) + k$ or equivalently, $\sigma_0\left(f_1(\ell_0)\right) - k \ge \sigma_k\left(f_1(\ell_k)\right)$. Furthermore, we have $\sigma_0\left(f_1(\ell_0)\right) \le \max\{0, \sigma_0\left(f_1(\ell_0)\right), \ldots, \sigma_0\left(f_d(\ell_0)\right)\} = M$. Hence, we obtain $\sigma_k\left(f_1(\ell_k)\right) \le \sigma_0\left(f_1(\ell_0)\right) - k \le M - k$. So in particular, if $M = 0$, then we have $\sigma_k(f_1(\ell_k)) \le -k$.

In the induction step, we assume that for all $0 \le k \le n$, we have $\sigma_k(f_{i-1}(\ell_k)) \le -k$ if $M = 0$ and $\sigma_k(f_{i-1}(\ell_k)) \le \gamma_{i-1} \cdot M \cdot k^{i-2} - \frac{k^{i-1}}{(i-1)!}$ if $M > 0$. To show that the inequations also hold for i, we first transform $\sigma_k(f_i(\ell_k))$ into a telescoping sum.

$$\sigma_k\left(f_i(\ell_k)\right) = \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}(\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right))$$

For all $0 \le j \le k-1$, the step from (ℓ_j, σ_j) to $(\ell_{j+1}, \sigma_{j+1})$ again has the form $(\ell_j, \sigma_j) \to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} (\ell'_j, \sigma'_j) \to_{\mathcal{T}'_>} (\ell_{j+1}, \sigma_{j+1})$ for some configuration (ℓ'_j, σ'_j). Since f is an MΦRF and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing, we obtain $\sigma_j(f_{i-1}(\ell_j)) \ge \sigma'_j(f_{i-1}(\ell'_j))$ and $\sigma_j(f_i(\ell_j)) \ge \sigma'_j(f_i(\ell'_j))$. Moreover, since the transitions in $\mathcal{T}'_>$ are decreasing, we have $\sigma'_j(f_{i-1}(\ell'_j)) + \sigma'_j(f_i(\ell'_j)) \ge \sigma_{j+1}(f_i(\ell_{j+1})) + 1$. So together, this implies $\sigma_j(f_{i-1}(\ell_j)) + \sigma_j(f_i(\ell_j)) \ge \sigma_{j+1}(f_i(\ell_{j+1})) + 1$ or equivalently, $\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right) < \sigma_j\left(f_{i-1}(\ell_j)\right)$. Hence, we obtain

$$\begin{aligned}\sigma_k\left(f_i(\ell_k)\right) &= \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}(\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right)) \\ &< \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}\sigma_j\left(f_{i-1}(\ell_j)\right).\end{aligned}$$

If $M = 0$, then we obviously have $\sigma_0(f_i(\ell_0)) \leq 0$ for all $1 \leq i \leq d$. For $k \geq 1$, we obtain

$$\begin{aligned} & \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1} \sigma_j\left(f_{i-1}(\ell_j)\right) \\ \leq\ & 0 + \sum_{j=0}^{k-1} -j && \text{(by the induction hypothesis)} \\ \leq\ & -k+1. \end{aligned}$$

Hence, we have $\sigma_k\left(f_i(\ell_k)\right) < -k+1$ and thus, $\sigma_k\left(f_i(\ell_k)\right) \leq -k$.

If $M > 0$, then we obtain

$$\begin{aligned} & \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1} \sigma_j\left(f_{i-1}(\ell_j)\right) \\ \leq\ & 2 \cdot M + \sum_{j=1}^{k-1} \sigma_j\left(f_{i-1}(\ell_j)\right) && \text{(as } \sigma_0\left(f_i(\ell_0)\right) \leq M \text{ and } \sigma_0\left(f_{i-1}(\ell_0)\right) \leq M\text{)} \\ \leq\ & 2 \cdot M + \sum_{j=1}^{k-1} (\gamma_{i-1} \cdot M \cdot j^{i-2} - \tfrac{j^{i-1}}{(i-1)!}) && \text{(by the induction hypothesis)} \\ =\ & 2 \cdot M + \gamma_{i-1} \cdot M \cdot \left(\sum_{j=1}^{k-1} j^{i-2}\right) - \tfrac{1}{(i-1)!} \cdot \left(\sum_{j=1}^{k-1} j^{i-1}\right) \\ \leq\ & 2 \cdot M + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} - \tfrac{1}{(i-1)!} \cdot \left(\tfrac{k^i}{i} - k^{i-1}\right) && \text{(by Lemma 28 and 30)} \\ =\ & 2 \cdot M + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} + \tfrac{k^{i-1}}{(i-1)!} - \tfrac{k^i}{i!} \\ \leq\ & 2 \cdot M \cdot k^{i-1} + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} + \tfrac{k^{i-1}}{(i-1)!} - \tfrac{k^i}{i!} \\ \leq\ & M \cdot k^{i-1} \cdot \left(\underbrace{2 + \tfrac{\gamma_{i-1}}{i-1} + \tfrac{1}{(i-1)!}}_{\gamma_i}\right) - \tfrac{k^i}{i!} && \text{(as } M \geq 1\text{)} \\ =\ & M \cdot k^{i-1} \cdot \gamma_i - \tfrac{k^i}{i!}. \end{aligned}$$

Hence, (2) is proved.

In the case $M = 0$, (2) implies $\sigma_n(f_i(\ell_n)) \leq -n \leq -\beta = -1 < 0$ for all $1 \leq i \leq d$ which proves the lemma.

Hence, it remains to regard the case $M > 0$. Now (2) implies

$$\sigma_n(f_i(\ell_n)) \ \leq\ \gamma_i \cdot M \cdot n^{i-1} - \tfrac{n^i}{i!}. \tag{3}$$

We now prove that for $i > 1$ we always have $i! \cdot \gamma_i \geq (i-1)! \cdot \gamma_{i-1}$.

$$\begin{aligned}
& i! \cdot \gamma_i \\
= {} & i! \cdot \left(2 + \tfrac{\gamma_{i-1}}{i-1} + \tfrac{1}{(i-1)!}\right) \\
= {} & i! \cdot 2 + i \cdot (i-2)! \cdot \gamma_{i-1} + i \\
\geq {} & (i-1) \cdot (i-2)! \cdot \gamma_{i-1} \\
= {} & (i-1)! \cdot \gamma_{i-1}.
\end{aligned}$$

Thus,

$$d! \cdot \gamma_d \geq i! \cdot \gamma_i \quad \text{for all } 1 \leq i \leq d. \tag{4}$$

Hence, for $n \geq \beta = 1 + d! \cdot \gamma_d \cdot M$ we obtain:

$$\begin{aligned}
& \sigma_n(f_i(\ell_n)) \\
\leq {} & \gamma_i \cdot M \cdot n^{i-1} - \tfrac{n^i}{i!} && \text{(by (3))} \\
= {} & \tfrac{n^{i-1}}{i!} \cdot (i! \cdot \gamma_i \cdot M - n) \\
\leq {} & \tfrac{n^{i-1}}{i!} \cdot (\beta - 1 - n) && \text{(by (4))} \\
< {} & 0 && \text{(since } n \geq \beta)
\end{aligned}$$

Finally, to show that $\beta \in \mathbb{N}$, note that by induction on i, one can easily prove that $(i-1)! \cdot \gamma_i \in \mathbb{N}$ holds for all $i \geq 1$. Hence, in contrast to γ_i, the number $i! \cdot \gamma_i$ is a natural number for all $i \in \mathbb{N}$. This implies $\beta \in \mathbb{N}$. □

A.2 Proof of Theorem 20

Proof. We prove Theorem 20 by showing that for all $t \in \mathcal{T}$ and all $\sigma_0 \in \Sigma$ we have

$$|\sigma_0| \left(\mathcal{RB}'(t)\right) \geq \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0) \, (\to^* \circ \to_t)^k \, (\ell, \sigma)\}. \tag{5}$$

The case $t \notin \mathcal{T}'_>$ is trivial, since $\mathcal{RB}'(t) = \mathcal{RB}(t)$ and $\mathcal{RB}$ is a runtime bound.

Now we prove (5) for a transition $t_> \in \mathcal{T}'_>$, i.e., we show that for all $\sigma_0 \in \Sigma$ we have

$$\begin{aligned}
|\sigma_0| \left(\mathcal{RB}'(t_>)\right) &= \textstyle\sum_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum_{t \in \mathcal{T}_\ell} |\sigma_0| \left(\mathcal{RB}(t)\right) \cdot |\sigma_0| \left(\mathcal{SB}(t, \cdot)(\beta_\ell)\right) \\
&\geq \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0) \, (\to^* \circ \to_{t_>})^k \, (\ell, \sigma)\}.
\end{aligned}$$

So let $(\ell_0, \sigma_0) \, (\to^* \circ \to_{t_>})^k \, (\ell, \sigma)$ and we have to show $|\sigma_0| \left(\mathcal{RB}'(t_>)\right) \geq k$. If $k = 0$, then we clearly have $|\sigma_0| \left(\mathcal{RB}'\,(t_>)\right) \geq 0 = k$. Hence, we consider $k > 0$. We represent the evaluation as follows:

$$\begin{array}{llll}
(\ell_0, \sigma_0) & \to^{\tilde{k}_0}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_1, \tilde{\sigma}_1) & \to^{k'_1}_{\mathcal{T}'} \\
(\ell_1, \sigma_1) & \to^{\tilde{k}_1}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_2, \tilde{\sigma}_2) & \to^{k'_2}_{\mathcal{T}'} \\
 & & \ldots & \\
(\ell_{m-1}, \sigma_{m-1}) & \to^{\tilde{k}_{m-1}}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_m, \tilde{\sigma}_m) & \to^{k'_m}_{\mathcal{T}'} \\
(\ell_m, \sigma_m) & & &
\end{array}$$

So for the evaluation from (ℓ_i, σ_i) to $(\tilde{\ell}_{i+1}, \tilde{\sigma}_{i+1})$ we only use transitions from $\mathcal{T} \setminus \mathcal{T}'$, and for the evaluation from $(\tilde{\ell}_i, \tilde{\sigma}_i)$ to (ℓ_i, σ_i) we only use transitions from $\mathcal{T}'$. Thus, $t_>$ can only occur in the following finite sequences of evaluation steps:

$$(\tilde{\ell}_i, \tilde{\sigma}_i) \to_{\mathcal{T}'} (\tilde{\ell}_{i,1}, \tilde{\sigma}_{i,1}) \to_{\mathcal{T}'} \cdots \to_{\mathcal{T}'} (\tilde{\ell}_{i,k_i'-1}, \tilde{\sigma}_{i,k_i'-1}) \to_{\mathcal{T}'} (\ell_i, \sigma_i). \tag{6}$$

For every $1 \le i \le m$, let $k_i \le k_i'$ be the number of times that $t_>$ is used in the evaluation (6). Clearly, we have

$$\sum_{i=1}^{m} k_i = k. \tag{7}$$

By Lemma 18, all functions $f_1, \ldots, f_d$ are negative after executing $t_>$ at least $1 + d! \cdot \gamma_d \cdot \max\{0, \tilde{\sigma}_i(f_1(\tilde{\ell}_i)), \ldots, \tilde{\sigma}_i(f_d(\tilde{\ell}_i))\}$ times in an evaluation with $\mathcal{T}'$. If all the f_i are negative, then $t_>$ cannot be executed anymore as f is an MΦRF for $\mathcal{T}'_>$ with $t_> \in \mathcal{T}'_>$ and $\mathcal{T}'$. Thus, for all $1 \le i \le m$ we have

$$1 + d! \cdot \gamma_d \cdot \max\left\{0, \tilde{\sigma}_i\left(f_1(\tilde{\ell}_i)\right), \ldots, \tilde{\sigma}_i\left(f_d(\tilde{\ell}_i)\right)\right\} \ge k_i. \tag{8}$$

Let t_i be the entry transition reaching $(\tilde{\ell}_i, \tilde{\sigma}_i)$, i.e., $\tilde{\ell}_i \in \mathcal{E}_{\mathcal{T}'}$ and $t_i \in \mathcal{T}_{\tilde{\ell}_i}$. As $(\ell_0, \sigma_0) \to_{\mathcal{T}}^* \circ \to_{t_i} (\tilde{\ell}_i, \tilde{\sigma}_i)$, by Definition 12 we have $|\sigma_0|\,(\mathcal{SB}(t_i, v)) \ge |\tilde{\sigma}_i(v)|$ for all $v \in \mathcal{PV}$ and thus,

$$\begin{aligned}
&|\sigma_0|\left(\mathcal{SB}(t_i, \cdot)(\beta_{\tilde{\ell}_i})\right) \\
\ge\ & |\tilde{\sigma}_i|\left(\beta_{\tilde{\ell}_i}\right) && \text{(since } \beta_{\tilde{\ell}_i} \in \mathcal{B}\text{)} \\
\ge\ & \tilde{\sigma}_i(\beta_{\tilde{\ell}_i}) \\
\ge\ & 1 + d! \cdot \gamma_d \cdot \max\left\{0, \tilde{\sigma}_i\left(f_1(\tilde{\ell}_i)\right), \ldots, \tilde{\sigma}_i\left(f_d(\tilde{\ell}_i)\right)\right\} \\
& && \text{(by definition of } \lceil\cdot\rceil \text{ and } \beta_{\tilde{\ell}_i}\text{)} \\
\ge\ & k_i && \text{(by (8))}
\end{aligned}$$

In the last part of this proof we need to analyze how often such evaluations $(\tilde{\ell}_i, \tilde{\sigma}_i) \to_{\mathcal{T}'}^* (\ell_i, \sigma_i)$ can occur. Again, let t_i be the entry transition reaching $(\tilde{\ell}_i, \tilde{\sigma}_i)$. Every entry transition t_i can occur at most $|\sigma_0|\,(\mathcal{RB}(t_i))$ times in the complete evaluation, as $\mathcal{RB}$ is a runtime bound. Thus, we have

$$\begin{aligned}
|\sigma_0|\left(\mathcal{RB}'\left(t_>\right)\right) &= \sum_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum_{t \in \mathcal{T}_\ell} |\sigma_0|\,(\mathcal{RB}(t)) \cdot |\sigma_0|\,(\mathcal{SB}(t, \cdot)(\beta_\ell)) \\
&\ge \sum_{i=1}^{m} |\sigma_0|\left(\mathcal{SB}(t_i, \cdot)(\beta_{\tilde{\ell}_i})\right) \\
&\ge \sum_{i=1}^{m} k_i && \text{(as shown above)} \\
&= k && \text{(by (7))}
\end{aligned}$$

□

A.3 Proof of Theorem 24

Let $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$. First note that for every evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^k (\ell', \sigma)$ there is obviously also a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^k (\ell, \sigma)$. To obtain the evaluation with $\mathcal{T}$ one simply has to remove the labels from the locations. Then the claim follows because the guards of the transitions in $\mathcal{T}'$ always imply the guards of the respective original transitions in $\mathcal{T}$ and the updates of the transitions have not been modified in the transformation from $\mathcal{T}$ to $\mathcal{T}'$.

For the other direction, we show by induction on $k \in \mathbb{N}$ that for every evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^k (\ell, \sigma)$ there is a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^k (\ell', \sigma)$ where either $\ell' = \ell$ or $\ell' = \langle \ell, \varphi \rangle$ for some constraint φ with $\sigma(\varphi) = \texttt{true}$.

In the induction base, we have $k = 0$ and the claim is trivial. In the induction step $k > 0$ the evaluation has the form

$$(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \to_{t_2} \cdots \to_{t_{k-1}} (\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$$

with $t_1, \dots, t_k \in \mathcal{T}$. By the induction hypothesis, there is a corresponding evaluation

$$(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1})$$

with $t_1', \dots, t_k' \in \mathcal{T}'$ where $\ell_{k-1}' = \ell_{k-1}$ or $\ell_{k-1}' = \langle \ell_{k-1}, \varphi \rangle$ for some constraint φ with $\sigma_{k-1}(\varphi) = \texttt{true}$. We distinguish two cases:

Case 1: $t_k \notin \mathcal{T}_{SCC}$. If $\ell_{k-1}' = \ell_{k-1}$ and $\ell_k \notin \mathcal{E}_{\mathcal{T}_{SCC}}$, then t_k has not been modified in the transformation from $\mathcal{P}$ to $\mathcal{P}'$. Thus, we have the evaluation $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$ with $t_k \in \mathcal{T}'$. If $\ell_{k-1}' = \ell_{k-1}$ and $\ell_k \in \mathcal{E}_{\mathcal{T}_{SCC}}$, then for $t_k = (\ell_{k-1}, \tau, \eta, \ell_k)$, we set $\ell_k' = \langle \ell_k, \texttt{true} \rangle$ and obtain that $t_k' = (\ell_{k-1}, \tau, \eta, \ell_k') \in \mathcal{T}'$. So we get the evaluation $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\ell_{k-1}, \sigma_{k-1}) \to_{t_k'} (\ell_k', \sigma_k)$. Finally, we regard the case $\ell_{k-1}' = \langle \ell_{k-1}, \varphi \rangle$ where $\sigma_{k-1}(\varphi) = \texttt{true}$. As $t_k = (\ell_{k-1}, \tau, \eta, \ell_k) \in \mathcal{T} \setminus \mathcal{T}_{SCC}$, and $\mathcal{T}_{SCC}$ is an SCC, there is a $t_k' = (\langle \ell_{k-1}, \varphi \rangle, \varphi \wedge \tau, \eta, \ell_k) \in \mathcal{T}'$. Then $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\langle \ell_{k-1}, \varphi \rangle, \sigma_{k-1}) \to_{t_k'} (\ell_k, \sigma_k)$ is an evaluation with $\mathcal{T}'$. The evaluation step with t_k' is possible, since $\sigma_{k-1}(\varphi) = \texttt{true}$ and $\sigma_{k-1}(\tau) = \texttt{true}$ (due to the evaluation step $(\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$). Note that the step with t_k' also results in the state σ_k, because both t_k and t_k' have the same update η.

Case 2: $t_k \in \mathcal{T}_{SCC}$. Here, ℓ_{k-1}' has the form $\langle \ell_{k-1}, \varphi \rangle$ where $\sigma_{k-1}(\varphi) = \texttt{true}$. As ℓ_k is part of the SCC and hence has an incoming transition from $\mathcal{T}_{SCC}$, at some point it is refined by Algorithm 2. Thus, for $t_k = (\ell_{k-1}, \tau, \eta, \ell_k)$, there is some $t_k' = (\langle \ell_{k-1}, \varphi \rangle, \varphi \wedge \tau, \eta, \langle \ell_k, \alpha_{\ell_k}(\varphi_{new}) \rangle) \in \mathcal{T}'$ where $\alpha_{\ell_k}(\varphi_{new})$ is constructed as in Line 8. This leads to the corresponding evaluation $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\langle \ell_{k-1}, \varphi \rangle, \sigma_{k-1}) \to_{t_k'} (\langle \ell_k, \alpha_{\ell_k}(\varphi_{new}) \rangle, \sigma_k)$. Again, the evaluation step with t_k' is possible, since $\sigma_{k-1}(\varphi) = \texttt{true}$ and $\sigma_{k-1}(\tau) = \texttt{true}$ (due to the evaluation step $(\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$). And again, the step with t_k' also results in the state σ_k, because both t_k and t_k' have the same update η. Finally, note that we have $\sigma_k(\alpha_{\ell_k}(\varphi_{new})) = \texttt{true}$. The reason is that $\models (\varphi \wedge \tau) \to \eta(\varphi_{new})$ and $\sigma_{k-1}(\varphi \wedge \tau) = \texttt{true}$ implies $\sigma_{k-1}(\eta(\varphi_{new})) = \texttt{true}$. Hence, we also have $\sigma_k(\varphi_{new}) = \sigma_{k-1}(\eta(\varphi_{new})) = \texttt{true}$. Therefore, $\models \varphi_{new} \to \alpha_{\ell_k}(\varphi_{new})$ implies $\sigma_k(\alpha_{\ell_k}(\varphi_{new})) = \texttt{true}$. □

A.4 Proof of Theorem 25

Let $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$ result from $\mathcal{P}$ by Algorithm 3. As in the proof of Theorem 24, for every evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^k (\ell', \sigma)$ there is also a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^k (\ell, \sigma)$, which is obtained by removing the labels from the locations.

For the other direction, we show that for each evaluation $(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \to_{t_2} \cdots \to_{t_k} (\ell_k, \sigma_k)$ with $t_1, \ldots, t_k \in \mathcal{T}$ there is a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^k (\ell'_k, \sigma_k)$ in $\mathcal{P}'$. To obtain this evaluation, we handle all evaluation fragments separately which use programs $\mathcal{Q}$ from $\mathcal{S}$. This is possible, since different programs in $\mathcal{S}$ do *not* share locations, i.e., entry and outgoing transitions of $\mathcal{Q}$ cannot be part of another $\mathcal{Q}'$ from $\mathcal{S}$. Such an evaluation fragment has the form

$$(\ell_i, \sigma_i) \to_{t_{i+1}} (\ell_{i+1}, \sigma_{i+1}) \to_{t_{i+2}} \cdots \to_{t_{n-1}} (\ell_{n-1}, \sigma_{n-1}) \to_{t_n} (\ell_n, \sigma_n) \qquad (9)$$

where t_{i+1} is an entry transition to $\mathcal{Q}$, t_n is an outgoing transition from $\mathcal{Q}$, and the transitions $t_{i+2}, \ldots, t_{n-1}$ belong to $\mathcal{Q}$. By Theorem 24 it follows that there is a corresponding evaluation using the transitions $t'_{i+2}, \ldots, t'_{n-1}$ from the refined version of $\mathcal{Q}$, such that with the new redirected entry transition t'_{i+1} and the new redirected outgoing transition t'_n we have

$$(\ell_i, \sigma_i) \to_{t'_{i+1}} (\ell'_{i+1}, \sigma_{i+1}) \to_{t'_{i+2}} \cdots \to_{t'_{n-1}} (\ell'_{n-1}, \sigma_{n-1}) \to_{t'_n} (\ell_n, \sigma_n) \qquad (10)$$

Thus, by substituting each evaluation fragment (9) in an evaluation of $\mathcal{P}$ by its refinement (10), we get a corresponding evaluation in $\mathcal{P}'$.

□

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. **413**(1), 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009
4. Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. ACM Trans. Comput. Log. **14**(3), 22:1–22:35 (2013). https://doi.org/10.1145/2499937.2499943

5. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: A formal verification framework for static analysis. Softw. Syst. Model. **15**(4), 987–1012 (2015). https://doi.org/10.1007/s10270-015-0476-y
6. Albert, E., Bofill, M., Borralleras, C., Martín-Martín, E., Rubio, A.: Resource analysis driven by (conditional) termination proofs. Theory Pract. Log. Program. **19**(5–6), 722–739 (2019). https://doi.org/10.1017/S1471068419000152
7. Albert, E., Genaim, S., Martin-Martin, E., Merayo, A., Rubio, A.: Lower-bound synthesis using loop specialization and Max-SMT. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 863–886. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_40
8. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
9. Avanzini, M., Moser, G.: A combination framework for complexity. In: van Raamsdonk, F. (ed.) RTA 2013. LIPIcs, vol. 21, pp. 55–70. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.55
10. Avanzini, M., Moser, G., Schaper, M.: TcT: Tyrolean complexity tool. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 407–423. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_24
11. Ben-Amram, A.M., Genaim, S.: Ranking functions for linear-constraint loops. J. ACM **61**(4), 26:1–26:55 (2014). https://doi.org/10.1145/2629488
12. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 601–620. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_32
13. Ben-Amram, A.M., Doménech, J.J., Genaim, S.: Multiphase-linear ranking functions and their relation to recurrent sets. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 459–480. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_22
14. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 99–117. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_6
15. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005). https://doi.org/10.1007/11523468_109
16. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4), 13:1–13:50 (2016). https://doi.org/10.1145/2866575
17. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Grove, D., Blackburn, S.M. (eds.) PLDI 2015, pp. 467–478 (2015). https://doi.org/10.1145/2737924.2737955
18. Clang Compiler. https://clang.llvm.org/
19. Doménech, J.J., Genaim, S.: "iRankFinder". In: Lucas, S. (ed.) WST 2018, p. 83 (2018). http://wst2018.webs.upv.es/wst2018proceedings.pdf
20. Doménech, J.J., Gallagher, J.P., Genaim, S.: Control-flow refinement by partial evaluation, and its application to termination and cost analysis. Theory Pract. Log. Program. **19**(5–6), 990–1005 (2019). https://doi.org/10.1017/S1471068419000310

21. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauss, M. (ed.) RTA 2011. LIPIcs, vol. 10, pp. 41–50. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011). https://doi.org/10.4230/LIPIcs.RTA.2011.41
22. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 275–295. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_15
23. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
24. Frohn, F., Giesl, J.: Complexity analysis for Java with AProVE. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 85–101. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_6
25. Frohn, F., Giesl, J., Hensel, J., Aschermann, C., Ströder, T.: Lower bounds for runtime complexity of term rewriting. J. Autom. Reason. **59**(1), 121–163 (2016). https://doi.org/10.1007/s10817-016-9397-x
26. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Barrett, C.W., Yang, J. (eds.) FMCAD 2019, pp. 221–230 (2019). https://doi.org/10.23919/FMCAD.2019.8894271
27. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM Trans. Program. Lang. Syst. **42**(3), 13:1–13:50 (2020). https://doi.org/10.1145/3410331
28. Gallagher, J.P.: Polyvariant program specialisation with property-based abstraction. In: VPT@Programming. EPTCS, vol. 299, pp. 34–48 (2019). https://doi.org/10.4204/EPTCS.299.6
29. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2016). https://doi.org/10.1007/s10817-016-9388-y
30. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probability programs. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 269–286. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_16
31. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 156–166. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_10
32. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012). https://doi.org/10.1145/2362389.2362393
33. Hoffmann, J., Das, A., Weng, S.-C.: Towards automatic resource bound analysis for OCaml. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017, pp. 359–373 (2017). https://doi.org/10.1145/3009837.3009842
34. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
35. Kaminski, B.L., Katoen, J.-P., Matheja, C.: Expected runtime analysis by program verification. In: Barthe, G., Katoen, J.-P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 185–220. Cambridge University Press (2020). https://doi.org/10.1017/9781108770750.007
36. Königsberger, K.: Analysis 1, 6th edn. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-642-18490-1

37. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
38. Leike, J., Heizmann, M.: Ranking templates for linear loops. Log. Methods Comput. Sci. **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015
39. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
40. Moser, G., Schaper, M.: From Jinja bytecode to term rewriting: a complexity reflecting transformation. Inf. Comput. **261**, 116–143 (2018). https://doi.org/10.1016/j.ic.2018.05.007
41. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
42. KoAT: Web Interface, Experiments, Source Code, Binary, and Docker Image. https://aprove-developers.github.io/ComplexityMprfCfr/
43. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. J. Autom. Reason. **51**(1), 27–56 (2013). https://doi.org/10.1007/s10817-013-9277-6
44. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_20
45. RaML (Resource Aware ML). https://www.raml.co/interface/
46. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reason. **59**(1), 3–45 (2017). https://doi.org/10.1007/s10817-016-9402-4
47. Srikanth, A., Sahin, B., Harris, W.R.: Complexity verification using guided theorem enumeration. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017, pp. 639–652 (2017). https://doi.org/10.1145/3009837.3009864
48. TPDB (Termination Problems Data Base). https://github.com/TermCOMP/TPDB
49. Wang, D., Hoffmann, J.: Type-guided worst-case input generation. Proc. ACM Program. Lang. **3**(POPL), 13:1–13:30 (2019). https://doi.org/10.1145/3290326
50. Yuan, Y., Li, Y., Shi, W.: Detecting multiphase linear ranking functions for single-path linear-constraint loops. Int. J. Softw. Tools Technol. Transfer **23**(1), 55–67 (2019). https://doi.org/10.1007/s10009-019-00527-1

Alice in Wineland: A Fairy Tale with Contracts

Dilian Gurov[1], Christian Lidström[1(✉)], and Philipp Rümmer[2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
clid@kth.se
[2] Uppsala University, Uppsala, Sweden

Abstract. In this tale Alice ends up in Wineland, where she tries to attend the birthday party of one of its most beloved inhabitants. In order to do so, she must learn about contracts and how important they are. She gets exposed to several contract languages, with their syntax and semantics, such as pre- and post-conditions, state machines, context-free grammars, and interval logic. She learns for what type of properties they are appropriate to use, and how to formally verify that programs meet their contracts.

1 Down the Rabbit-Hole

It was a sunny summer afternoon and Alice was once again sitting on the bank; this time she was alone, since her sister did not like much to sit in the sun. To keep herself occupied, Alice had brought the book she had found in her father's workshop the other day: it was a curious book, thick and heavy, and so worn from many years of study that it almost fell apart in her hands when she tried to open it. What had caught her attention was the picture on the cover page, showing a smiling keyhole in beautiful royal blue colour. 'I didn't even know that keyholes could smile,' thought Alice by herself, and started reading.

Even more curious than the outside was the inside of the book. Being a modern child, Alice had a solid grasp of micro-, nano-, and pico-computers, and had already written several programs herself—and a book about computers it was, as Alice had realised quickly—but the words passing by, as she was going from page to page, were so strange and unknown that she quickly started to feel all numb. She read about syntax and semantics, about functions with three or more possible values, she saw humongous trees growing to the sky, and many weird symbols that seemed to be written upside-down. She came across pages over pages filled with equations, sometimes in black and sometimes in grey colour (or maybe the print had faded over the years), and on one page she even found an egg that had the words 'Real World' written on it. Many times Alice stumbled over prose seemingly familiar, but used in a way that was utterly confusing,

We wish to thank Lewis Carroll, whose books inspired us to write this paper. Also, many thanks to Hurt Sandra Hedström for the wonderful state machine illustration.

© Springer Nature Switzerland AG 2022
W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 229–242, 2022.
https://doi.org/10.1007/978-3-031-08166-8_11

among those long and tedious considerations that involved banks and credit cards; and several times even mentioned contracts!

‘Oh, there is no use in reading the book,’ thought Alice, increasingly desperate, ‘this book is perfectly idiotic!’ when she suddenly noticed the White Rabbit rushing past her, mumbling its usual ‘Oh dear! Oh dear! I shall be late!’ Alice, knowing how fast the Rabbit used to walk, went instantly after it and asked, ‘Late for what?’ ‘Oh dear! Why, for Reiner’s birthday party, of course!’ And down it went into the rabbit-hole, and down went Alice after the Rabbit.

2 Before and After the Footman

Alice was falling, and falling, and falling. That was not a very remarkable thing by itself, as Alice was already used to the journey through the rabbit hole from her previous adventures, and she was not in the least afraid—what caught Alice unprepared were rather the strange writings she could glimpse flying by on the tunnel wall, the likes of which she had never noticed before. In the beginning, Alice’s fall was too fast to read any of the words, but as she started to pay closer attention, Alice could make out individual letters; and soon she recognised the sentences, symbols, and pictures she had seen in the book with the keyhole! ‘Oh dear!’ thought Alice, ‘this will be a tricky adventure!’

Thump! As she was still contemplating about syntax, semantics, and most of all about contracts and banks, the fall came to a sudden end, and Alice found herself sitting on an open plain with a little house in it. THUMP! There was a most extraordinary noise going on within—a constant howling and sneezing, and every now and then a great crash, as if a dish or kettle had been broken to pieces. This must be it, Alice thought, the place of Reiner’s birthday party!

Alice went timidly up to the door, and knocked, but there was no answer. For a minute or two she stood looking at the house, and wondering what to do next, when suddenly a footman in livery came running out of the wood. He rapped loudly at the door with his knuckles, and a second later it was opened by another footman. The first footman produced from under his arm a great letter, nearly as large as himself, and this he handed over to the other, saying in a solemn tone, ‘For the Professor. An invitation to wine tasting!’

The next time Alice looked, one of the footmen was gone, and the other one was sitting on the ground near the door, staring stupidly up into the sky.

Alice really wanted to join the party, so she knocked a second time, but to no avail. ‘There’s no sort of use in knocking,’ said the footman, idling beside her, ‘and for many reasons. First, as you can see, it is me who admits people to the party, and I’m on the same side of the door as you are.’

‘Pray, how can I enter then?’ cried Alice, suddenly afraid she might have arrived too late. ‘I will miss all the fun!’

‘There is much you have to learn first,’ said the footman, ‘before Reiner will receive you. Few are deemed worthy, and you know next to nothing. At this party, everything that is done or happens has a beginning and an end. It will happen only when the beginning is possible, and it can finish only when the outcome is satisfactory.'

'This was not an encouraging opening for a lecture.' Alice replied, rather shyly, 'I—I don't understand a word of what you said.'

The footman gave a deep sigh. 'It is hard work teaching you humans even the simplest things. You will make a complete fool of yourself at the party!' He hesitated, sighed again, and then started to explain.

'The world is evolving as the result of operations or actions applied to it: opening the door, entering the house, taking a seat, or emptying a glass of wine. You might think of such operations as just names, but they carry some meaning as well. We need to have a common understanding of the operations, lest we end up emptying a bottle of wine when we really wanted to write a well-reasoned research article.

We therefore describe the meaning of operations using *contracts,* which will enable civilised people to converse without the danger of any misunderstanding. As an example that even you, my child, will understand, consider the action of sitting down:

Operation:	Take a seat
Pre:	Person is standing $\wedge$ Person was offered seat $\wedge$ Seat is free
Post:	Person is sitting

This contract gives three pieces of information: the *name* of an operation, a *pre-condition* that has to be satisfied before we can even consider starting the operation, and a *post-condition* that describes what we can expect after the operation has finished. The contract does not tell us *how* we can take a seat, which will depend on whether you are a human, a caterpillar, or a tortoise, but it will abstractly describe the assumptions and the result of sitting down.'

A question had formed in Alice's head, and at this point she managed to interrupt the monologue of the footman: 'But I much prefer to sit down whenever I want, not only when somebody has invited me to do so! What use has a contract if the pre-condition is not satisfied?'

The footman felt uneasy, confronted with ignorance of this extent, and put on an indignant face. 'A contract will not tell you anything about the outcome of an operation when you fail to take the pre-condition into account. Even though it might appear, at face value, that you could take seat on every free chair, doing so uninvited might have the most unintended side effects!

But your question has certain merits, as it highlights the slightly surprising semantics of pre-conditions, an element of confusion that can be traced through the literature. In the logic of Hoare [8], and in various methods for specifying programs (for instance [11,17]), a post-condition applies when the pre-condition is satisfied at the point when the operation occurs; nothing is said about cases in which the pre-condition does not hold. In our case, if a person was never invited to sit down, no conclusions can be made about the effect of taking a seat. The outcome of sitting down is one of the possible effects, but neither mandated nor forbidden. In other approaches, however, pre-conditions are interpreted differently and more strictly, namely as necessary conditions for being able to embark upon an operation in the first place (for instance in the *Design by Contract* setting of Meyer [12], and in [10]).'

Sensing insecurity on the side of the footman, Alice put forward a bold suggestion: 'If that is so, wouldn't it be better to rename such conditions of the second sort to 'necessary pre-condition' or 'enabling condition'? Just imagine what a mess you will otherwise get when you associate one operation with two contracts? You will never know which pre-conditions you have to follow, and which ones you are free to ignore!'

As upset as he was about the complete lack of respect shown by this student, the footman couldn't help but secretly agree, Alice had a point. Since he was used to following rules, and not question them, he hastily changed the topic.

'You might wonder, then, how operations relate to the software programs you have read about in your book. In a program, we use *functions* or *procedures* to capture the operations (you might also call them *actions*) taking place in the real world. The effect of an operation can now, very concretely, be described using the variables in the program. The pre-condition will be a condition about the state variables, and the procedure inputs, whereas the post-condition will relate the values of state variables *before* calling the procedure with the values of the variables *after* the call (and with the inputs, and maybe the returned result). Such pre-/post-condition contracts were proposed by Meyer [12] in the context of *Design by Contract,* and represent the most common form of contract. Formal syntax for contracts of this form is provided by several programming and specification languages, including Eiffel, JML, and ACSL. The distinction between pre-conditions, which have to be ensured by the *caller* of an operation, and post-conditions, which are the responsibility of the *callee* and describe the result and effect of an operation, reflects the principle of modular design in which the different components of a program collaborate on clearly stated terms. Modular design enables us to use *Hoare logic* or *Dynamic logic* (and many other approaches) to verify rigorously that each procedure, and the program altogether, satisfies its contracts and therefore will be free of bugs!'

Having come to the end of this complicated explanation, the footman was very satisfied with himself, and expectantly he looked at Alice. But Alice had become confused long ago, and lost track of the many abbreviations used by the footman; all the while a new question had come into her mind and kept her occupied. Slowly, with a lot of pausing and thinking, she tried to explain her doubts.

'That is all fine, but must be an idea pursued by the academic gentry, which has never written a program longer than ten lines. A real procedure will do many things, it will read and update the values of hundreds of variables, and it can itself call many other procedures; how could one ever describe the complete effect of such a procedure using one post-condition? And what is worse, since the contract of a procedure has to capture also the effects of all internally called procedures, it will not at all be modular! If the procedure f calls procedure g, then to write the contract of f we already have to know what g does!'

The footman was taken even more aback than before by this blasphemy; and at the same time did not fully grasp what Alice meant. Rather stiffly he retorted: 'The intention of a contract is to specify the *overall effect* of an operation. Pre- and post-conditions are a means for describing how, upon a procedure call, the final values of the variables relate to their initial values. Contracts *enable* the modular design of a program (or of the world around us), since we can now talk about the effect of operations without having to consider their implementation. If a procedure calls another procedure, this is an implementation detail that the contract should abstract away from. We can build a system modularly by first introducing its operations, then equip each of the operations with a contract describing the intended behaviour, and later we can implement the operations (as procedures) independently of each other. A contract does not have to fully describe the effect of an operation either, it is enough if it mentions its essential features, and leaves away the unnecessary details.'

Alice, who had in the meanwhile ignored the rambling of the footman as she was absorbed in her own thoughts, continued her argument: 'But now imagine we want to write a contract for the task of organising Reiner's party. This will be a huge undertaking: we first have to select a good day and time, then send out all the invitations, then organise all the wine (and other less important things), then wait for all the people to arrive, then ask them to sit down, then pour wine, and so on! The party will of course contain many instances of operations like 'Take a seat,' 'Pour wine,' and 'Empty glass,', so that the pre- and post-condition of the party will repeat the pre-/post-conditions of those sub-operations many times over. The contract will be like a telescope that unfolds and becomes longer and longer, until nobody can see its beginning and end anymore! And think of all the empty bottles, how should we dispose of them?'

At this point, the footman decided that enough was enough, he had wasted already too much time with this lecture. Sitting down on the grass again, he merely mumbled: 'Go and talk to the Caterpillar! You will find it where the smoke is, and it will help you!' With this, he resumed his study of the clouds, leaving Alice alone in a rather confused state.

3 The State of the Caterpillar

As the footman had said, the Caterpillar was to be found exactly where the smoke was. It was sitting on top of a large mushroom, with its eyes closed and its arms folded, quietly smoking its long hookah. 'Good day!' Alice greeted the creature. 'I was told that you can tell me more about contracts. You see, I won't be invited to Reiner's party with only what I know now.' The Caterpillar did not move. It only released yet another cloud of smoke. 'If I want to specify,' Alice went on, 'that a computer program is to engage in certain sequences of operations, or that only certain sequences are allowed, how shall I do that? It seems to me cumbersome to use for this the pre- and post-conditions that the footman is so fond of.'

The Caterpillar finally opened its eyes. 'And because of this, little girl, you disturb my peace?' It cleared its throat and continued in a languid, sleepy voice.

'Well, let me think... This means that when executing the program, whether an operation is to be (or can be) executed next will depend on the history of operations performed up to that point. In that case one needs a notion of *state* as a means to organise the operations in the desired order, and so, it is only natural to use Finite Automata for this, or more generally, State Machines (see for instance [9] for an introduction). Due to their graphical representation, state machines can be a useful visual specification language. But there also exist other formalisms for specifying sequences of operations, such as process algebras (as used, e.g., in [14]) and grammars (see Sect. 4 below), and hierarchical variants of state machines such as Statecharts [7].'

Here, Alice raised her hand—upon which the Caterpillar raised its eyebrows. It was not very fond of being interrupted, but it allowed Alice to ask her question while it took a smoke. 'Can I use a state machine to specify that guests should only drink wine while seated? I wouldn't like them to get drunk and stumble over other guests, you know.' 'Oh, absolutely!' the Caterpillar replied. 'And here is how such a state machine could look.' And it drew, with the mouthpiece of the hookah, a picture on the surface of the mushroom (see Fig. 1).

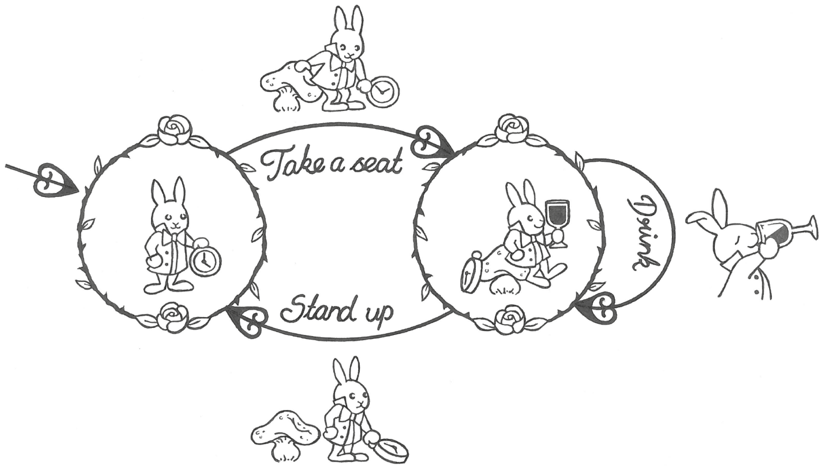


Fig. 1. A contract presented as a state machine, specifying that guests shall be sitting while drinking.

'How marvellous,' exclaimed Alice, 'that is so beautiful!' (Maybe even more so than the drawings of John Tenniel, she thought.) 'But can't this also be achieved with pre- and post-conditions as the footman wanted, instead of drawing circles and arrows?' 'One surely can, my child, one surely can,' replied the Caterpillar.

Here, the Caterpillar made a long pause, exhaling a cloud of smoke. 'You see, a picture is just so much easier to understand. But once you have drawn it, and you wish to use pre- and post-conditions, you simply have to introduce a new global variable[1] to represent the state of the state machine. This *state* variable is initialised at the start of program execution to the initial state of the state machine. And then, you have to encode every transition of the state machine into the contract of the procedure that implements the operation with which the transition is labelled. You can achieve this by adding a conjunct to the pre-condition, to capture when the operation is enabled, and a conjunct to the post-condition, to capture how the state variable is changed.'

'Hm,' Alice mumbled, writing something in her notebook. 'Would this then be a good contract for the drink operation?' And she showed her notes to the Caterpillar.

Operation:	Drink wine
Pre:	Person is sitting $\wedge$ Glass has wine
Post:	Person is sitting

'Yes, indeed,' replied the Caterpillar. But Alice was still wondering. 'If I understand correctly,' she said, 'the state machine is for the whole program and not just local to one procedure. And we also seem to assume that procedures and operations are the same thing. But how about *procedure contracts* that prescribe many operations to be performed, and in certain orders? Can I also use state machines to write such contracts?'

'Hm, hm.' The Caterpillar was pleased to have such an astute student, but was at the same time annoyed by her many questions. 'Yes. Such a situation would arise, for instance, when we have a *main* procedure, which is responsible to order the operations, and we want to write a contract for this procedure. You could then indeed use a state machine like the one in Fig. 1 to formulate the contract. Or, if you prefer to write the contract in textual form, you could present it in some contract specification language such as ConSpec [1] (see Fig. 2).'

```
CONTRACT STATE
   bool sitting = false;

AFTER take_a_seat()
PERFORM
   !sitting  -->  { sitting = true; }

AFTER drink()
PERFORM
   sitting  -->  { skip; }

AFTER stand_up()
PERFORM
   sitting  -->  { sitting = false; }
```

Fig. 2. A state machine in textual form.

[1] Or a so-called *ghost* variable, if your specification system supports them: these are variables that are not part of the program, but are used for specification purposes.

'I see,' said Alice. (But the picture was indeed much nicer, she thought.) 'So sitting is the state variable, and is initialised to false in the beginning. And to take a seat, sitting must be false, but as a result of the operation it will become true?' 'Yes, indeed,' the Caterpillar replied. 'But what if the operation is attempted when sitting is true?' . 'Well, then the contract will be violated, Alice.' 'Oh, yes, I remember! So then, drinking is only allowed when sitting is true, and the operation does not change this.'

'And still' Alice kept wondering, 'instead of using a state machine to specify which other procedures are called and in what order, can't I simply specify the *overall effect* of calling the main procedure in terms of pre- and post-conditions? The footman was insisting on this, you see!'

'Hm, hm,' mumbled the Caterpillar. 'Yes, you can—and typically you would do exactly this. But sometimes you want your contract to be more abstract, as for instance when the very *purpose* of the procedure is to call certain other procedures in a certain order, regardless of what exactly they do (i.e., how they change the state). Another case is when the overall effect of calling the procedure simply cannot be captured in terms of changing the values of the variables—for example, when the effect concerns (interaction with) external entities.'

'Right. But I do have one last question,' said Alice. (Thank God, thought the Caterpillar, as it was beginning to get tired from all the profound questions that Alice was asking.) 'Will the pictures of the state machines always be small enough that we can draw them on a mushroom?' 'No, not always,' the Caterpillar replied. 'Sometimes the state machine may have a large or even an infinite state space. Such state machines are best represented *symbolically*, e.g., as a symbolic Kripke structure (see for instance [5]), by means of two predicates over states: a unary predicate $\mathsf{I}(s)$ that captures the *initial states*, and a binary predicate $\mathsf{T}(s, s')$ that captures the *transition relation*. This is for instance how one represents programs in the TLA framework [10] and the NuSMV model checker [4].

'And finally,' the Caterpillar continued after a brief pause, 'there is the question of how to formally *verify* that a given procedure meets its contract, when the latter is stated as a state machine.' Alice, who really wanted to make sure that Reiner's guests won't drink excessively and make fools of themselves at the party, looked very interested. 'Yes!' she exclaimed, 'I suppose, if we encoded the state machine with pre- and post-conditions, we could simply use procedure-modular deductive verification?'

'Indeed,' replied the Caterpillar. 'Another possibility would be to check statically that the state machine *simulates* the program's execution. Essentially, this amounts to conjoining the program with the state machine (i.e., forming their automata-theoretic product) and checking that the state machine never blocks (i.e., never has to take an operation that is not offered from its current state).' Here, Alice looked puzzled. 'Is this also a procedure-modular method?' she asked. But the Caterpillar could take no more. 'End of class!' it shouted grumpily, and with this it turned its back to Alice.

4 The March Hare Rules

After the lecture by the Caterpillar, Alice thought that, surely, she now knows almost everything there is to know about contracts. Satisfied, she started heading back to the party, but realised that it would probably be improper to show up without some sort of gift—a bottle of wine, or perhaps two. She was reminded of the March Hare, and the many parties with the Mad Hatter and the Dormouse. Maybe he would be able to help, Alice thought, and started heading towards his house to pay another visit.

When she arrived, she was immediately greeted by the March Hare, who invited her in. 'My dear friend', Alice started, 'I am attending a party, and I am in need of wine, in haste.' The March Hare led her down into the cellar, where, sure enough, there was a giant wine rack on one of the walls. 'I have enough wine for any party', the March Hare said, 'but I cannot just give it away.' 'Please, I promise you, for any bottle I take, I will put another one back, as soon as the party is over,' Alice replied.

Not convinced, the March Hare insisted that she make the promise formal, and started lecturing Alice about *context-free grammars* (CFGs). 'Context-free languages are defined by context-free grammars. Such grammars consist of *production rules*, which are sentences over terminating and non-terminating symbols. The terminating symbols are symbols representing the basic units that we want to reason about. This could be the program states, for example, or some set of abstract actions. The non-terminating symbols refer to other production rules, possibly even recursively, and are substituted when evaluating what possible sequences may occur. For this reason, context-free grammars can express many properties that cannot be expressed in the other formalisms, some of which you have already seen.'

As the March Hare went on and on, Alice grew impatient. Eventually, she interrupted the monologue, and although not entirely sure she understood, presented to the March Hare the following contract, in the form of a CFG:

$$C \rightarrow \varepsilon \mid \text{TakeBottle}\ C\ \text{PutNewBottle}$$

She explained her reasoning. The contract is denoted by the production rule C. To make sure the production stops, either because party guests have had just enough wine, or, perhaps more likely, because the shelf has run out of bottles, we use the symbol ε to say that no action is taken. Alice's contract to the March Hare, C, then says that either she doesn't take any bottle at all, or at first she takes one bottle, in the end replaces it with a new bottle, and in between she again fulfils the same contract, resulting in the same amount of bottles being put back as was initially taken out.

'That looks good,' said the March Hare, 'and I can agree to those terms. But before you go, let me first tell you a bit more about such contracts.'

The March Hare continued: 'It is often the case when specifying sequences of operations, that contracts become unwieldy. Consider a procedure which performs its task by calling several other procedures, which all have their own contracts specifying the sequences of operations they will produce. The contract for the top-level procedure will then, naturally, consist of some combination of all those sequences produced by the called procedures. Instead of restating the full sequences, or the formulas representing them, it would be preferable if we had some way of directly referring to the contracts of the called procedures.

Now, for contracting purposes, the non-terminating symbols of the context-free grammars may also serve the purpose of referring to the contracts of other parties involved. For example, for two mutually recursive procedures a and b, with contracts given by non-terminating symbols A and B, respectively. Modelling only the events of calling and terminating the other procedure, we could give their contracts by the following grammar:

$$\begin{aligned} A &\rightarrow \varepsilon \mid \mathrm{call}(b)\ B\ \mathrm{term}(b) \\ B &\rightarrow \varepsilon \mid \mathrm{call}(a)\ A\ \mathrm{term}(a) \end{aligned}$$

We are thus able to specify contracts more concisely, when they depend on results of other procedures. You may already be familiar with the concept of function modularity for verification purposes (see Sect. 2), and by using CFGs in this way we achieve a similar modularity in the contracts themselves.'

'Okay,' said Alice.

'From an assume-guarantee viewpoint, the possible productions of terminating symbols of a contract are what is guaranteed, under the assumption that the other procedures whose contracts are referred to, produce what is specified by the grammar. In this way, assumptions are made explicit, whereas in other formalisms, such as pre- and post-conditions, similar assumptions implicitly exist on called procedures, but are never explicitly stated. A drawback, however, of directly referring to other contracts in this way, is that they are no longer independent of each other, and changes in one contract, as may happen regularly during development, will affect all contracts directly or indirectly referring to that contract.'

'Does this not also mean, then,' Alice remarked, 'that verifying the correctness of such contracts is not an easy task?'

'Why, yes...' the March Hare said, 'but I have some ideas.' Listening to the explanation, Alice learned of the concept of undecidability. Some problems are in general infeasible to solve, and for two languages defined by CFGs, deciding whether one is included in the other is precisely such a problem [9]. This did not stop the March Hare from fleshing out the idea. 'Let us say, that we are only interested in certain actions. If we ignore the data of the program, then the program could be translated, statement by statement, into a CFG producing all those sequences of actions which the program could possibly produce, and then some.' Alice looked confused, but the March Hare continued. 'You see, since we do not take any data into account, our analysis of the control-flow will be an *overapproximation*. But if the sequences produced by this grammar can also be produced by the CFG of the contract, then we can still be sure that the program satisfies the contract.'

'But,' objected Alice, 'you just said a moment ago that we cannot decide whether one such language is a subset of another.'

'While this will not be possible to verify every time, if we are lucky, often it will.' The March Hare went on to explain that if we do not limit ourselves to automatic proofs, but allow humans (or other creatures) to interact with the prover, many more possibilities arise, and that there already are ideas proposed for solving the problem at hand in this way [16]. 'Or...' The March Hare looked deflated. 'Or we restrict ourselves to simpler classes of languagues.' The March Hare suggested the use of visibly pushdown languages [2]. For such languages, Alice soon learned, inclusion is always decidable in exponential time. 'Then you will have your answer in the end, although it might take a very long time. Let's just not restrict ourselves too much, or we will end up back at finite state-machines!' The March Hare laughed, and ran away, leaving Alice alone in the wine cellar.

5 The Logic of the Mad Hatter

Alice, equipped with both knowledge and wine, again started making her way back to the party. As she was walking along a path in the forest, she happened upon the Mad Hatter. Alice started telling him about everything she had learned. The Mad Hatter soon stopped her, and said, 'Yes, yes. That all seems very useful. But how would you go about specifying that if a bottle of wine has been emptied, eventually a new one will be brought to the table? Such a contract seems to me to be of utmost importance!' Alice thought for a minute, but could not find an obvious answer in the logics she had gotten to know.

'Or how about: there shall never be fewer than five full bottles of wine at each table?' the Mad Hatter continued. 'An even more important property!'

The Mad Hatter started rambling about this thing called Interval Temporal Logic (ITL) [6], and how it could be used to specify properties over intervals, or finite sequences of states. 'At the *heart* of this, is an operation called *chop*, and much like when the Queen of Hearts applies it to heads and bodies, it separates an interval into two!'

The Mad Hatter continued 'Now, if E represents the fact that a bottle has been emptied, and N that a new bottle is brought to the table, then the logical formula $\neg E \vee (E {}^\frown N)$ means that either no bottle is emptied, or it is, and if so, a new bottle is eventually brought to the table.'

Alice had to stop to think. She understood the connectives of negation and disjunction, since their meaning here was similar to how she had seen them used before. She recalled what the Mad Hatter had said about the chop operator—that it was a way of concatenating two separate intervals, such that the first interval ends where the second begins. She also recalled that atomic formulas are evaluated in the first state of an interval, and are true for the entire interval if they hold in this state.

'I see!' Alice exclaimed. 'If the interval starts with an emptied bottle, we want there to be a second interval starting with a new bottle being brought, and this should happen when the first interval ends.' She thought a bit more. 'And since the intervals are finite, in particular the first one, the state containing the new bottle must eventually occur.'

'Correct. Let's now say, the state of our table consists of the number of wine bottles on it. We can represent this by the variable *bottles*. At any point in time, we thus want to assert that $bottles \geq 5$.'

Alice also soon understood, that by using the operator *, she could specify that an assertion shall repeatedly hold. 'So by writing F^*, I say that it should be possible to divide the interval into smaller parts, such that each subinterval ends in the state where the next one begins, and the formula F holds in each part, right?'

'That is correct. Now, let's see if can use this to specify what we want.'

'Oh, I know—I know!' Alice exclaimed. 'We simply write $(bottles \geq 5)^*$. That was too easy!'

'Aha, not quite,' the Mad Hatter grinned. 'Would your formula not hold for an interval where there are at least five bottles only in the first state?'

Alice thought for a bit, and realised her mistake. 'You are right, since one possible way to split the interval is into a single part.'

'You see, there is one more operator that we need to talk about, called *next*—and by the way, we will also need a formula that holds for all possible intervals, let's call it true. Now, about *next*, it is true if the formula that comes after holds in the second state. We can use this to reason about the length of intervals...' Alice was again becoming impatient—and it must have showed, because the Mad Hatter interrupted himself mid-sentence. 'But enough of that for now. In this case, all we need is a simple little trick. Instead of saying something shall always hold, we state that its negation must never hold. Using this equivalence, you should be able to specify the property with what you already know.'

'That is neat. How about this, then: $\neg(\mathsf{true}^\frown\neg(bottles \geq 5))$?' Alice asked. 'It should not be the case that eventually, there is fewer than five bottles of wine.'

The Mad Hatter nodded. 'Let me tell you one more thing, before you head off,' he said. 'Since you know about pre- and post-conditions, this might be familiar to you,' (see Sect. 2). The Mad Hatter explained that ITL could be used in a similar way, to get something more closely resembling a contract. 'As with other types of assertions, we would often like to specify that for our formula to apply, some pre-condition must be met. But for this pre-condition, we are not so interested in the full sequence of preceding states. Thus, in interval-based contracts, the pre-condition can be an assertion over singleton states, whereas the post-condition specifies the full traces to be produced.' The Mad Hatter explained that much like Hoare logic has been used to prove correctness of contracts in the style of pre- and post-conditions, extensions of it can be used to prove the correctness of contracts based on ITL [13].

As the Mad Hatter noticed that Alice was barely listening anymore, they said their farewells, and went their separate ways.

'And now it is high time for me to go to Reiner's party,' thought Alice. 'Enough of all these contract languages. And I really want to finally meet Reiner!' She then saw the White Rabbit rush past her. 'Wait!' she shouted after it, 'I can't run as fast as you!' And they disappeared one after the other in the forest.

6 Epilogue

Alice was sound asleep on the bank, with the book [3] still in her hands, when her sister came. ‘Alice, Alice, wake up, I brought you a marshmallow!’ Alice promptly jumped on her feet. She then recalled her dream. ‘Sister, dear—you will never believe what I just dreamt!’ And she told her sister about Wineland and all the creatures she had met, about Reiner and the party she never got to.

‘First, there was this footman, you know,’ Alice started her story. ‘He knew an awful lot about doing things. He said that to do something properly, you first need to know when you can do it (he called it a pre-condition), and then you need to know how things would change afterwards (he called it a post-condition). These two conditions he called a Contract. We talked a lot about these contracts. But I thought them rather limited. Then there was the grumpy Caterpillar, who was sitting on a mushroom and smoking a long hookah. We talked about contracts with states. It drew funny pictures on the mushroom, with circles and arrows, and called them State Machines. And then there was the March Hare. He told me about contracts written as grammars. But not the English grammar we learn about in school, you know. These grammars are called context-free, and with them you could talk about taking bottles and then returning them, one-for-one. And finally, there was the Mad Hatter, who was really quite mad, indeed! He told me about something very strange, he called it Interval Logic. He used some funny letters to write contracts. With them you could make sure that the guests at Reiner’s party will always have enough to drink.’

Alice’s sister looked rather perplexed. She didn’t understand a word of what Alice was saying. And then she suddenly got very agitated. ‘This is all very, very strange,’ she said. ‘While you were asleep, Alice, I heard this funny song on the wireless. It must have been about your dream! It went on like this:’

And if you go chasing rabbits
And you know you’re going to fall,
Tell ’em a hookah smoking caterpillar
Has given you the call.
Call Alice
When she was just small.

‘Yes, this is indeed very strange!’ Alice exclaimed. ‘Maybe I heard it too, from a distance, and that’s why I dreamt all this? But after all,’ she continued, thoughtfully, ‘I am really happy that you woke me up. I should think that *applying* all these contracts for Reiner’s party would have made a complete mess!’

And still, Alice wondered, in Wineland they all assumed that everything you do, you do in a sequence. But how would the contracts be if we also considered things that we do simultaneously? This, and many other questions crossed Alice’s mind. She had still to learn a lot about contracts, and such mysterious languages as Separation Logic [15], in which one can write contracts for concurrent programs. Alice had even heard rumours about contracts that were *smart,* although that was surely an exaggeration that the mad creatures in Wineland must have come up with.

References

1. Aktug, I., Naliuka, K.: ConSpec - a formal language for policy specification. Sci. Comput. Program. **74**(1–2), 2–12 (2008). https://doi.org/10.1016/j.scico.2008.09.004
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of ACM Symposium on Theory of Computing (STOC 2004), pp. 202–211. Association for Computing Machinery (2004). https://doi.org/10.1145/1007352.1007390
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
4. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_44
5. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-10575-8
6. Della Monica, D., Goranko, V., Montanari, A., Sciavicco, G.: Interval temporal logics: a journey. Bull. Eur. Assoc. Theor. Comput. Sci. EATCS **3**(105), 81–107 (2011)
7. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Pearson international edition, Addison-Wesley, Boston (2007)
10. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
11. Leavens, G.T., et al.: JML reference manual (2008)
12. Meyer, B.: Applying "Design by Contract". IEEE Comput. **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
13. Nakata, K., Uustalu, T.: A hoare logic for the coinductive trace-based big-step semantics of while. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 488–506. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_26
14. Oortwijn, W., Gurov, D., Huisman, M.: Practical abstractions for automated verification of shared-memory concurrency. In: Beyer, D., Zufferey, D. (eds.) VMCAI 2020. LNCS, vol. 11990, pp. 401–425. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_19
15. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
16. Rot, J., Bonsangue, M., Rutten, J.: Proving language inclusion and equivalence by coinduction. Inf. Comput. **246**, 62–76 (2016). https://doi.org/10.1016/j.ic.2015.11.009
17. Wing, J.M.: A Two-Tiered Approach to Specifying Programs. Ph.D. thesis, Technical Report TR-299 (1983)

Teaching Design by Contract Using Snap!

Marieke Huisman and Raúl E. Monti(✉)

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{m.huisman,r.e.monti}@utwente.nl

Abstract. With the progress in deductive program verification research, new tools and techniques have become available to support design-by-contract reasoning about non-trivial programs written in widely-used programming languages. However, deductive program verification remains an activity for experts, with ample experience in programming, specification and verification. We would like to change this situation, by developing program verification techniques that are available to a larger audience. In this paper, we present how we developed program verification support for Snap!. Snap! is a visual programming language, aiming in particular at high school students. We added specification language constructs in a similar visual style, designed to make the intended semantics clear from the look and feel of the specification constructs. We provide support both for static and dynamic verification of Snap! programs. Special attention is given to the error messaging, to also make this as intuitive as possible. Finally, we outline how program verification in Snap! could be introduced to high school students in a classroom situation.

Keywords: Verification · Software · Education

1 Introduction

Research in deductive program verification has made substantial progress over the last years: tools and techniques have been developed to reason about non-trivial programs written in widely-used programming languages, the level of automation has substantially increased, and bugs in widely-used libraries have been found [9,24,28]. However, the use of deductive verification techniques remains the field of expert users, and substantial programming knowledge is necessary to appreciate the benefits of these techniques.

We feel that it is important to change this situation, and to make deductive program verification techniques accessible to novice programmers, because specifying the intended behaviour of a program explicitly (including the assumptions that it is making on its environment) is something that programmers should learn about from the beginning, as an integral part of the process leading from design to implementation. Therefore, we feel that it is important that the *Design-by-Contract* approach [23] (DbC), which lies at the core of deductive program verification is taught in first year Computer Science curricula, as is

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 243–263, 2022.
https://doi.org/10.1007/978-3-031-08166-8_12

done already at several institutes, see e.g. CMU's *Principles of Imperative Computation* freshmen course, which introduces *Design-by-Contract* combined with run-time checking [7,26]. In this paper, we take this even further, and make the *Design-by-Contract* idea accessible to high school students, in combination with appropriate tool support.

Concretely, this paper presents a *Design-by-Contract* approach for Snap!. Snap! is a visual programming language targeting high school students. The design of Snap! is inspired by Scratch, another widely-used visual programming language. Compared to Scratch, Snap! has some more advanced programming features. In particular, Snap! provides the possibility to create parametrised reusable blocks, basically modelling user-defined functions. Also the look and feel of Snap! aims at high school age, whereas Scratch aims at an even younger age group. Snap! has been successfully integrated in high school curricula, by its integration in the *Beauty and Joy of Computing* course [12]. This course combines programming skills with a training in abstract computational thinking. We feel that the skills taught in the Beauty and Joy of Computing also provide the right background to introduce *Design-by-Contract*.

In an earlier paper [16], we already presented this position and approach to teaching *Design-by-Contract* using Snap!. This paper further motivates and explains our proposal. We also expand on the analysis of our Snap! extension by presenting further alternatives to our block designs. Moreover, we propose a series of lessons and exercises based on our Snap! extension and targeting school curricula.

The first step to support *Design-by-Contract* for Snap! is to define a suitable specification language. The visual specification language that we propose in this paper is built as a seamless extension of Snap!, i.e. we propose a number of new specification blocks and natural modifications of existing ones. These variations capture the main ingredients for the *Design-by-Contract* approach, such as pre- and postconditions. Moreover, we also provide blocks to add assertions in a program, as well as the possibility to specify loop invariants (which are necessary to support static verification of programs with loops). The choice of specification constructs is inspired by existing specification languages for *Design-by-Contract*, such as JML [17], choosing the most frequently used constructs with a clear and intuitive meaning. Moreover, all verification blocks are carefully designed to reflect the intended semantics of the specifications in a visual way. Below, in Sect. 3, we discuss pros and cons of different options for the visualisation of these specification blocks, and motivate our choice. In addition, we have also extended the standard expression pallets of Snap! with some common expressions to ease specifications such as quantifications over lists and implications, and with the specification-only expressions to refer to a return value of a function, and to the value of a variable in the function's pre-state.

A main concern for a programmer, after writing the specification of the intended behaviour of their programs, should be to validate that these programs behave according to their specification. Therefore, we provide two kinds of tool support for *Design-by-Contract* in Snap!: (i) runtime assertion checking [8], which checks whether specifications are not violated during a particular program execution, and (ii) static verification (or deductive verification) [19], which verifies that all possible program executions respect its specifications. The runtime assertion checker is built as an extension of the standard Snap! execution mechanism. As mentioned, it only checks for single executions, but has the advantage that it can be used quickly and provides intuitive feedback. The deductive verification support is built by providing a translation from a Snap! program into Boogie [2]. This requires more expertise to use it, and moreover, in the current set-up, the verification is done inside Boogie, and the error messages are not translated back. Improving this process will be a future challenge.

Another important aspect to take into account for a good learning experience are the error messages that indicate that a specification is violated. This is an important challenge, also because clear error messages are still a big challenge for existing *Design-by-Contract* checking tools (both runtime and static). We have integrated these messages in Snap!'s standard error reporting system, again sticking to the look and feel of standard Snap!. Moreover, we have put in effort to make the error messages as clear as possible, so that also a relative novice programmer can understand why the implementation deviates from the specification. Of course, improvements in error reporting in other verification tools can also lead to further improvements in our tool.

Finally, we sketch how the specification language and corresponding tool support could be introduced to high school students. Unfortunately, because of the Covid-19 situation, we have not been able yet to try our plan in a high school setting.

To summarise, this paper contributes the following:

– We propose a visual specification language, seamlessly integrated into Snap!, which can be used to specify the behaviour of Snap! programs, including the behaviour of user-defined blocks (functions).
– We provide tool support to validate whether a Snap! program respects its specification. Our tool support enables both dynamic and static verification, such that high school students immediately get a feeling for the different techniques that exist to validate a specification (and the efforts that are required).
– Verification errors are reported as part of the standard Snap! error messaging system.
– We outline a set of exercises and challenges that can be used to make high school students with basic Snap! knowledge familiar with the *Design-by-Contract* approach and the different existing validation techniques.

The remainder of this paper is organised as follows. Section 2 provides a bit more background information on Snap! and *Design-by-Contract*. Section 3 then discusses the visual specification language, and Sect. 4 the verification result reporting mechanisms. Section 5 discusses the tool support we provide, whereas our training plan for *Design-by-Contract* with Snap! is described in Sect. 6. Finally, Sect. 7 concludes, including related and future work.

2 Background

2.1 Snap!

Snap! is a visual programming language. It has been designed to introduce children (but also adults) to programming in an intuitive way. At the same time, it is also a platform for serious study of computer science[14][1]. Snap! actually re-implements and extends Scratch [27]. Programming in Snap! is done by dragging and dropping blocks into the coding area. Blocks represent common program constructs such as variable declarations, control flow statements (branching and loops), function calls and assignments. Snapping blocks together, the user builds a script and visualises its behaviour by means of turtle graphics visualisation, called *sprites*. Sprites can change shape, move, show bubbled text, play music, etc. For all these effects, dedicated blocks are available.

The Snap! interface divides the working area into three parts: the pallet area, the scripting area, and the stage area, see Fig. 1[2]. On the left, it shows the various programming blocks. Blocks are organised into pallets that describe their natural use. For instance, the *Motion* pallet contains blocks that allow you to define moves and rotations of your sprites, the *Variables* pallet contains blocks for declaring and manipulating variables, and so on. In Snap!, variables are dynamically typed; the main types supported in Snap! are Booleans, integers, strings and lists.

Blocks are dragged and dropped from the pallets into the scripting area, located at the centre of the working area. The script in the scripting area defines the behaviour of a *sprite*, i.e. here the Snap! program is constructed. Blocks can be arranged by snapping them together, or by inserting them as arguments of other blocks. Blocks can only be used as arguments if their shapes match with the shape of the argument slots in the target block. These shapes actually provide a hint on the expected evaluation type of a block. For example, an operation block corresponding to a summation

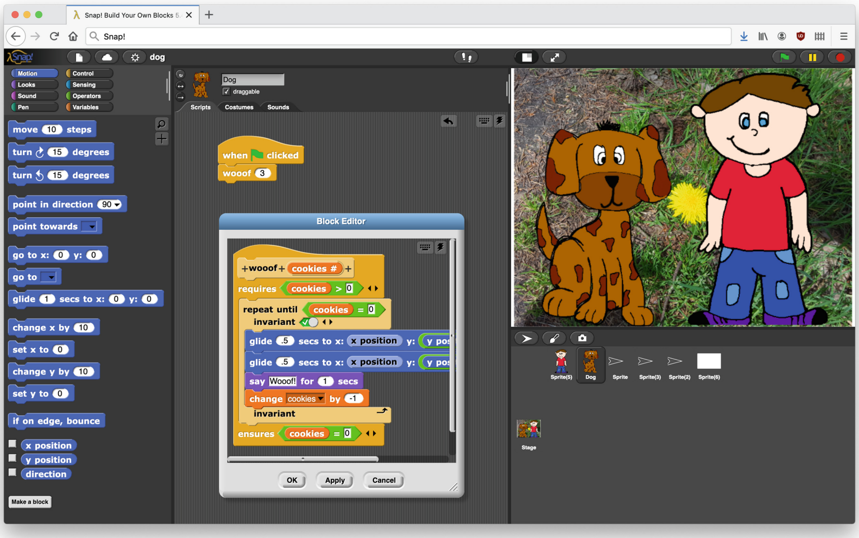
shows rounded slots for its summands (for integer values), while an operation block corresponding to a conjunction

shows diamond slots indicating that it expects boolean operands.

The behaviour of the script is shown in the stage area located in the rightmost part of the screen.

In addition, at the bottom of the pallet area, there is a "Make a block" button. This allows the user to define his or her *Build Your Own Block* (BYOB) blocks. When pressed, a new floating "Block Editor" window pops out with a new coding area, in which the behaviour of the personalised block can be defined (similar to how a script is made in the scripting area). Figure 2 shows the definition of a BYOB block that will make the dog sprite jump and woof for each cookie it is fed. Once defined, the BYOB block becomes available to be used just as any other predefined block.

[1] http://ddi-mod.uni-goettingen.de/ComputerScienceWithSnap.pdf.
[2] Thanks Flor for the drawings!.



```
when [flag] clicked
wooof (3)
```

```
+wooof+ (cookies #) +
requires <(cookies) > [0]>
repeat until <(cookies) = [0]>
  invariant <✓>
  glide (.5) secs to x: (x position) y: (y pos
  glide (.5) secs to x: (x position) y: (y pos
  say [Wooof!] for (1) secs
  change [cookies] by (-1)
  invariant
ensures <(cookies) = [0]>
```

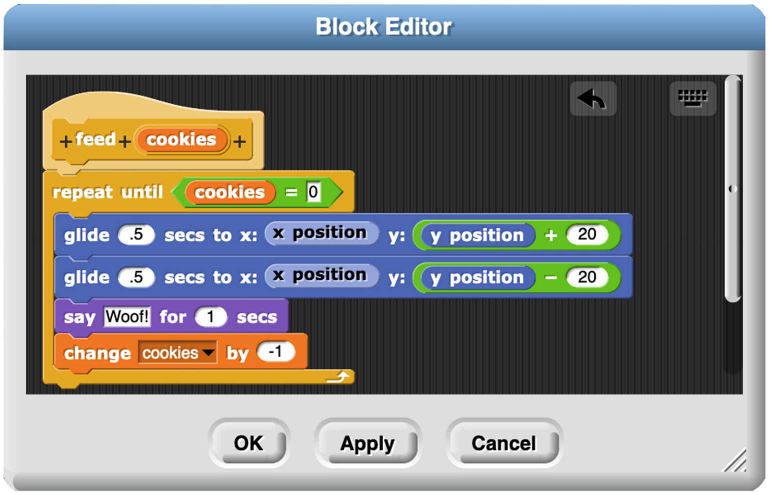
Fig. 1. The Snap! working area.

2.2 Program Verification

The basis of the *Design-by-Contract* approach [21] is that the behaviour of all program components is defined as a formally defined contract. For example, at the level of function calls, a function contract specifies the conditions under which a function may be called (the function's *precondition*), and it specifies the guarantees that the function provides to its caller (the function's *postcondition*). There exist several specification languages that have their roots in this *Design-by-Contract* approach. For example the Eiffel programming language has built-in support for pre- and postconditions [21], and for Java, the behavioural interface language JML [18] is widely used. As is common for such languages, we use the keyword *requires* to indicate a precondition, and the keyword *ensures* to indicate a postcondition.

If a program behaviour is specified using contracts, various techniques can be used to validate whether an implementation respects the contract. Here we distinguish in particular between *runtime assertion checking* and *deductive program verification*, which we will refer to as *static verification*.

Runtime assertion checking validates an implementation w.r.t. a specification at runtime. This means that, whenever during program execution a specification is reached, it will be checked for this particular execution that the property specified indeed holds. In particular, this means that whenever a function will be called, its precondition will be checked, and whenever the function returns, its postcondition will be checked. An advantage of this approach is that it is easy and fast to use it: one just runs a program and checks if the execution does



```
+ feed + cookies +
repeat until cookies = 0
    glide .5 secs to x: x position y: y position + 20
    glide .5 secs to x: x position y: y position − 20
    say Woof! for 1 secs
    change cookies by -1
```

Fig. 2. A BYOB block editor.

not violate the specifications. A disadvantage is that it only provides guarantees about a concrete execution.

In contrast, static verification aims at verifying that all possible behaviours of a function respect its contract. This is done by applying Hoare logic proof rules [15] or using Dijkstra's predicate transformer semantics [10]. Applying these rules results in a set of first-order proof obligations; if these proof obligations can be proven it means that the code satisfies its specification. Advantage of this approach is that it guarantees correctness of all possible behaviours. Disadvantage is that it is often labour-intensive, and often many additional annotations, such as for example loop invariants, are needed to guide the prover.

3 Visual Program Specifications

This section discusses how to add visual specification constructs to Snap!. Our goal was to do this in such a way that (1) the intended semantics of the specification construct is clear from the way it is visualised, and (2) that it smoothly integrates with the existing programming constructs in Snap!

Often, Design-by-Contract specifications are added as special comments in the code. For example, in JML a function contract is written in a special comment, tagged with an @-symbol, immediately preceding the function declaration. The tag ensures that the comment can be recognised as part of the specification. There also exist languages where for example pre- and postconditions are part of the language (e.g., Eiffel [22], Spec# [3]). We felt that for our goal, specifications should be integrated in a natural way in the language, rather than using comments. Moreover, Snap! does not have a comment-block feature. Therefore, we introduce variations of the existing block structures, in which we added suitable slots for the specifications. This section discusses how we added pre- and postconditions, and in-code specifications such as asserts and loop invariants to Snap!. In addition, to have a sufficiently expressive property specification language, we also propose an extension of the expression constructs. In particular, we provide support for specification-only expressions to represent the result and the old value of an expression, as well as quantified expressions. For all our proposals, we discuss different alternatives, and motivate our choice.

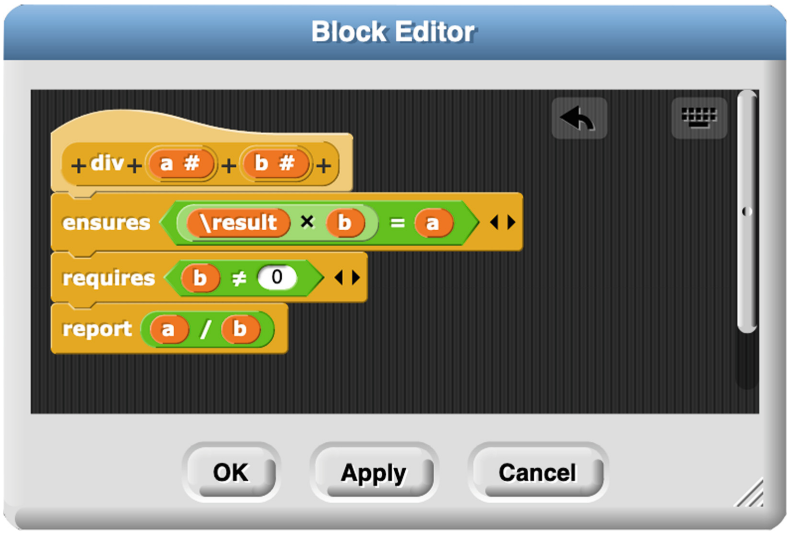
3.1 Visual Pre- and Postconditions

To specify pre- and postconditions for a BYOB script, we identified the following alternatives:

1. Individual pre- and postcondition blocks, which can be inserted in the script, just as any other control block (see Fig. 3 for an example). The advantage of this approach is that the user has flexibility in where to use the specifications, and the user explicitly has to learn where to position the pre- and the postconditions. However, the latter could also be considered as a disadvantage, because if the block is put in a strange position, the semantics becomes unclear from a visual point of view. Moreover, it even allows the user to first specify a postcondition, followed by a precondition.
2. Disconnected, dedicated pre- and postconditions blocks, defined on the side (see Fig. 4). This resembles the special comment style as is used by many deductive verification tools. Drawback is that it clutters up the code, and the connection between the BYOB block and the specification block is lost. Moreover, there is no clear visual indication of when the specified properties should hold.
3. A variation of the initial block, with a slot for a precondition at the start of the block, and a slot for a postcondition at the end of the block (Fig. 5). This shape is inspired by the c-shaped style of other Snap! blocks, such as blocks for loops. The main advantage is that it visualises at which points in the execution, the pre- and the postconditions are expected to hold. In addition, it also graphically identifies which code is actually verified. Moreover, the shapes are already familiar to the Snap! programmer. If the slots are not filled, default pre- and postconditions `true` can be used.

Taking into account all advantages and disadvantages of the different alternatives, we decided to implement this last option as part of our DbC-support for Snap!.

Also for the shape of the pre- and postcondition slots, we considered two possible alternatives. A first alternative is to use a single diamond-shaped boolean slot, using the shape of all boolean blocks in Snap!. An arbitrary boolean expression can be built and dragged into this slot by the user. A second alternative is



```
+div+ a # + b # +
ensures \result × b = a
requires b ≠ 0
report a / b
```

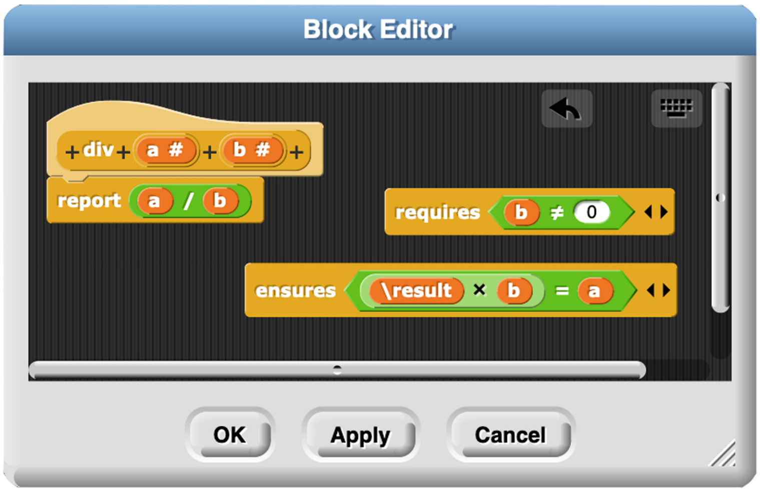
Fig. 3. Separate pre- and postcondition blocks. (Snap! seems to implement mathematical real numbers.)

to add a multiple boolean-argument slot, where we define the semantics of the property to be the conjunction of the evaluation of each of these slots. This is similar to how Snap! extends a list or adds arguments to the header of a BYOB. We opted for the last alternative, because it makes it more visible that we have multiple pre- or postconditions for a block, and it also makes it easier to maintain the specifications.

3.2 Visual Assertions and Loop Invariants

For static verification, pre- and postconditions are often not sufficient, and we need additional in-code specifications to guide the prover. These can come in the form of assertions, which specify properties that should hold at a particular point in the program, and loop invariants. While adding assertions for static verification to guide the prover might be a challenge for high school students, they can also be convenient for runtime assertion checking to make it explicit that a property holds at a particular point in the program. As such, intermediate asserts have an intuitive meaning, and can help to “debug” specifications – therefore, we have decided to support them in our prototype.

Visual Assertions. To specify assertions, both the property specified and the location within the code are relevant. To allow the specification of assertions at arbitrary places in a script, we define a special assertion block assert ◂▸ similar to all other control blocks. The body of the assertion block consists of a multiple boolean-argument slot, similar to how we did this for pre- and postconditions.



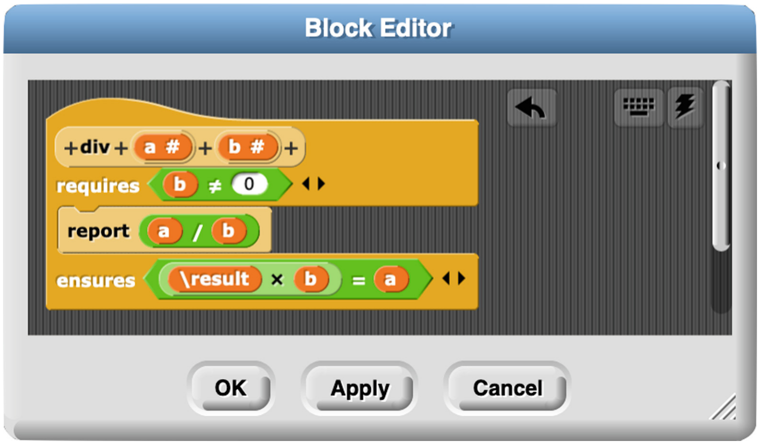
```
+div+ a # + b # +
report a / b
requires b ≠ 0
ensures \result × b = a
```

Fig. 4. Detached contracts

Visual Loop Invariants. Loop invariants are necessary for static verification of programs with loops [29]. A loop invariant should hold at the beginning and end of every loop iteration. Typically, in textual Design-by-Contract languages, loop invariants are specified just above the loop declaration.

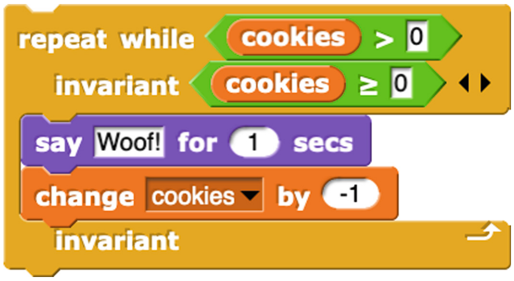
We considered several options for specifying loop invariants in Snap!. One option is to require that the loop invariant is the first instruction of the loop body. However, this does not visually indicate that the invariant also has to hold at the end of every iteration (including the last one). Therefore, we opted for another variant, where we have a (multi-boolean argument) slot to specify the loop invariant in the c-shaped loop block.

This slot is located just after the header where the loop conditions are defined. In addition, the c-shaped loop block repeats the word invariant at the bottom of the block (see Fig. 6) to visually indicate that the invariant is checked after each iteration. We also considered to use an arrow for this purpose, emerging from the invariant declaration and moving along the c-shape up to the bottom. A final option we considered was to implement a ghost placeholder at the end of the c-shaped block, which would be automatically filled with the loop invariant declared at the top. However, we did not further explore this option because we feared it could create confusion about where to enter the loop invariant and it could also use a lot of space with redundant code if not carefully implemented.



```
+div+ a # + b # +
requires b ≠ 0
  report a / b
ensures \result × b = a
```

Fig. 5. Extended block with contracts



```
repeat while cookies > 0
  invariant cookies ≥ 0
  say Woof! for 1 secs
  change cookies by -1
  invariant
```

Fig. 6. Visual loop invariants.

3.3 Visual Expressions

As mentioned above, properties are expressed using Snap!'s visual expression language, extended with several specification-specific constructs. Therefore we have introduced some specification-only keywords, as commonly found in Design-by-Contract languages.

– An *old* expression is used in postconditions to indicate that a variable/expression should be evaluated in the pre-state of the function, for example to specify a relation between the input and output state of a function. To support this, we introduced an operator block \old ▯ with a slot for a variable name.
– A *result* expression refers to the return value of a function inside its postcondition, to specify a property about the result value of a reporter BYOB. This is supported by the introduction of a constant \result operator.

In addition, we also introduced syntax to ease the definition of complex Boolean expressions, adding the operator blocks implies, ≥, ≤ and ≠, as well as syntax to write more advanced Boolean expressions, introducing support for quantified expressions.

In Design-by-Contract languages, universally quantified expressions are typically written in a format similar to: $\forall x \in \mathit{Domain} : \mathit{filter}(x) : \mathit{assertion}(x)$. For example, to specify that an array *ar* is sorted, one would write something like $\forall i \in \mathit{Int} : 0 < i \wedge i < |ar| : ar[i-1] \leq ar[i]$. We find this notation very general and not always suitable for runtime assertion checking: for instance, in many language, it is allowed to leave the *filter* expression empty, i.e. quantify over an unbounded range. However, experience shows that this is hardly ever needed, and requiring an explicit range avoids many mistakes. Therefore, our quantified expressions in Snap! require the user to explicitly specify the range (See Fig. 7).

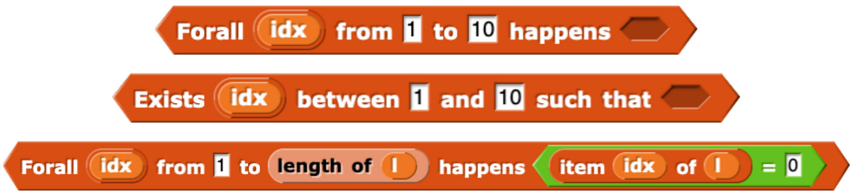

```
Forall (idx) from 1 to 10 happens < >
Exists (idx) between 1 and 10 such that < >
Forall (idx) from 1 to (length of (l)) happens <(item (idx) of (l)) = 0>
```

Fig. 7. Global and Existential quantifiers templates, and an example application.

3.4 Discussion

Based on feedback from one of the reviewers of this paper, we realised that we might have to deviate more from the standard terminology as is used in the *Design-by-Contract* community, as this might be confusing for high school students, without any former experiences in this area.

Some remarks that were made are the following:

- Consider renaming **ensures** to something more intuitive, such as **promises** or **checks**.
- The loop block with invariant could be misinterpreted: students might think that the loop should stop repeating if the invariant does not hold anymore.
- It might be unclear to what state **old** is referring, and it might be worth to investigate if it is possible to make this more intuitive, for example by using a naming convention to denote earlier program states, as in Kaisar [5].
- Rename quantifier blocks such as **Forall .. from .. to .. happens ..** to something like **All .. from .. to .. satisfy ..** .

As future work, we plan to do some experiments with high school students to understand what is the best and most intuitive terminology for them.

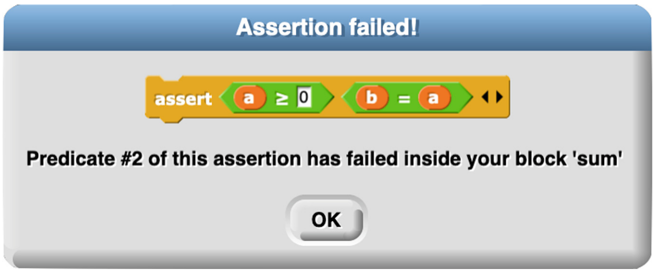
4 Graphical Approach to Verification Result Reporting

Another important point to consider is how to report on the outcome of the verification. We have to consider two aspects: (1) presenting the verdict of a passed verification, and (2) in case of failure, giving a concrete and understandable explanation for the failure. The latter is especially important in our case, as we are using the technique with unexperienced users. Snap! provides several possibilities to present script output to the user, and we discuss how these can be used to present the verification outcomes.

1. Print the value of a variable in the stage using special output blocks. This has the advantage that the stage is the natural place to look at to see the outcome of the script. A drawback is that you loose the connection with the script, to indicate where the verification failed, and that it deviates from the intended use of the stage. Therefore, we decided not to choose this option.
2. Use sounds, using special sound blocks. This is a quick way to indicate that there is verdict, but does not provide any indication of the reason of verification failure. However, it might be an interesting option to explore for vision-impaired users.
3. Use block glowing: when a script is run, the script glows with a green border, when the script fails to execute due to some error the block glows with a red border. This glowing can be reduced to a single block, that caused the failure.
4. Have speech bubbles emerge at specific points in the script that describe the cause of failure. This has the advantage that the failing block can easily be singled out by the location of the bubble, while the cause of failure is described by the text inside the bubble.
5. Use pop-up notification windows. These windows are used by Snap! to show help information about blocks for instance. They are also used as confirmation windows for removing BYOB blocks. These windows have the advantage that a failing block can be printed inside them even when the failing script is not currently visible to the user.

We opted for a combination of alternatives:

1. In order to alert about a contract violation, or any assertion invalidated during runtime assertion checking, we opted for option 5. This allows to be very precise about the error, even when the BYOB body is not currently visible to the user (See Fig. 8). A possible extension of this solution would be that clicking **ok** would immediately lead the user to the corresponding code block.
2. In order to alert of errors while compiling to Boogie, such as making use of dynamic typing or nested lists in your Snap! BYOB code, we opted for option 4. We find this option less invasive than a pop-up window but still as precise, and we can be sure that the blocks involved will be visible since static verification is triggered from the BYOB editor window (See Fig. 9). Notice that the identification of the results of static verification is not something we do from within Snap!, since our extension only returns a compiled Boogie code which has to be verified with Boogie separately.


Assertion failed!

assert ⟨a ≥ 0⟩ ⟨b = a⟩

Predicate #2 of this assertion has failed inside your block 'sum'

Fig. 8. Failure notification for runtime assertion checking.

Although not currently implemented, we think that verification options 2 and 3 would be interesting alternatives for reporting successful verification.

5 Tool Support

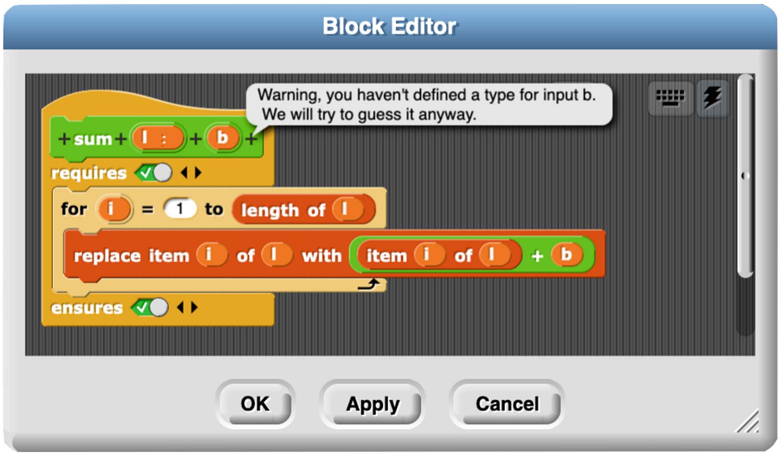
We have developed our ideas into a prototypal extension to Snap! which can be found at https://gitlab.utwente.nl/m7666839/verifiedsnap/. A couple of running examples for verification, including the solutions to the exercise sheet from Sect. 6, have been added to the *lessons* folder under the root directory. The extension uses the same technology as the original Snap!. After downloading the project to a computer, one can just run the "snap.html" file found at the root directory by using most common web-browsers that support JavaScript.

Our extension supports both runtime assertion checking and static verification of BYOB blocks. Runtime assertion checking is automatically triggered when executing BYOB blocks in the usual way. For static verification, a dedicated button located at the top right corner of the BYOB editor window allows to trigger the compilation of the BYOB code into an intended equivalent Boogie code. The compiled code can be then downloaded to verify it with Boogie. Boogie can either be downloaded[3] and run locally or can be run on the cloud at https://rise4fun.com/Boogie/.

The newly introduced verification blocks were naturally distributed along the existing block pallets. These blocks can be used for both types of verification and consist of an `implication` block, `less-or-equal`, `greater-or-equal`, and `different than` blocks, an `assertion` block, a `result` and an `old` block, two types of `looping blocks with invariants`, and `global` and `existential quantification` blocks.

Runtime assertion checking has been fully integrated into the normal execution flow of a Snap! program, and thus there are no real restrictions on the BYOB that can be dynamically verified. When any of the newly defined blocks is reached, the block is loaded into the stack of the executing process and is evaluated just as

[3] https://github.com/boogie-org/boogie.



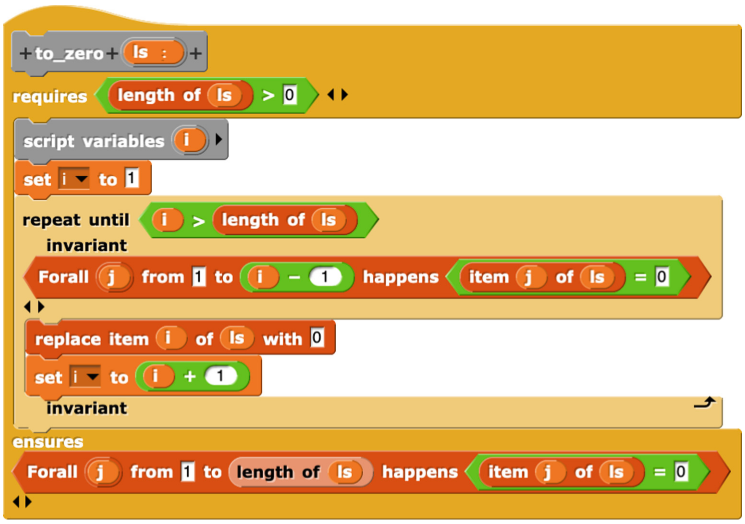
Warning, you haven't defined a type for input b.
We will try to guess it anyway.

```
+sum+ l : + b +
requires
for i = 1 to length of l
    replace item i of l with item i of l + b
ensures
```

Fig. 9. Static verification compilation notification.

any other Snap! block. To make this evaluation possible, a considerable amount of code was introduced to define each block's behaviour. In some cases, such as the implication or the assertion block, the implementation simply consists on a few lines addition to the original scripts of the Snap! project. For other blocks, such as the verifiable loops, the implementation effort was considerably bigger, since it involved implementing the execution logic and evaluation flow of the body of these blocks. Nevertheless, these cases only involved naturally introducing new code in the existing code base. The implementation of the `result` and `old` block, and the implementation of the pre- and postcondition evaluations, turned to be more involved and required modifying the evaluation process of the BYOB blocks. For instance the evaluation of `report` blocks, i.e., blocks that return a value to the programmer of the calling block, was modified in order to, on one hand catch the reported value of the block for the purpose of evaluating `result` blocks in the postconditions, and on the other hand to make sure that the postcondition is evaluated at every possible exit point of a BYOB block.

Static Verification. As we decided to develop only prototypal support, we restrict the kind of BYOB blocks that can be verified with Boogie. We have restricted data types to be integers, booleans and list of integers. In Snap! there is no such concept as integers. The restriction is introduced on the verification level by interpreting all Snap! real numbers as integers. Furthermore, we do not support dynamic typing of variables in the sense that we statically check that the inferred type of a variable does not change during the execution of a program. Finally, we do not target to compile every available Snap! block into Boogie and focus only on an interesting subset for the sake of teaching *Design-by-Contract*. The encoding of the blocks into Boogie is fully implemented into the new *verification.js* module at the *src* directory of the Snap! extension. A considerable part of the encoding is straightforward since a direct counterpart for the construct



```
+to_zero+ ls : +
requires  length of ls > 0
script variables i
set i to 1
repeat until  i > length of ls
  invariant
  Forall j from 1 to i − 1 happens item j of ls = 0
  replace item i of ls with 0
  set i to i + 1
  invariant
ensures
Forall j from 1 to length of ls happens item j of ls = 0
```

Fig. 10. A BYOB block that sets all elements of a list to zero.

can be found in the Boogie language. Nevertheless, some special attention was needed in the following situations:

1. There is no built-in support for lists, nor sequences in Boogie. Thus, we encode lists of integers as maps from integers to integers, and their length as a separate integer variable. Some operations on lists have a complex encoding. For instance, inserting an element in a list is encoded as shifting the tail of the list towards the end using a loop in order to make place for the new element. Lists initialisation is encoded as successive assignments to the map.
2. For ranges, such as those used for global and existential quantification, we also use maps. In this case, if the range bounds are statically known then the initialisation is encoded as successive assignments to the map. If the bounds are not known, then we use a loop and appropriate loop invariants for the initialisation.
3. Parameters in Boogie's procedures are immutable. This is not the case in Snap! where for instance the content of a list can be modified within a BYOB block. This required to encode the inputs parameters of BYOBs as global variables, which Boogie allows to modify inside a procedure. To avoid checking whether variables may be modified, we applied this transformation to global variables to all parameters.
4. Local variables in Boogie are declared at the starting point of the function and are statically typed. As a consequence, the compilation from the dynamically typed Snap! language is not direct and even involves a type inference mechanism.

To illustrate, List.1.1 shows our Boogie compiled code for the BYOB block of Fig. 10.

```
// This code has been compiled from a verifiable Snap!
    project.

var ls_length : int;
var ls: [int]int;

procedure to_zero ()
  modifies ls;
  modifies ls_length;
  requires (ls_length >= 0);
  ensures (forall j : int ::
    (1 <= j && j <= ls_length) ==> (ls[j-1] == 0));
{
  var i: int;

  i := 1;
  while (!(i > ls_length))
    invariant (forall j : int ::
      (1 <= j && j <= (i-1)) ==> (ls[j-1] == 0));
  {
    ls[i-1] := 0;
    i := (i+1);
  }
}
```

Listing 1.1. Boogie compiled code from Snap! block 10

6 Sketch of Teaching Plan

The current situation due to the COVID-19 pandemic jeopardised our plans to test our approach on teaching program verification in high-schools. We have developed an initial lesson plan, containing a sequence of exercises and learning goals to teach program verification to students while assisted by our Snap! with verification extension. The intention of this section is to serve as initial guidance to build more complete plans for teaching software verification that can be integrated into the computer science curricula at high schools. To further develop this lesson plan, we plan to collaborate with didactical experts, that have experience with high school teaching of computer science. The exercises sheet containing the exercises that we mention for each lesson, and the kick-off Snap! projects for each exercise, can be found inside the *lessons* directory at [30]. Along with each lesson we specify which topics the teachers should introduce, such that the students should be able to carry out the exercises.

We assume that students that take on this plan will already have some experience with Snap! and know their way around the coding area. In fact, we target students from the last years of high-school which are already following a lecture assisted by Snap!.

Plan

We divide our teaching plan in lessons, starting from what we expect to be the simplest and moving towards more involved or unintuitive aspects. We describe each lesson by its goals and examples of Snap! verification exercises to assist the learning.

Lesson 1. The goal of this first lesson is to understand the concept of runtime assertion, as validating expectations that we have from the program at certain points of its execution.

We propose two sets of exercise. The first one already contains assertions at certain points of the code. The students are asked to modify the inputs to the code in a way that they obtain cases where the assertions hold and cases where they do not.

In the second set of exercises, the assertions are missing from the code, and the students are asked to define them theirselves and to validate them with different inputs.

Exercise *I* of the exercises sheet may serve as inspiration for preparing exercises for this lesson.

We recommend that the teacher introduces the concept of *predicate*, presents the assertion block assert to the students and compares its behaviour against other blocks which do not stop the execution when the predicate does not hold. It should be clear that not fulfilling the predicate will not be tolerated by the code.

Lesson 2. The goal of this lesson is to introduce the concept of contract. The student is expected to learn the difference between pre- and postconditions for a method/function and how do these play the role of specifying the expected behaviour of the block of code.

We also expect from this lesson that the student obtain some initial practice and intuition on how to correctly translate a natural language specification into corresponding pre- and postconditions formulae.

We propose to carry out this lesson in three steps: first introduce preconditions as contracts with the caller which we can rely upon when developing our code. Then we continue with postconditions showing how they can help to figure out if the code behaves as specified. Then we integrate pre- and postconditions as formal specifications of the code behaviour and we stress their importance by examples where not defining them may result in unexpected behaviour.

Exercises *II, III* and *IV* of the exercises sheet may serve as inspiration for each of the steps of this lesson.

We recommend that the teacher introduces the new initial look of a BYOB block and explains how we can initially check for several predicates in the 'requires' list of the block, while we may use the 'ensures' list to check for predicates when leaving the block. The teacher should also introduce the special blocks \result and \old assisted by examples.

Lesson 3. The goal of this lesson is that students realise that runtime assertion is testing, and that this is different from static verification, which offers full guarantees on the contract fulfilment.

We propose to make the students test BYOB examples where the error is not easy to spot by assertion checking. Maybe offering misleading tests that do not discover the mistake is another way of creating a false feeling of correctness. The next step is to use static verification by compiling the same Snap! code to Boogie. This should show them that their belief is mistaken and trigger them to spot the error. It will also show them the limits of testing and the importance of static verification.

You can use exercise *V* of the exercises sheet for inspiration on the type of exercises to undertake this lesson.

It is important to introduce the students with the differences between *assertion checking* and *static verification*. Most importantly, static verification should not be seen as magic but as logical reasoning with formal guarantees. It is also important to indicate the students where to find the new Boogie compilation button, located at the top-right corner of the scripting window and BYOB coding window, and explain that this will translate the Snap! code into an equivalent Boogie code that they can then verify. It is recommended to introduce Boogie and a little bit of its language to show them that their specifications are indeed present and verified in the compiled code. Of course the students will need guidance to interpret Boogie's output and map it to their original Snap! code.

Lesson 4. In this lesson we will introduce *loop invariants*. The goal is that students learn where and when an invariant is validated during a loop verification, and that they are necessary for static verification to succeed. Another goal is to explain the students how to use *quantifiers* to specify properties about lists. In fact, the use of quantifiers to define loop invariants is specially tricky and will require some practice from the students.

Example exercise *VI* of the exercises sheet may inspire other exercises to accompany this lesson.

This lesson is about an advanced verification topic. Students may need help to understand the results of quantification on empty ranges, thus it is important to introduce this while presenting the quantification blocks. Furthermore we recommend presenting these blocks separately from the exercises, using single block examples. We also recommend to show examples of loop invariants that fail on entering the loop, along with examples that fail after looping some amount of times, and finally examples of edge cases due to, for instance, mistakes on the quantification bounds.

Lesson 5. For the last lesson we propose a bigger project involving a game, such as for example Crisis, which is a multiplayer strategy game. The goal is to demonstrate the use of formal verification in the context of a project as well as to motivate the students with something more fun than disconnected exercises. Moreover, the difficulty of the verification tasks can be increased in comparison with the previous lessons, taking advantage on the motivation of eventually playing the game.

We propose to present the game with a careful description of its rules, trying to remove ambiguities that may result in unnecessary difficulties to translate into formal specifications. We also propose to hand the students an incomplete Snap! implementation of the game. The incomplete parts may consist of BYOB blocks which still need to be specified, developed, or fixed, in order to be able to play the game.

Exercise *VII* of the exercises sheet may serve as inspiration for game-kind projects.

7 Conclusions

In this paper we presented a prototypal program verification extension to the Snap! tool. The extension is intended to support the teaching of *Design-by-Contract* in the later years of high schools. For this reason, we payed considerable attention to the didactic aspects of our tool: the looks and feel of the extension should remain familiar to Snap! users, the syntax and structure of the new blocks should give a clear intuition about their semantics, and the error reporting should be precise and expressive. Whether we succeeded will have to be evaluated in practice.

Our extension allows to analyse BYOB blocks both by runtime assertion checking and static verification. Runtime assertion checking is fully integrated into Snap! and there is no limitation on the kind of blocks that can be analysed. Static verification compiles the Snap! code into a Boogie equivalent code and the verification needs to be run outside of Snap!. Moreover, we make some restrictions on the kind of BYOB blocks we can compile, in order to keep the complexity of the prototype low. As future work we would like to lift these restrictions as much as possible by integrating the remaining Snap! blocks into the compilation and by allowing other data types to be used. Also, we would like to integrate the verification into Snap!, translating Boogie messages back to the Snap! world, to help student to interpret them.

We also plan to extend Snap! with a Sequence or Array library. This will allow to teach students to verify codes that may commit an 'index out of bound' error. Currently, the implementation of lists in Snap! hides this kind of mistakes and works around the problem by returning a special value whose behaviour is not clearly specified. Nevertheless this behaviour gets around quite well, and the student will usually not notice any mistake.

In the last section of our work we sketch a plan of lessons including goals and exercises to teach *Design-by-Contract* to high-school students with some previous knowledge of Snap!. We were not able to test this plan given the current pandemic. As it is still preliminary, it should be improved with the help of a didactic expert on high school teaching of computer science. Moreover, a considerable amount of extra teaching material and teacher guidance should be developed to accompany this plan. Finally, in a follow up work, we expect to be able to test the plan in classrooms and analyse if the learning goals are met.

Related Work. Computer science curricula that uses blocks programming are widely and freely available [4,6,11,13,25]. Nevertheless, they don't seem to include any topics around design and verification of code. Also, the words 'test' or 'testing' are rare and, where mentioned, they are not sufficiently motivated. The drawbacks of teaching coding with blocks without paying attention to design nor correctness has already been analysed [1,20]. We have not found any work on teaching these concepts in schools, nor implementations on block programming that allow to support the teaching of design by contract.

References

1. Aivaloglou, E., Hermans, F.: How kids code and how we know: an exploratory study on the Scratch repository. In: Proceedings of the 2016 ACM Conference on International Computing Education Research, pp. 53–61 (2016)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
4. The Beauty and Joy of Computing. An AP CS Principles Course. https://bjc.edc.org/, Accessed Feb 2022
5. Bohrer, B., Platzer, A.: Structured proofs for adversarial cyber-physical systems. ACM Trans. Embed. Comput. Syst. **20**(5s), 93:1–93:26 (2021)
6. The Creative Computing Curriculum. http://creativecomputing.gse.harvard.edu/guide/, Accessed Feb 2022
7. Cervesato, I., Cortina, T.J., Pfenning, F., Razak, S.: An approach to teaching to write safe and correct imperative programs – even in C (2019). https://www.cs.cmu.edu/~fp/papers/pic19.pdf
8. Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language. PhD thesis, Department of Computer Science, Iowa State University, Ames. Technical Report 03–09 (2003)
9. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
10. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Upper saddle River (1976)
11. Factorovich, P., Sawady, F.: Actividades para aprender a Program. AR: Segundo ciclo de la educación primaria y primero de la secundaria. Miller Ed Buenos Aires (2015)

12. Garcia, D., Harvey, B., Barnes, T.: The beauty and joy of computing. Inroads **6**(4), 71–79 (2015)
13. CS First. https://csfirst.withgoogle.com/s/en/home. Accessed Feb 2022
14. Harvey, B., Mönig, J.: Snap! reference manual (2017). http://snap.berkeley.edu/SnapManual.pdf
15. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
16. Huisman, M., Monti, R.E.: Teaching design by contract using snap! In: 2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG), pp. 1–5. IEEE (2021)
17. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Springer, Boston (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
18. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. **55**(1–3), 185–208 (2005)
19. Leino, K.R.M.: Towards reliable modular programs. Technical report, California Institute of Technology (1995)
20. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.: Habits of programming in Scratch. In: Rößling, G., Naps, T.L., Spannagel, C. (eds.) Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, 27–29 June 2011, pp. 168–172. ACM (2011)
21. Meyer, B.: Eiffel: a language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988)
22. Meyer, B.: Eiffel: The Language. Prentice-Hall, Upper Saddle River (1991)
23. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
24. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
25. An Introduction to Programming. A Pencil Code Teacher's Manual. https://manual.pencilcode.net/, Accessed Feb 2022
26. Pfenning, F., Cortina, T.J., Lovas, W.: Teaching imperative programming with contracts at the freshmen level (2011). https://www.cs.cmu.edu/~fp/papers/pic11.pdf
27. Resnick, M., et al.: Scratch: programming for all. Commun. ACM **52**(11), 60–67 (2009)
28. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10
29. Türk, T.: Local reasoning about while-loops. In: Joshi, R., Margaria, T., Müller, P., Naumann, D., Yang, H. (eds.) VSTTE 2010. Workshop Proceedings, pp. 29–39. ETH Zürich (2010)
30. Snap! extension for runtime assertion checking and static verification. https://gitlab.utwente.nl/m7666839/verifiedsnap/. Accessed Feb 2022

On the Notion of Naturalness in Formal Modeling

Eduard Kamburjan[1(✉)] and Sandro Rama Fiorini[2]

[1] University of Oslo, Oslo, Norway
eduard@ifi.uio.no
[2] IBM Research, Rio de Janeiro, Brazil
srfiorini@ibm.com

Abstract. We investigate what it means for a formal model to be natural using theories from cognitive science and linguistics. Intuitively, naturalness describes that the formal model fits the domain it is modeling – it is not an intrinsic property of the formal model, but a property that is assigned to it by some human interpreter who is making sense of it. Our main observation is that for each formal model, two sense-making processes are possible: First, the process that interprets the formal model as a symbol in the application domain and assigns it a domain concept. Second, the process that interprets the formal model as a symbol in the engineering domain and assigns it a concept describing an engineering view. Naturalness is described as the similarity of these two mental concepts, i.e., the cognitive complexity to map the domain concept to the engineering concept. We discuss these ideas and formalize then using conceptual spaces, a similarity-based concept representation theory based on cognitive semantics.

1 Introduction

Formal methods is a research field spanning programming languages, logics and other formalisms. At its core, it provides tools for *formal modeling*, the development of a formal representation of a system or a design, as well as tools for its analysis. Despite the highly impressive machinery developed to analyze models, there is little research on the modeling process itself. This is far from being an irrelevant aspect – to-date, there is no way to precisely justify modeling decisions and argue why one modeling language is more suited for one task over another.

Modeling studies are notoriously imprecise in this point, resorting on vague descriptions that core concepts of the modeled domain are "naturally" expressed in the chosen language or that the modeling language has a small "representational distance" [27,43] or "is a good match" [28] for the domain. The common idea expressed is that the mental concept of the modeler and the concept as formalized in the language are somehow similar. These arguments are not about the mathematical structure – at their core they are arguments about cognition: a formal model is "natural", if the mental process to recognize the structure of the domain in the formal model requires little cognitive effort.

© Springer Nature Switzerland AG 2022
W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 264–289, 2022.
https://doi.org/10.1007/978-3-031-08166-8_13

The contribution of this work is a cognitive view on formal modeling to give a framework that allows us to reason about modeling decisions and about cognitive processes during modeling. As our main focus, we make the notion of naturalness more precise. To do so, we fix naturalness as the cognitive complexity of the mapping from the mental concept that arises from interpreting the formal model as an expression in the application domain to the mental concept that arises from interpreting the formal model as an engineering artifact, i.e., based on its functionality. For our framework we draw on three tools from cognitive science and linguistics: (1) semiotics to describe the sense-making processes, (2) conceptual spaces to represent mental concepts and (3) metaphorical mappings to describe the relation of mental concepts.

After introducing the basic ideas behind semiotics in Sect. 2, we reflect in Sect. 3 on the classical view on models as abstractions of reality. We then show the inadequacy of the view that the essence of modeling is abstraction to explain why formally equivalent models are perceived with different naturalness. We also distinguish between *non-perceptional* and *perceptional* naturalness – the first is concerned with the similarity of mental concepts, while the later is concerned with more syntactic properties, such as code formatting. The focus here is on *non-perceptional* naturalness.

We aim to provide a way to argue more precisely about modeling, based on our experiences with formal methods – the choice of conceptual spaces is due to their elegant mathematical structure, not a commitment to a model of cognition. We introduce conceptual spaces and their use to describe metaphors in Sect. 4. The Theory of Conceptual Spaces [17] is a concept representation framework having conceptual similarity as its main feature. In Sect. 5 we then use conceptual spaces to make the introduced notions more precise. Additionally, we propose a framework for the mental processes when writing and reading formal models and show where abstraction has its place during the modeling process. Section 6 revisits the examples from previous section and discusses them, and further examples, through the lens of newly introduced framework. Finally, Sect. 7 concludes.

We do not target models in the broadest possible sense, but instead concentrate on a certain set of scenarios that form the core of use cases for formal models: First, we assume that the formal model has a connection to a domain. Such a connection can be direct, as in formalization of domains with ontologies, or indirect, e.g., in a data model for a database or a programmed application. Second, we distinguish between three roles in the modeling, i.e., the creation of the formal model: (1) the *domain expert* who understands the domain, but little of the formalism used for formal modeling; (2) the *technical expert*, who understands the formalism, but little of the domain; and (3) the *modeler* whose task is understanding both formalism and domain, as well as on communicating with domain expert and technical expert.

A Note on Terminology. Due to the interdisciplinary nature of this work, some of the terms are overloaded, most prominent "model" and "modeling". To avoid misunderstandings we use the term *formal model* for digital or physical artifacts in some formal language and the term *formal modeling* for the cognitive process that produces this artifact. We use the word "concept" for mental models (in contrast to, e.g., [20], where "model" is used instead). If "model" is used without further specification, "formal model" is meant.

2 Background: A Very Short Primer on Semiotics

Semiotics is the study of *signs*: symbols, their meaning and the processes that connect meaning and symbol. In this section, we give an overview over its main notions, as far as we need them. More precisely, we adopt the triadic model going back to Peirce [35], with the terminology by Ullmann [46] and some adaptations to avoid name clashes with computer science notions. For a readable general introduction we refer to Chandler [8].

For our purposes, a sign is something that is interpreted as signifying something else to somebody. It consists of three components (Fig. 1): (1) a *symbol*, the form the sign takes, which can be, e.g., a word, a sound or an image; (2) a *concept*, the sense made from the sign, in our case a mental concept; and (3) a *thing*, something the sign refers to, which can be, e.g., another sign or some physical domain entity. The sense making process that connects symbol, concept and thing is cognitive and *needs* an interpreter: *"Nothing is a sign unless it is interpreted as a sign"* [35, 2.172]. We stress two details about this model of signs: First, signs are *not* necessarily psychological – while in this work the concept will indeed be a mental concept, this is not true of general signs. Second, signs are triadic – they are *not* the sum of the diadic relations, but arise from the interactions of all three components.

As an example, the word `Tree` is a symbol, the mental representation of trees is a concept and physical trees are a thing. Together, they form a sign. For formal modeling, we can see this triad as follows: the (real or thought-of) system is the thing, the symbol is the formal model of it, and the concept is the mental view of the modeler or reader.

We introduce a more precise notion of concept in Sect. 4. For now we use them as an intuitive term for "mental representation used in cognition". Concepts may have properties and have connections to other concepts. For example, the concept

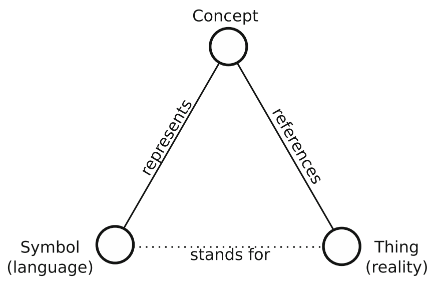


Fig. 1. Adopted semiotic triangle.

"car" has the context of vehicles, the concept "Unit Test" is a concept in Java programming, and the concept "guarded fragment" exists in first-order logic.

Things, i.e., the reality of domain entities, are not a central point in this work. We assume that they exist and that agents can construct conceptual and symbolic descriptions of domain entities. We will rather explore in more detail the representation relation between symbol and concept. Similarly, we do not discuss further that the mental concept is a sign in itself, beyond the observation that the conceptual spaces we introduce in Sec. 4 are symbols on their own.

3 The Inadequacy of Abstraction for Explanation

A formal model is commonly defined along the lines of

> A mathematical representation of a relevant part of a system, design or domain, used for communication and/or a certain analysis.

For example, Peled [36] defines modeling as *"representing a system in terms of mathematical objects that reflect its observed properties. [...] Modelling usually involves the process of abstraction, i.e., simplifying the description of the system, while preserving only a limited number of the original details."* Definitions of modeling are often mere accessory to the formal method and defined ad-hoc – for a more systematic definition of general models we turn to philosophy, where Stachowiak [41, p. 131–133][1] gives the following definition: A model stands for an original (its *Abbildungsmerkmal* or mapping feature), a model is not covering all attributes of the original (its *Verkürzungsmerkmal* or reduction feature, which we identify with abstraction) and a model is standing for its original only for a specific purpose (its *Pragmatisches Merkmal* or pragmatic feature).

In the rest of this work, we understand a *formal model* as a model in the sense of Stachowiak, that is additionally expressed as some mathematical language with formal syntax and an interpretation in terms of other mathematical objects. For example, we consider both programming languages like Java or the λ-calculus, as well as logics as mathematical languages.

Such definitions, which we call the *abstraction-centered*, are emphasizing the the relation of a model to (a possibly perceived or thought-off) reality through abstraction. A model is seen as a clear, partial representation of the world which can be expressed using mathematics and used for a certain intent.

Abstraction-centered definitions are suited to describe what a formal model *is* but are not able to explain why models *are developed* in the way they are in practice. More often than not, models have elements that are unrelated to the things being modeled. For example, consider the Java code in Fig. 2 that models a car with some axles.[2] The class is using a certain serialization framework to read and write objects as XML. There are several points that are not explained by abstraction-centered views:

[1] We restrict ourselves to the above setting and do not investigate, e.g., epistemological questions. English translations of the features are taken from Kühne [29].

[2] We consider Java as being formalized to a sufficient degree to consider it formal.

```
@XmlRootElement(name = "Car")
@XmlType(propOrder={"nrAxels", "name", "pos", "velocity"})
public class Car {
  private int nrAxels;
  private String name;
  private Pair<Integer, Integer> pos;
  private Pair<Integer, Integer> v;

  public Car() {
    this.nrAxels = 0;  this.name = "";
    this.position = null;  this.velocity = null;
  }
  public Car(int nrAxels, String name,
             Pair<Integer, Integer> pos,
             Pair<Integer, Integer> v) {
    setNrAxels(nrAxels);
    ...
  }
  public void drive(Logger log){
    double dist = Math.sqrt(v.component1()*v.component1() +
                            v.component2()*v.component2())
    pos = new Pair<>(pos.component1() + v.component1(),
                     pos.component2() + v.component2());
    log.log(Level.ALL, "Car␣"+name+"␣drove␣"+dist);
  }

  @XmlAttribute
  public int getNrAxels(){ return nrAxels; } // + other getters
  public void setNrAxels(int nrAxels){this.nrAxels = nrAxels;}
  //+ other setters
}
```

Fig. 2. A car with axles as a model in Java.

– The method `drive` takes as parameter a logging instance for debugging. Its existence is not related to cars at all.
– The field `nrAxles` has a setter and is initialized with 0. Certainly a vehicle with 0 axles is not a car, and once a car is build the number of axles does not change[3]. Yet, marshaling in Java requires a default constructor and setters for all fields.

There are modeling decisions in this code that we examine in detail in later parts of this article. For now, it suffices to observe that the Java class `Car` is related to the concept of a car through more than abstraction, as it has elements that do not occur in the domain concept. Furthermore, we cannot separate them

[3] We are sure the interested reader can find situations where the number of axles does change in the lifetime of a car. We assume that this class is written for an application that does not consider any of these situations.

clearly: the field `nrAxles` should be declared final, given our assumption that the number of axles of a car does not change, but we cannot do that given the marshalling requirements. In that case, a reader of this model might wonder whether that is a feature of the domain or a feature of the language.

This brings us to the core of the problem: a model does not only have a relation to our view on reality, it also has a relation to our view on the formalism. The given code can be read in two ways: The first, which domain experts and car enthusiasts would take, is the one of `Car`-as-cars, an expression of how the modeler thinks about cars. The second, which technical experts would take, is the one of `Car`-as-code, a construct that can be understood by examining how the modeler thinks about Java (and further, technical context). If the technical expert knows nothing about cars, it is as hard for him/her to judge its correct abstraction just as it is hard for the domain expert to detect technical bugs in it. The modeler, given the task to mediate between domain and formalism, is caught in-between: to the modeler, the class is *both* `Car`-as-cars and `Car`-as-code.

This makes the modeling job more difficult: the domain expert is not familiar with XML marshaling – communicating requires the modeler to think of the model *only* as an expression of the relevant parts of the domain concept car. However, the expression is partially formed by the requirements of the formalism and the domain expert must accept that `nrAxles` cannot be final. Communicating with other technical experts requires to think of the model *only* as a concept within the engineering/language domain. This too can be challenging: the modeler may willingly break established design patterns within the formalism so the model is conceptually nearer to the domain.

Let us next make these notions more formal by using the semiotic framework established in the previous section. We introduce three entities: (a) the notion of a symbolic, formal model $\mathfrak{M}$; (b) a conceptual, mental structure $\mathfrak{M}^\delta$ denoting the domain aspects coded in $\mathfrak{M}$; and (c) a conceptual, mental structure $\mathfrak{M}^\alpha$ denoting the engineering, technical aspects coded in $\mathfrak{M}$. For example, considering that $\mathfrak{M} = \texttt{Car}$ then $\mathfrak{M}^\delta$ refers to the conceptualization of notion such as that cars have axles, position and velocity; and $\mathfrak{M}^\alpha$ refers to the conceptualization of a Java class with four attributes, with a `drive` method and so on. The semiotic view is given in Fig. 3: $\mathfrak{M}$ is the symbol for *both* sense-making processes. The first sense-making process (of the domain expert) has the concept $\mathfrak{M}^\delta$ and real cars as things, while the second one (of the engineering expert) has the concept $\mathfrak{M}^\alpha$. The modeler must perform, depending on the current situation perform one of these sense-making processes or possibly switch from one to another. The theoretical question when analyzing this situation is to relate the two sense-making processes.

3.1 The Concept-Centered View on Models

We introduce now the first part of our contribution. Both domain expert and technical expert do not ignore the other's view, but for them one of the views dominates. For the modeler, the domination effect is either not strong, does not exist at all, or shifts depending on the situation. In any case, it is out of the question to ignore one of them. This situation begs the question how the to two sense-making processes of Fig. 3 interact within *one* agent. More precisely, with this view on models, which we call *concept-centered*, the central question is:

> Given a formal model $\mathfrak{M}$, how do $\mathfrak{M}$-as-a-domain-concept ($\mathfrak{M}^\delta$) and $\mathfrak{M}$-as-an-artifact-concept ($\mathfrak{M}^\alpha$) relate to each other?

In this work, we explore what it means for $\mathfrak{M}$ to be *natural*. As a start we say that $\mathfrak{M}$ is natural for a person if $\mathfrak{M}$-as-an-artifact-concept is easy to map on $\mathfrak{M}$-as-a-domain-concept.

We stress that we indeed relate two concepts to each other and stress the difference between $\mathfrak{M}$-as-an-artifact-concept and $\mathfrak{M}$: The former describes the mental representation of $\mathfrak{M}$ by properties from the engineering/formalism domain. The latter is just syntax.

Another aspect that is lost in abstraction-centered views is the choice of the modeling language. This is especially true for programming languages, which are (mostly) all Turing-complete and, thus, equally expressive and have the same pragmatic feature (in the sense of Stachowiak). Abstraction is not able to act as an explanation for the choice of the modeling language for a certain situation, if several languages are able to support the needed analyses. We remind that we understand abstraction as the relation between the model and the modeled thing, not implementation-hiding constructs such as interfaces within parts of the model.

3.2 Concepts and Syntax

The concept-centered view on models is not antithetical to other views but rather stresses cognitive processes related to modeling. We distinguish between the mental processes concerned with concepts and the mental processes concerned with deriving these mental concepts from perception.

The first class of processes describes the representation of concepts and their processing; e.g., what constitutes a car and what is the relation of the concept car

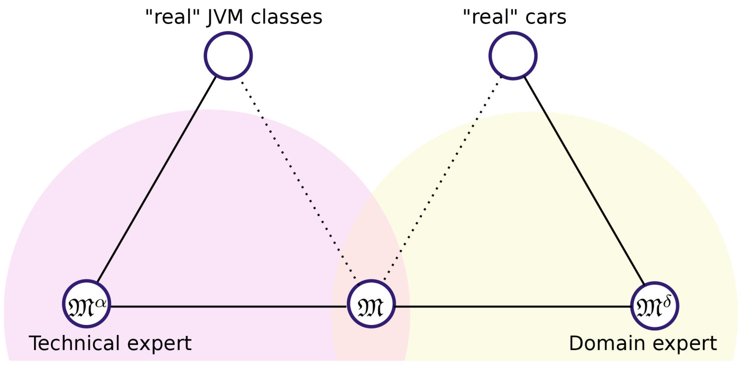


Fig. 3. The concept-centered view on models for two agents.

with other concepts. The second class connects these processes with perception; e.g., is the thing that one perceives representing the concept of cars. The distinction is not sharp, but it is useful in our context as it allows us to specifically target syntactic questions. As an example, we give several logic characterizations of the following statement:

A car has 4 wheels.

First, consider the following first-order formula:

$$\begin{aligned}\forall x.\ \Big(&\mathsf{Car}(x) \to \exists w_1, w_2, w_3, w_4.\\ &\big(\mathsf{hasPart}(x, w_1) \land \mathsf{hasPart}(x, w_2) \land \mathsf{hasPart}(x, w_3) \land \mathsf{hasPart}(x, w_4)\\ &\ \land \mathsf{Wheel}(w_1) \land \mathsf{Wheel}(w_2) \land \mathsf{Wheel}(w_3) \land \mathsf{Wheel}(w_4)\\ &\ \land w_1 \neq w_2 \land w_1 \neq w_3 \land w_1 \neq w_4 \land w_2 \neq w_3 \land w_2 \neq w_4 \land w_3 \neq w_4\big)\Big)\end{aligned}$$

There are several syntactical effects here that make it easy to grasp what is modeled, before the stage where we can ask whether the model is natural or not. I.e., syntactical effects describe how perceived structures give rise to the model-as-a-concept:

– Conjuncts are grouped together in units that correspond to one statement, e.g., "all parts are wheels" corresponds to the second line,
– the order of these units is almost the same as in the description,
– the literal 4 occurs in both natural language description and formalization,
– indentation gives a clear structure that partitions the unit visually, and
– the variable names clearly related to their intended meaning.

We assume that the predicate and relation symbols are inherently meaningful and that it is not sensible to have the predicate that expresses "x is a car" have any other name. Next, we turn to the units in more detail. The unit

$$w_1 \neq w_2 \land w_1 \neq w_3 \land w_1 \neq w_4 \land w_2 \neq w_3 \land w_2 \neq w_4 \land w_3 \neq w_4$$

describes that the four variables are different. We imagine that readers familiar with fist-order logic have not read all clauses in detail. Instead the pattern of pairwise inequality is recognized and the unit is perceived as one statement.

Now, consider the following, equivalent first-order formula:

$$\begin{aligned}\forall x.\ \Big(&\mathsf{Car}(x) \rightarrow \exists \mathit{cake}, \mathfrak{Y}_6, x_2, \mathit{Schiff}.\\ &\big(\ \mathsf{Wheel}(\mathit{Schiff}) \land \mathfrak{Y}_6 \neq x_2 \land \mathfrak{Y}_6 \neq \mathit{Schiff}\\ &\ \land \mathsf{hasPart}(x, \mathit{Schiff}) \land \mathit{cake} \neq \mathfrak{Y}_6 \land \mathsf{Wheel}(\mathfrak{Y}_6) \land \mathsf{Wheel}(x_2)\\ &\ \land \mathsf{hasPart}(x, x_2) \land \mathit{cake} \neq \mathit{ship} \land \mathsf{Wheel}(\mathit{cake}) \land \mathsf{hasPart}(x, \mathfrak{Y}_6)\\ &\ \land \mathsf{hasPart}(x, \mathit{cake}) \land x_2 \neq \mathit{ship} \land x_2 \neq \mathit{cake}\big)\Big)\end{aligned}$$

While one can see it represents the same concept once understood, it is probably harder to derive the mental concept in the first place given how the variables and restrictions are written. We call the distance of concept and formal model *perceptional naturalness*. We will discuss this example in more details when we have formalized perceptional naturalness. Next, we briefly discuss the role of the language in more detail.

Formal models are expressed in some modeling language and the choice of the language plays a role in how a concept is expressed. Thus, the choice of the language has an influence on both the naturalness of the formal model and its perceptional naturalness. We focus on naturalness here, for perceptional naturalness it suffices to note that formal languages such as Whitespace or Malbolge [34] and C are all equally expressive languages, yet Whitespace or Malbolge are highly unnatural in any sense of the word.

It is out of scope for this work to discuss computational thinking in detail and to investigate the differences in programming paradigms; e.g., between functional, declarative and imperative programming. Instead, we discuss in more detail how the units introduced in the above example related to the role of language. Returning to the four-wheeled car, we can give an alternative notation for the same formula:

$$\forall x.\ \mathsf{Car}(x) \rightarrow \exists w_1, w_2, w_3, w_4.\ \bigwedge_{i \in 1..4} \big(\mathsf{Wheel}(w_i) \land \mathsf{hasPart}(x, w_i)\big) \land \bigwedge_{\substack{i,j \in 1..4\\ i \neq j}} w_i \neq w_j$$

Is this formula a first-order logic formula? Syntactically it is not, but it is straightforward to expand all the introduced shortcuts and retrieve a "pure" first-order formula[4]. We argue that it is still a first-order logic formula, instead the shortcuts form a *conceptual library* of patterns that are employed anyway. E.g., the grouping into units. Similar conceptual libraries are known in other minimal languages, e.g., the church encoding of the natural numbers in the lambda-calculus. Similarly to programming language libraries, these conceptual libraries are a summary of useful patterns that are repeated in many programs.

When the modeler starts to express a concept in a certain language, the conceptual (and programming) libraries are included in the expression. I.e., when

[4] Extensions of a simple formalism may be less straightforward than expected, as the study of Quinlan et al. [37] on the use of BNF grammars in practice shows.

arguing about the naturalness of a formal model, the libraries must be included in the discussion, as the concept must be adapted to both language and libraries. In this sense, no programming or modeling language is truly minimal, it just gives the user a bigger freedom in the choice of libraries in turn for a higher reliance on these libraries.

3.3 A Complete View on Models

We have so far discussed two views on models: (1) The abstraction-centered view emphasizes the relation of a formal model the thing it stands for. It emphasizes the reduction feature. (2) The concept-centered view emphasizes the dual nature of a formal method and the relation between $\mathfrak{M}$-as-a-domain-concept and $\mathfrak{M}$-as-an-artifact. It emphasizes the mapping feature. For completeness' sake, we mention a third view that emphasizes the pragmatic feature, which we call *purpose-centered*: A model is a mathematical expression made for a certain (business-)purpose. In industrial practice, this view is more relevant than the others: as businesses aim to make money, the availability of trained personal, computational resources, etc. is critical in the choice of what language is chosen and how a model is designed. I.e., one may have a model that is less natural than possible and less abstract than possible, but no employee is able to produce such a model in a reasonable amount of time and the model is good enough for business-purposes.

4 Conceptual Spaces

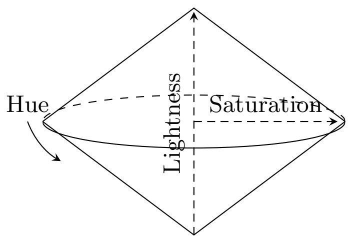
To examine naturalness, we must be able to analyze the relationship between $\mathfrak{M}^\delta$ and $\mathfrak{M}^\alpha$; In search for tools to examine and describe these mental concepts we turn to cognitive science. This research field is, among others and briefly summarized, concerned with explaining and constructing cognitive activity. Cognitive activity uses some information to reach some goals. While the nature of how this information and these goals are represented mentally is an elusive mystery, there are numerous theories of *modeling* representations. The following section, and indeed this whole work, does, thus, *not* claim that the used notion of concepts is "real", in the sense that the cognitive activity uses conceptual spaces for representation, they are a model *for* the representations.

The theory used here is the Theory of Conceptual Spaces of Gärdenfors [17,18], which is based on geometric structures and motivated existing cognitive phenomena, such as similarity[5]. In short, conceptual spaces (CS) are metric spaces where concepts are represented as regions, objects as points and dimensions are ways in which these can be compared. Similar concepts are grouped closer together in a conceptual space. For example, the concept Apple could be represented as a region in a CS where the dimensions are shape, color, and weight. In this space, the region for Oranges would be closer to Apple than Pineapple,

[5] This is in line with a tradition to describe concepts/categories not by common features, but by distance between instances, following Wittgenstein's family resemblance [49] and Rosch' prototype theory [38].

for example. The relevance of Conceptual Spaces to our discussion resides in its proposal as a framework for cognitive semantics. In cognitive semantics, the meaning of linguistic expressions is given by *mental* entities, which are grounded in reality through perception. Conceptual spaces provides a structure for such mental entities. That fits our view in which (symbolic) models, such as Java classes, are interpreted as mental models of domain or technical entities.

A considerable part of what makes CS a powerful representation framework is the way in which the dimensions of the metric space are structured. Certain dimensions, particularly perceptual ones, always appear together, forming *quality domains*. For example, hue, saturation, and luminosity (HSL) are integral[6] to each other, forming a color domain (to the right). In addition to that, the Theory specifies that *natural properties* are convex regions in quality domains. The notion of natural property is loosely defined as those natural for the purpose of usual problem-solving tasks. So, for example, the property Red in the HSL space should be convex. Indeed, studies with color perception in different cultures showed that regions for basic colors in HSL space are indeed approximate convex regions [40]. Studies with other perceptual domains paint a similar picture [17].



Complex concepts, such as Apple, can be more precisely defined as collections of regions in quality domains. Those include perceptual domains, but can also include non-perceptual domains, such as price and shelf life.

The Theory is not intended to be complete. One important aspect for our discussion and that is not well established is related to conceptual spaces uniqueness: how many conceptual spaces there are in an agent's mind? Some works assume a single a conceptual space with a high number of dimensions in which all concepts are represented (e.g., [1]). Some works assume smaller conceptual spaces, sometimes one for each concept (e.g., [e.g., [15]]). Furthermore, distinct agents are normally assumed to have distinct conceptual spaces, which can be aligned by mapping conceptual spaces to symbolic structures and then by symbolic communication [48]. In this paper, we assume each agent (e.g. modeler or expert) has a collection of subjective conceptual spaces focused in specific topics, which may or may not be result from the projection of a universal subjective conceptual space. We also assume that structures of these conceptual spaces are mapped to language structures, which can be communicated symbolically through written artifacts. Furthermore, we assume that the decoding of these artifacts induce a conceptual space.

Other cognitive phenomena can be explained in terms of operations in conceptual spaces. For example, taxonomic reasoning can be defined in terms of region embeddings/projections. Contextual effects can be explained in terms of dimension weighting (see [17] for more details).

[6] I.e., it is not possible to assign a value to an object in one dimension without assigning one in the others.

Concept Composition. From the tools developed for Conceptual Spaces, we require those related to concept composition and introduce these next.

Only in the most simple case can we model the composition of two concepts as their product space, i.e., intersection of the regions in their properties, because only the most simple composition shares properties. This is, for example, the case for "red car", which is composed from "red things" and "car". Such compositions are described by intersecting the region for "red" in the color property of "car" and leaving the rest of the concept of "car" unchanged.

A more common case is that while intersection can be applied to some regions, other regions are *incompatible*. The classical example here is "stone lion". The material property of "lion" has an empty intersection with "stone", so instead the property is *replaced*. However, some of the properties of "lion", namely all which are concerned with living things, are not compatible with things made of stone and are consequently removed from the composed concept. Indeed, the only remaining property of "lion" is its shape.

A similar situation arises in natural language with metaphors, where the composition of concepts cannot be described by removal, addition and interactions of properties, as the concepts share no properties [31]. Metaphors are *"mappings across conceptual domains"* [30] that preserve some cognitive structure not directly accessible on a lexical level, instead they *"preserve the cognitive topology (...) of the source domain, in a way consistent with the inherent structure of the target domain."* [30].

The similarities between metaphors and models run deeper than their shared property of mapping across domains. Both metaphors and models are, in the words of Steen [42], *"not a matter of language but of thought"*. A model is a model because it is thought of as such; there is nothing in a Java program or a first-order formula that gives it its mapping feature without a mind to perform the mapping. Similarly, the adequacy of a model is a property that is inherently non-lexical and cannot be judged without a cognitive approach.

To handle such situations in the Conceptual Spaces framework, [18, Ch. 13] describes two mechanisms: for shared domains, the concept is *projected* on the shared dimensions (in the example above, the lion is projected on its "form" dimension). This is not abstraction, which is removing properties based on the context of the formal modeling. For non-shared domain, a *metaphorical mapping* is used: a homeomorphism between the regions of the two concepts, i.e., an isomorphism preserving structural/topological properties. Through the homeomorphism, structures from one domain can be applied in the other one, which may not posses such structures.

5 Naturalness in Conceptual Spaces

We have so far established a semiotic view on models, argued that naturalness must be explained as a relation between $\mathfrak{M}^\alpha$ and $\mathfrak{M}^\delta$, and introduced Conceptual Spaces and metaphors as a tool to describe such relations between concepts.

Now we can revisit the notions discussed in Sec. 3. To do so, we discuss mental processes that relate artifacts and mental representations, which differ between *enmodeling* as a mental process to generate a mental concept for a given context (which, most likely, is more simple or suited) and *encoding*, for the process that encodes this model in a formal model or symbol.

5.1 Redefining $\mathfrak{M}^\delta$ and $\mathfrak{M}^\alpha$ and Other Mental Models

As we stated earlier, both $\mathfrak{M}^\delta$ and $\mathfrak{M}^\alpha$ are mental representations that base the production/understanding of artifacts $\mathfrak{M}$. Taking Conceptual Spaces as our framework for mental representation, we must then define their nature in terms of conceptual constructs.

We start by introducing $\mathfrak{C}^\delta$ as a region in a conceptual spaces denoting the full conceptualization of a domain concept. It spans properties in perceptual and non-perceptual quality domains representing the overall experience a person might have with exemplars of such concept. For example, for a car expert, $\mathfrak{C}^\delta$ captures aspects related to specialist and common-sense knowledge about cars.

In contrast, $\mathfrak{M}^\delta$ is a region in a conceptual space constructed by domains and subproperties derived from $\mathfrak{C}^\delta$ that are relevant to the task at hand. In our Java car example, $\mathfrak{M}^\delta$ include domains regions related to axles, position and velocity.

Similarly, we introduce $\mathfrak{C}^\alpha$ a region in conceptual spaces denoting one's general knowledge about the constructs and structures in the target formal model. In our Java example, $\mathfrak{C}^\alpha$ equates to the general notion one has of Java classes. Points in this conceptual space denote individual possible Java objects. Its domain structure is more elusive, given its abstract nature. Examples of quality domains include memory position, hash encoding and use of logging.

Finally, $\mathfrak{M}^\alpha$ represents the formalism and task-dependent mental model of the $\mathfrak{M}$. While $\mathfrak{M}^\delta$ might be a property region on a domain denoting the range of possible number of axles a car might have, $\mathfrak{M}^\alpha$ would have a counterpart region on a integer domain with no region defined.

In the following, we use $\mathfrak{C}^\mathrm{x}/\mathfrak{M}^\mathrm{x}$ when δ and α concepts are interchangeable.

5.2 The Place of Abstraction

We next discuss the relation of $\mathfrak{M}^\mathrm{x}$ and $\mathfrak{C}^\mathrm{x}$ and the process involved with their construction. Modeling is a mental process that starts with a concept (i.e., a region in a conceptual space) and ends with an (physical) artifact expressing this concept in a certain context and a certain (formal) language. We hypothesize that it consists of two main steps: *enmodeling* and *encoding*. In reverse, the mental process that starts with an artifact and ends with a concept consists of *decoding* and *demodeling*. Figure 4 depicts these processes for $\mathfrak{M}^\mathrm{x}$ and $\mathfrak{C}^\mathrm{x}$.

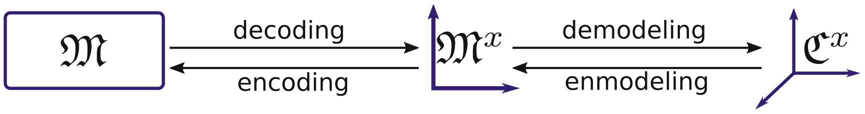


Fig. 4. The mental processes and representations to produce and understand models. The two right representations are cognitive spaces, denoted by blue axes. (Color figure online)

Enmodeling. Enmodeling takes $\mathfrak{C}^{\mathrm{x}}$ and adapts it to the target context and target language. Adaptation to the context is mainly abstraction: the removal and rescaling of dimensions and properties that are not relevant for the context. Adaption to the language is a more elusive process: we conjecture it to be similar to the effects known in linguistics that language influences the way concepts are formed and expressed in natural language. The result of enmodeling is again a concept, a region in a conceptual space. Due to the adaption to the language, it is *not* a subconcept of the one we started with.

Encoding. Encoding starts with the adapted concept and ends with the artifact. It is the generation of artifacts from mental concepts *after* adaption of the concepts. These two processes are not independent: obviously, enmodeling is influenced by the target language and is guided by the expression of concepts in this language. Their relation is also not necessarily sequential, but we conjecture that enmodeling starts before encoding, and that encoding ends after enmodeling. For our discussion, it suffices to regard them as separate and ordered.

Decoding. Decoding starts with the artifact and ends with a adapted concept that is specific to its context (i.e., the application in question where the artifact is used). It is the opposite of encoding and the resulting concept still contains traces of the artifact, as it is done in a certain language and context (which the ending concept is adapted to). Decoding contains numerous mechanisms, for example it is the part that is concerned with *perception.* It also may involve higher cognition mechanisms, like memory.

Demodeling. Demodeling is the opposite process of enmodeling, it starts with a concept that contains traces of the artifact language and ends with a full, not context-specific context. It relates $\mathfrak{M}^\delta$ to a $\mathfrak{C}^\delta$ and has, as one of its main parts, the task to recognize the modeled concept in the artifact. For example, it is during demodeling, when the car expert recognizes the Java class as a *specific* concept from the car domain. The moment when the car enthusiast recognizes the Java class as something from the car domain is during decoding. The main difference of enmodeling and demodeling is their direction w.r.t. complexity of the concept: enmodeling reduces complexity (e.g., by removing a dimension), while demodeling increases it.

Demodeling and enmodeling are not monolithic processes and contain subprocesses which may take the opposite direction w.r.t. complexity as the overall process. The exact subprocesses are, however, not of importance for the phenomena we aim to describe here. The important detail is that (de/en)modeling and (de/en)coding can be distinguished: (de/en)modeling is an internal transformation of mental concepts, while (de/en)coding is their relation with the artifact.

5.3 Formal Models as Metaphors

The end of the decode-demodeling process is a concept independent of the artifact, but in our setting with domain view and engineering view, there are two process with *one* beginning, namely the artifact, and *two* ends.

This means that each involved person, engineer and domain expert, have their own decoding and demodeling process when examining a single formal modeling artifact. Not only are the processes different: the concepts are different as well. At the end of the engineer's demodeling, the concepts describe the artifact in purely technical term. For example, it describes the class in terms of properties of Java classes (e.g., final or not, number of fields) and is, in the extreme case where the engineer has no knowledge about the modeled domain, free of any domain dimensions.

If the artifact is examined by an engineer and a domain expert, the artifact essentially becomes a message. Here, we are however interested in our modeler, for whom the artifact is a concept in both the domain context and the technical context. For enmodeling, the modeler also starts with $\mathfrak{C}^\delta$. It is not possible to start with $\mathfrak{C}^\alpha$, as the main task is to model a domain situation, not to produce (some) working code. For encoding, the modeler needs to operate on $\mathfrak{M}^\alpha$, as for this task the technical knowledge is dominant. Thus, $\mathfrak{M}^\delta$ is an *intermediate* concept, constructed during enmodeling.

We refine our view on enmodeling into two steps: (1) *abstraction*, a process that generates $\mathfrak{M}^\delta$ from $\mathfrak{C}^\delta$ by adapting it to the application scenario and (2) *adaptation*, a process that reformulates $\mathfrak{M}^\delta$ by relating it to the chosen language and technical framework. Similarly, when reading a model, the modeler decodes the artifact into $\mathfrak{M}^\alpha$, then disperses the technical framework to arrive at the abstracted domain concept $\mathfrak{M}^\delta$ and finally relates it to the final concept $\mathfrak{C}^\delta$.

Note that $\mathfrak{C}^\alpha$ is not constructed in this process. However, the modeler is able to suppress the domain side and construct $\mathfrak{C}^\alpha$ directly from $\mathfrak{M}^\alpha$, i.e., act as a technical expert.

As discussed, these extreme views of the technical expert and the domain expert are unlikely to occur. Every domain expert has some basic linguistic knowledge and *must* be able to construct some $\mathfrak{M}^\alpha$. However, due to the lack of technical knowledge, $\mathfrak{M}^\alpha$ is rudimentary – it is comparatively hard to disperse the language specifics of the model to reveal the underlying domain structures.

Metaphors. We see that formal modeling requires to compare the structure of different concepts. Following up on on Lakoff's observation that metaphors are mappings across conceptual domains that preserve cognitive topology, we also see that formal models are merely metaphors themselves: The structure of $\mathfrak{M}^\delta$ must be preserved in $\mathfrak{M}^\alpha$. To define naturalness we can, thus, use the mechanisms

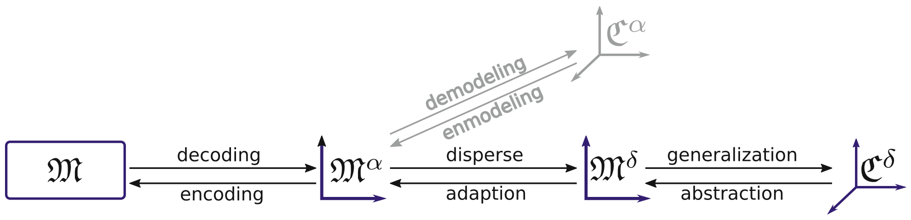


Fig. 5. The mental processes and representations for the formal method expert.

already discussed for concept composition and metaphors through conceptual spaces (Fig. 5).

It is interesting to note that when we see models as metaphors, we use the structure of the formal model to explain effects in the domain. This is the opposite direction from metaphors in everyday use in computer science, which use the structure of some "domain" to explain the formal model. For example, the notion of a *stack* [9] uses the structure of the domain (being able to add on top) to illustrate the computational concept.

Signs. We have now two mental concepts for each sense-making process: the "raw" concept and the adapted concept. The semiotic triad we use to introduce semiosis, however, has no place for the adapted concept. Our solution is that adapted concepts play a role in two sense-making processes, as illustrated in Fig. 6: the adapted concepts. $\mathfrak{M}^\alpha$ and $\mathfrak{M}^\delta$ are a *concept* for the first sense-making process (the one for en-/decoding) and a *symbol* for the second sense-making process (the one for en-/demodeling). Such a sequence of two sense-making processes which share the same thing and where one concept is the symbol of the other, can be seen as an instance of "successive interpretants"(successive concepts) in the Peircean theory of signs.

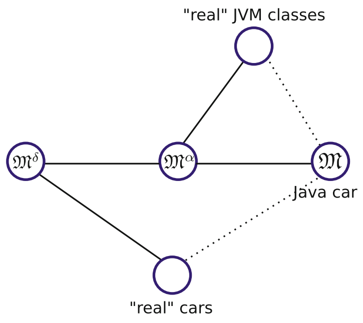


Fig. 6. The semiotic relationships for formal models.

5.4 Naturalness and Perceptional Naturalness

The previous section established a framework for the cognitive processes for formal modeling. Now we use the ideas from cognitive linguistics on concept composition to define of notions of naturalness and perceptional naturalness. Intuitively, a model is natural if it is a good metaphor: there is a metaphorical mapping from $\mathfrak{M}^\alpha$ to $\mathfrak{M}^\delta$ that requires little cognitive effort to map structures from the artifact-view to the domain-view. We now make our notion of naturalness more precise:

> Let $\mathfrak{M}$ and $\mathfrak{N}$ be two formal models for the same aspect of a domain, i.e., the same $\mathfrak{M}^\delta = \mathfrak{N}^\delta$. Let μ be a metaphorical mapping from $\mathfrak{M}^\alpha$ to $\mathfrak{M}^\delta$ and ν the corresponding metaphorical mapping from $\mathfrak{N}^\alpha$ to $\mathfrak{N}^\delta$.
> We say that $\mathfrak{M}$ is more natural than $\mathfrak{N}$ if μ has a lower cognitive complexity than ν.

In short: a model is more natural than another if it is easier to recover the domain conceptual structure from the artifact conceptual structure. Cognitive complexity denotes the effort needed to perform the metaphorical mapping, which in our setting we interpret as the computational complexity of the metaphorical mapping, if we see the metaphorical mapping as a function between two metric spaces. Using the computational complexity in cognition has a certain appeal when comparing the mind with computers [47], but here we do not use it for general assumption about cognition, but to measure the complexity of a certain cognitive task. For a more detailed discussion on computational complexity effects in cognition we refer to Isaac et al. [26].

In this setting, the complexity of the metaphorical mapping is harder to grasp, as several components are moving: both conceptual spaces are influenced by mental changes, i.e., they change their shape through learning. For example, the space for the language gains more dimensions as the domain expert gains more experience with it. Furthermore, operations within the mapping become computationally cheaper if they are performed more often – analogously to results in natural languages, where less frequent language fragments have higher complexity (in terms of the logic needed to formalize it) [45]. Thus, the computational model in terms of needed operations may also change over time. Consider the example of the first-order logic formula formalizing a car with four wheels. The unit describing that the four wheels are different requires conscious reading of all conjuncts for the novice, i.e., a *linear* complexity in the length of the formula, but after more exposure to the usual patterns in logical modeling, this is reduced into one reasoning step, i.e., *constant* complexity.

If we fix a threshold for low cognitive complexity, then we can give a definition of naturalness that does not require a second model to compare with.

> Let $\mathfrak{M}$ be a formal model. Let μ be the metaphorical mapping from $\mathfrak{M}^\alpha$ to $\mathfrak{M}^\delta$. We say that $\mathfrak{M}$ is natural if μ has a low cognitive complexity.

If we accept the P-cognition thesis that "cognitive capacities are limited to those functions that can be computed in polynomial time" [47], than, in our eyes, a sensible assumption would be that μ is natural if it is even less complex than polynomial. One obvious candidate would be linearity, but we leave this question open.

Let us return to the example in Fig. 2, in particular the number of axles. Let us assume that in $\mathfrak{M}^\alpha$, the field of a Java class, is represented as a property with the dimensions "type" = integers, "modifier"=private and "name"=nrAxles. Note that nrAxles here is purely symbolic and not connected to the concept of axles at all. In $\mathfrak{M}^\delta$ the number of axles is just a single dimension over an interval in the natural numbers.

The mapping μ is of low cognitive complexity: exactly one property is mapped onto directly one dimension and both the name and type of the field are directly related to the domain dimension. In terms of concept composition, one can simply perform a *property replacement*. The additional dimension of the modifiers in the engineering domain can just be removed in the mapping – it must not be disentangled from the other dimensions.

Now consider a Java class where the number of axles is modeled as following:

```java
private int wheels;
private int wheelsPerAxis;
...
public int getAxles() { return wheels/wheelsPerAxis; }
```

This is less natural: $\mathfrak{M}^\alpha$ now has three different properties and $\mathfrak{M}^\delta$ must include the notion of wheels, i.e., be more precise in its representation of axles. The mapping is more complex: three properties are mapped onto one property. The property of the method is furthermore more complex and involves arithmetic.

Next, let us examine the use of `int` as a type for the number of axles. It is rather unnatural, because it allows to create `Car` instances that cannot be mapped to points in the conceptual space $\mathfrak{M}^\delta$ for car; e.g., those with a negative number of axles. This is obviously not a metaphor: the structure provided by the formal model does not carry over the application domain. It is a consequence of a, possibly conscious, modeling decision to use integers, as these are easily available in the language, while, for example, ranges are not (in Java). This is an example of how none of the processes is performed in isolation – the choice of the target language already influences the enmodeling process. The use of integers is still relatively natural, as (a) integers are often used to overapproximate ranges and (b) one dimension is mapped onto one other.

Finally, we discuss the serialization framework. It is completely foreign to the domain of cars, but some parts are more unnatural than others. The annotations, such as @XmlAttribute are unnatural, but they are easy to ignore – the cognitive mapping just removes the dimensions related to annotations. It is also unnatural that nrAxels can be changed. But while it is also effectively removed in the cognitive mapping, this requires more cognitive complexity compared to the removal of annotations, because it models possible behavior that cannot be *fully* ignored. Indeed, it contradicts the domain – and the cognitive mapping must thus involve more domain reasoning why this contradicting behavior can be ignored. We expect that over time, i.e., after working with the model for some time, memory and association effects will make it more natural as the reason becomes part of the memory.

Similarly, the *cake* variable in our FOL example increases complexity, as it breaks the context: it implies that the context contains notions of baking, which activates the wrong memories and makes it harder to understand what symbols carry domain information and which do not. More generally, syntax highlighting is a technique to increase perceptional naturalness by reducing the cognitive burden required to build the concept $\mathfrak{M}^\alpha$.

Perceptional Naturalness. While we support the idea that naturalness is mainly associated with dispersion and adaption mappings, there are formal language features that are more perceptual but that also influence how easily a formal artifact can be understood. Consider the block structure in the first-order example in Sect. 3 or the use of syntax highlighting. These are features of formal models—some of them *ad hoc*—that helps decoding.

These effects are associated to what we call *perceptional naturalness*. In our view, an artifact is perceptually natural if it has *low decoding complexity*. Since decoding is outside the scope of our Conceptual Space-based framework (decoding is not purely conceptual), we do not investigate decoding complexity further.

Note that the mutability of `nrAxles` above does not fall under perceptional naturalness: the mutability of the field is unnatural, not the absence of a **final** modifier and the presence of the setter. In general we consider most iconicity effects to fall under perceptional resemblance, but iconicity plays little role for the formal languages that we consider here. For example, the only iconicity in the car formula is the occurrence of four different variables for the four wheels.

Further Details. Naturalness is defined in terms of understanding the model, i.e., how easy it is to disperse the language structure, but still related to adaption, i.e., how easy it is to model something: natural pairs of concepts are easy to compose (by replacing the domain properties by engineering properties) and easy to decompose. It is able to explain why something is natural to *express* is a certain way, as well as able to explain why something is natural to *understand*.

Naturalness is more important to the modeler than to the domain expert or the technical expert: the technical expert is not interested in $\mathfrak{M}^\delta$, except when it relies on his common sense for explanations[7]. For the domain expert, the domain view is dominant, so the domain expert decodes into $\mathfrak{M}^\delta$ *almost directly*, as the domain expert has too few dimensions and properties to build a sensible concept $\mathfrak{M}^\alpha$. For the domain expert, there is no sharp difference between dispersal and generalization – naturalness and perceptional naturalness merge.

6 Discussion

Consequences for Interdisciplinary Modeling Studies. We once again stress that naturalness is a purely mental notion and, as such, different for every person: it

[7] For example, we can assume any programmer to have some knowledge about cars.

is not possible to reason about the naturalness of an artifact per se, as without an interpreter no sense-making processes occur.

However, we can reason about naturalness is a restricted context beyond a specific person: given a certain domain and application, we can assume that the domain concepts have similar structures for different people working in a field, due to common education and experiences. Thus, if one such person perceives a model as natural, it is likely that this generalizes within the target group.

We can, thus, also make assessments of naturalness of formal modeling *languages*: a language is more natural than another, if there is a more natural model in it. We can approximate this by trying to map the core concepts of the domain on constructs within the existing language. For example, consider a mail service to send letters. It is more natural to model such a service using the actor concurrency model than, let's say, in a shared memory model with semaphores, because the basic language feature of *asynchronous messages* shares structure with *sending a letter*, because both may be reordered and require, in general, no waiting for a response. In contrast, to model the same property requires a more complex formulation when using semaphores.

This is, in essence, the underlying assumptions why domain-specific languages work in practice: if the vocabulary and constructs are fixed and the target group shares education and experiences, then they find it natural to express themselves in it, i.e., to write natural formal models, if the language has primitives for common relations and actions.

It follows from the above that the modeler is *not* able to judge the naturalness of the formal model until the modeler is trained enough to align the domain concepts with the one of established domain experts. This confirms our experiences in the common setting where the modeler starts as a technical expert and acquires domain knowledge until the modeler can take the role of the mediating formal method expert: The first iterations of a formal model are mainly useful to find out where the preliminary intuition of the formal method expert is still wrong. Yet, we found early prototypical models of critical importance to establish a successful interdisciplinary collaboration: these models train both technical expert (i.e., the modeler to-be) and the domain expert to use models for communication and, thus, lower the cognitive complexity needed for both when working on common artifact. We conjuncture, based on these experiences, that *common decoding experiences* are more important for formal modeling than establishing *common knowledge up front.*

Lastly, we note that the metaphors established by the formal model can transfer novel structures into the domain: For example, the notion of a logical group is used recently for infrastructure in railways by Schön [39], but stems from its formal modeling as a common object-oriented pattern to group objects for communication [28].

Objective Naturalness. Our notion of modeling and naturalness is subjective, relying on the inner workings of the mind, and we do not investigate *objective* naturalness, which would directly connect the semiotic symbol with the semiotic thing (Fig. 1). Indeed, it is questionable whether such a notion could exist. One can argue that domains have inherent structure, which should be natural to any model and modeler acquainted with the domain. This brings our discussion back to the above point about domain-specific languages, which aim to provide a natural model *for any mind*, and we stress that this is not the same as a natural model *without an involved mind*, which does not exist in our framework[8].

Empirical Evaluation of Naturalness. As such, any precise, objective assessment the naturalness would have to rely on direct measurement of cognitive complexity of individual artifact-person (or artifact-mind) pairs. That would in turn require direct access to mental representations, which is still beyond the present state of the art. On the other hand, indirect characterizations of naturalness across formalisms, artifacts and mind types might still be possible within the realm of experimental Cognitive Sciences. We let this issue for future work.

User Studies in Formal Methods. Formal methods, as well as related disciplines, rarely perform user studies that target understanding and tend to reuse theories from human-computer interaction. Consequently, they are restricted to usability questions. For example, Hentschel et al. [23] propose a new tool for interactive theorem proving that is motivated by enabling the user to understand the formal system better:

> *"To improve the efficiency of understanding intermediate proof situations, therefore, promises considerable gains in the overall human user time spend..."*

However, their study is purely *performative* and only investigates whether the tool increases the performance with respect to time and correctness. It does not investigate whether the tool indeed improves understanding.

Similarly, Harkes [22] discusses the problems when evaluating domain-specific languages, where the standard approach in that field is to discuss (1) performance and (2) generality, because these are simple to measure and simple to argue over. This problem of arguing about languages is particularly explicit in the work of Myers et al. [33] on natural programming, which argues that programming languages and environments should be "natural":

> *"By natural, we mean faithfully representing nature or life"*

We regard this definition as little useful in practice, as it gives no detail to why a model is more natural than another and ignores that naturalness differs between individuals. Note, however, that Myers et al. are interested in program development and are *"aiming for the language and environment to work the way that nonprogrammers expect"*. They are not considering single models/programs.

While not a user study, the presentation of Leuschel [32] is worth mentioning in this context: it argues that the reason why the B-method is so successful in

[8] In the semiotic framework there is no such thing as a model *at all* without an involved mind, as a model is a sign and a sign needs an interpreter.

modeling railway systems is that railway systems are modeled as graph structures and the B-method is well-suited to operate on such graph structures. We interpret this as a naturalness argument in our sense: given a B-model, it is easy to retrieve the domain view from the computational structures.

Further Related Work. We are interested in formal models from the perspective of cognitive linguistic and largely ignore the actual evaluation or runtime semantics. Indeed, we do not require a computer or (runtime) semantics in the first place. Tanaka-Ishii [44] gives a more detailed view on programs, where the symbols signify their semantics, which are, in turn, again signs. The sense-making process in that setting is not mental, but physical. Other works on semiotics in computing, such as the one by Andersen [2], also focus on the relation of the sign to the execution itself. In contrast, there is a tradition of Semiotic Engineering in Human-Computer Interaction (HCI) [12] that sees computers as devices for communication, not computation and draws some parallels between the development of programming and the evolution of "natural" languages [5][9]. This research has also resulted in some cognitive guideline for dimensions for *usability* of programming languages environments [6]. From these dimensions *Closeness of mapping* comes closest to naturalness. Similarly, for business processing modeling languages understandability has been investigated by Fahland et al. [14].

[9] investigate metaphors for programming, which has a rich vocabulary of metaphors such as *thread* or *throw/catch*. They observe that for programming, the metaphors are, in the terms of [25], better explained through rather *comparative* theories. Comparative theories of metaphors see metaphors as ways to emphasize and expose preexisting similarities between two domains. In contrast, we use an *interactive* theory of metaphors, where the metaphors creates the similarity. Furthermore, Colburn and Shute discuss metaphors in the opposite direction: While formal models are computer scientific structures that are metaphors for some domain, their metaphors are terms from some domain for computer scientific structures. In subsequent work, this approach is applied to types [10]. Metaphors are also used widely in HCI [4].

Another connection between the philosophy of science and formal modeling has been explored by Hähnle [21], who notes that black boxes have, in general, a negative connotation in philosophy, as they prevent the investigation of its content, while they have, again in general, a positive connotation in computer science, because they hide complexity.

Works in in Ontology Engineering in Computer Science also touch in some of the notions we discussed here. Guarino [19] proposed that ontologies specified as logical theories should approximate the set of intended models (i.e. first-order models) in the the domain, without investigating how the mismatch might occur. Guizzardi [20] suggests a similar distinction between mental models of domain concepts and artifacts, as well as their representation as symbolic specifications.

[9] On the problems of applying the theory of evolution to developments of programming languages we refer to [11].

Also, he summarizes a collection of mappings to characterize how well a formal model covers a domain. However, the work also does not investigate in more detail how artifact-specific constructs affect ontologies. Furthermore, some works in ontology also incorporate Conceptual Spaces as a representation construct [1,20], however we go further in representing the formal artifact itself.

7 Conclusion

This article presents a cognitive view on formal modeling motivated by the observation that several effects in formal modeling that cannot be analyzed by focusing on abstraction, i.e., the reduction feature of models.

At the core are two proposals. (1) That there are two sense-making processes associated with a formal model, one that interprets the formal model as a concept in the application domain and one that interprets it as a concept in the engineering domain. We represent these mental concepts using conceptual spaces. (2) That a model is more natural than another, if it is a better metaphor, i.e., it retains more structure from the engineering view in the domain view.

The notion of naturalness is our main contribution: a formal model is more natural than another if it is cognitively easier to map the extracted artifact concept on the extracted domain concept. As naturalness is concerned with mental processes starting and ending with mental concepts, we also introduce *perceptional naturalness* as a notion of complexity to measure the difference between the artifact itself and the artifact concept it is decoded into. This captures a wide range of effects of more syntactical nature, e.g., formatting and naming.

Contrary to prior work, we do not focus on programming, i.e., execution, or user interfaces, but on a single aspect of formal modeling. We hope that a cognitive view on formal modeling can lead to better designed modeling languages, better designed qualitative user studies and help to build a body of experiences in formal modeling.

Future Work. The natural next step is to design user studies to empirically test our view. A promising start to do so is to investigate are common modeling experiences for interdisciplinary modeling efforts. Furthermore, as we are motivated by the difficulties of justifying and precisely argue about modeling decisions, we also plan to reinvestigate recent successful formal modeling projects, namely `FormbaR` [28], the `GeoAssistant` [13], and the core ontology for robotics and automations [16,24], in particular its positioning part [7], and present the underlying modeling decisions using the framework presented here.

We conjecture that investigating the connection with semiotics in more detail can give further insights into modeling. For example, the situation of the modeler can be seen as multiple parallel signifying processes (cf. Bateman [3, Fig. 2]). Furthermore, the relation of $\mathfrak{M}^\delta$ and $\mathfrak{C}^\delta$ has similarities to the relation of the dynamic and final interpretant of Peirce [8].

Acknowledgement. This work was partially supported by the Research Council of Norway via the `SIRIUS` Center (237898) and the `PeTWIN` project (294600). The authors thank Lars Tveito and Michael Lienhardt for feedback on early drafts of this article.

References

1. Aisbett, J., Gibbon, G.: A general formulation of conceptual spaces as a meso level representation. Artif. Intell. **133**(1–2), 189–232 (2001)
2. Andersen, P.B.: A semiotic approach to programming. In: Learning in Doing: Social, Cognitive and Computational Perspectives, pp. 16–67. Cambridge University Press, Cambridge (1994)
3. Bateman, J.A.: Peircean semiotics and multimodality: towards a new synthesis. Multimodal Commun. **7**(1), 20170021 (2018)
4. Blackwell, A.F.: The reification of metaphor as a design tool. ACM Trans. Comput. Hum. Interact. **13**(4), 490–530 (2006)
5. Blackwell, A.F.: 6,000 years of programming language design: a meditation on eco's perfect language. In: Diniz Junqueira Barbosa, S., Breitman, K. (eds.) Conversations Around Semiotic Engineering, pp. 31–39. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56291-9_5
6. Blackwell, A.F., et al.: Cognitive dimensions of notations: design tools for cognitive technology. In: Beynon, M., Nehaniv, C.L., Dautenhahn, K. (eds.) CT 2001. LNCS (LNAI), vol. 2117, pp. 325–341. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44617-6_31
7. Carbonera, J.L., et al.: Defining positioning in a core ontology for robotics. In: IEEE/RSJ, pp. 1867–1872. IEEE (2013)
8. Chandler, D.: Semiotics: The Basics, 3rd edn. Routledge, Abingdon (2017)
9. Colburn, T., Shute, G.: Metaphor in computer science. J. Appl. Logic **6**(4), 526–533 (2008)
10. Colburn, T.R., Shute, G.M.: Type and metaphor for computer programmers. Techné Res. Phil. Technol. **21**, 71–105 (2017)
11. Crafa, S.: Modelling the evolution of programming languages. CoRR, abs/1510.04440 (2015)
12. de Souza, C.S., Leitão, C.F.: Semiotic engineering methods for scientific research in HCI. Synth. Lect. Human-Center. Inf. **2**, 1–122 (2009)
13. Din, C.C., Karlsen, L.H., Pene, I., Stahl, O., Yu, I.C., Østerlie, T.: Geological multi-scenario reasoning. In: 32nd Norsk Informatikkonferanse, NIK. Bibsys Open Journal Systems, Norway (2019)
14. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: the issue of understandability. In: Halpin, T., et al. (eds.) BPMDS/EMMSAD -2009. LNBIP, vol. 29, pp. 353–366. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01862-6_29
15. Fiorini, S.R., Abel, M.: Part-whole relations as products of metric spaces. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, pp. 55–62. IEEE (2013)
16. Fiorini, S.R., et al.: A suite of ontologies for robotics and automation [industrial activities]. IEEE Rob. Autom. Mag. **24**(1), 8–11 (2017)
17. Gärdenfors, P.: Conceptual Spaces: The Geometry of Thought. MIT press, Cambridge (2004)

18. Gärdenfors, P.: The Geometry of Meaning: Semantics Based on Conceptual Spaces. MIT press, Cambridge (2014)
19. Guarino, N.: Formal ontologies and information systems. In: Formal Ontology in Information Systems, Proceedings of FOIS 1998. IOS Press (1998)
20. Guizzardi, G.: Ontological foundations for structural conceptual models. PhD thesis, University of Twente (2005)
21. Hähnle, R.: Colorful boxes. In: The 7th Biennial Conference of the Philosophy of Science in Practice, pp. 147–148. University of Ghent, Faculty of Arts and Philosophy (2018)
22. Harkes, D.: We should stop claiming generality in our domain-specific language papers. In: The Art Science, and Engineering of Programming, p. 3 (2018)
23. Hentschel, M., Hähnle, R., Bubel, R.: An empirical evaluation of two user interfaces of an interactive program verifier. In: ASE, pp. 403–413. ACM (2016)
24. Ora, I.E.E.E., WG,: IEEE standard ontologies for robotics and automation. IEEE Std. **1872**, 1–60 (2015)
25. Indurkhya, B.: Metaphor and cognition: an interactionist approach. In: Studies in Cognitive System (1992)
26. Isaac, A.M.C., Szymanik, J., Verbrugge, R.: Logic and complexity in cognitive science. In: Baltag, A., Smets, S. (eds.) Johan van Benthem on Logic and Information Dynamics. OCL, vol. 5, pp. 787–824. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06025-5_30
27. Johnsen, E.B., Steffen, M., Stumpf, J.B., Tveito, L.: Resource-aware virtually timed ambients. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 194–213. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_12
28. Kamburjan, E., Hähnle, R., Schön, S.: Formal modeling and analysis of railway operations with active objects. Sci. Comput. Program. **166**, 167–193 (2018)
29. Kühne, T.: Matters of (meta-)modeling. Softw. Syst. Model. **5**(4), 369–385 (2006)
30. Lakoff, G.: The Contemporary Theory of Metaphor, 2nd edn., pp. 205–251. Cambridge University Press, Cambridge (1993)
31. Lakoff, G., Johnson, M.: Metaphors We Live By. University of Chicago Press, Chicago (1980)
32. Leuschel, M.: The unreasonable effectiveness of B for data validation and modelling railway systems. RSSRail, Keynote (2017)
33. Myers, B.A., Pane, J.F., Ko, A.J.: Natural programming languages and environments. Commun. ACM **47**(9), 47–52 (2004)
34. Olmstead, B.: Reference Malbolge interpreter (1998). https://www.lscheffer.com/malbolge_interp.html, Accessed 29 oct 2021
35. Peirce, C.S.: The Collected Papers of Charles Sanders Peirce. Harvard University Press, Harvard (1935)
36. Peled, D.A.: Software testing. In: Software Reliability Methods. TCS, pp. 249–278. Springer, New York (2001). https://doi.org/10.1007/978-1-4757-3540-6_9
37. Quinlan, D., Wells, J.B., Kamareddine, F.: BNF-style notation as it is actually used. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) CICM 2019. LNCS (LNAI), vol. 11617, pp. 187–204. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23250-4_13
38. Rosch, E., Mervis, C.B.: Family resemblances: studies in the internal structure of categories. Cogn. Psychol. **7**(4), 573–605 (1975)
39. Schön, S.: Formalisierung von betrieblichen Regelwerken. In: SRSS 2021 Tagungsband, TU Darmstadt (2021). (in German)
40. Sivik, L., Taft, C.: Color naming: a mapping in the IMCS of common color terms. Scand. J. Psychol. **35**(2), 144–164 (1994)

41. Stachowiak, H.: Allgemeine Modelltheorie. Springer, Heidelberg (1972). (in German). https://doi.org/10.1007/978-3-642-69706-7_56
42. Steen, G.J.: The contemporary theory of metaphor - now new and improved! Rev. Cogn. Linguist. **9**(1), 26–64 (2011)
43. Stehr, M.-O., Meseguer, J.: Pure type systems in rewriting logic: specifying typed higher-order languages in a first-order logical framework. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) From Object-Orientation to Formal Methods. LNCS, vol. 2635, pp. 334–375. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39993-3_16
44. Tanaka-Ishii, K.: Semiotics of Programming. Cambridge University Press, Cambridge (2009)
45. Thorne, C.: Studying the distribution of fragments of English using deep semantic annotation. In: 8th Workshop in Semantic Annotation ISA 8 (2012)
46. Ullmann, S.: Semantics: An Introduction to the Science of Meaning. Basil Blackwell, Oxford (1972)
47. van Rooij, I.: The tractable cognition thesis. Cogn. Sci. **32**(6), 939–984 (2008)
48. Warglien, M., Gärdenfors, P.: Semantics, conceptual spaces, and the meeting of minds. Synthese **190**(12), 2165–2193 (2013)
49. Wittgenstein, L.: Philosophical Investigations. Basil Blackwell, Oxford (1953)

The Karlsruhe Java Verification Suite

Jonas Klamroth[1], Florian Lanzinger[2], Wolfram Pfeifer[2], and Mattias Ulbrich[2](✉)

[1] FZI Research Center for Information Technology, Karlsruhe, Germany
klamroth@fzi.de

[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{florian.lanzinger,wolfram.pfeifer,mattias.ulbrich}@kit.edu

Abstract. Thanks to the deductive verifier KeY, the formal verification of Java programs has a long-standing tradition at Karlsruhe. The design of KeY implies some properties that can restrict its use in real-world application cases: (1) Verifying long, code-intensive methods with many instructions or bit-wise operations is difficult even if their behaviour is not overly complex, and (2) tracking formal guarantees through unverified code is difficult if not impossible using KeY.

To mitigate these weak spots, we introduce the *Karlsruhe Java Verification Suite*, a collection of formal Java verification tools that work with the Java Modeling Language (JML). Complementing KeY, the suite comprises *JJBMC*, a bounded model checker for Java and JML and the *Property Checker*, a type checker for user-defined property types.

In this paper, we first discuss formally how tools sharing a common specification language can share distributed obligations in a general setting, and then specialise this to the case of Java and our tool suite.

In a case study, we show that the Karlsruhe Java Verification Suite can verify a program that none of the three components could have proved alone.

Keywords: Software verification · Modular design · Design by contract · Software bounded model checking · Pluggable type systems · Deductive verification · Refinement types

1 Introduction

The KeY project [1] was initiated more than 20 years ago at Karlsruhe University, with Reiner Hähnle one of the founders of the project. The deductive proof engine for the formal verification of the correctness of formally specified Java code has since been an important player in the world of formal analysis of Java programs. KeY is still an active project that is now co-developed in Karlsruhe, Darmstadt and Gothenburg.

Over the years we have observed that when applying formal Java verification closer to practical application cases, there are properties of KeY which make it hard to apply KeY easily in practical situations:

© Springer Nature Switzerland AG 2022
W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 290–312, 2022.
https://doi.org/10.1007/978-3-031-08166-8_14

1. *Some parts of a program to be verified may not fall into the fragment that KeY can handle well.*
   While the symbolic execution of Java programs in KeY models the Java semantics very precisely, it does not scale very well and verifying larger sections of code may hamper the verification significantly even if the code is loop-free and has a simple specification. Since KeY models the bounded integer types in Java using unbounded mathematical integers and modulo operations, verification of programs using bit-wise operations (like XOR) are also difficult.
   It would be beneficial if the power of KeY could be complemented by verification routines that are particularly good on these domains.
2. *It may be necessary to track formal guarantees through code outside the verified core.*
   Usually one can identify a critical core of a program onto which formal verification is applied to guarantee its correctness. However, it is not unusual that data leaves this verified core, is processed in code areas with a lower criticality level (like a user interface) and then, later, reenters the verified core. Since KeY does not scale well enough to verify entire code bases, it would be beneficial to complement KeY with approaches which scale well and allow verified properties to be propagated through large code bases.

In this paper, we present how two approaches and their corresponding tools complement KeY with functionalities that fill precisely these two gaps. It is a deliberate decision that they are not tightly integrated into KeY, but collaborate with KeY using the *Java Modeling Language (JML)* [19] as their common specification language. The rationale behind this loose coupling is that thus no technical tool-specific encoding or implementation details must be considered outside each verification tool. Using the common specification language JML as the interface makes it possible to easily incorporate other tools than ones presented here into the approach.

JJBMC [2] complements the deductive verification engine in KeY by a component for bounded model checking. It translates JML specifications into assumptions and assertions in the code, which can then be analysed using bounded model checking. While bounded model checking can in general not fully prove properties about programs with loops or recursion, it is well-suited to loop-free programs with many cases, which often slow KeY's proof search to a crawl. Unlike KeY, which models Java bounded integers using unbounded mathematical integers, JJBMC can deal well with bit-wise operators.

The *Property Checker* [18] brings together the expressive power of formal specification and verification with KeY and the scalability of lightweight verification using decidable type systems by translating type qualifiers whose correctness cannot be shown by the type system into JML annotations. We will show how this combination can be used to reduce the specification and verification overhead of proving program correctness.

Since KeY and these two approaches are currently actively developed at KIT in Karlsruhe, the combination of approaches and tools form the *Karlsruhe Java Verification Suite*. The collaboration between the tools works by a *distribution of proof obligations* among them. We discuss for a simple while language with embedded assumptions and assertions, how the assertions to be proved can be distributed between different approaches and when such a distribution is correct. We formally prove this in Theorem 1 in Sect. 4. We then (in Sect. 5) informally lift this result from the while language to JML-annotated Java and show how the tool collaboration there can follow the same principles as in the simpler language. We discuss a few points necessary for an application of the approach in the field (semantic coherence, proof management).

We have implemented a small wine-store example illustrating the benefits of the collaboration within the Karlsruhe Java Verification Suite. The critical core is a sorting routine which is verified using KeY, supported by JJBMC for long linear code and bit-level operations within it. The application has a graphical user interface outside the verified core. The Property Checker is used to propagate the sortedness property through the non-core code. By joining forces, the three tools can together show that the contracts are satisfied. No tool would have been able to show this alone.

The main contributions of this paper are the following:

- an approach to combine verification tools that follow different formal analysis approaches but share a common specification language,
- a formalisation of this approach for a while language with embedded assumptions and assertions and a formal correctness proof,
- the description of an instantiation of the combination idea with three tools (KeY, JJBMC, Property Checker) that collaborate using JML,
- a case study for the JML combination which the tools can only verify collaboratively.

2 The Java Modeling Language

The Java Modeling Language (JML) [19] serves as lingua franca for the different tools in the Karlsruhe Java Verification Suite. Therefore, this section provides a short introduction to its concepts. JML is a behavioural specification language for sequential Java programs and the de facto standard in the Java verification community. It is designed to be close to the Java language and thus comparatively easy to write and understand for Java developers. JML follows the principle of *design by contract* [21]. This means that specifications are written at the method level using *contracts*, which abstract from the method's behaviour. A contract in general consists of a precondition and a postcondition with the semantics that *if* the precondition holds at the method call, the postcondition must hold after returning from the method. The use of contracts allows one to divide the complexity of large software into smaller parts, which can then be reasoned about individually (*modular reasoning*). However, to be able to verify

```
/*@ normal_behavior
  @  requires 0 <= a < array.length;
  @  requires 0 <= b < array.length;
  @  ensures array[a] == \old(array[b]);
  @  ensures array[b] == \old(array[a]);
  @  assignable array[a], array[b];
  @*/
public static void swap(int[] array, int a, int b) {
  int tmp = array[a];
  array[a] = array[b];
  array[b] = tmp;
}
```

Fig. 1. A formally specified `swap` method as an example JML method contract.

the contracts using a deductive program verifier, it is often necessary to provide additional helper specifications. The most prominent ones are *loop invariants* and *framing clauses*: The former can be used to conduct induction proofs over the number of loop iterations, while the latter are used to specify an upper bound of the heap locations written by the method. These auxiliary specifications are often difficult to write and error prone. In addition to contracts, JML also supports inlined specification statements like explicit assertions or assumptions.

To be transparent to Java compilers, all JML annotations are embedded into Java comments that start with an 'at' sign after the comment delimiter (i.e. `//@` and `/*@`). JML intends to be precise on the one and concise on the other hand; therefore it imposes some useful defaults for specification, for example that fields, parameters and return values are non-null by default.

Figure 1 provides an example of a JML method contract. The keyword `normal_behavior` is used to specify that the method always terminates and does not throw an exception. Under the precondition (`requires`) that the given indices `a` and `b` are in the bounds of the array, the contract states that after execution of the method the two elements will be swapped (`ensures`). The operator `\old(...)` is used inside the postconditions to evaluate an expression in the method's pre-state rather than in its post-state. Furthermore, a framing clause (`assignable`) is given: The method is at most allowed to write to the heap locations of the two elements of the array.

A number of formal tools to reason about JML specifications has been implemented over the years, with KeY and *OpenJML* [8] the most actively developed tools today. In addition to deductive verification, OpenJML also supports runtime assertion checking. There are also other dynamic verification tools for JML like for example JMLUnitNG [24], which allows users to create unit tests with test oracles automatically generated from JML specifications.

To enable parts of the specification only for specific tools, JML brings the feature of *annotation markers*: If an annotation contains the tool name prior to the 'at' sign (e.g. `//+KeY@`), this annotation is only to be considered by the named tool, other tools have to ignore it. Likewise, specific JML clauses can be explicitly disabled for some tools via `//-<toolname>@`. Therefore, some assertions can be discharged by specific tools, while other tools may assume them afterwards. Of course, one has to be careful not to conduct an unsound circular proof.

3 Tools in the Karlsruhe Java Verification Suite

The specification language JML is the common denominator and the communication means by which the components of the verification suite can interact and combine and exchange their verification results. In the following we represent the main three components of the suite (co-)developed in Karlsruhe: *KeY* as a full fledged deductive verification tool at the heart of the tool suite, *JJBMC* as a more versatile, flexible, well-scaling bounded verification tool for more lightweight static checking and the *Property Checker* as a means to check lightweight formal properties in a well-scaling type checker.

While the presentation in this paper and in the case study in Sect. 6 focus on these tools developed at KIT, the described approach is by no means limited to them. On the contrary, since the only requirement is support of the JML language, other tools that operate using this language can be naturally incorporated as well. In particular, the deductive JML verification engine OpenJML fits seamlessly into the tool suite.

3.1 KeY

KeY [1] is a tool for deductive verification of Java programs which are formally specified in JML. At its core is a sequent calculus working on Java Dynamic Logic (JavaDL) formulas. This logic features the modal operators $[p]$ and $\langle p\rangle$ (‘box p’ and ‘diamond p’) parametrised by a Java program p. The formula $[p]\psi$ is valid iff starting in any pre-state, either the program p does not terminate or it does terminate and ψ holds in the post-state of its execution. In contrast to that, $\langle p\rangle\psi$ is valid iff the program terminates and ψ holds in the state afterwards. In dynamic logic, the Hoare triple $\{\phi\}p\{\psi\}$ with a precondition ϕ, a program p, and a postcondition ψ can be expressed as $\phi \rightarrow [p]\psi$. In general, dynamic logic is more expressive than Hoare logic, since the formulas can contain nested modalities again, which enables the specification of, for instance, the equivalence of two programs. In KeY, multiple modalities are used for example to formulate proof obligations for information flow in a very intuitive and natural fashion, whereas in Hoare logic additional constructs like Hoare Quadruples would have to be introduced for this.

Besides modalities, JavaDL extends first order dynamic logic by a type hierarchy suitable for Java. In particular, the types Heap, Object, Field, and LocationSet are included to be able to model and reason about memory properties of Java programs using the theory of arrays [20] and dynamic frames [16].

The usual workflow in KeY is as follows: After loading the method contract to be proven, KeY creates a JavaDL proof obligation whose validity entails the correctness of the method wrt. the contract. Next, the program is symbolically executed by applying a series of sequent calculus rules that transform the code inside modalities into substitutions outside of them. Eventually, symbolic execution terminates in a proof tree with one or more branches which contain only first-order formulas without modalities. While in theory, the validity of formulas already in first-order logic (and thus also in JavaDL) is undecidable, in practice, it is possible to find a proof for many instances even automatically by using well-designed heuristics built into KeY. However, in case the automatic proof search fails, KeY provides the possibility to apply rules interactively, which further increases the number of provable instances.

3.2 JJBMC

JJBMC [2] is a command-line tool for the verification of JML-annotated Java code based on the bounded model checker JBMC [9]. The tool provides an automatic translation of JML specifications to pure Java code with additional assertions, assumptions and non-deterministic value assignments. This translation is a purely syntactical replacement function $\mathit{trans}(\cdot) : \mathit{JML} \cup \mathit{Java} \rightarrow \mathit{Java}$. It relies on the base idea of translating a method contract as first assuming the precondition, then executing the method body, and finally asserting the postcondition. This can be formally expressed as follows:

$$\mathit{trans}\left(\begin{array}{l}\texttt{/*@ requires } R\texttt{;}\\ \quad\texttt{@ ensures } E\texttt{; */}\\ \texttt{\{ } B \texttt{ \}}\end{array}\right) = \begin{array}{l}\mathit{trans}(\texttt{assume } R)\texttt{;}\\ \mathit{trans}(B)\texttt{;}\\ \mathit{trans}(\texttt{assert } E)\texttt{;}\end{array}$$

The transformation *trans* is recursively defined on all statement and expression constructors. While some Java expressions are their own translation directly, some JML-specific expressions like quantifiers require a more sophisticated translation which may involve additional code, like loops in the case of a quantification over an integer range. The JML example presented in Fig. 1 gets translated into the Java code in Fig. 2.

The bounded model checker JBMC is then able to analyse the result of the translation and thus verify each method wrt. its contract. By using JBMC as a back end, JJBMC inherits the bounded analysis semantics of JBMC:

The key idea of bounded verification is to consider only program runs which are bounded by a given threshold in loop iterations and recursive method calls. In particular, this allows the bounded analysis to unroll loops, inline method calls and thus create a finite program. While this brings along several advantages like the possibility to leave out auxiliary specification as well as being a fast and fully automatic approach, this also means that results obtained in this manner can only ever be valid up to the given threshold. If the program contains loops which may have more iterations or contains arrays which are bigger than the threshold, the result is only partial. By partial we mean that although the tool

```
public static void swapVerf(int[] array, int a, int b) {
  assume(array != null);
  assume(0 <= a && a < array.length);
  assume(0 <= b && b < array.length);
  int old0 = array[b];
  int old1 = array[a];
  int tmp = array[a];
  array[a] = array[b];
  array[b] = tmp;
  assert array[a] == old0;
  assert array[b] == old1;
}
```

Fig. 2. Result of JJBMC's JML-to-Java transformation for the `swap` method of Fig. 1. `assume` refers to a static verification-only method declared by JBMC.

signals a successful verification, there may still be a violation of the specification for runs of the program that exceed the threshold.

In JBMC (and, hence, also in JJBMC), all data is modelled in a bit-precise fashion using bit vectors. This encoding has the advantage that bit-wise logical or shift operations can easily be formulated and reasoned about. When representing bounded Java integers using mathematical integers like in KeY, it is still possible to encode such operations, but makes reasoning significantly more difficult.

JJBMC can be used to fully verify loop-free code. But it can also be used as a means to gain confidence about a specification before conducting a full formal proof. The fully automatic bounded model checking approach allows one to check specifications early on even when auxiliary specifications like loop invariants or method contracts of subroutines are still missing. This provides a early feedback opportunity while engineering JML specifications.

3.3 Property Checker

Pluggable type systems [7] are type systems which extend a language's existing type system without changing its run-time semantics. The *Checker Framework* [10, 22] is a framework for the creation of pluggable Java type systems using Java's annotation mechanism. For example, the annotation *@NonNull* and the base type `Object` can be combined into the type *@NonNull* `Object` of all objects which are not `null`. A type consisting of an annotation and a base type is called a *qualified type*, and an annotation which occurs in a qualified type is called a *qualifier*.

The advantages of pluggable type systems as a verification tool are that they are simple to use and that the type checker's run time generally scales very well with program size. On the other hand, they only provide conservative estimations of the property they are designed to show. So a nullness type checker will reject all programs in which a `NullPointerException` may occur, but it may also reject some NPE-free programs. Consider the excerpt in Fig. 3 from

```
boolean is_less_equal(@NonNull VarInfo v1, @NonNull VarInfo v2) {
  @Nullable Invariant inv = null; @Nullable PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    inv = instantiate(slice);
  }
  if (inv != null) {
    @SuppressWarnings("nullness")
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 3. Example of a false positive of the Checker Framework's Nullness Checker. (See the original file at https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/PptTopLevel.java\#L2087)

the Daikon Invariant Generator [11], which uses the Checker Framework to avoid `NullPointerExceptions` at run time. Our presentation of this example is taken from [18, Sec. 2]. The Checker Framework reports that the variable `slice` may be null when `slice.is_inv_true` is called. This is a false positive, because the implementation ensures that if the variable `inv` is non-null, the variable `slice` is also non-null.

Deductive verification, as provided e.g. by KeY, has the exact opposite advantages and disadvantages: It is rare that the correctness of a correct program cannot be proven using KeY's calculus. On the other hand, using KeY requires more expertise than using a type checker and both KeY's run time and the time required to write a correct specification scale badly with program size and complexity.

The *Property Checker* [18] is a generic framework for pluggable type systems with user-defined properties developed in the Checker Framework. This checker can be instantiated with a hierarchy of *property qualifiers*, which are qualifiers whose semantics is defined by a single Boolean expression, which may depend on the typed variable (called the *subject*). For example, *@NonNull* could be made into a property qualifier via the property *subject* $\neq$ *null*. More elaborate examples for property type qualifiers can be found in the case study in Sect. 6.2. The Property Checker also supports qualifier hierarchies and parametrised qualifiers. For instance, a qualifier *@GreaterEq(int a)* can be defined by the property *subject* $\geq a$. A suitable subtyping hierarchy can be defined via $@\mathit{GreaterEq}(a) \preceq @\mathit{GreaterEq}(b) :\Longleftrightarrow a \geq b$. However, these features are not needed in the case study.

A type which is qualified with a property qualifier is called a *property type*. Property types can thus be seen as a kind of refinement types, a *refinement type* being a subtype which restricts its base type by demanding that all of its instances fulfil some property [12,23].

```
//@ requires v1 != null && v2 != null;
boolean is_less_equal(VarInfo v1, VarInfo v2) {
  Invariant inv = null; PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    //@ assume slice != null;
    inv = instantiate(slice);
  }
  if (inv != null) {
    //@ assume inv != null;
    //@ assert slice != null;
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

Fig. 4. (Simplified) translation of Fig. 3.

Defining qualifiers using Boolean expressions allows the Property Checker to translate occurrences of qualifiers in a program to JML specifications, which in turn allows us to combine the scalability of type systems with the power of deductive verification: The Property Checker checks for a conservative estimation of the desired properties using simple subtyping rules over the declared hierarchy. Any occurrences of property qualifiers whose correctness cannot be established using this estimation are translated to JML assertions, to be discharged by some other JML tool. In addition, any occurrences of property qualifiers whose correctness can be established are translated to JML assumptions to aid in the proof search. The translation of our example from Fig. 3 is seen in Fig. 4: All qualifier occurrences proven by the type checker have been translated into assumptions, and all occurrences not proven into assertions. Thus, we can discharge most proof obligations using the scalable, easy-to-use type checker, but still rely on the full power of deductive verification for the trickier proof obligations.

4 Distributing Proof Obligations

For multiple verification tools to be able to collaboratively prove a program correct, we must first clarify how proof obligations for a program can be distributed between different tools while keeping a sound verification approach.

To this end, in this section we will not work with Java and JML, but study a simpler while language with assertion and assumption statements. For this language, we will prove that it suffices for a program to be correct that any assertion in the program be proven by any one tool while all other assertions can be taken as assumptions (i.e. their statements can be used in the verification process without having to be shown).

$$\begin{aligned}
C &::= x := T \mid \mathit{if}\ B\ \mathit{then}\ C\ \mathit{else}\ C \mid \mathit{while}\ B\ \mathit{do}\ C \mid C\ ;C \mid \\
&\qquad \mathit{assert}\ L : B \mid \mathit{assume}\ B \\
T &::= x \mid T + T \mid T * T \mid T - T \mid 0 \mid 1 \\
B &::= \neg B \mid B \wedge B \mid T = 0 \mid T > 0
\end{aligned}$$

Fig. 5. Grammar for the while language with assertions and assumptions.

The considered while language is according to the grammar in Fig. 5, in which $x \in \mathit{Var}$ is a placeholder for a variable name from the set of variables Var. Assertions are garnished with a label L, which must be a unique character string within the entire program. The set $\mathit{labels}(P)$ collects all assertion labels that occur in P. The set *Programs* is the set of all syntactically correct programs that can be produced from the non-terminal C in the above grammar.

This deterministic language has the usual intuitive semantics with the set of states of an execution being $\mathit{State} = \mathbb{Z}^{\mathit{Var}}$, the set of all variable assignments. Intuitively, whenever the execution reaches an assumption whose condition fails, execution silently halts. Whenever the execution reaches an assertion whose condition fails, it raises an error. A program is correct if it does not raise an error for any initial state.

For the purposes of this paper, we only consider programs that always terminate, but assume termination silently without showing it.[1] We formally define the semantics of such programs using *assertion/assumption-traces* (short *aa-traces*): An aa-trace is a finite sequence of pairs in $A = \{\mathit{assert}, \mathit{assume}\} \times \mathit{Bool}$. Each element in the sequence denotes that an assertion/assumption has been reached and whether its condition evaluated to true or false. This is encoded as a function $[\![\cdot]\!](\cdot) : \mathit{Programs} \times \mathit{State} \to A^* \times \mathit{State}$ whose definition is shown in Fig. 6.

This definition is well-founded (despite the recursive definition for loops) since we silently only consider terminating programs. Note that failing assertions and assumptions do not end an aa-trace but let the execution continue. The formal definition of a correct program capturing the intuitive notion takes this into account by requiring that no assertion fails before an assumption has failed:

Definition 1 (Correct Program). *A program $P \in \mathit{Programs}$ is called correct if the trace $[\![P]\!](\sigma)$ for each initial state $\sigma \in \mathit{State}$*

1. *does not contain the pair $(\mathit{assert}, \mathit{false})$* or
2. *contains the pair $(\mathit{assert}, \mathit{false})$ only at a position after an occurrence of $(\mathit{assume}, \mathit{false})$.*

Remember that in this section, we want to show that it suffices that each assertion is covered by one verification approach while all other verification engines are allowed to consider it an assumption. Therefore, we introduce the

[1] It would have been possible to extend the following definitions also to non-terminating programs, but would have reduced readability without adding much insight.

$$\begin{aligned}
[\![x := T]\!](\sigma) &= (\epsilon, \sigma[x \mapsto T^\sigma]) \qquad (T^\sigma \text{ evaluates expression } T \text{ in } \sigma)\\
[\![c_1; c_2]\!](\sigma) &= (s_1 \_ s_2, \sigma_2) \text{ with } (s_1, \sigma_1) = [\![c_1]\!](\sigma),\\
&\qquad\qquad\qquad\qquad (s_2, \sigma_2) = [\![c_2]\!](\sigma_1)\\
[\![\mathit{if}\ b\ \mathit{then}\ t\ \mathit{else}\ e]\!](\sigma) &= \begin{cases} [\![t]\!](\sigma) & \text{if } \sigma \models b\\ [\![e]\!](\sigma) & \text{otherwise}\end{cases}\\
[\![\mathit{while}\ b\ \mathit{do}\ c]\!](\sigma) &= \begin{cases} (s_1 \_ s_2, \sigma_2) & \text{if } \models b \text{ with } (s_1, \sigma_1) = [\![c]\!](\sigma),\\ & \qquad (s_2, \sigma_2) = [\![\mathit{while}\ b\ \mathit{do}\ c]\!](\sigma_1)\\ (\epsilon, \sigma) & \text{otherwise}\end{cases}\\
[\![\mathit{assert}\ L : b]\!](\sigma) &= \begin{cases} ([(\mathit{assert}, \mathit{true})], \sigma) & \text{if } \sigma \models b\\ ([(\mathit{assert}, \mathit{false})], \sigma) & \text{if } \sigma \not\models b\end{cases}\\
[\![\mathit{assume}\ b]\!](\sigma) &= \begin{cases} ([(\mathit{assume}, \mathit{true})], \sigma) & \text{if } \sigma \models b\\ ([(\mathit{assume}, \mathit{false})], \sigma) & \text{if } \sigma \not\models b\end{cases}
\end{aligned}$$

where $s \_ u$ denotes the concatenation of two sequences s and u.

Fig. 6. Definition of program semantics using assertion/assumption-traces.

concept of *program variants*. A variant P' of a program $P \in \mathit{Programs}$ is a program that can be produced from P by rewriting arbitrarily many assertions into assumptions of the same conditions. For instance, `assert a; assume b` is a variant of `assert a; assert b` (but not vice versa). Since executions in variants take the same path as in the original program, but encounter potentially fewer assertions, every variant P' of a correct program P is also correct. In the following, we also want to refer to a verification tool t, which we consider to be a partial function $t : \mathit{Programs} \nrightarrow \mathit{Bool}$ which returns whether the argument is a correct program. We assume all tools are sound wrt. Definition 1.

Theorem 1 (Distributed Proof Obligations).
Given a program $P \in \mathit{Programs}$ and a set of sound verification tools T, let for any $t \in T$ the program P_t denote the variant assigned to tool t.

If $t(P_t) = \mathit{true}$ for all $t \in T$ and $\bigcup_{t \in T} \mathit{labels}(P_t) = \mathit{labels}(P)$, then P is correct.

This theorem formally captures the goal of this section: The condition $\bigcup_{t \in T} \mathit{labels}(P_t) = \mathit{labels}(P)$ encodes that every assertion (identified by its label in $\mathit{labels}(P)$) has not been rewritten into an assumption in at least one variant P_t. If all variants can be proven correct by their respective sound tool t, the joint verification effort proves that the input program is correct.

Proof (of Theorem 1). Let P be a program according to the requirements of Theorem 1, i.e. every tool reports that its respective variant P_t is correct. Let us assume that P is incorrect. There would then be an initial state $\sigma_x \in \mathit{State}$

with trace $s = [\![P]\!](\sigma_x)$, which is a counterexample to the correctness of P, i.e. the first failing verification statement is an assertion, not an assumption. Let us call the label of that assertion L_x and its position in the trace x. All entries in the trace before x are either $(\mathit{assert}, \mathit{true})$ or $(\mathit{assume}, \mathit{true})$.

Let us inspect a variant P_t with $L_x \in \mathit{labels}(P_t)$ covering the assertion under observation, and its trace $s_t = [\![P_t]\!](\sigma_x)$. (Variant P_t must exist as every label must be covered by some tool.) We notice that throughout the execution of the programs P and P_t, the same operations have been executed and the same path has been taken. This implies that the conditions checked in assertions are the same, the only possible difference between s and s_t is that some 'assert' elements in the aa-trace s have been replaced by 'assume' in s_t. This implies that the xth entry in s_t must also be the first failing entry in that aa-trace, the same as in s. But failing this assertion entails that P_t is not a correct program which is a contradiction against the assumed soundness of the tool t which reported P_t to be correct. So, while P_t contains additional assumptions, all failing assertions will still be reported, as the assumptions are justified by other tools and, thus, do not restrict the traces of the program. $\square$

This result allows us to distribute proof obligations in form of assertions in a while program among a set of verification approaches. This principle of distributed assertions is not limited to while programs and it remains as future work to extend the formal setting to (recursive) function invocations, their abstraction for a formal modular analysis following design-by-contract and to the more sophisticated features of the Java language (exceptions, non-standard control flow, etc.).

5 Tool Interaction and Integration

The central feature that allows the tools in the Karlsruhe Java Verification Suite to be integrated is the use of a common specification language: JML. In Sect. 4, we have seen how the proof obligations dispersed throughout a while program in the form of assertions (in a common language) can be distributed among a set of verification tools by building a variant for each tool. In a modular context following the design-by-contract principle (e.g. when using JML), there are fewer explicit assertions in the code and, usually, the specification goes into method contracts. Clauses in method contracts can most naturally also serve as such specification distribution points. This ties in with the concept of assumption variants from Sect. 4, as the clauses in contracts between caller and callee have both a nature of assertion and of assumption that go hand in glove. The schematic sequence diagram in Fig. 8 sketches this relationship for a method call to $m()$ with a contract with precondition *pre* (asserted by caller, assumed by callee) and postcondition *post* (asserted by callee, assumed by caller). If we now verify the two methods n and m with separate tools we can see that the tool for n assumes the conditions verified by the tool covering m and vice versa. Covering different methods modularly with different JML verification tools is thus a special case of the proof obligation distribution sketched in Sect. 4.

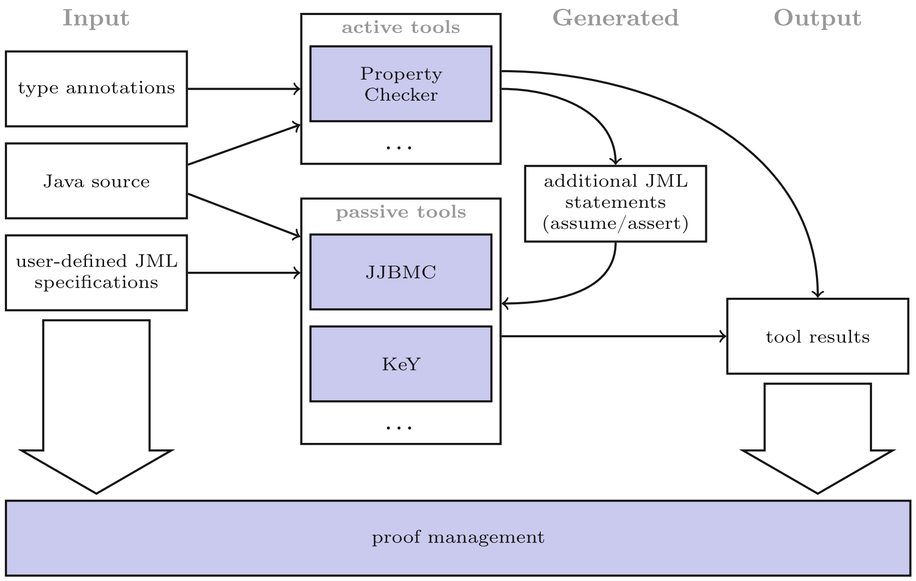


Fig. 7. Vision for interaction between the tools.

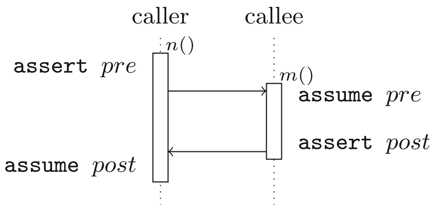


Fig. 8. Dual nature of specification clauses in design-by-contract.

Approaches for modular deductive verification often require extensive specifications, which in many cases is considered a major downside. But for once, the specification overhead can also be considered an advantage, as the explicit contracts allow one to distribute the verification overhead between different tools without requiring even more specification.

There are two ways in which a formal verification tool can interact via its specification language: either *passively* by interpreting specifications or *actively* by emitting specifications. Figure 7 shows the basic workflow and interaction between the tools. While JJBMC and KeY passively read and interpret JML specifications, the Property Checker does not itself digest JML specifications, but produces additional JML annotations to be verified or assumed by other tools. In either case, the tools integrate by distributing the proof responsibilities between them. In the passive case, it is the user who decides which tool has to cover which assertion of the specification. In the active case, it is the tool that decides if other tools must verify a property (if it cannot be discharged) or may assume it (if it can be discharged).

5.1 JML Semantics

One major challenge for the collaboration of the different tools in the suite is the potentially different semantics that different tools might implement for annotations in the common language JML. There are a few points in the semantics of JML which have a canonical answer within each approach, but not necessarily the same one for all systems. Two such semantics questions shall illustrate the challenge:

When do which object invariants have to hold? This is still an active research area and the answer to the question heavily depends on the technique the tool implements to deal with heap framing (e.g. separation logic, region logic, dynamic frames, ownership, ...). While JML originally proposed an ownership approach, KeY adopted dynamic frames. It is very difficult to bring these two concepts together in general.

What is the meaning of arithmetic operations in specifications? Due to the nature of encoding data using bit vectors, JJBMC naturally uses strict Java 32-bit integer semantics for arithmetic operations in specifications while JML by default assumes integer arithmetic to be performed on non-overflowing mathematical integers. Fortunately, KeY has switches that allow it to treat integers compatibly to the bounded model checker.

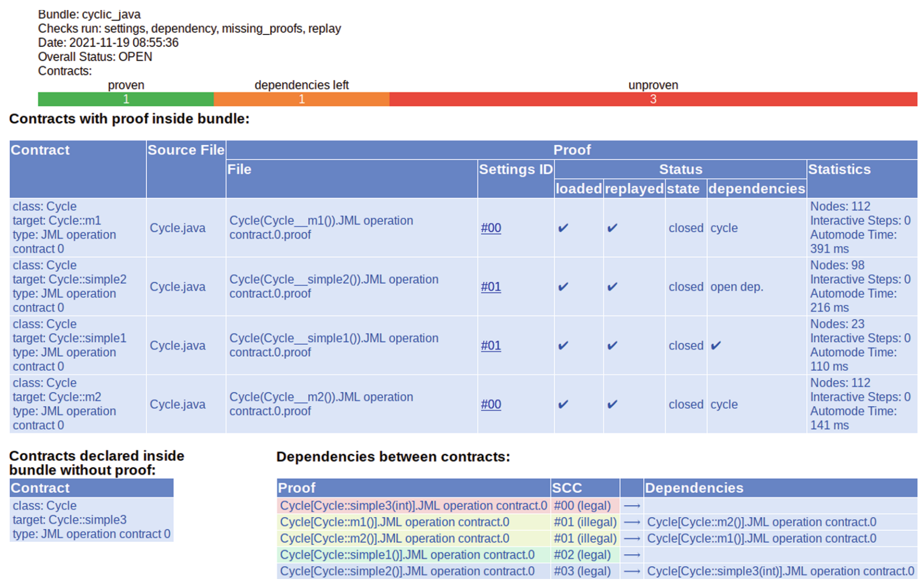
We have made sure (by manual review) that such differences do not compromise the validity of our case study. It remains as future work to either base all tools on the same semantic footing, even out differences in the specifications or at least to detect and report discrepancies. The envisioned Proof Management system (see Sect. 5.3) seems to be the ideal point to integrate this into the tool suite.

5.2 Formulating Program Variants Using JML

While assigning method contracts to different tools provides a natural process to distribute proof obligations between tools, it is also convenient to be able to follow the idea of assertion distribution from Sect. 4 more closely in such a setting.

Indeed, the JML annotation marker feature, which allows users to assign a JML annotation to specific tools, serves as a perfect means to express this distribution. If there are two JML-based verification tools A and B, then the annotation `/*+A@ assert` ϕ`;*//*+B@ assume` ϕ`;*/` makes sure that the same condition ϕ is interpreted as an assertion by A and as an assumption by B.

In the case study in Sect. 6, we have used this feature to combine bit-precise reasoning provided by JJBMC with more sophisticated reasoning in KeY. Figure 12 shows the `swap` method that exchanges the ath and bth element of an



Bundle: cyclic_java
Checks run: settings, dependency, missing_proofs, replay
Date: 2021-11-19 08:55:36
Overall Status: OPEN
Contracts:

proven	dependencies left	unproven
1	1	3

Contracts with proof inside bundle:

Contract	Source File	Proof						
		File	Settings ID	Status				Statistics
				loaded	replayed	state	dependencies	
class: Cycle target: Cycle::m1 type: JML operation contract 0	Cycle.java	Cycle(Cycle__m1()).JML operation contract.0.proof	#00	✔	✔	closed	cycle	Nodes: 112 Interactive Steps: 0 Automode Time: 391 ms
class: Cycle target: Cycle::simple2 type: JML operation contract 0	Cycle.java	Cycle(Cycle__simple2()).JML operation contract.0.proof	#01	✔	✔	closed	open dep.	Nodes: 98 Interactive Steps: 0 Automode Time: 216 ms
class: Cycle target: Cycle::simple1 type: JML operation contract 0	Cycle.java	Cycle(Cycle__simple1()).JML operation contract.0.proof	#01	✔	✔	closed	✔	Nodes: 23 Interactive Steps: 0 Automode Time: 110 ms
class: Cycle target: Cycle::m2 type: JML operation contract 0	Cycle.java	Cycle(Cycle__m2()).JML operation contract.0.proof	#00	✔	✔	closed	cycle	Nodes: 112 Interactive Steps: 0 Automode Time: 141 ms

Contracts declared inside bundle without proof:

Contract
class: Cycle target: Cycle::simple3 type: JML operation contract 0

Dependencies between contracts:

Proof	SCC		Dependencies
Cycle[Cycle::simple3(int)].JML operation contract.0	#00 (legal)	⟶	
Cycle[Cycle::m1()].JML operation contract.0	#01 (illegal)	⟶	Cycle[Cycle::m2()].JML operation contract.0
Cycle[Cycle::m2()].JML operation contract.0	#01 (illegal)	⟶	Cycle[Cycle::m1()].JML operation contract.0
Cycle[Cycle::simple1()].JML operation contract.0	#02 (legal)	⟶	
Cycle[Cycle::simple2()].JML operation contract.0	#03 (legal)	⟶	Cycle[Cycle::simple3(int)].JML operation contract.0

Fig. 9. Example of a report generated by our proof management tool.

array, implemented using XOR operations. JJBMC considers the swapping property in the JML annotation as an assertion, whereas KeY is allowed to assume it. JJBMC can prove this property easily while proving it is out of scope for KeY (which models Java integers using mathematical integers and cannot deal with bit-level operations). But with KeY being allowed to assume this property, it can then proceed and use it to prove a more sophisticated permutation property which would have been out of scope for JJBMC.

5.3 Proof Management

In the current state of the Karlsruhe Java Verification Suite, it has to be ensured by the user that the verification responsibilities are distributed soundly among the tools according to Theorem 1. While for simple cases, the proof coverage and the absence of cycles can still be checked manually, for large projects, it is essential to have some kind of proof management. For multiple KeY proofs, this gap is already closed by a command line tool which can be used to check that (a) the proofs can be reloaded and checked, and also match the Java source code and JML specifications, (b) there are no illegal[2] cyclic dependencies in the proofs, (c) all contracts a proof depends on are proven as well, and (d) the settings are compatible. This last point ensures that the semantics, for instance the meaning of arithmetic operations, is the same for each proof.

[2] To be able to reason about (mutually) recursive methods, cyclic dependencies are allowed as long as a termination witness is provided for each of them.

The proof management tool generates an HTML report from a set of input proofs that contains all the information described above. Figure 9 shows an example of such a report for a source file containing 5 contracts, of which 4 are proven. However, one contract still has open dependencies, which means that the proof uses another contract that is still not discharged. In addition, there is an illegal cycle detected; therefore the two contracts in the cycle are considered unproven.

At the moment, this tool manages only KeY proofs. For the future however, we envision a proof management tool that also integrates results from JJBMC and the Checker Framework and possibly other tools such as OpenJML, and which therefore serves as a connection between the tools of the Karlsruhe Java Verification Suite. The goal is that if the proof management report indicates that all proofs have been done, then the proof obligations have been distributed successfully and the project has been correctly verified.

6 Case Study

We implemented a small Java-based shop GUI (shown in Fig. 10) to illustrate how the different components of the Karlsruhe Java Verification Suite can collaborate to prove a system correct. In the application, the user can buy different types of wine. Whenever they click on the button beside a wine, a number representing its price is added to the shopping basket at the top of the GUI. The items in the shopping basket are always kept in sorted order. The code contains a sorting routine whose verification falls clearly in the domain of deductive verification with KeY. However, there is a base case which is better handled using JJBMC. The client program uses GUI-specific code and could not even be loaded into KeY. Luckily, the sortedness property of the basket can be propagated through the GUI code using property types. All components of the Karlsruhe Java Verification Suite have to join forces to prove this program correct.

Figure 11 shows a class diagram for the case study. In addition to the `GUI` class, we have a `Shop` class containing the products and prices as well as a `Basket` class which encapsulates the user's shopping basket in an instance of `ImmutableArray`. `ImmutableArray` uses a sorting algorithm provided by the class `Quicksort`.

6.1 Library Code: Quicksort with Explicit Base Case

As a library function for the web shop, we provide a method to sort the elements of an array. We follow a common practice in algorithm engineering by efficiently sorting the given array with Quicksort until we reach a small enough array (less than six elements) for which we employ a specialised sorting network. This approach proves to be faster than Quicksort in practice. We also use the in-place `swap` method using repeated XOR operations from Fig. 12 to swap elements in the array. The interplay between KeY and JJBMC for this method has already been explained in Sect. 5.2.

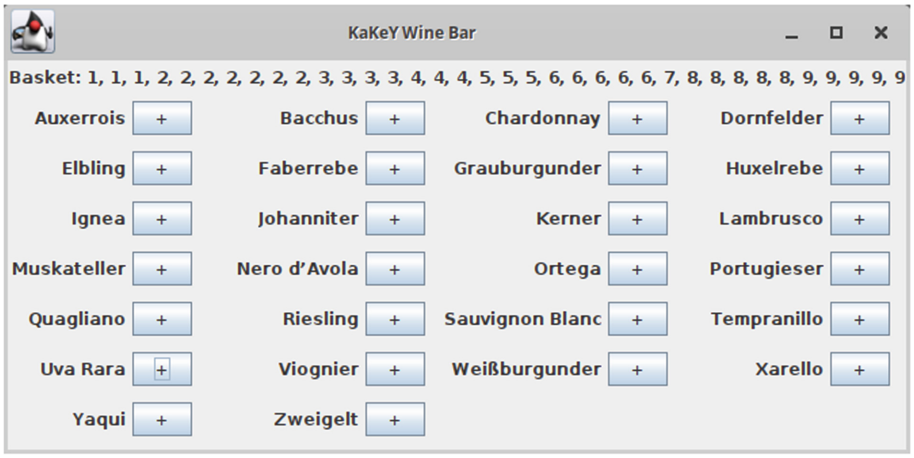


Fig. 10. Graphical user interface for the case study.

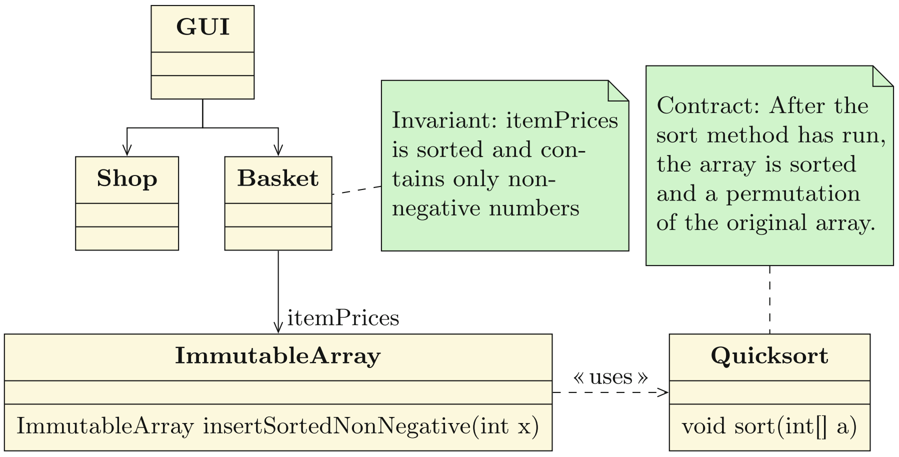

GUI
Shop
Basket
Invariant: itemPrices is sorted and contains only non-negative numbers
Contract: After the sort method has run, the array is sorted and a permutation of the original array.
itemPrices
ImmutableArray
ImmutableArray insertSortedNonNegative(int x)
« uses »
Quicksort
void sort(int[] a)

Fig. 11. Class diagram for the case study.

The sorting network implementing the base case with at most 5 elements (taken from [6]) is free of loops, but contains 12 consecutive `if` statements: It is thus a natural fit for bounded model checking, whereas KeY chokes on the input due to the large number of paths it has to traverse during symbolic execution.[3] In contrast, the actual Quicksort implementation with several unbounded loops and arrays is where KeY shines. This routine is a perfect example of how different tools can play together in order to conduct proofs that neither of them would be able to find alone.

[3] There are verification tools which avoid this exponential blowup of proof obligations, but other tools relying on symbolic execution would suffer under the same problem.

```
void swap(int[] arr, int a, int b) {
  if(a != b) {
    arr[a] ^= arr[a];
    arr[a] ^= arr[b];
    arr[b] ^= arr[a];
    //+KeY@   assume arr[a]==\old(arr[b]) && arr[b]==\old(arr[a]);
    //+JJBMC@ assert arr[a]==\old(arr[b]) && arr[b]==\old(arr[a]);
  }
}
```

Fig. 12. In-place swap of integers using XOR operations. (Its JML contract is the one shown in Fig. 1.)

$$\begin{aligned}
@\mathit{Sorted} &\triangleq \mathit{subject} \neq \mathit{null} \land \mathit{subject.arr} \neq \mathit{null} \land \texttt{Util.isSorted}(\mathit{subject.arr})\\
@\mathit{NonNegArray} &\triangleq \mathit{subject} \neq \mathit{null} \land \mathit{subject.arr} \neq \mathit{null} \land \texttt{Util.isNonNeg}(\mathit{subject.arr})\\
@\mathit{NonNeg} &\triangleq \mathit{subject} \geq 0
\end{aligned}$$

Fig. 13. Property qualifier definitions. The identifier *subject* refers to the variable receiving the type annotation.

The base case verifies in JJBMC within 13 min.[4] Sortedness is shown for bound $k = 6$, the permutation property for $k = 5$ (greater bounds resulted in a timeout). The analysis of the swap method is instantaneous. For the verification of the different parts of the Quicksort algorithm, KeY runs in automatic proof search mode for about 11 min in total (see footnote 4). For the permutation property, a few manual rule applications (which can be captured in a proof script) are needed to guide quantifier instantiation in the prover.

6.2 Library Code: Immutable Arrays with Sortedness and Non-negativity

In the application, we want to be able to obtain immutable array instances which are known to only contain non-negative numbers in sorted order. Since references to arrays with such properties will also be handed around in GUI code, the plan is to capture these conditions as property types to not have to load GUI code into the deductive verifier. The required qualifiers are shown in Fig. 13, where the qualifiers *@Sorted* and *@NonNegArray* can be applied to instances of the class `ImmutableArray`, and *@NonNeg* can be applied to int values. The type property definitions make use of the helper methods `Utils.isSorted`, which returns `true` if and only if the array is sorted, and `Utils.isNonNeg`, which returns `true` iff all elements in the array are ≥ 0.

Next, we use these qualifiers to specify the method `insertSortedNonNegative()` (shown in Fig. 14) which takes a non-negative array and a non-negative number, returns a new sorted non-negative array. Since the implemen-

[4] On a PC with an AMD Ryzen 7 PRO 4750U (8x1.7 GHz) CPU and 32 GB RAM.

```
public static
@NonNegArray ImmutableArray
insertSortedNonNegative(@Sorted @NonNegArray ImmutableArray ia,
                        @NonNeg int newElement) {
  int[] newArr = new int[ia.arr.length + 1];
  System.arraycopy(arr, 0, newArr, 0, ia.arr.length);
  newArr[ia.arr.length] = newElement;
  Quicksort.sort(newArr);
  return new ImmutableArray(newArr);
}
```

Fig. 14. The `ImmutableArray` methods which need to be proven in KeY.

```
(ActionEvent e) -> {
  basket.prices = basket.prices.insertSortedNonNegative(price);
  ...
}
```

Fig. 15. An assignment which passes along a type property and can be verified by the Property Checker.

tation of property types is currently limited to immutable objects, the method returns a new object instead of modifying the argument array.

To obtain the well-typedness of the method, its return type must be correct, i.e. the returned array must (a) be sorted, and (b) contain only non-negative elements. The type-checking algorithm behind the Property Checker cannot look inside the type qualifier definitions and take their semantics into account. Instead, it only checks whether syntactic typing rules are respected; e.g., the right-hand side of an assignment must evaluate to a subtype of the left-hand side's type. Thus, the Property Checker is unable to prove the well-typedness of this method. The fact that the returned array is sorted could be shown by the checker if `Quicksort.sort()` were specified using property types. But since it was specified directly in JML instead, the checker cannot use its specification. The fact that the returned array contains only non-negative elements follows from the fact that `Quicksort.sort()` returns a permutation of the original array, and also cannot be established by the checker. Hence, it translates the type qualifiers of the returned value into JML assertions. The types of the method parameters, on the other hand, are guaranteed and are translated into JML assumptions. KeY is then able to discharge the proof obligations that arise from the assertions. This requires 85 s in KeY's automatic mode[5] and two manual quantifier instantiations.

6.3 Client Code: The Web Shop GUI

The method in the class `Quicksort` has now been formally verified, (Sect. 6.1) and a formal connection between type qualifiers and their defining predicates has

[5] On a PC with an AMD Ryzen 7 PRO 4750U (8x1.7GHz) CPU and 32GB RAM.

been established (Sect. 6.2). The remaining classes in the case study do not establish any type properties, but only use them and pass objects having the property along. The well-typedness of these classes can be proven by the Property Checker. For example, consider the action listener in Fig. 15, which is executed whenever the user presses a button. It updates the shopping basket with the price of the newly chosen product. The fact that this assignment preserves the type properties of the shopping basket – i.e. that `basket.prices` is sorted and non-negative – follows directly from the well-typedness of `insertSortedNonNegative()` and is verified by the Property Checker. The total run time of the Property Checker for this case study is 8 s (see footnote 5).

Unlike KeY, the Property Checker supports code using features of Java 8 and later like lambda functions, making it well-suited to this kind of client code, which would otherwise have to be rewritten for KeY and specified in JML just for this relatively superficial formal treatment.

6.4 Conclusion

This case study demonstrated how multiple verification tools can be combined to prove the correctness of a program that is not wholly accessible to either of the tools. While sorting routines generally fall in KeY's domain, highly optimised routines like the one analysed here often have base cases with long `if` cascades, which are onerous to prove in KeY. JJBMC has no problem with this kind of code, but can only show bounded correctness in code with loops. But since the base case is only used for arrays with bounded size, JJBMC can give us a total correctness guarantee. Our program also has a graphical user interface, which we can only analyse in KeY or JJBMC after writing lots of boilerplate JML specifications for the GUI code. Even then, we still have to do a lot of work just to prove how the sortedness property is propagated through the GUI methods. Using a pluggable type system and the Property Checker, this task is less burdensome: We simply annotate all variables which should be sorted with an appropriate qualifier and run the Property Checker. On the other hand, the Property Checker is unable to reason about the methods which establish the sortedness property, which we instead have to prove in KeY or JJBMC. Thus, all components of the Karlsruhe Java Verification Suite have to join forces to prove this program correct.

7 Related Work

An overview over existing approaches to combine multiple verifiers is discussed by Beyer and Wehrheim [5]. In their classification system, our approach would fit into the category 'cooperation of tools viewed as black box objects', since our tools only communicate via an interface (JML in our case), do not need to know anything about their internals, and cooperate on intermediate results of the verification (for instance, well-typedness information from type checker is passed via additional assumptions to KeY).

In [14], Jacobs presents a technique to construct a correctness witness from multiple partial analysis results. As opposed to our work for combining a type system, an interactive deductive verifier, and a bounded model checker, the technique presented there is targeted towards model checkers and static analysers which have an explicit notion of visited and checked states and record these states via so called abstract reachability graphs. On a practical level, as they implemented their technique using the CPAChecker framework [4], their technique works for programs written in C, while our approach works for Java.

There are many other deductive verification approaches which combine different tools to check the correctness of one program.

Type checkers for refinement type systems like *LiquidHaskell* [23] use SMT solvers internally, which allows them to be more powerful than conventional stand-alone type checkers. *LiquidJava* [13] applies this idea to Java: Refinements similar to our property types are specified using Java annotations and the proof obligations arising from them are translated to SMT.

Hybrid type checking [17] combines static type checks at compile time with dynamic checks at run time. The static type checking process has three possible results: 1. *definitely well-typed*, where the program's type safety was able to be established at compile time, 2. *definitely ill-typed*, where the compile-time checks found a definitive error, and 3. *unknown*, where some parts of the program were proven to be type-safe, and the other parts had dynamic run-time casts automatically inserted where necessary. JJBMC can distinguish between definitive incorrectness and bounded correctness. KeY too can in some cases generate counterexamples for incorrect programs using SMT. In contrast to hybrid type checking, which combines a compile-time type checker with run-time checks, our approach combines multiple compile-time tools, but it could also be extended to include run-time tools where the compile-time verification fails.

RustBelt [15] is an approach which proves the soundness of Rust's ownership type system. Programs written in a safe subset of Rust are proven to always be sound, and for library code using unsafe features, verification conditions for Coq [3] are generated. Thus, the correctness of a Rust program using such a library is proven by a combination of the Rust type checker and Coq. In contrast to RustBelt, which is focused on ownership properties, our approach is based on the Java Modeling Language, allowing us to prove general functional properties.

8 Conclusion and Future Work

We introduced the Karlsruhe Java Verification Suite, a collection of Java verification tools (co)-developed in Karlsruhe built around the deductive verifier KeY, which next to KeY includes the bounded model checker JJBMC and a type checker called the Property Checker. We showed how proof obligations can be soundly distributed between different verification tools and how this distribution can be implemented for JML. We also demonstrated using a small case study how the Karlsruhe Java Verification Suite can be used to verify the correctness of a program which would have required a large refactoring and specification overhead if we had only used KeY.

In the future, we plan to investigate how other kinds of verification tools can be used to expand the Karlsruhe Java Verification Suite. We also plan to refine an existing proof management tool such that it can be used to orchestrate proofs with proof obligations distributed over the tool suite. In particular, the management tool has to be able to deal with differences in the interpretation of JML between different verification tools.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 60–80. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
5. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: survey and unifying component framework. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 143–167. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_8
6. Bingmann, T., Marianczuk, J., Sanders, P.: Engineering faster sorters for small sets of items. Softw. Pract. Exp. **51**(5), 965–1004 (2021). https://doi.org/10.1002/spe.2922. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2922
7. Bracha, G.: Pluggable type systems. In: OOPSLA 2004 Workshop on Revival of Dynamic Languages, October 2004
8. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
9. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: a bounded model checking tool for verifying java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10
10. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.: Building and using pluggable type-checkers. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 681–690. Association for Computing Machinery (2011). https://doi.org/10.1145/1985793.1985889
11. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015

12. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI 1991, pp. 268–277. Association for Computing Machinery, New York (1991). https://doi.org/10.1145/113445.113468
13. Gamboa, C., Santos, P.A., Timperley, C.S., Fonseca, A.: User-driven design and evaluation of liquid types in Java. CoRR abs/2110.05444 (2021). https://arxiv.org/abs/2110.05444
14. Jakobs, M.-C.: PART$_{\text{PW}}$: from partial analysis results to a proof witness. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 120–135. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_8
15. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. **2**(POPL), 1–34 (2017). https://doi.org/10.1145/3158154
16. Kassios, I.T.: Dynamic frames: support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_19
17. Knowles, K., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. **32**(2), 1–34 (2010). https://doi.org/10.1145/1667048.1667051
18. Lanzinger, F., Weigl, A., Ulbrich, M., Dietl, W.: Scalability and precision by combining expressive type systems and deductive verification. Proc. ACM Program. Lang. **5**(OOPSLA), Article no: 143 (2021). https://doi.org/10.1145/3485520
19. Leavens, G.T., et al.: JML Reference Manual, May 2013. http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf. Revision 2344
20. Mccarthy, J.: Towards a mathematical science of computation. In: In IFIP Congress, pp. 21–28. North-Holland (1962). https://doi.org/10.1007/978-94-011-1793-7_2
21. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
22. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: International Symposium on Software Testing and Analysis, ISSTA, pp. 201–212. ACM, Association for Computing Machinery (2008). https://doi.org/10.1145/1390630.1390656
23. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton Jones, S.: Refinement types for Haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014, pp. 269–282. Association for Computing Machinery, September 2014. https://doi.org/10.1145/2628136.2628161
24. Zimmerman, D.M., Nagmoti, R.: JMLUnit: the next generation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 183–197. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_13

Further Lessons from the JML Project

Gary T. Leavens[1(✉)], David R. Cok[2], and Amirfarhad Nilizadeh[1]

[1] University of Central Florida, Orlando, FL, USA
Leavens@ucf.edu, af.nilizadeh@knights.ucf.edu
[2] Safer Software Consulting, Rochester, NY, USA
david.r.cok@gmail.com
http://www.cs.ucf.edu/~leavens

Abstract. JML is an ambitious project in formal specification and verification that has aimed to bring value to Java programmers. As an international, cooperative effort, JML has been a uniting force, bringing together tools such as KeY, OpenJML, and many others. Specification language designers can learn much from the experience gathered by the JML team. The lessons of the JML project can be useful to others designing specification languages and tools, not only for object-oriented languages such as Java, but more generally.

Keywords: JML · Behavioral interface specification language · Formal specification language design · Formal methods

1 Introduction

The Java Modeling Language (JML) [93] project is an ambitious formal specification language project, with continuing contributions since its beginning about 1997. JML is a Behavioral Interface Specification Language (BISL) for Java. As a BISL [89,147] [99, Sec. 1.1], JML can specify all of the details of a Java interface, including privacy, exceptions, and the types of formal parameters; thus a user of a JML specification has all the information needed to write (and verify) code that uses that interface. Consequently, JML is most useful for recording detailed designs.

The JML project is more than language design. It includes research on specification languages and verification techniques, tool development, applying the tools in academic and industrial settings, sharing educational materials, and encouraging broad use through workshops.

This paper summarizes some of the lessons we have learned in the course of the JML project. The lessons can be categorized in two groups: general lessons for projects that aim to develop specification languages, discussed in Sect. 2, and lessons about the details of JML itself, discussed in Sect. 3. The remainder of this introduction gives some background on the goals and history of the project.

G. T. Leavens—Much of the early work on JML was supported by the US National Science Foundation and carried out while Leavens was at Iowa State University.

1.1 Goals of JML

As stated in the paper on the preliminary design of JML [93, p. 1], the main goal of the JML project has been to understand how to design specification languages and tools that are “practical and effective for production software environments.” In detail, this means [93, p. 4] that JML should be able to:

1. document “the interfaces and behavior of existing software,” no matter how it was designed,
2. be “easily understandable” by Java programmers, and
3. be given a “rigorous formal semantics.”

These goals are discussed in Sects. 2.3–2.5. The first two goals have been accomplished to some extent, but the last one has remained elusive.

In addition to these overall goals, the project also develops tools and works to disseminate their use.

1.2 Project History

The JML project grew out of work on Larch/C++ [35,89], as Leavens and his research group (at Iowa State) realized that creating tools and verifying C++ programs was very difficult. Java, which started to appear around 1996 [67], was (at the time) a simple alternative to C++ that offered many advantages for verification (such as type safety, lack of pointer arithmetic, and array bounds checking) and simplicity (e.g., lack of include files and macros). Java’s simplicity had definite advantages for tool support. At Iowa State University, Leavens worked with professor Albert Baker and PhD student Clyde Ruby on the initial design of JML [92,93].

Larch/C++ was a BISL for C++ in the Larch family, meaning that it described program states using abstract (mathematical) values specified in the Larch Shared Language (LSL) [70,71]. However, experience with Larch/C++ led to the conclusion that understanding LSL was an impediment to wider use of such a Larch-style specification language. The alternatives seemed to be to (a) use a standard mathematical vocabulary for abstract values, as in VDM [86], or (b) use values from the programming language itself in reasoning, as in Eiffel [112]. Since the Eiffel approach seemed to be easier for programmers to learn and use, it was adopted for JML. However, to stay close to the VDM (and Larch) approach, JML was initially designed with a library of modeling types such as sets, sequences, and maps, which were implemented as pure (side-effect free) types in Java. This turned out not to be a good decision for static verification, as discussed below (see Sect. 3.3).

During Java’s early years, Leavens’s group at Iowa State was not the only research group interested in formal methods for Java. An important influence was the group at DEC Systems Research Center (SRC), which included Rustan Leino and Greg Nelson. This group developed an “extended static checker” (ESC) tool for Java [50], with a formal specification language. Leavens visited this group[1] and discussed formal specification and verification with Leino and Nelson, and they suggested that, with some

[1] The date of Leavens’s visit to DEC SRC is unclear, but was probably in 1998.

changes, the ideas that Leavens was advocating for specification of Java programs could be used to create a common specification language for their ESC tool as well as Iowa State's runtime assertion checker. An important syntactic change resulting from their suggestions was that method specifications should appear before a method's header, since that is where Java programmers were already putting documentation comments (which describe the functionality of a method).

Another influential group of researchers in the early days of JML was Bart Jacobs's Verificard group at Nijmegen in the Netherlands [85]. This group developed the first verifier for Java programs, LOOP [83], using the PVS theorem prover. This group adopted JML as their specification language [17], finding that it was an advantage to use JML's runtime assertion checker and ESC to debug their specifications and code before attempting to verify the code. Members of this group, especially Erik Poll and Marieke Huisman, made substantial contributions to JML [18,31]. The group's initial motivation for verification of Java programs was to verify smart cards in the JAVA CARD dialect of Java, since once code is built into such a smart card, it could not easily be changed [130].

Michael Ernst, who was at MIT in the early years of JML, was also interested in tools for Java. He developed a dynamic invariant inference tool, later called Daikon, which output invariants in JML notation [60,124]. Daikon is still available from its website [61]; the tool has also been extended to apply to other languages, in addition to Java.

Patrice Chalin, who was on the faculty at Concordia University in Montreal at the time, was another early influence on JML, particularly in terms of runtime assertion checking (RAC). With his students he recommended changes to the semantics of JML so that references were non-null by default, arithmetic overflow was considered to be incorrect in Java code, and exceptions in assertions should be regarded as errors [27–30,32], arguing that the first two of these changes would make JML's semantics easier to understand and that the third change would make the semantics of RAC and ESC more similar, and thus users would be less likely to have their specifications misinterpreted by different tools.

Peter Müller, now a professor at ETH, made significant contributions to the semantics of JML. While working on his PhD under Arnd Potezsch-Heffter at Fernuniversität Hagen in Germany, he spent the summer of 1998 at Iowa State University and collaborated with Leavens on semantic issues in JML, particularly for framing. His dissertation work on the Universe Type System [113], which allowed for modular reasoning about frame conditions with information hiding [114], was incorporated into JML and sparked a long line of research about the verification of invariants [3,4,6,10,55,56,108,109,115].

David Cok joined the JML project as an open source developer in 2000 and continues as a contributor to JML's design and semantics and to tools supporting the use of JML. He contributed to the Iowa State suite of tools (notably jmldoc), worked with Joseph Kiniry to extend the DEC ESC/Java tool (as ESC/Java2 [44,45]) to be fully integrated with JML and Java 4, and he led the work to base tools for JML on the OpenJDK compiler, producing the OpenJML [40,41,43,47] tool. His continued development of the tools eventually made him the principal developer of OpenJML.

The KeY project was started in 1998 at the University of Karlsruhe in Germany [2,16, p. XI]. The project was originally led by Drs. Rainer Hähnle, Peter Schmitt, and Wolfram Menzel, with PhD students (at the time) Wolfgang Ahrendt and Bernhard Beckert participating and continuing in project leadership roles afterwards. The original goal was to apply logic to software engineering concerns with an initial focus on the UML's Object Constraint Language (OCL). Sometime in 2001, the KeY project started work on using JML as a way to specify Java programs [138]. The KeY project found that the main advantage of JML over OCL was that JML is more expressive, because, as a BISL, it is more closely tied to Java: "we noticed that many requirements that we wanted to express could not be formalized in OCL: frame and depends-on information of methods, loop invariants, loop termination conditions, etc." [138, p. 7].

1.3 Tools for JML

At present, two tools are the most prominent and well-supported for static verification of Java code with JML:

– the KeY tool does static deductive verification of Java programs specified in (the KeY dialect of) JML, though its support of the full Java language extends only to about Java 4, as it was originally designed to be used for JAVA CARD programs, and
– the OpenJML tool which has mostly been supported by David Cok, but which handles most of Java and attempts to keep up with the current Java language.

KeY only supports static verification, but has excellent documentation [2] and can be used for interactive proofs (although it currently focuses on automated proofs). OpenJML integrates both static verification (in the non-interactive style of ESC) with RAC and other tools for JML.

An early summary of tool support for JML was presented in the paper "An Overview of JML Tools and Applications" [25].

One of the goals of JML has been to serve as a specification language that coordinates many different tools; the idea is that JML should serve as a common specification language shared by many tools, so that both users and tool developers would benefit. Several tools were implemented at Iowa State University (ISU) by Leavens's PhD students, including Clyde Ruby, Curtis Clifton, and Yoonsik Cheon, with contributions from other students. David Cok was an important outside contributor to this software. These ISU tools for JML were based on the MultiJava compiler [39], which itself was based on the Kopi open source Java compiler. The main tool from ISU was the runtime assertion checker (jmlc [36]). The ISU group also built a tool to display specifications and javadoc comments together as web pages (jmldoc, originally written by David Cok, but revised by Arun Raghavan [131]) and an influential unit testing tool that used specifications as test oracles during runtime assertion checking (jmlunit [37]).

The DEC SRC group's Extended Static Checker for Java (ESC/Java) tool was reimplemented for then current Java 4 and using JML as its input language as the tool ESC/Java2 by Joseph Kiniry and David Cok [44,45]. ESC/Java2 performed automated verification, using an SMT solver (replacing the Simplify solver), making it much easier to use than interactive verification tools such as LOOP [83], which required expertise both in JML and the PVS interactive theorem prover.

As mentioned above, the KeY project [88] provides a solidly supported and documented tool for the deductive verification of JML-annotated Java programs. It aims for automatic verification, but a user can apply interactive theorem-proving which may be needed for the verification of more sophisticated heavyweight specifications.

KeY uses JML for formal specifications, but also supports some extensions going beyond current JML. The KeY project, like the LOOP project, began as a tool for the JAVA CARD language; however, now it handles Java through Java 4 (without generics) and has an upgraded parser supporting more recent Java versions in progress. A unique feature is that KeY encodes proof obligations in Java Dynamic Logic [15] and includes facilities for logic-based symbolic execution. There is extensive documentation in the KeY book [2, 16].

The rapid evolution of Java, especially the addition of generics, annotations, and other features in Java 5 made many of the early ISU tools, such as jmlc, out of date with respect to Java. The Kopi project on which the Multijava compiler (and thus the ISU tools) were based, was not revised to keep pace with Java. One strategy for keeping up with Java was to use an extensible compiler, such as JastAdd [57]. Some explorations of how to use JastAdd were done [73], but the JML developer community never really bought into JastAdd, perhaps because there was no guarantee that the JastAdd system would track Java's evolution.[2]

To help the JML tools keep up with Java's evolution, David Cok (and the JML community) settled on the idea of extending OpenJDK [126], creating the OpenJML suite of JML tools [40]. Although not designed as an extensible compiler framework, OpenJDK serves as a suitable base for extension due to the considerable advantage that it would necessarily track the evolution of Java, because it is the official compiler for Java.

Most of the work on OpenJML has focused on the ESC tool, since that does static verification. However, RAC is a useful complement to ESC and OpenJML does feature a RAC tool.

Henrique Rebêlo, working at the Federal University of Pernambuco, Brazil, developed a RAC tool for JML [136] based on aspect-oriented programming tools (i.e., AspectJ) and the ISU RAC tool, jmlc. Rebêlo's tool, ajmlc, featured good error messages for runtime assertion violations and client-side contract checking, in which preconditions are checked at the call site of a method (instead of at the beginning of a method body's execution) [135]. Contract-aware checking allows the RAC to take privacy information into account at the call site, only checking the precondition(s) visible to the client calling a method. The ajmlc tool is now incorporated into AspectJML [133, 134].

Considering this history leads to two initial lessons.

Lesson 1. *Academics are only strongly motivated to work on tools if they can make novel changes, but such changes make specifications less portable among tools.*

Lesson 2. *Supporting a current, industrially used language enables broader use, but the cost of keeping the tools current with a rapidly evolving language is substantial.*

[2] In fact, the JastAdd Java compiler has not kept up with Java's evolution, although JastAdd itself provides the capability to do so.

As can be seen from these lessons, an effort like the JML project faces several difficulties. To have industrial influence the tools need to be maintained to support the rapidly evolving Java language, but academics often have other priorities. Nevertheless, the JML project has had an influence. One example of academic work relevant to the wider Java community is the use of KeY to identify an implementation flaw in Java's sorting algorithm [68]. Amazon Web Services (AWS) has used OpenJML for formal verification of important customer-facing software [47]. Other international companies have internal groups applying proof-based verification or have proprietary contracts with researchers in the field.

1.4 Related Work on Assessing Specification Language Projects

An earlier paper on lessons from the JML project was presented at the *VSTTE* conference in 2005 [94]. That paper focused on JML's role as an integrator of tools, which was promoted as a model for the verifying compiler grand challenge. The advantages of BISLs were touted for unifying several different tools. The paper noted problems with JML's "draconian" rules for pure methods, citing the work on observational purity as a possible way forward [9,116]. Other problems noted were these: the treatment of frame axioms, specifying and verifying "callbacks" [94, Sec. 5] (which has now assumed more importance due to Java's inclusion of lambda expressions), specifying and verifying concurrent programs, and the challenges of keeping pace with Java's evolution (as in Lesson 2).

Leavens, Leino, and Müller published a set of specification and verification challenges for sequential OO programs [95], which built on an earlier assessment by Jacobs et al. [84]. This paper led to several other papers that tried to address some of these challenges [95,106,109,143], while others remain challenges in JML (and similar languages) to this day (for example, see Lesson 9 and Sects. 3.5 and 3.6 below).

Hatcliff et al. [77] give a comprehensive overview of BISLs and highlight specification challenges outstanding at the time of that paper (2012).

Boerman et al. evaluated the compatibility of OpenJML's ESC tool and KeY's static verifier [21]. They reported both syntactic and semantic incompatibilities, and some tendency for the different tools to generate new dialects of JML (as mentioned in Lesson 1).

Cok has performed specification and verification of industrial code, which gave rise to recommendations and implementations of new language features. It also identified some outstanding language design problems, some of which are repeated here as still outstanding [42,43,47].

2 Lessons About Developing Specification Languages and Tools

2.1 Effect on Other Specification Language Projects

JML has had a strong influence on subsequent specification languages. The languages discussed below started development after JML and borrowed syntax and semantics from JML. ACSL, ACSL++ and Spec# are BISLs that have adjusted JML's ideas to be appropriate for their target programming languages. Given the subsequent concurrent development of these languages, there has been significant cross-fertilization, with new features, concepts, and techniques being borrowed in all directions. This interaction is a significant successful contribution of the JML project. (Some additional specification languages are described in Sect. 5.)

ACSL and Frama-C. ACSL is a BISL for C [13] and there is a C++ variant designed for C++ (ACSL++), both part of the Frama-C ecosystem[14,66]. These languages and tools are used for research and industrial application on C and C++ programs. The Frama-C ecosystem includes many language analysis tools, going beyond the static and runtime checking discussed in this paper.

Spec#. Microsoft developed the C# language as a combination of features from Java and C++. Researchers (including Leino) at Microsoft Research defined the Spec# specification language as a BISL for C# and built powerful tools for static and runtime checking [11]. Spec# featured excellent tool support and a novel ownership system based on an `owner` ghost field, which allowed for ownership transfer and a flexible treatment of invariants in what is known as the "Boogie methodology" [10,108].

2.2 The Costs and Benefits of Specification

The JML project aims to increase the value users obtain from formal specifications and thereby improve safety and security of software through the use of formal specifications. An earlier position paper [94, p.137], stated that the main problem is to "give users enough value to justify the cost of specification." With the tremendous advances in automated tools for verification and other quality enhancement and debugging tasks, users obtain more benefits from verification, but the cost and effort of writing specifications is still high.

In our experience, formal specifications are roughly as long (say, in lines of text) as the code that they specify. This suggests that the cost of writing a formal specification is roughly the same as the cost of writing code, meaning that adding formal specifications roughly doubles the initial cost of software development. However, there is some evidence [75, Myth 5] that the effort put into specification decreases the total cost of software—in part because if specifications are created early in the development process, they enforce careful thinking about the problem and thereby enable a cleaner and quicker implementation with some problems avoided early in the software process.

Solutions to the perceived problem of the cost of specification are not easily found. For specifying detailed designs with JML, the effort involves thinking about the way that interfaces between software modules *should* work. Since specifications capture the intent of software interfaces, there are cases where they cannot be inferred from the behavior of code. However, we do believe that inferring specifications from code (including tests) can be helpful in reducing the cost of specification, as in many cases the inferred specification will be what is intended, and in other cases the specifier will be able to edit or adjust the inferred specifications.

Lesson 3. *Tools that infer specifications from code and defaults that match most common uses could reduce the length of and the burden of writing specifications.*

On the other hand, we do not believe that making specifications shorter would significantly decrease the effort or skill needed to write specifications, since good detailed designs require thought and experience. Furthermore, formal specifications should be read more often than they are written, as software modules usually have many clients and the specification should only need to be written a few times, so ease of understanding specifications is more important than ease of writing them.

2.3 Documenting Existing Java Programs (Goal 1)

Specifying and verifying programs as they are designed and written can be accomplished in two ways. One could write programs in a language designed for specification and verification, such as Dafny [107, 110], in which one verifies as one writes the program, and then has Dafny automatically translate the program to an industrially used language. Or one could start by designing and writing a program in an industrially used language, writing specifications and verifying them more or less concurrently; however, in practice, the verification lags a bit. In either case, the discipline of creating a design that is verifiable requires a clarity of thought and modularity of design that is beneficial for the software development endeavor overall [75].

However, JML can also be used to verify legacy programs. Here, the program was designed and written, perhaps carefully, perhaps not, but certainly not with an eye towards verification. As a result there are more side-effects, more connections between otherwise independent modules, more implicit assumptions, unstated data invariants, etc., than would be desirable for verification. The exercise of verifying a legacy program can be very useful in identifying and documenting these implicit assumptions and even in finding latent bugs.

JML was designed to support the verification of legacy programs. This implies that the language must have enough features to be able to express the messiness of legacy programs, and not just enough to work with correct-by-construction techniques.

Experience with verifying legacy programs shows that JML is largely up to the task, but illustrates a few observations:

– Complexity of verification arises not from the details of verifying intricate mathematical algorithms but from the scale of interactions across a program.
– Handling frame conditions well is important, since they are important in managing interactions. In particular, better solutions for handling obervational purity are needed, as many situations arise where methods have side-effects on hidden state.
– Good defaults and specification inference are needed to reduce the burden of writing specifications.

2.4 Being Easily Understood (Goal 2)

The goal of having JML be readily understood drove the design to use Java syntax and semantics as much as possible. This goal was largely achieved; students easily grasp the main syntax and semantics of JML along with Java. This similarity caused two problems however, which have led to some adjustment of this design decision.

First, Java's semantics for two's complement integer arithmetic operations are modulo operations that do not alert the user about overflow or underflow. If the specifications and the code have the same arithmetic semantics, then overflow bugs missed in the code will be missed just as easily in the specifications. In fact, users more often read the specifications as having mathematical (unbounded integer) semantics. Consequently, JML's default semantics for specifications is the mathematical (bigint) semantics and the default semantics for Java code is "safe" semantics—standard Java arithmetic with warnings about overflow and underflow [26].

Second, the focus on using Java syntax led to the use of Java classes to model mathematical concepts (such as sets and maps) in JML. As described in Sect. 3.3, even carefully written Java classes have complex semantics. Furthermore, some pure methods of Java's built-in types, like `Object`'s `equals` method are sometimes overridden by code with benevolent side-effects [128, Sec. 5]; for example, string equality in (some versions of) Java puts strings being compared into a table so that the string `equals` method could compare indexes into the table instead of individual characters. Therefore, we now believe that a better choice would be to use some mathematical types for specification purposes.

2.5 Formally Defining JML's Semantics (Goal 3)

The goal to document a "rigorous formal semantics" for JML, though conceptually straightforward, has proved difficult, despite some efforts [2, 16, 80, 85, 98]. An important problem with achieving a "rigorous formal semantics" is the problem of obtaining agreement on JML's semantics [99] from all groups of researchers involved. This was the topic of several sessions at JML workshops, but due to lack of agreement and differences in research needs, some different dialects of JML have arisen, tailored to different tools such as OpenJML and KeY. The problem is that codifying the semantics might come at the expense of flexibility in research directions. Furthermore, defining the semantics formally has not been a high priority compared to tool development and research questions.

In any case, the biggest impediment to having a complete formal semantics for JML is that it includes all of (sequential) Java, which is a large and rapidly evolving language. Efforts to formally specify JML have so far used a subset of Java. For example, the CoreJML effort [98] formalized only a small subset of Java. Since JAVA CARD is a subset of sequential Java with a set of features useful in Java smart cards, several projects have formalized JML with respect to a semantics of JAVA CARD, notably the VerifiCard project [80, 85] (which produced the LOOP tool) and the KeY project [2, 138].

The rapid evolution of Java itself has made it difficult for tools to track the Java language, so it is not surprising that formal semantics have lagged behind Java's evolution. This leads to an additional lesson from the project.

Lesson 4. *Formalization of semantics is easier with a language that is not evolving rapidly, but there is more interest in a popular programming language.*

2.6 Tool Design with Intermediate Languages

An important innovation in tool architecture was incorporated into Spec# [11], namely the use of an intermediate language, BoogiePL [9,49]. BoogiePL is similar to Dijkstra's guarded command language [52] and has a simple and well-understood translation into logic. By using BoogiePL as an intermediate language in the Spec# tools, the researchers were able to describe the verification conditions for Spec# programs at a high level and have BoogiePL take care of the translation into the input language used by an SMT solver. Moreover, they were able to change their specification language and its translations quickly and with less effort than if they had directly translated Spec# into the SMT input language. OpenJML's ESC tool is more difficult to modify because it uses its own internal intermediate language and its own translation to SMT. (OpenJML started development before BoogiePL was defined.)

Several other specification language tools have used BoogiePL as an intermediate language, including the verifier for Dafny [107,110].

Why3 [62] is a programming and specification language platform that uses an intermediate language, WhyML, which can fill the same role in a verifier as BoogiePL. As a platform, Why3 can transform verification conditions stated in WhyML into the input language of many different theorem provers [20]. The Krakatoa system used Why (an earlier version of Why3) to verify JAVA CARD programs annotated in a subset of JML [111]. This is possible because WhyML includes features such as exceptions and heap references. Why3 is used in the Frama-C collection of tools and in other tools originating in INRIA and CEA-LIST in France.

The KeY tool does not use BoogiePL or Why3, since Java Dynamic Logic seems to be useful as an intermediate language [138].

A good tool will provide a user with detailed information about proof failures, such as the source code location and character of failing assertions and the details of counterexamples to proofs expressed in terms of source code names. The problem with an architecture in which a tool uses an intermediate language to invoke a logical solver is that the intermediate language may encode names and logical assertions in ways unknown to the invoking tool. Thus a good intermediate language must also convert the details of solver's results back into information that the invoking tool can understand and represent to the user.

2.7 Coordinating Different Kinds of Tools

We believe that an important contribution of JML is showing how a single specification language can coordinate many different tools, in particular both static tools (such as ESC, LOOP, and KeY) and dynamic checkers (i.e., RAC tools). For this to work, it is important that the tool developers understand this goal and agree to parse and ignore (or warn about) language features that their tool cannot handle. If this goal is not agreed upon, then tool developers will be tempted to make changes to the specification language to work better with their tool, eventually resulting in incompatible versions of the language.

Technically, it is important to make the semantics of the specification language align as closely as possible among all tools, to minimize the differences among them.

One technical contribution in this area addresses the question of how undefined expressions should be handled in specifications [12,34]. In JML, the solution adopted is to consider such undefined expressions to be errors [28,30]. This semantics corresponds to allowing exceptions that occur during runtime checking of assertions to propagate out of those assertion checks, which is easy in RAC. However, this solution requires specifiers to write specifications so that undefined expressions do not occur. In essence, specifiers are responsible for writing assertions that protect against undefined expressions [102].

Lesson 5. *Considering undefined expressions to be an error in assertions helps to unify static verification and RAC.*

Another technical contribution that aligns semantics among tools is client-aware checking [135]. Client-aware checking is a technique for RAC that checks preconditions at the point of a method call (i.e., by the caller), aligning the semantics used in RAC with that used during static verification. To explain a bit, during static verification a verifier checks the precondition of a method call at the call site, using the part of the method's specification (i.e., the specification cases) visible there; thus for consistency, RAC should also make the same checks of the same specifications. Furthermore, client-aware checking during RAC helps smooth over the differences between different semantics of behavioral subtyping, as the notion of behavioral subtyping used by Findler and Felleisen [63] and by Spec# [11], both align with JML's more flexible semantics for behavioral subtyping when client-aware checking is taken as the semantics of RAC.

Lesson 6. *Client-aware checking helps reconcile static verification and RAC, as well as some differences in semantics between specification languages.*

From a project-wide perspective, however, the decision to coordinate multiple tools is not without its drawbacks. For the specification language designers, there is a constant tension on the design (from the needs of the different kinds of tools) that must be handled; in essence this is a management problem. For example, RAC prefers executable specifications, whereas deductive verification prefers provable formulations.

Another drawback is that the language tends to become too large, as the design becomes a grab bag of features needed for each tool; and when the language is large, giving it a formal semantics and documenting it becomes difficult. The problem of having a large specification language has certainly affected JML. Feature interactions also multiply as a specification language becomes larger, and discovering such unwanted feature interactions becomes harder. Current work on a 2nd edition of JML [46] aims to both codify language features that have shown their usefulness and to deprecate features that have not.

Lesson 7. *While it is useful for a specification language to coordinate several tools, this exerts pressure on the specification language to become larger.*

2.8 Specifying Java's Libraries

Whenever one writes a program in Java, one uses many classes and interfaces from the Java built-in libraries. For static verification it is necessary that these libraries have JML specifications, otherwise calls to library methods will stop any proof from proceeding.[3] Unfortunately, the Java class libraries are quite large, and sometimes change (slightly) with releases of Java; this makes it difficult for an academic project like JML to both specify these libraries and keep the specifications up to date [95, Sec. 6.1].

A solution to this problem of specifying the Java class libraries could involve tool support. Several different kinds of tool could help. First, it might be possible to translate the English documentation into JML, but in our experience, the natural language technology is not yet at that level of precision, nor is the natural language documentation always sufficiently complete. Second, it might be possible to infer a specification that the code implements. Such inferred specifications, while they would have to be adjusted by a human in some cases, could reduce the time and effort needed to write specifications for the Java class libraries. In many cases, such as methods that set or get the value of a field of a class, the inferred specification would be exactly what is desired, so inferring such specifications could save specifiers a great deal of time. The main drawbacks are the need for manual review and the possibility of inferring an incorrect specification from code that is incorrect. (That is, if the code is incorrect, then one would infer an incorrect specification from it, which would be easy to miss.)

Specifications could be inferred from code in one of several ways. First, one could infer specifications using a dynamic tool, like Daikon [60], which mines execution traces to suggest specifications. A tool such as OpenJML's ESC could then be used to check such specifications [65, 124]. Second, one could infer specifications from the implementation of the code, as discussed above. One approach, taken in the Houdini tool [65], is to guess specifications and then check to see if they are satisfied by the code. Another strategy for inferring specifications from code is to mine intended preconditions for methods from uses of those methods [132], then, with the mined precondition, one could infer the code's strongest postcondition [141] using a strongest postcondition program transformer [53]. Given a method postcondition and invariants for any loops in the method, one can infer the strongest postcondition that the method's code ensures. Clearly this will not help with abstract methods (those that have no code), but the join (with **`also`**) of the specifications for all of the implementations could be used for those, or a user could be notified that such methods need to be specified manually. That leaves the problem of inferring loop invariants.

To infer loop invariants, one could use various logical techniques [54, 69, 78, 140]. Currently no JML tool statically infers invariants, although Daikon [61] can infer invariants using its dynamic techniques.

Lesson 8. *For static verification to be practical, it is necessary to have specifications of commonly-used library functions. A specification inference tool would lessen the effort involved.*

3 Language Design Contributions and Remaining Issues

JML has made several contributions to specification language design, but some problems remain.

[3] For validity, a library should also be verified against its specifications, but we are more concerned with verifying client code in this section.

3.1 Behavioral Subtyping and Supertype Abstraction

In the 1980s and 1990s, object-oriented (OO) programming was at the height of its popularity and Java was designed as an OO programming language. The dynamic dispatch (message passing) mechanism of OO programming languages makes verification of method calls, such as `o.m()`, more difficult, because the language will call the code written for the dynamic type of the receiver object, `o`, which may even be a type that is unknown when the program is being written or verified. This problem is solved by *supertype abstraction* [97,100], which means reasoning about method calls using the specifications associated with the static type of the receiver object (which should be a supertype of the dynamic type of the receiver); such reasoning is valid if each subtype obeys the specifications of its supertype(s), that is, if all subtypes are behavioral subtypes [97]. JML uses specification inheritance to force all subtypes to be behavioral subtypes [51,97], thus making supertype abstraction valid.

JML's semantics for behavioral subtyping [97] is explicitly different than Eiffel's semantics, and follows the work of Leavens's dissertation [100] in using a more general technique for proving behavioral subtyping [33,97] that aligns with specification inheritance [51]. JML's specification inheritance forces subtypes to be behavioral subtypes, making supertype abstraction valid, so one can statically reason about dynamically-dispatched method calls using the specifications associated with the static type of the receiver object [51,97,101]. However, Eiffel allows the arguments of overriding methods to be covariantly specialized (i.e., to be declared to have parameter types that are subtypes of the corresponding parameter types of the method they override); thus Eiffel allows programs to violate behavioral subtyping [51, p. 266].

One might think that supertype abstraction limits the ability of a program to take advantage of more refined specifications in subtypes, however, one can still do that and use supertype abstraction by down-casting (in the Java code) a receiver object to the subtype(s) in question (which must necessarily be known to the program) and then using supertype abstraction for the method calls on the subtype(s), which can thus take advantage of the more refined specification. This is shown in Fig. 1, where it is assumed that there is a type `Staff` that is a supertype of the types `Doctor` and `Nurse` and that the methods `getTitle` and `isChief` are defined for the types `Doctor` and `Nurse`, respectively, but have little in common. The code in Fig. 1 shows that the down-casts would be needed in Java, and that once the down-cast is accomplished, one can use supertype abstraction to reason about the corresponding method calls. If there were other subtypes of `Staff` with similar methods, these could be handled in the same way by down-casting and then using supertype abstraction on these subtypes.

A problem with supertype abstraction is that it interacts with other aspects of JML in ways that may be too restrictive. One such aspect is JML's notion of a pure method, which prohibits all assignments to pre-existing storage in a method. However, there are some methods, such as `Object`'s methods `equals` and `toString`, which need to be pure, so that they can be used in specifications, but can usefully be implemented with benevolent side effects. One solution to this problem is to weaken JML's definition of pure methods by using a notion of observational purity, allowing benevolent side effects; such a solution has been worked out by Naumann [116]. Another solution,

```
/*@ requires p instanceof Doctor
  @      || p instanceof Nurse;
  @ ensures \result ==>
  @      (p instanceof Doctor
  @       && ((Doctor)p).getTitle().startsWith("Head"))
  @    || (p instanceof Nurse && ((Nurse)p).isChief());
  @*/
public /*@ pure @*/ boolean isHead(Staff p) {
  if (p instanceof Doctor) {
      Doctor doc = (Doctor) p;
      return doc.getTitle().startsWith("Head");
  } else {
    Nurse nrs = (Nurse) p;
    return nrs.isChief();
  }
}
```

Fig. 1. Java code with JML annotations showing how to take advantage of refined specifications in a subtype.

which is perhaps simpler and also takes into account Lesson 9 (see Sect. 3.3), is to change JML to use a collection of built-in types for specifications, so that such Java methods would not need to be used in specifications.

3.2 Specification Cases and Specification Inheritance

A contribution related to OO programming is that JML uses specification inheritance to force subtypes to be behavioral subtypes [51,90,97]. Method specifications from supertypes are inherited as "specification cases" [91,145,147]. In JML a specification case is a method specification consisting of requires and ensures clauses, along with assignable clauses and other method specification features [99, Sec. 9.2]. For example, consider the code in Figs. 2, 3, 4, and 5. The `setAge` method of `Animal` inherits specification cases from both `NormalSetAge` and `ExceptionalSetAge`, and thus the implementation must follow both specifications, changing the age to the argument given when that is between 0 and 150 and leaving the age unchanged when the argument is (strictly) less than 0.

```
public interface Age {
    //@ model instance int age;
}
```

Fig. 2. The interface `Age` that declares a model field `age`.

One way in which specification cases are used in JML is to write different specification cases for a method's normal and exceptional behavior. JML's syntax separates spec-

```
public interface NormalSetAge extends Age {
    /*@  requires 0 <= a && a <= 150;
      @  assignable age;
      @  ensures age == a;    @*/
    public void setAge(int a);
}
```

Fig. 3. The interface `NormalSetAge`.

```
public interface ExceptionalSetAge extends Age {
    /*@   requires a < 0;
      @   assignable \nothing;
      @   ensures age == \old(age);   @*/
    public void setAge(int a);
}
```

Fig. 4. The interface `ExceptionalSetAge`.

```
public class Animal implements Gendered,
           NormalSetAge, ExceptionalSetAge {
    protected boolean _gen; //@ in gender;
    /*@ protected represents gender
      @          = (_gen ? "female" : "male");
      @*/

    protected int _age = 0; //@ in age;
    //@ protected represents age = _age;

    //@ requires g.equals("female")||g.equals("male");
    //@ ensures gender.equals(g);
    public Animal(String g) {
        _gen = g.equals("female");
    }

    public /*@ pure @*/ boolean isFemale()
    { return _gen; }

    // specification is inherited
    public void setAge(int a) {
        if (a >= 0) { _age = a; }
    }
}
```

Fig. 5. The class `Animal`.

ification cases with the keyword **also**. For example, Fig. 6 shows an equivalent specification of `setAge` that combines the two specification cases that would be inherited (in the class `Animal` from the interfaces `NormalSetAge` and `ExceptionalSetAge`.

```
    /*@  requires 0 <= a && a <= 150;
      @  assignable age;
      @  ensures age == a;
      @ also
      @  requires a < 0;
      @  assignable \nothing;
      @  ensures age == \old(age);    @*/
    public void setAge(int a) {
        if (a >= 0) { _age = a; }
    }
```

Fig. 6. Explict specification of `setAge` using **also**. This is equivalent to the specification that is inherited by `setAge` in Fig. 5.

From the client's point of view, a combination of specification cases with **also** is a join, as the client can use any (or all) of them [97]. From the implementation's point of view, this combination is a meet, as the implementation must satisfy all specification cases.

To help visualize specification cases, imagine a single specification case with a single pre- and postcondition. As shown in Fig. 7, the precondition determines a set of states for which the method must be defined; one can visualize one pre-state as a point in this set.[4] For each specified pre-state, the postcondition determines a set of acceptable post-states, which one can visualize as a set of points above each pre-state (visualized in the figure as a vertical line of points). An implementation of the specified method, `m`, must deliver a post-state in the specified area for each specified pre-state;[5] thus the graph shows that the method is correct, since it delivers one of the specified post-states for each specified pre-state.

Figure 8 shows two specification cases joined with **also** (e.g., as in Fig. 6). For both the red and blue specifications, each postcondition specifies an acceptable set of post-states for each pre-state. What happens if the preconditions overlap? For each pre-state that is allowed by both the red and blue specifications (and thus is shown as purple), a correct implementation must deliver a post-state in both the red and blue sets of acceptable post-states, thus it must deliver a post-state in the purple intersection area to satisfy both postconditions. For example, consider the class `Human`, which is declared (in Fig. 9) to be a subtype of `Animal`. For a call such as `h.setAge(75)`, where `h` is an

[4] The figure shows the sets of pre-states and post-states as connected, but that is just for convenience in the figure; it is not necessary or required by JML.

[5] The figure shows the method as delivering a connected set of post-states, but again that is not necessary.

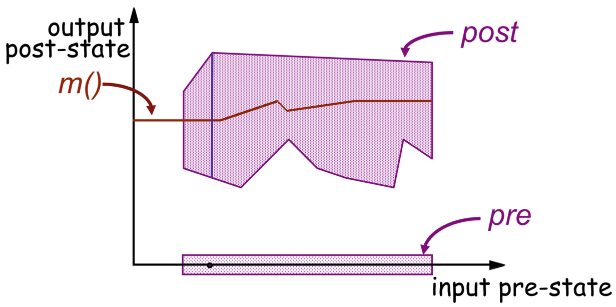


Fig. 7. Visualizing the meaning of a single specification case and correctness of an implementation. The dot on the horizontal axis represents a specific pre-state, and the vertical line above it represents the set of specified post-states allowed for that pre-state by the postcondition.

object of type `Human` the `setAge` method must obey the normal specification given in `NormalSetAge` and also the added specification case in `Human`'s `setAge` method specification (following **`also`**).

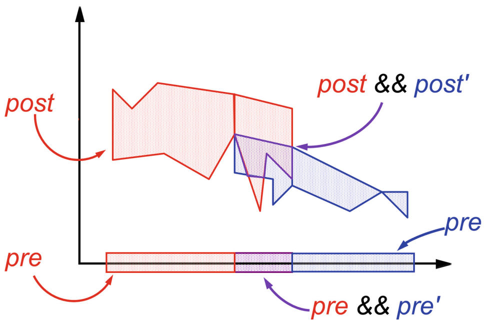


Fig. 8. A visualization of the meaning of the join (with `also`) of two specification cases.

JML uses **`also`** to combine inherited specification cases (possibly from multiple supertypes) with any specifications given by an overriding method's declaration. The `setAge` method of `Animal` illustrates this, as it inherits specifications from both `NormalSetAge` and `ExceptionalSetAge`. As in Fig. 8, a correct implementation must satisfy each such specification case. Therefore, a client of a method can use any specification case(s) available to reason about a method call, since all are satisfied by the implementation. For example, if a method has both a normal and exceptional specification case, then the client can make sure that the call satisfies the precondition of the normal specification case and can validly assume the postcondition of that specification case after the call [97].

While this way of combining specification cases [144,145,147] is the least restrictive method that ensures plug-compatibility [33], it does have one significant drawback,

```
public class Human extends Animal {
    //@ public model boolean discount; //@ in age;
    protected boolean _discount = false; //@ in age;
    //@ protected represents discount = _discount;

    /*@ also
      @   requires 65 <= a && a <= 150;
      @   assignable age;
      @   ensures discount;    @*/
    public void setAge(int a) {
        super.setAge(a);
        if (65 <= _age) { _discount = true; }
    }
}
```

Fig. 9. The class `Human` that adds behavior to the `setAge` method it inherits from its superclass `Animal`.

which is that it is easy to create method specifications that are unsatisfiable. In Fig. 10, one can see that where the blue and red preconditions overlap, the intersection of their sets of acceptable post-states is empty (as shown in Fig. 11), thus for states in the intersection of the two preconditions, it is impossible for an implementation to satisfy the specification. Some JML tools now check for such problems, but the specifier needs to take them into account when designing types that may eventually have subtypes.

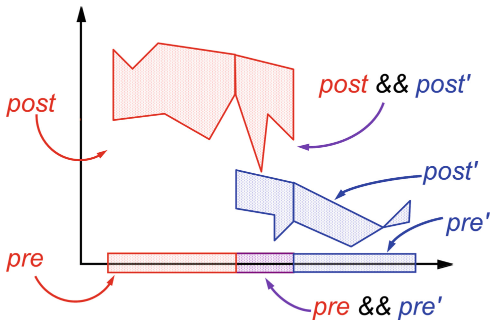


Fig. 10. A visualization of the meaning of the join (with `also`) of two specification cases that produces an unsatisfiable specification.

JML's specification cases (which allow multiple sets of pre- and postconditions) follow the work of Wing on Larch/CLU [147] and the capsules of the Fresco specification language by Wills [144–146]. Although Fresco features specification inheritance, it makes behavioral subtyping optional.

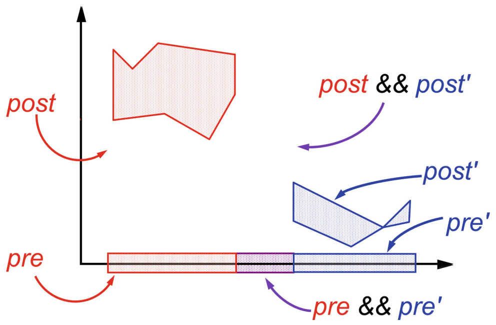


Fig. 11. A visualization of the meaning of the join that has resulted in an unsatisfiable specification.

The semantics of multiple specification cases in JML is not without controversy, however—see Sect. 4.4.

3.3 Types Used in Specification

As noted above, in an effort to make JML easily understood by programmers, JML followed Eiffel [112] by using a set of Java classes with pure methods to aid specification; these included sets, sequences, and maps.

For static verification, several researchers have pointed out problems with JML's use of Java types with (pure) Java methods to specify programs. It could also be said that JML's definition of a "pure method" is not sufficiently restrictive, since such methods are allowed to allocate storage (e.g., to create a string or array), which both modifies the heap and complicates the semantics used in program verification. Another problem with pure methods in JML is that they are not necessarily deterministic (i.e., functional) [23]. Due to these problems, JML's approach has not been followed by more recent specification languages. A better alternative, found in VDM [64,87], the Larch family of BISLs [71,72], and Dafny [105, ch. 10], is to have a set of built-in collection types (e.g., for sets, sequences, and maps), which are purely mathematical, are operated on by mathematical functions, and thus have a simple semantics; these collection types could then be emulated during RAC.[6]

To examine the problems with JML's approach more closely, first consider using non-primitive types (i.e., reference types) from the Java program under consideration. Doing so avoids creating a parallel set of specification types that replicate the information stored in these Java types. However, using such types in assertions leads to changes in the heap occurring during assertion evaluation, since JML's definition of pure methods allows for allocation of new objects; such heap allocation considerably complicates the semantics and thus makes developing tools for static verification more difficult. Most tellingly, all such types inherit `java.lang.Object`'s `toString` and `equals` methods, which one might think should be pure, but cannot be pure according

[6] An alternative solution, which could be used if one is designing a new programming language from scratch [129], is to design all types to have a value semantics.

to JML's definition. To see why, recall that JML requires overriding methods to obey the specification of the methods they override; thus, if the `toString` and `equals` methods are specified as pure methods, which they must be to allow use in specification, then such benevolent side effects would need to be disallowed—and not just in `Object` but in every class that derives from `Object`, namely every Java class. Some library classes, such as Java's `String` class, have used benevolent side effects in their code (in some versions of Java); in any case, imposing a no-side-effect restriction on all Java classes is unacceptable. The problem is that if these methods are not specified as pure, then they cannot be used in JML specifications, but the `equals` method in particular is definitely needed for specifications.

One solution to this problem would be to use actually pure Java classes for specification. However, this would not help verify the existing Java code that has behavior (e.g., if-tests) that rely on Java's `equals` method or Java's built-in types, such as strings.

A better solution seems to be to build into JML some mathematical value types. The semantics for such types could be designed to be mathematically simple, which would aid in static verification and could also be simulated during RAC. An extra advantage is that static verification tools could map such types directly into the corresponding types of theorem provers (such as SMT solvers), which would ease the task of building such tools and might result in performance improvements. For example, built-in types for maps and strings could be modeled directly in SMT solvers. The semantics of such built-in types would be more like Java's value types (i.e., primitive types such as **`boolean`**) and thus would be simpler to model mathematically, since a faithful model would not involve any heap allocation.

The syntax for operating on such built-in, mathematical types in assertions could also be made to be more mathematical, which may be either an advantage or disadvantage.

Lesson 9. *For static verification that is also adaptable to RAC, it is best if the specification language includes a built-in set of mathematical collection types and numerical types.*

3.4 Frame Conditions

A persistent semantic problem in reasoning about object-oriented (OO) programs is how to deal with frame axioms, which say what locations a method may change during its execution, and thus, implicitly, what locations must be preserved [22]. Müller proposed an ownership type system (Universes) with a relevant invariant semantics [113, 115] that JML adopted. However, Universes made transfer of ownership somewhat difficult.

The designers of Spec# created a more flexible ownership methodology based on a field in each object that indicated the object's owner [10, 108]. This "Boogie methodology" made ownership transfer easier and more flexible than the Universe type system. Spec# adopted the notion of behavioral subtyping, but in a way that was slightly less flexible than JML, since an overriding method could not change the precondition of the method(s) being overridden; thus, inheritance of specifications boils down to conjunction of postconditions [11, pp. 56–57].[7]

Research in this area is ongoing, with several mechanisms proposed, including separation logic [82,137] and region logic [4,5,7]. Region logic is better adapted to the SMT theorem provers used in JML's ESC. A flexible variant of region logic is the dynamic frames approach, in which frame axioms are specified in terms of specification-only state (ghost fields); this approach has been adopted by Dafny [103–105], the work of Smans et al. [142], and also by the KeY tool [2].

3.5 Specifying Lambda Expressions

Java 8 introduced lambda expressions; anticipating this, JML had previously added "model programs" to implement the grey-box approach [24] to specifying and verifying such higher-order features [139]. However, applying the copy rule as advocated in the original paper about model programs cannot be applied to recursive methods and the grey-box approach has not seen significant use in JML's tools.

Some researchers have advocated specifying higher-order functions using logics or specification features that are themselves higher-order, such as the ability to refer to the specifications of function parameters [48,59], but this leads to specifications that are difficult to write and understand, and are also incompatible with the first-order theorem proving technology used in ESC. There is continuing research in the formal methods community on how to specify and verify such higher-order expressions. For example, in recent work by Müller's group at ETH, Rust's ownership type system is used to help deal with framing closures [148].

3.6 Class and Object Invariants

There has been a long line of work on the specification and verification of class and object invariants [3,4,6,10,55,56,108,109,113,115]. The essential problem is that, in JML's semantics, class and object invariants are assumed at the beginning of every (non-helper) method, but these invariants are not the responsibility of clients. Instead, there must be some methodology that guarantees that invariants established by constructors and at the end of each method are still valid when they are assumed at the start of each (non-helper) method.

The key problems are representation exposure and argument exposure [38,113, 125]. *Representation exposure* occurs when references to mutable objects contained in an object's own fields are aliased by clients, as could happen when a reference to an array that is used to store the elements of a set is returned by a method. *Argument exposure* occurs when a method puts a reference to a mutable argument object into one

[7] The reason given for not allowing a loosening of preconditions in an overriding method is that "The run-time checks evoked by the method contract are thus also inherited." [11, p. 56]. However with client-aware-checking of preconditions [135], the precondition of the receiver's static type are checked at call sites, which allows JML's more flexible form of specification inheritance to still enforce precondition checks based on static type information at call sites as would be done in Spec#.

of its fields. In both of these situations, a client could hold a reference to an object A that forms part of another object, o, allowing the client to change A without calling any methods on o, thereby violating an invariant of o. Similar problems could occur if the fields of o are not hidden from clients; that is, if a client can set a public field of o, then the client can change the state of o in ways that might violate its invariants, but in that case the invariant must be maintained by all code that has visibility to change it [109].

Callbacks can also raise problems with invariants: if a method operating on a receiver object o temporarily violates an invariant of o, then calls another method, which eventually calls back to a method on o, then that call would find that o's invariant was violated. One solution, adopted in JML, is to require that all invariants be re-established when a method calls any other method [115], but this rule tends to be onerous, even unworkable at times, in practice. A more flexible solution is adopted by the Boogie methodology: not assuming object invariants on entry to every method, but specifying when they need to hold and verifying that the invariant holds at each program point where it is needed [10, 108].

4 Controversies and Continuing Discussions

There are some features of JML that are controversial and/or have not been adopted by later specification languages. The controversies and points of discussion listed in this section describe some of the features of JML that some users and researchers have objected to or that need more discussion.

4.1 Interface Specification

The idea of a BISL is found in Wing's dissertation [147]. The advantage of a BISL is that it allows precise documentation of an API in a particular programming language, including such details as privacy, types, parameter passing modes, and exceptions. The alternative is to have a more general specification language that is *not* tailored to a particular programming language. The advantage of such a specification language, such as VDM, Z, or UML, is that it can be used without change for different kinds of programs, and thus tools designed for that specification language also can be used with different programming languages.

A related technique, adopted by Whiley [129] and Dafny, is to use different code generators for a single specification language. Each such code generator would output a verified program in a different programming language, though the code-generation step itself is not verified, just the original program.

4.2 Visibility in Specifications

Java has a system of four levels of visibility for user-declared entities. As a BISL, JML also has the same four levels with the same meanings. The idea is that, for example, private specifications can be used to document design decisions within a class, which can be hidden from all other classes. Similarly, protected specifications can be used to document design decisions that are visible to subclasses (and other classes in the same package) but hidden from other clients. However, it makes no sense to have a specification be more visible that its owning entity [96].

However, the visibility rules for specifications are controversial, as they add complexity to JML; one can write several specifications for the same method, for example, and the tools must enforce several rules related to visibility of these specifications. Most other specification languages do not have such complex visibility rules; for example, "in Spec# method specifications have the same visibility as the method itself, and invariants are always private" [96, Sec. 6].

The original rules for visibility [96] stated that a specification with visibility V could only mention names with visibility V. Leavens and Müller justified these rules by showing how violating these rules could lead to problems. For instance, suppose that an invariant with visibility V could refer to a name N with visibility greater than V (i.e., that is visible to more parts of the program than the invariant), then N would need to obey a hidden invariant. That is, clients could violate the invariant by initializing or changing N; such invariants would destroy modularity and cause maintenance problems. Conversely, if a specification with visibility V could refer to a name N that is less visible than V, then parts of the program that could see the specification would not be able to understand it [112], which would be a maintenance problem as clients could depend on state that was intended to be hidden. For these reasons, specifications are not allowed to refer to names that are either less visible or more visible than themselves [96].

However, these rules cause complexity in JML specifications. Consider a simple public getter method that returns the value of a private field, `f`. If JML had no visibility rules, then the public postcondition could be `ensures \result == f;`. But JML's visibility rules do not allow this, as a private field cannot be used in a public specification. However, JML does allow **`represents`** clauses such as the following

```
private represents f_pub = f;
```

where `f_pub` is a public model field; this represents clause says that the value of the public field `f_pub` is the value of the private field `f`. To avoid this extra specification overhead, JML includes the modifier `spec_public`, which can declare that a (Java) private field, like `f`, is to be considered public for specification purposes. (In essence, `spec_public` is sugar for the above workaround with a represents clause.)

In practice, JML users can easily become entwined in a tangle of visibility errors. Thus many people will write specifications by declaring all fields to be `spec_public`. It is unclear whether declaring fields to be `spec_public` gives the modularity and maintainability guarantees desired [96] or is just an inconvenience. As mentioned above, most specification languages do not allow writing so many different levels of specifications, so perhaps simpler rules would give most of the benefits of JML's rules. We leave this question for future work.

4.3 Specification Placement Before Methods

JML follows Java's documentation comment convention in placing method specifications in front of the method being specified. Unfortunately, this means that the declarations of a method's formal parameters, which can be used in that method's specification, follow the specification, which goes against normal practice in programming languages. Thus the JML convention has been a continuing point of discussion, as some say that it makes specifications difficult to read.

4.4 Semantics of Multiple Specification Cases

Programmers typically expect that multiple specification cases act like an if-then-else or case statement in a programming language; that is, the specification case whose precondition matches first is obeyed and all the rest do not matter. However, that is not the semantics of multiple specification cases in JML. As explained in Sect. 3.2, all specification cases whose preconditions hold must be obeyed by the implementation. From the client's point of view this makes sense, as a client can pick any specification case (e.g., from some supertype, due to specification inheritance) to use in reasoning about a call (and the client does not need to worry about a specification case not applying because of some ordering). As noted in that Sect. 3.2, this may lead to some specifications being unsatisfiable, e.g., if the postconditions for two specification cases conflict when both preconditions hold (as shown in Fig. 11), but that would not be a problem if JML's semantics were more like the semantics of if-then-else statements. In addition, an if-then-else flavor of semantics for specification cases would require defining a total order on all specification cases, even ones inherited from multiple supertypes. JML's current semantics does not require such an ordering, which simplifies the semantics. However, JML's semantics suffers from the complementary problem of creating unsatisfiable specifications when specification cases conflict. This topic has been a continuing discussion item, especially with new users of JML.

4.5 Default Specifications

The design of user-friendly default specifications could lead to unsoundness. Default specifications are those assumed for a method when the user writes no specifications. As libraries typically have no specifications at the start of a project, the choice of defaults can have a big effect on incremental progress in verification. Traditionally JML has chosen very conservative defaults, ones that are known to be true, in order to ensure soundness in the absence of specifications. For example, the default postcondition is `ensures true` and the default frame condition is `assigns \everything`.

While these defaults are sound, they are also useless. A user must write specifications for any method that is actually used, or little will be provable. This has led to calls for better defaults, for example, that methods be assumed pure (`assigns \nothing`), even though this is in general unsound.

This tension is as yet unresolved. It is hoped, that at least for methods with source code available, specification inference can provide better defaults, but for library methods the problem remains.

5 Related Work on Specification Languages

In addition to the specification languages (ACSL and Spec#—Section 2.1) and other related work discussed above (Sect. 1.4), there have been several other formal specification language efforts that can be compared with JML.

Spark/Ada [8] is a BISL for Ada that restricts Ada to a safe subset and has tools to support proof-based verification. Ada was developed (in the 1970s)s) as a safer programming language than existing alternatives for systems programming. The SPARK subset of Ada and its associated tools provide verification capability to its industrial base. However, since the SPARK subset of Ada does not allow the use of dynamic dispatch, it does not concern itself with behavioral subtyping.

Dafny is a language designed to aid static verification [104,105,110]. It can then be compiled (a non-verified step, as yet) to a variety of target languages, so that the originally verified program can be compiled with other software or run in a variety of environments. Because it is designed from scratch for verification, rather than being designed to accommodate legacy code, the language and its semantics are simpler than those of the other languages described here. It also serves as a test-bed for specification language research and as a relatively simple language for education. Dafny was begun at Microsoft and is now an open-source, GitHub-based project, but most current development is sponsored by Amazon AWS.

Dafny, as noted above (see Sect. 3.3), has a major difference from JML in that Dafny has several built-in types, such as sets, multisets, sequences, strings, and maps, that can be used in specifications [105, ch. 10]. For behavioral subtyping, Dafny requires that an overriding method have a specification that is stronger than the method that it overrides, but the rule used for this is that the overriding method's precondition "must be implied by" the precondition of the method it overrides and the overriding method's postcondition "must imply" the postcondition of the method it overrides [105, Sec. 14.2], which is stronger (i.e., less flexible) than necessary for behavioral subtyping, although it makes supertype abstraction valid.

Stainless [58,76] (previously Leon [19]) is a specification and verification tool for Scala programs. It follows the Eiffel tradition of integrating the programming and specification languages, unlike JML. Stainless benefits from the expressiveness of Scala and the mostly functional design of Scala's built-in datatypes, which makes using Scala's built-in types more suitable for verification. Thus, compared with Lesson 9 there is less need for a separate library of mathematical types for use in specifications. Stainless uses a different verification technique from that used by JML and other verifiers based on verification conditions; these techniques could possibly be adopted by JML's static verification tools in the future. Stainless can also be used for RAC, and checks preconditions at call sites (as in Lesson 6). The design of Stainless does not seem to consider dynamic dispatch of object-oriented methods or make any provision for behavioral subtyping.

Whiley is an "open platform" [129, p. 238] for research on verifying compiler languages and tools. Originally the Whiley compiler was designed with its own intermediate language, and the programming language has been, like Dafny, designed from scratch to ease static verification. Whiley is a hybrid language with an imperative outer layer and a functional core that is pure (i.e., free of effects). Unlike Java, for built in collection types Whiley uses a value semantics, so that the value of any expression is copied during assignments and parameter passing (although Whiley also has references); this reduces aliasing and makes the built in types suitable for specification (in a way that is similar to Dafny's collection types). In Whiley specifications are not subject to behavioral subtyping.

6 Future Work for the JML Project

6.1 Tool Improvements

A priority for JML tools is to keep the OpenJML tool suite up to date with the latest version of Java. This is especially important for teaching, as it helps motivate students if the tool works with the version of Java used in industry.

To make the tools useful, they must be able to work with all of the built-in Java types, including strings and floating point numbers, which are commonly used. This goal has been the focus of some recent work on OpenJML.

As noted above, for broad use, the tools must also be able to verify programs that use the Java class libraries (see Lesson 8). However, to specify the Java class libraries, it would be helpful to have a tool that could infer specifications, which would also mean a tool to infer loop invariants. A good first step would be to work with Daikon [61] to see the extent to which it can handle loop invariant inference.

6.2 Semantic Extensions

As computers control more devices in the physical world, such as airplanes, medical devices, and self-driving cars, more computing systems will become safety-critical. Since safety-critical systems can more easily justify the cost of specification, it will become more important to extend JML to such systems. Haddad previously did some work extending JML to real-time systems [74], but certainly more remains to be done.

For control systems, concurrent programs, and systems built around state-machine abstractions, it is important to easily specify finite state machines [79]. There has been some work in adding temporal logic to JML [81], but certainly more could be done along these lines.

Although the Nijmegen and KeY groups both worked on JAVA CARD and have been concerned with the security of such programs, the area of computer security is another area that would be fruitful for specification and verification.

6.3 Documentation and Outreach

It is clear to us that, for students and those new to JML, better documentation, including examples and tutorials, is urgently needed. The modern way of making tutorial material is to create videos. Wolfgang Ahrendt gives a recorded lecture in one of the few videos available on YouTube about JML [1]. It would be helpful to provide a tutorial document or a series of videos for educational purposes for students who are new to programming or formal verification, to help them learn JML and its tools. Furthermore, this documentation can be used for teaching in formal verification schools that are held yearly, similar to the tutorial on PVS [127] that is taught in the Summer School on Formal Techniques every year and has had a significant impact for new researchers in formal methods.

For students, it would also be useful to have a website to run the latest JML compiler on a (small) program, and an IDE for JML. There was formerly a page for OpenJML's ESC on `rise4fun.com`, but that site is no longer active. There has been an Eclipse plugin for JML, but it is not up to date. Packaging releases of OpenJML (or other tools) in a container might be something that would help students as well.

Another area of research is in measurement of the benefits of formal methods. Historically, most of the focus of JML teams at different times has been on improving the language or building new tools. However, there are a few works that show the benefits of using JML for helping to solve real-world and academic research problems. For example, Nilizadeh et al. used different features of OpenJML to solve some problems in automated program repair [117, 118, 120–122] and fuzzing [123]. Also, Nilizadeh et al. investigated a tool to automatically translate counterexamples generated by OpenJML's ESC into unit tests [119]. These academic works may encourage other researchers to learn and use JML and its tools.

For tool implementers, a formal definition of JML is needed (Lesson 4).

7 Conclusions

The easiest lessons from the JML project to apply are probably the technical ones. These technical lessons include:

Lesson 5 Consider undefined expressions to be an error in assertions, which helps unify static verification and RAC.
Lesson 6 Client-aware checking helps reconcile static verification and RAC, especially with respect to behavioral subtyping.
Lesson 8 There should be a specification of the programming language's built-in libraries, and to do that a specification inference tool would be helpful.
Lesson 9 A specification language should include a built-in set of mathematical collection types and numerical types.

Less easily applied are the political and managerial lessons to be learned from JML. The most important of these relates to a key characteristic of JML: that it coordinates several different tools (Lesson 7). The trouble is that coordinating several tools tends to make the specification language bigger. Adding to this problem is the motivation of academic participants: they want to create novel results, and so try to extend the language in new directions (Lesson 1), and they are more strongly motivated if the programming language is popular, but working with a popular programming language may mean that the language evolves too quickly for academics to track (Lesson 4). These lessons are perhaps inherent tradeoffs in specification using a BISL and may imply that a commercial entity needs to be involved. For reference, we include these other lessons below.

Lesson 1 Academics are only strongly motivated to work on tools if they can make novel changes, but such changes make specifications less portable among tools.
Lesson 2 Supporting a current, industrially used language enables broader use, but the cost of keeping the tools current with a rapidly evolving language is substantial.
Lesson 3 Tools that infer specifications from code and defaults that match most common uses could reduce the length of and the burden of writing specifications.

Lesson 4 Formalization of semantics is easier with a language that is not evolving rapidly, but there is more interest in a popular programming language.

Lesson 7 While it is useful for a specification language to coordinate several tools, this exerts pressure on the specification language to become larger.

Acknowledgments. Thanks to David Pearce for comments on an earlier draft of this paper. Thanks to all participants in the JML project (many of whom are named above). Thanks to the US National Science Foundation for funding early work on JML through several grants (at Iowa State University).

References

1. Ahrendt, W.: The Java Modeling Language - a basis for static and dynamic verification, June 2018. https://youtu.be/9ItK0jxJ0oQ
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Banerjee, A., Naumann, D.A.: Local reasoning for global invariants, part ii: dynamic boundaries. J. ACM **60**(3), 19:1–19:73 (2013). https://doi.org/10.1145/2485981. http://doi.acm.org/10.1145/2485981
4. Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part i: region logic. J. ACM **60**(3), 18:1–18:56 (2013). https://doi.org/10.1145/2485982. http://doi.acm.org/10.1145/2485982
5. Bao, Y.: Reasoning about frame properties in object-oriented programs. Technical report, CS-TR-17-05, Computer Science, University of Central Florida, Orlando, Florida, December 2017. The author's dissertation. https://goo.gl/WZGMiB
6. Bao, Y., Leavens, G.T.: A methodology for invariants, framing, and subtyping in JML. In: Müller, P., Schaefer, I. (eds.) Principled Software Development, pp. 19–39. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98047-8_2
7. Bao, Y., Leavens, G.T., Ernst, G.: Unifying separation logic and region logic to allow interoperability. Form. Asp. Comput. **30**, 381–441 (2018). https://doi.org/10.1007/s00165-018-0455-5
8. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, New York (2003)
9. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
10. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. J. Object Technol. **3**(6), 27–56 (2004). http://tinyurl.com/m2a8j
11. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
12. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Informatica **21**(3), 251–269 (1984)
13. Baudin, P., et al.: ACSL: ANSI C Specification Language (2008ff). http://frama-c.com/download/acsl_1.4.pdf

14. Baudin, P., et al.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
15. Beckert, B.: A dynamic logic for Java Card. In: Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A. (eds.) Workshop on Formal Techniques for Java Programs (FTfJP). Technical report 269, Fernuniversität Hagen (2000)
16. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
17. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_21
18. van den Berg, J., Poll, E., Jacobs, B.: First steps in formalising JML: exceptions in predicates. In: Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A. (eds.) Formal Techniques for Java Programs. Proceedings of the ECOOP'00 Workshop. Technical report, Fernuniversität Hagen (2000). http://www.cs.ru.nl/~erikpoll/publications/ftfjp00.ps.gz
19. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala. SCALA 2013. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2489837.2489838
20. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wrocław, Poland (2011). https://hal.inria.fr/hal-00790310
21. Boerman, J., Huisman, M., Joosten, S.: Reasoning about JML: differences between KeY and OpenJML. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 30–46. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_3
22. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Softw. Eng. **21**(10), 785–798 (1995). http://doi.ieeecomputersociety.org/10.1109/32.469460
23. Breunesse, C.B., Poll, E.: Verifying JML specifications with model fields. In: Formal Techniques for Java-like Programs (FTfJP), pp. 51–60. No. 408 in Technical report, ETH Zurich, July 2003. http://www.cs.ru.nl/~erikpoll/publications/ftfjp03.pdf
24. Büchi, M., Weck, W.: A plea for grey-box components. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems Workshop. University of Central Florida (1997). https://www.cs.ucf.edu/~leavens/FoCBS/buechi.html
25. Burdy, L., et al.: An overview of JML tools and applications. In: Arts, T., Fokkink, W. (eds.) Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 80, pp. 73–89. Elsevier (2003). http://www.sciencedirect.com/science/journal/15710661
26. Chalin, P.: Back to basics: language support and semantics of basic infinite integer types in JML and Larch. Technical report, CU-CS 2002–003.1, Computer Science Department, Concordia University (2002). http://www.cs.concordia.ca/~faculty/chalin/papers/TR-CU-CS-2002-003.1.pdf
27. Chalin, P.: Improving JML: for a safer and more effective language. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 440–461. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_25 http://www.springerlink.com/content/26cpmd9b3vbgd2et
28. Chalin, P.: Logical foundations of program assertions: What do practitioners want? In: Proceedings of the 3rd International Conference on Software Engineering and Formal Method (SEFM). IEEE Computer Society, Los Alamitos, California (2005). http://www.cs.concordia.ca/~chalin/papers/TR-2005-002-r2.pdf

29. Chalin, P.: Towards support for non-null types and non-null-by default in Java. In: Workshop on Formal Techniques for Java-like Programs (FTfJP) (2006). http://www.disi.unige.it/person/AnconaD/FTfJP06/paper03.pdf
30. Chalin, P.: A sound assertion semantics for the dependable systems evoluation verifying compiler. In: International Conference on Software Engineering (ICSE), Los Alamitos, California, pp. 23–33. IEEE, May 2007. http://dx.doi.org/10.1109/ICSE.2007.9
31. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_16 https://tinyurl.com/3z2vk55n
32. Chalin, P., Rioux, F.: Non-null references by default in the Java modeling language. In: Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS 2005). ACM Software Engineering Notes, vol. 31, no. 2. ACM (2005)
33. Chen, Y., Cheng, B.H.C.: A semantic foundation for specification matching. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 91–109. Cambridge University Press, New York (2000)
34. Cheng, J.H., Jones, C.B.: On the usability of logics which handle partial functions. In: Morgan, C., Woodcock, J.C.P. (eds.) Proceedings of the Third Refinement Workshop. Workshops in Computing Series, pp. 51–69. Springer, Berlin (1991)
35. Cheon, Y., Leavens, G.T.: A quick overview of Larch/C++. J. Object-Oriented Program. **7**(6), 39–49 (1994)
36. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java modeling language (JML). In: Arabnia, H.R., Mun, Y. (eds.) Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2002), Las Vegas, Nevada, USA, 24–27 June 2002, pp. 322–328. CSREA Press (2002). ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf
37. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: the JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47993-7_10 https://tinyurl.com/4tk2nzzd
38. Clarke, D.G., Noble, J., Potter, J.M.: Simple ownership types for object containment. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 53–76. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_4
39. Clifton, C.: MultiJava: design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical report, 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. ftp://ftp.cs.iastate.edu/pub/techreports/TR01-10/TR.pdf. The author's masters thesis
40. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
41. Cok, D.R.: (2018). http://www.openjml.org
42. Cok, D.R.: Reasoning about functional programming in Java and C++. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, pp. 37–39. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3236454.3236483
43. Cok, D.R.: JML and OpenJML for Java 16. In: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2021, pp. 65–67. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3464971.3468417

44. Cok, D.R., Kiniry, J.: ESC/Java2: uniting ESC/Java and JML. progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system, May 2004. Presented at CASSIS 2004 and submitted for publication
45. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_6
46. Cok, D.R., Leavens, G.T., Ulbrich, M.: Java Modeling Language (JML) Reference Manual, 2nd edn (2022, in progress). https://www.openjml.org/documentation/JML_Reference_Manual.pdf
47. Cok, D.R., Tasiran, S.: Practical methods for reasoning about java 8's functional programming features. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 267–278. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_15
48. Damm, W., Josko, B.: A sound and relatively complete Hoare-logic for a language with higher type procedures. Acta Informatica **20**(1), 59–101 (1983). http://dx.doi.org/10.1007/BF00264295
49. DeLine, R., Leino, K.R.M.: BoogiePL: a typed procedural language for checking object-oriented programs. Technical report, MSR-TR-2005-70, Microsoft Research (2005). ftp://ftp.research.microsoft.com/pub/tr/TR-2005-70.pdf
50. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998
51. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pp. 258–267. IEEE Computer Society Press, Los Alamitos (1996). http://doi.ieeecomputersociety.org/10.1109/ICSE.1996.493421. A corrected version is ISU CS TR #95-20c. http://tinyurl.com/s2krg
52. Dijkstra, E.W.: Guarded commands, nondeterminancy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
53. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall Inc, Englewood Cliffs (1976)
54. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2013, pp. 443–456. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2509136.2509511
55. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_18
56. Drossopoulou, S., Francalanza, A., Müller, P.: A unified framework for verification techniques for object invariants. In: International Workshop on Foundations of Object-Oriented Languages (FOOL 2008) (2008). http://fool08.kuis.kyoto-u.ac.jp/drossopoulou.pdf
57. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Sci. Comput. Program. **69**(1–3), 14–26 (2007). https://doi.org/10.1016/j.scico.2007.02.003
58. EPFL, Lausanne, Switzerland: Stainless Verification Framework (2022). https://epfl-lara.github.io/stainless/intro.html
59. Ernst, G.W., Navlakha, J.K., Ogden, W.F.: Verification of programs with procedure-type parameters. Acta Informatica **18**(2), 149–169 (1982)
60. Ernst, M., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 99–123 (2001). http://doi.ieeecomputersociety.org/10.1109/32.908957

61. Ernst, M., et al.: Daikon website. https://plse.cs.washington.edu/daikon/. Accessed Sept 2021
62. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
63. Findler, R.B., Felleisen, M.: Contract soundness for object-oriented languages. In: OOPSLA 2001 Conference Proceedings. Object-Oriented Programming, Systems, Languages, and Applications, 14–18 October 2001, Tampa Bay, Florida, USA, pp. 1–15. ACM, New York (2001)
64. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools in Software Development. Cambridge University Press, Cambridge (1998)
65. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29 http://www.springerlink.com/content/nxukfdgg7623q3a9
66. (2011ff). https://frama-c.com
67. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. The Java Series. Addison-Wesley, Reading (1996). http://www.aw.com/cp/javaseries.html
68. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
69. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 281–292. Association for Computing Machinery, New York (2008). https://doi.org/10.1145/1375581.1375616
70. Guttag, J.V., Horning, J.J.: Report on the larch shared language. Sci. Comput. Program. **6**(2), 103–134 (1986)
71. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Springer, New York (1993). https://doi.org/10.1007/978-1-4612-2704-5
72. Guttag, J.V., Horning, J.J., Wing, J.M.: The Larch family of specification languages. IEEE Softw. **2**(5), 24–36 (1985)
73. Haddad, G., Leavens, G.T.: Extensible dynamic analysis for JML: a case study with loop annotations. Technical report, CS-TR-08-05, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, April 2008
74. Haddad, G., Leavens, G.T.: Specifying subtypes in safety critical Java programs. Concurr. Comput. Pract. Exp. **25**(16), 2290–2306 (2013)
75. Hall, A.: Seven myths of formal methods. IEEE Softw. **7**(5), 11–19 (1990)
76. Hamza, J., Voirol, N., Kunčak, V.: System FR: formalized foundations for the Stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360592
77. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012). https://doi.org/10.1145/2187671.2187678. http://doi.acm.org/10.1145/2187671.2187678
78. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_7
79. Hubbers, E., Oostdijk, M., Poll, E.: From finite state machines to provably correct JavaCard applets. In: Workshop of IFIP WG 11.2 - Small Systems Security. IFIP (2003). http://www.cs.ru.nl/~erikpoll/publications/sec03.pdf

80. Huisman, M., Jacobs, B.: Java program verification via a hoare logic with abrupt termination. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 284–303. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46428-X_20
81. Hussain, F., Leavens, G.T.: temporaljmlc: a JML runtime assertion checker extension for specification and checking of temporal properties. Technical report, CS-TR-10-08, UCF, Department of EECS, Orlando, Florida, July 2010
82. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2001, pp. 14–26. ACM, New York (2001). http://doi.acm.org/10.1145/360204.375719
83. Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M., Hensel, U., Tews, H.: Reasoning about Java classes (preliminary report). In: OOPSLA 1998 Conference Proceedings. ACM SIGPLAN Notices, vol. 33, no. 10, pp. 329–340. ACM, October 1998
84. Jacobs, B., Kiniry, J., Warnier, M.: Java program verification challenges. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 202–219. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39656-7_8
85. Jacobs, B., Meijer, H., Poll, E.: VerifiCard: a European project for smart card verification. Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI) (2001). https://repository.ubn.ru.nl/bitstream/handle/2066/130369/130369.pdf
86. Jones, C.B.: Program specification and verification in VDM. Technical report, UMCS-86-10-5, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, November 1986
87. Jones, C.B.: Systematic software development using VDM. International Series in Computer Science, Prentice-Hall Inc., Englewood Cliffs (1986)
88. The KeY project. https://www.key-project.org. Accessed Sept 2021
89. Leavens, G.T.: An overview of Larch/C++: behavioral specifications for C++ modules. In: Kilov, H., Harvey, W. (eds.) Specification of Behavioral Semantics in Object-Oriented Information Modeling, chap. 8, pp. 121–142. Kluwer Academic Publishers, Boston (1996). An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011
90. Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 2–34. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_2
91. Leavens, G.T., Baker, A.L.: Enhancing the pre- and postcondition technique for more expressive specifications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1087–1106. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_8
92. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a Java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA 1998), October 1998. http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html
93. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). http://doi.acm.org/10.1145/1127878.1127884
94. Leavens, G.T., Clifton, C.: Lessons from the JML project. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 134–143. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69149-5_15
95. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects Comput. **19**(2), 159–189 (2007). http://dx.doi.org/10.1007/s00165-007-0026-7

96. Leavens, G.T., Müller, P.: Information hiding and visibility in interface specifications. In: International Conference on Software Engineering (ICSE), Los Alamitos, California, pp. 385–395. IEEE, May 2007. http://dx.doi.org/10.1109/ICSE.2007.44
97. Leavens, G.T., Naumann, D.A.: Behavioral subtyping, specification inheritance, and modular reasoning. TOPLAS **37**(4), 13:1–13:88 (2015). https://doi.org/10.1145/2766446. http://doi.acm.org/10.1145/2766446
98. Leavens, G.T., Naumann, D.A., Rosenberg, S.: Preliminary definition of Core JML. CS Report 2006-07, Stevens Institute of Technology, September 2006. http://www.cs.stevens.edu/~naumann/publications/SIT-TR-2006-07.pdf
99. Leavens, G.T., et al.: JML Reference Manual, May 2008. http://www.jmlspecs.org
100. Leavens, G.T., Weihl, W.E.: Reasoning about object-oriented programs that use subtypes (extended abstract). In: Meyrowitz, N. (ed.) OOPSLA ECOOP 1990 Proceedings. ACM SIGPLAN Notices, vol. 25, no. 10, pp. 212–223. ACM (1990). http://doi.acm.org/10.1145/97945.97970
101. Leavens, G.T., Weihl, W.E.: Specification and verification of object-oriented programs using supertype abstraction. Acta Informatica **32**(8), 705–778 (1995). http://dx.doi.org/10.1007/BF01178658
102. Leavens, G.T., Wing, J.M.: Protective interface specifications. Formal Aspects Comput. **10**(1), 59–75 (1998). http://dx.doi.org/10.1007/PL00003926
103. Leino, K.R.M.: Specification and verification of object-oriented software (2008). http://research.microsoft.com/en-us/um/people/leino/papers/krml190.pdf. Lecture notes from Marktoberdorf Internation Summer School. http://research.microsoft.com/en-us/um/people/leino/papers/krml190.pdf
104. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
105. Leino, K.R.M., Ford, R.L., Cok, D.R.: Dafny reference manual, July 2021. https://github.com/dafny-lang/dafny/blob/master/docs/DafnyRef/out/DafnyRef.pdf
106. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 615–622. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1529282.1529411
107. Leino, K.R.M., Monahan, R.: Dafny meets the verification benchmarks challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_8
108. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24851-4_22 http://www.springerlink.com/content/ttfnjg36yq64pah8
109. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 26–42. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_4 https://tinyurl.com/4xfc2989
110. Leino, K.R.M., et al.: Dafny github site. https://github.com/dafny-lang/dafny. Accessed Sept 2021
111. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. J. Logic Algebraic Program. **58**(1–2), 89–106 (2004). http://dx.doi.org/10.1016/j.jlap.2003.07.006
112. Meyer, B.: Object-Oriented Software Construction, vol. 2. Prentice Hall, New York (1997)
113. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45651-1 http://tinyurl.com/jtwot

114. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular specification of frame properties in JML. Concurr. Comput. Pract. Exp. **15**(2), 117–154 (2003). https://doi.org/10.1002/cpe.713. ftp://ftp.cs.iastate.edu/pub/techreports/TR02-02/TR.pdf
115. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. **62**(3), 253–286 (2006). http://dx.doi.org/10.1016/j.scico.2006.03.001
116. Naumann, D.A.: Observational purity and encapsulation. Theor. Comput. Sci. **376**(3), 205–224 (2007)
117. Nilizadeh, A.: Test overfitting: challenges, approaches, and measurements. Technical report, University of Central Florida, Computer Science (2021)
118. Nilizadeh, A.: Automated program repair and test overfitting: measurements and approaches using formal methods. In: 2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE (2022, in press)
119. Nilizadeh, A., Calvo, M., Leavens, G.T., Cok, D.R.: Generating counterexamples in the form of unit tests from Hoare-style verification attempts. In: IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE). IEEE (2022, in press)
120. Nilizadeh, A., Calvo, M., Leavens, G.T., Le, X.B.D.: More reliable test suites for dynamic APR by using counterexamples. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), Los Altos, pp. 208–219. IEEE (2021). https://doi.org/10.1109/ISSRE52982.2021.00032
121. Nilizadeh, A., Leavens, G.T.: Be realistic: automated program repair is a combination of undecidable problems. In: 2022 IEEE/ACM International Workshop on Automated Program Repair (APR). IEEE (2022, in press)
122. Nilizadeh, A., Leavens, G.T., Le, X.B.D., Păsăreanu, C.S., Cok, D.R.: Exploring true test overfitting in dynamic automated program repair using formal methods. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), Los Altos, pp. 229–240. IEEE (2021). https://tinyurl.com/bn3ecw98
123. Nilizadeh, A., Leavens, G.T., Păsăreanu, C.S.: Using a guided fuzzer and preconditions to achieve branch coverage with valid inputs. In: Loulergue, F., Wotawa, F. (eds.) TAP 2021. LNCS, vol. 12740, pp. 72–84. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79379-1_5 https://tinyurl.com/4xzxxrn2
124. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: integrating daikon and ESC/Java. In: Proceedings of RV'01, First Workshop on Runtime Verification. Elsevier (2001). http://dx.doi.org/10.1016/S1571-0661(04)00256-7
125. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054091
126. Oracle: OpenJDK (2021). http://openjdk.java.net/. Accessed Sept 2021
127. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
128. Pearce, D.J.: JPure: a modular purity system for java. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 104–123. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19861-8_7
129. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 238–248. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_13
130. Poll, E., van den Berg, J., Jacobs, B.: Specification of the JavaCard API in JML. In: Domingo-Ferrer, J., Chan, D., Watson, A. (eds.) Smart Card Research and Advanced Application Conference (CARDIS 2000), pp. 135–154. Kluwer Academic Publishers (2000)

131. Raghavan, A.D.: Design of a JML documentation generator. Technical report, 00-12, Iowa State University, Department of Computer Science, July 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-12/TR.ps.gz
132. Rajan, H., Nguyen, T.N., Leavens, G.T., Dyer, R.: Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence. In: ICSE 2015: The 37th International Conference on Software Engineering: NIER Track, pp. 579–582, May 2015. https://tinyurl.com/jpemux34
133. Rebêlo, H.: AspectJML website (2021). https://www.cin.ufpe.br/~hemr/aspectjml/. Accessed Sept 2021
134. Rebêlo, H., et al.: AspectJML: modular specification and runtime checking for crosscutting contracts. In: Proceedings of the 13th International Conference on Modularity, MODULARITY 2014, pp. 157–168. ACM, New York (2014). https://doi.org/10.1145/2577080.2577084. http://doi.acm.org/10.1145/2577080.2577084
135. Rebêlo, H., Leavens, G.T., Lima, R.M.: Client-aware checking and information hiding in interface specifications with JML/Ajmlc. In: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, SPLASH 2013, pp. 11–12. ACM, New York (2013). https://doi.org/10.1145/2508075.2514569.. http://doi.acm.org/10.1145/2508075.2514569
136. Rebêlo, H., Soares, S., Lima, R., Ferreira, L., Cornélio, M.: Implementing Java modeling language contracts with AspectJ. In: SAC 2008: Proceedings of the 2008 ACM Symposium on Applied computing, pp. 228–233. ACM, New York (2008). http://doi.acm.org/10.1145/1363686.1363745
137. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, Los Alamitos, California, pp. 55–74. IEEE Computer Society Press (2002). http://dx.doi.org/10.1109/LICS.2002.1029817
138. Schmitt, P.H.: A short history of KeY. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) Deductive Software Verification: Future Perspectives. LNCS, vol. 12345, pp. 3–18. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_1 https://tinyurl.com/3xbrdwbr
139. Shaner, S.M., Leavens, G.T., Naumann, D.A.: Modular verification of higher-order methods with mandatory calls specified by model programs. In: International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada, pp. 351–367. ACM, New York (2007). http://doi.acm.org/10.1145/1297027.1297053. http://doi.acm.org/10.1145/1297027.1297053
140. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_57
141. Singleton, J.L., Leavens, G.T., Rajan, H., Cok, D.: Poster: an algorithm and tool to infer practical postconditions. In: 2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden, pp. 313–314. ACM (2018). https://doi.org/10.1145/3183440.3194986
142. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. In: Huisman, M. (ed.) Formal Techniques for Java-like Programs (FTfJP 2008), pp. 1–12. Radboud University, Nijmegen, Technical report ICIS-R08013, Radboud University (2008)
143. Svendsen, K., Birkedal, L., Parkinson, M.: Verifying generics and delegates. ECOOP 2010 - Object-oriented Programming, p. 175 (2010)
144. Wills, A.: Capsules and types in fresco: program validation in smalltalk. In: America, P. (ed.) ECOOP 1991: European Conference on Object Oriented Programming. LNCS, vol. 512, pp. 59–76. Springer, New York (1991). http://dx.doi.org/10.1007/BFb0057015

145. Wills, A.: Specification in fresco. In: Stepney, S., Barden, R., Cooper, D. (eds.) Object Orientation in Z, chap. 11, pp. 127–135. Workshops in Computing, Springer, Cambridge CB2 1LQ, UK (1992)
146. Wills, A.: Refinement in fresco. In: Lano, K., Houghton, H. (eds.) Object-Oriented Specification Case Studies, chap. 9, pp. 184–201. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (1994)
147. Wing, J.M.: A two-tiered approach to specifying programs. Technical report, TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science (1983)
148. Wolff, F., Bílý, A., Matheja, C., Müller, P., Summers, A.J.: Modular specification and verification of closures in Rust. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485522

Inference in MaxSAT and MinSAT

Chu Min Li[1,2] and Felip Manyà[3](✉)

[1] MIS, Université de Picardie Jules Verne, Amiens, France
[2] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
[3] Artificial Intelligence Research Institute (IIIA, CSIC), Bellaterra, Spain
felip@iiia.csic.es

Abstract. Logical calculi applied to solve SAT are unsound for MaxSAT and MinSAT because they preserve satisfiability but not the minimum and the maximum number of unsatisfied clauses, respectively. This paper overviews the complete resolution and tableau-style calculi that have been defined to solve MaxSAT and MinSAT, as well as their variants with hard and weighted soft clauses. These calculi provide an exact approach to solving MaxSAT and MinSAT problems.

1 Introduction

The Propositional Satisfiability problem (SAT) is the problem of deciding if there exists a truth assignment for a given propositional formula in conjunctive normal form (CNF) that evaluates the formula to true. This paper overviews inference systems of two optimization variants of SAT: Maximum Satisfiability (MaxSAT), which is the problem of finding a truth assignment that minimizes the number of unsatisfied clauses in a multiset of clauses, and Minimum Satisfiability (MinSAT), which is the problem of finding a truth assignment that maximizes the number of unsatisfied clauses. Notice that minimizing (maximizing) the number of unsatisfied clauses is equivalent to maximizing (minimizing) the number of satisfied clauses.

The inference rules applied in SAT are sound if they preserve satisfiability. Nevertheless, such rules are usually unsound in MaxSAT and MinSAT because they do not preserve the minimum and the maximum number of unsatisfied clauses between the premises and the conclusions, respectively. As a consequence, new complete inference systems for MaxSAT and MinSAT have had to be defined (see e.g. [19,25,37,43]).

In SAT, the conclusions of an inference rule are commonly added to the formula under consideration. In MaxSAT and MinSAT, after applying an inference rule, the premises are replaced by the conclusions. This is so because, by adding the conclusions to the formula under consideration, the number of unsatisfied clauses might increase and so the minimum or maximum number of unsatisfied clauses could not be preserved.

Given a multiset of clauses ϕ and an inference system $\mathcal{R}$, a refutation is a sequence of multisets of clauses $\phi_1, \dots, \phi_n$ such that $\phi_1 = \phi$, ϕ_{i+1} has been derived from ϕ_i $(1 \leq i < n)$ by applying an inference rule of $\mathcal{R}$, and ϕ_n is formed by k occurrences

This work has been supported by the French Agence Nationale de la Recherche, reference ANR-19-CHIA-0013-01, and Grant PID2019-111544GB-C21 funded by MCIN/AEI/10.13039/501100011033. The last author was supported by mobility grant PRX21/00488 of the Spanish *Ministerio de Universidades*.

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 350–369, 2022.
https://doi.org/10.1007/978-3-031-08166-8_16

of the empty clause. In MaxSAT, the soundness of $\mathcal{R}$ ensures that if there is a refutation from ϕ in which ϕ_n contains k empty clauses, then k is the minimum number of clauses of ϕ that can be unsatisfied. The completeness of $\mathcal{R}$ ensures that if the minimum number of clauses of ϕ that can be unsatisfied is k, then any refutation of ϕ_n contains k empty clauses. In MinSAT, soundness and completeness are defined analogously but considering the maximum number of unsatisfied clauses instead of the minimum.

This paper overviews the main complete inference systems that have been defined for MaxSAT and MinSAT in recent years. It also updates the contents about MaxSAT and MinSAT inference in our chapter of the Handbook of Satisfiability [36]. Firstly, we present the existing resolution-style calculi and variable elimination algorithms for MaxSAT and MinSAT [19,34,37]. Restrictions of these calculi are routinely applied in MaxSAT and MinSAT branch-and-bound solvers [1,40,49]. Furthermore, they have drawn the interest of the proof complexity community because they can become stronger than general resolution [16,17,35]. Secondly, we present the existing tableau-style calculi for MaxSAT and MinSAT [11,24,25,43]. They are interesting because they solve non-clausal MaxSAT and MinSAT problems without requiring any clausal form transformation that preserves the minimum or maximum number of unsatisfied clauses [42,44].

Although this paper is mainly theoretical, it is worth mentioning that MaxSAT and MinSAT offer a competitive generic problem solving formalism for combinatorial optimization. For example, MaxSAT and MinSAT have been applied to solve optimization problems in real-world domains as diverse as bioinformatics [26,52], circuit design and debugging [59], combinatorial auctions [49], combinatorial testing [6,9], community detection in complex networks [32], diagnosis [23], FPGA routing [62], planning [63], scheduling [15] and team formation [50,51], among others.

We currently have highly optimized MaxSAT solvers, due in part to the existence of an annual MaxSAT Evaluation (MSE) since 2006 [10,12]. The literature distinguishes two types of exact MaxSAT algorithms: Branch-and-Bound (BnB) and SAT-based algorithms. BnB algorithms apply restrictions of MaxSAT resolution and incorporate a bounding procedure based on detecting disjoint inconsistent cores with unit propagation [36]. Recently, they also incorporate clause learning [46,47]. SAT-based algorithms transform MaxSAT into a sequence of SAT instances that are solved with a Conflict-Driven Clause Learning (CDCL) SAT solver [13]. Moreover, there are efficient local search MaxSAT algorithms like SATLike 3.0 [20]. There has not been as much activity in MinSAT as in MaxSAT, but there exist a few MinSAT solvers [2,8,48].

This paper is structured as follows. Section 2 introduces definitions and notations used through the document. Section 3 reviews the MaxSAT resolution rule and describes variable elimination algorithms for MaxSAT and MinSAT. Section 4 reviews tableau-style calculi for MaxSAT and MinSAT. Finally, Sect. 5 presents some concluding remarks.

2 Preliminaries

A propositional formula is an expression constructed from propositional variables by means of the propositional connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$ in accordance with the following rules: i) each propositional variable is a propositional formula; and ii) if A and B are propositional formulas, then so are $(A \land B)$, $(A \lor B)$, $(A \rightarrow B)$, and $\neg A$. In the sequel, $\neg A$ is also denoted by $\overline{A}$.

A truth assignment is a mapping that assigns 0 (false) or 1 (true) to each propositional variable. A propositional formula is satisfied by an assignment if it is true under the usual truth-functional interpretation of the connectives and the truth values assigned to the variables. Given a multiset of propositional formulas ϕ, non-clausal MaxSAT (MinSAT) is the problem of finding an assignment of ϕ that minimizes (maximizes) the number of unsatisfied formulas. We use multisets of formulas instead of sets of formulas because duplicated formulas cannot be collapsed into a single formula because then the minimum (maximum) number of unsatisfied formulas might not be preserved.

Clauses are a particular type of propositional formulas defined as follows. A clause is a disjunction of literals, where a literal l_i is a variable x_i or its negation $\overline{x}_i$. A clausal MaxSAT instance is a multiset of clauses. Non-clausal MaxSAT (MinSAT) is called clausal MaxSAT (MinSAT) when all the formulas in the multiset are clauses.

A weighted formula is a pair (A, w), where A is a propositional formula and w, its weight, is a positive number. If A is a clause, (A, w) is a weighted clause. Given a multiset of weighted formulas (clauses) ϕ, non-clausal (clausal) weighted MaxSAT (MinSAT) is the problem of finding an assignment of ϕ that minimizes (maximizes) the sum of weights of unsatisfied formulas (clauses).

We can distinguish between hard and soft formulas. Soft formulas can be considered as formulas with weight 1, and hard formulas can be considered as formulas with infinite weight, and are usually represented with weight $\top$. Given a multiset of hard and soft formulas (clauses) ϕ, non-clausal (clausal) partial MaxSAT (MinSAT) is the problem of finding an assignment of ϕ that satisfies all the hard formulas and minimizes (maximizes) the number of unsatisfied soft formulas (clauses).

The weighted partial MaxSAT (MinSAT) problem is the combination of partial MaxSAT (MinSAT) and weighted MaxSAT (MinSAT). Given a multiset ϕ composed of hard formulas (clauses) and weighted soft formulas (clauses), non-clausal (clausal) weighted partial MaxSAT (MinSAT) is the problem of finding an assignment of ϕ that satisfies all the hard formulas (clauses) and minimizes (maximizes) the sum of weights of unsatisfied soft formulas (clauses).

3 Resolution-Style Calculi for MaxSAT and MinSAT

This section first presents the MaxSAT resolution rule and describes a variable elimination algorithm for MaxSAT. It also contains the extension of MaxSAT resolution to deal with weighted and hard clauses. Then, it describes how the MaxSAT resolution rule can be used to define a variable elimination algorithm for MinSAT.

It is important to highlight that the MaxSAT resolution rule is also sound for MinSAT because it preserves the number of unsatisfied clauses between the premises and the conclusions. The rule would be unsound for MinSAT if it only preserved the maximum but not the minimum number of unsatisfied clauses. Actually, any transformation that preserves the number of unsatisfied clauses is valid for both MaxSAT and MinSAT.

3.1 Resolution for MaxSAT

The resolution rule of Robinson [58] is unsound for MaxSAT. Accordingly, Bonet et al. [18, 19] and Heras and Larrosa [29] defined, independently, the following MaxSAT resolution rule:

$$\begin{array}{l} x \vee a_1 \vee \cdots \vee a_s \\ \overline{x} \vee b_1 \vee \cdots \vee b_t \\ \hline\hline a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_t \\ x \vee a_1 \vee \cdots \vee a_s \vee \overline{b_1} \\ x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \overline{b_2} \\ \cdots \\ x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_{t-1} \vee \overline{b_t} \\ \overline{x} \vee b_1 \vee \cdots \vee b_t \vee \overline{a_1} \\ \overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \overline{a_2} \\ \cdots \\ \overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \cdots \vee a_{s-1} \vee \overline{a_s} \end{array}$$

We say that the rule *cuts* the variable x. This inference rule concludes, apart from the conclusion where a variable has been cut, some additional clauses that contain one of the premises as subclause and are known as compensation clauses. Compensation clauses ensure that the number of unsatisfied clauses is preserved between the premises and the conclusions.

The tautologies concluded by the rule can be omitted and repeated literals in a clause can be collapsed into one. For example, we derive the empty clause ($\Box$) from x_1 and $\overline{x}_1$, x_2 from $x_1 \vee x_2$ and $\overline{x}_1 \vee x_2$, and $x_2 \vee x_3$, $x_1 \vee \overline{x}_2 \vee x_3$ and $\overline{x}_1 \vee x_2 \vee \overline{x}_3$ from $x_1 \vee x_3$ and $\overline{x}_1 \vee x_2$.

In the sequel, we use the following more compact representation of the rule [34]:

$$\begin{array}{l} x \vee A \\ \overline{x} \vee B \\ \hline\hline A \vee B \\ x \vee A \vee \overline{B} \\ \overline{x} \vee \overline{A} \vee B \end{array}$$

where $A = a_1 \vee \cdots \vee a_s$ and $B = b_1 \vee \cdots \vee b_t$ are disjunctions of literals. Note that $\overline{A}$ and $\overline{B}$ are not in clausal form [42]. In MaxSAT, the clausal forms of $\overline{A}$ and $\overline{B}$ are $\overline{A} = \{\overline{a_1}, a_1 \vee \overline{a_2}, \cdots, a_1 \vee \cdots \vee a_{s-1} \vee \overline{a_s}\}$ and $\overline{B} = \{\overline{b_1}, b_1 \vee \overline{b_2}, \cdots, b_1 \vee \cdots \vee b_{t-1} \vee \overline{b_t}\}$.

Note that an instance of MaxSAT resolution not only depends on the two premises and the cut variable. It also depends on the order of the literals in the premises. Moreover, this rule concludes one new clause not containing the variable x, except when this clause is a tautology.

The completeness of the calculus relies on the following notion of saturation: A multiset of clauses $\mathcal{C}$ is saturated w.r.t. variable x if, for every pair of clauses $c_1 = x \vee A$ and $c_2 = \overline{x} \vee B$ of $\mathcal{C}$, there is a literal l such that l is in A and $\overline{l}$ is in B. A multiset of clauses $\mathcal{C}'$ is a *saturation of* $\mathcal{C}$ w.r.t. x if $\mathcal{C}'$ is saturated w.r.t. x and $\mathcal{C}'$ can be obtained from $\mathcal{C}$ by applying the inference rule resolving x finitely many times.

The application of MaxSAT resolution to two clauses of the form $x \vee A$ and $\overline{x} \vee B$ of a multiset of clauses $\mathcal{C}$ that is saturated w.r.t. x only introduces compensation clauses because $A \vee B$ is a tautology. Note that the saturation of a multiset is not unique. For instance, the multiset $\{x_1,\ \overline{x}_1 \vee x_2,\ \overline{x}_1 \vee x_3\}$ has two possibles saturations w.r.t. variable x_1: the multiset $\{x_2,\ \overline{x}_2 \vee x_3,\ x_1 \vee \overline{x}_2 \vee \overline{x}_3,\ \overline{x}_1 \vee x_2 \vee x_3\}$, and the multiset $\{x_3,\ x_2 \vee \overline{x}_3,\ x_1 \vee \overline{x}_2 \vee \overline{x}_3,\ \overline{x}_1 \vee x_2 \vee x_3\}$. Nevertheless, completeness is independent of the saturation selected.

Bonet et al. [18,19] proved the completeness of MaxSAT resolution and proposed an exact variable elimination algorithm for MaxSAT. Figure 1 shows the pseudo-code of the algorithm: Given an input multiset of clauses C with n different variables, the algorithm returns the minimum number m of clauses of C that can be unsatisfied, and an optimal MaxSAT assignment I. Function $saturation(C, x)$ computes a saturation of C w.r.t. x. Function $partition(C, x_i)$ partitions C into two multiset, C_i and D_i so that C_i contains the clauses without occurrences of variable x_i, and D_i contains the clauses with occurrences of x_i. Function $max\_extension(x_i, I, D_i)$ computes a truth assignment for x_i as follows: if I satisfies all the clauses in D_i, including the case in which $D_i = \{\}$, then the function returns false (x_i is set to false, but if x_i is set to true, it also works); otherwise, either all the clauses of the form $x_i \vee A$ are satisfied or all the clauses of the form $\overline{x}_i \vee B$ are satisfied. In this case, x_i is set in such a way that all the clauses in D_i become satisfied.

```
**input:** C
$C_0 := C$
**for** $i := 1$ **to** $n$
    $C' := saturation(C_{i-1}, x_i)$
    $\langle C_i, D_i \rangle := partition(C', x_i)$
**endfor**
$m := |C_n|$
$I := \emptyset$
**for** $i := n$ **downto** 1
    $I := I \cup [x_i \mapsto max\_extension(x_i, I, D_i)]$
**output:** $m, I$
```

Fig. 1. An exact variable elimination algorithm for MaxSAT

The algorithm has two parts. In the first part, the algorithm successively saturates w.r.t. all the variables occurring in the input multiset. Once the current multiset is saturated w.r.t. a variable x_i, it partitions the resulting multiset into two multisets: C_i and D_i. The multiset C_i contains the clauses without occurrences of x_i, and the multiset D_i contains the clauses with occurrences of x_i. The algorithm continues saturating C_i w.r.t. one of the remaining variables, and ignores D_i. This process continues until all the variables are eliminated. At the end, C_n does not contain any variable, and the number of empty clauses in C_n is the returned minimum number of unsatisfied clauses. In the second part, the algorithm builds an optimal assignment as function $max\_extension(x_i, I, D_i)$ states.

Example 1. Let $\phi = \{\overline{x}_1, x_1 \vee x_2, x_1 \vee x_3, \overline{x}_3\}$ be a multiset of clauses. Resolving the first two clauses, we get $\{x_2, \overline{x}_1 \vee \overline{x}_2, x_1 \vee x_3, \overline{x}_3\}$. Resolving the second and third clause, we get a saturation of ϕ w.r.t. x_1: $\{x_2, \overline{x}_2 \vee x_3, \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3, \overline{x}_3\}$. Hence, $C_1 = \{x_2, \overline{x}_2 \vee x_3, \overline{x}_3\}$ and $D_1 = \{\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3\}$. Resolving the first two clauses of C_1, we get $\{x_3, x_2 \vee \overline{x}_3, \overline{x}_3\}$, which is a saturation of C_1 w.r.t. x_2. Hence, $C_2 = \{x_3, \overline{x}_3\}$ and $D_2 = \{x_2 \vee \overline{x}_3\}$. Resolving $\{x_3, \overline{x}_3\}$, we get the empty clause. Hence, $C_3 = \{\Box\}$ and $D_3 = \{\}$. So, the minimum number of unsatisfied clauses is 1, and $x_3 \mapsto false, x_2 \mapsto false, x_1 \mapsto true$ is an optimal assignment.

The idea behind the previous variable elimination algorithm is the following: Each clause $x \vee D$ in a saturation w.r.t. a variable x fulfills that, for each clause $\overline{x} \vee D'$, there is a literal l in D' such that $\bar{l}$ in D; and each clause $\overline{x} \vee D'$ fulfills that, for each clause $x \vee D$, there is a literal l in D such that $\bar{l}$ in D'. Thus, any assignment of a saturation w.r.t. x can be transformed into an assignment that satisfies all the clauses containing x and $\overline{x}$ if we assign correctly the value of x. In the case of MaxSAT, we saturate w.r.t. a variable x and then remove the clauses containing x and $\overline{x}$, because we know that any truth assignment that does not satisfy the clauses containing x and $\overline{x}$ will satisfy such clauses by flipping the value of x. In particular, any optimal MaxSAT assignment of the clauses not containing x and $\overline{x}$ can become an optimal MaxSAT assignment of the whole saturation w.r.t. x by assigning correctly the value of x. Because of that, after saturating w.r.t. a variable x, all the clauses containing variable x can be ignored. At the end, we obtain a multiset containing only empty clauses, and we have that the union of all the ignored clauses is satisfiable.

The proof complexity of resolution has been deeply investigated. For example, we know that there exists no polynomial-size resolution proof of the pigeon hole principle (PHP) [28]. Nevertheless, Ignatiev et al. [31] showed that there exist polynomial-size MaxSAT resolution proofs of PHP if PHP is encoded as a Partial MaxSAT instance using the dual rail encoding. Indeed, the combination of the dual rail encoding and MaxSAT resolution is a stronger proof system than either general resolution or conflict-driven clause learning [16]. More recently, it has been shown that MaxSAT resolution, when combined with certain rules, also produces polynomial-size MaxSAT resolution proofs of PHP. For example, MaxSAT resolution with the split rule (replace clause C with $\overline{x} \vee C$ and $x \vee C$) produces polynomial-size proofs of PHP, and this does not happen if MaxSAT resolution is replaced with resolution [17,35]. Interestingly, the combination of MaxSAT resolution with the split rule has also been used to create MaxSAT proofs from SAT refutations [57]. The challenge is to apply all these results to solve SAT more efficiently using MaxSAT approaches [7].

MaxSAT resolution has been extended to the multiple-valued clausal forms known as signed CNF formulas [14]. The defined signed MaxSAT resolution rules are complete and provide a logical framework for weighted constraint satisfaction problems (WCSP) [4]. Besides, some restrictions of the rules enforce the defined local consistency properties for WCSPs in a natural way [3,5]. A complete MinSAT resolution calculus for signed CNF formulas was defined in [45].

From a practical point of view, restrictions of MaxSAT resolution are routinely used in BnB MaxSAT and MinSAT solvers like ahmaxsat [1], akmaxsat [33], MaxSatz [40], MiniMaxSat [30] and MinSatz [48,49] to simplify the formulas at each node of the search tree and derive empty clauses. The application of MaxSAT resolution in BnB MaxSAT solvers is commonly combined with a lower bounding procedure that detects disjoint inconsistent subsets with unit propagation [38,39]. Narodytska and Bacchus [55] used MaxSAT resolution in SAT-based MaxSAT solvers, avoiding the use of cardinality constraints and obtaining very competitive results on industrial instances.

For ease of presentation, we have presented above the resolution rule for the unweighted case. The weighted version of the MaxSAT resolution rule is as follows:

$$\begin{array}{l} (x \vee A, u) \\ (\overline{x} \vee B, w) \\ \hline\hline (A \vee B, \min(u,w)) \\ (x \vee A \vee \overline{B}, \min(u,w)) \\ (\overline{x} \vee \overline{A} \vee B, \min(u,w)) \\ (x \vee A \vee \overline{B}, u - \min(u,w)) \\ (\overline{x} \vee \overline{A} \vee B, w - \min(u,w)) \end{array}$$

Conclusions with weight zero are omitted. A clause with weight w is equivalent to having w copies of that clause, and the weighted rule collapses $\min(u, w)$ applications of the unweighted rule.

In the partial case, we consider that hard clauses have weight $\top$ and apply the following rules:

$$\begin{array}{l} (x \vee A, u) \\ (\overline{x} \vee B, \top) \\ \hline\hline (A \vee B, u) \\ (x \vee A \vee \overline{B}, u) \\ (\overline{x} \vee B, \top) \end{array} \qquad \begin{array}{l} (x \vee A, \top) \\ (\overline{x} \vee B, w) \\ \hline\hline (A \vee B, w) \\ (x \vee A, \top) \\ (x \vee \overline{A} \vee B, w) \end{array} \qquad \begin{array}{l} (x \vee A, \top) \\ (\overline{x} \vee B, \top) \\ \hline\hline (A \vee B, \top) \\ (x \vee A, \top) \\ (\overline{x} \vee B, \top) \end{array}$$

Note that the derivation of an empty clause from two hard unit clauses corresponds to an unfeasible solution. This corresponds to the derivation of $(\square, \top)$, which is commonly represented by $\blacksquare$.

3.2 Resolution for MinSAT

MaxSAT resolution is sound for MinSAT because it preserves the number of unsatisfied clauses between the premises and the conclusions. Nevertheless, the elimination of variables must be defined differently to get completeness [37]. After saturating a MinSAT instance w.r.t. a variable x_i, we partition the saturation into two multisets as in MaxSAT: the multiset C_i of clauses without occurrences of x_i, and the multiset D_i of clauses with occurrences of x_i. In the next step, we consider the MinSAT instance ϕ' formed by both C_i and the multiset of clauses, say F_i, resulting of removing all the occurrences of the literals x_i and $\overline{x}_i$ in D_i; i.e., $F_i := \{A \mid x_i \vee A \in D_i\} \cup \{B \mid \overline{x}_i \vee B \in D_i\}$. It was proved in [37] that the MinSAT problem for ϕ can be reduced to the MinSAT problem for $\phi' = C_i \cup F_i$. In this way, after saturating all the variables, the number of empty clauses derived is equal to the maximum number of clauses that can be unsatisfied.

```
input: C
for $i := 1$ to $n$
    $C := saturation(C, x_i)$
    $\langle C_i, D_i \rangle := partition(C, x_i)$
    $F_i := \{A \,|\, x \vee A \in D_i\} \cup \{B \,|\, \overline{x} \vee B \in D_i\}$
    $C := C_i \cup F_i$
endfor
$m := |C|$
$I := \emptyset$
for $i := n$ downto $1$
    $I := I \cup [x_i \mapsto min\_extension(x_i, I, D_i)]$
output: $m, I$
```

Fig. 2. An exact variable elimination algorithm for MinSAT

Figure 2 shows the pseudo-code of the exact variable elimination algorithm for MinSAT proposed in [37]. Given an input multiset of clauses C with n different variables, the algorithm returns the maximum number m of clauses of C that can be unsatisfied, and an optimal MinSAT assignment I. Functions $saturation(C, x)$ and $partition(C, x_i)$ are defined as in MaxSAT. Function $min\_extension(x_i, I, D_i)$ computes a truth assignment for x_i as follows: if I unsatisfies a clause of D_i by setting x_i to false, then the function returns false; otherwise it returns true.

There are two crucial differences between the proposed MaxSAT and MinSAT algorithms: The first one is that, after saturating w.r.t. the variable under consideration, the algorithm continues saturating using the multiset $C_i \cup F_i$ instead of the multiset C_i. The second one is the way of computing an optimal assignment.

Example 2. We consider again the multiset of clauses $\phi = \{\overline{x}_1, x_1 \vee x_2, x_1 \vee x_3, \overline{x}_3\}$ of Example 1. Resolving the first two clauses, we get $\{x_2, \overline{x}_1 \vee \overline{x}_2, x_1 \vee x_3, \overline{x}_3\}$. Resolving the second and third clause, we get a saturation of ϕ w.r.t. x_1: $\{x_2, \overline{x}_2 \vee x_3, \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3, \overline{x}_3\}$. Hence, $C_1 = \{x_2, \overline{x}_2 \vee x_3, \overline{x}_3\}$, $D_1 = \{\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3\}$, and $F_1 = \{\overline{x}_2 \vee \overline{x}_3, x_2 \vee x_3\}$. So, the problem reduces to find the maximum number of unsatisfied clauses in $C_1 \cup F_1 = \{x_2, \overline{x}_2 \vee x_3, \overline{x}_2 \vee \overline{x}_3, x_2 \vee x_3, \overline{x}_3\}$.

We now resolve the second and fourth clause, and get $\{x_2, x_3, \overline{x}_2 \vee \overline{x}_3, \overline{x}_3\}$. We resolve the first and third clause, and get a saturation of $C_1 \cup F_1$ w.r.t. x_2: $\{\overline{x}_3, x_2 \vee x_3, x_3, \overline{x}_3\}$. Hence, $C_2 = \{\overline{x}_3, x_3, \overline{x}_3\}$, $D_2 = \{x_2 \vee x_3\}$, and $F_2 = \{x_3\}$ So, the problem reduces to find the maximum number of unsatisfied clauses in $C_2 \cup F_2 = \{\overline{x}_3, x_3, \overline{x}_3, x_3\}$.

Resolving the two first and the two last clauses of $\{\overline{x}_3, x_3, \overline{x}_3, x_3\}$, we get two empty clauses. Hence, $C_3 = \{\square, \square\}$, $D_3 = \{\}$, and $F_3 = \{\}$. So, the maximum number of unsatisfied clauses is 2, and $x_3 \mapsto true, x_2 \mapsto true, x_1 \mapsto true$ is an optimal assignment.

Table 1. α-formulas and β-formulas.

α	α_1	α_2
$A \wedge B$	A	B
$\overline{(A \vee B)}$	$\overline{A}$	$\overline{B}$
$\overline{(A \to B)}$	A	$\overline{B}$

β	β_1	β_2
$A \vee B$	A	B
$\overline{(A \wedge B)}$	$\overline{A}$	$\overline{B}$
$A \to B$	$\overline{A}$	B

4 Tableau-Style Calculi for MaxSAT and MinSAT

This section first introduces basic concepts of tableaux for SAT, then presents the tableau-style calculi for clausal and non-clausal MaxSAT defined so far and, finally, describes how MaxSAT tableaux can be used to solve MinSAT.

4.1 Tableau Calculi for SAT

In the uniform notation, all propositional formulas of the form $(A \circ B)$ and $\overline{(A \circ B)}$, where A and B denote propositional formulas and $\circ \in \{\vee, \wedge, \to\}$, are grouped into two categories so that the presentation and proofs are simplified: Those that act conjunctively, which are called α-formulas, and those that act disjunctively, which are called β-formulas. The different formulas in each category are displayed in Table 1. To complete a taxonomy of propositional formulas, excluding literals, we also need the propositional formulas of the form $\overline{\overline{A}}$.

Note that α is logically equivalent to $\alpha_1 \wedge \alpha_2$, β is logically equivalent to $\beta_1 \vee \beta_2$ and $\overline{\overline{A}}$ is logically equivalent to A. In SAT tableaux, these equivalences are used to reduce the problem of finding a satisfying assignment of α to that of finding a satisfying assignment of both α_1 and α_2, of β to that of finding a satisfying assignment of β_1 or β_2 and of $\overline{\overline{A}}$ to that of finding a satisfying assignment of A. Thus, using the expansion rules of Table 2 we obtain a complete tableau calculus for non-clausal SAT. We introduced the contradiction rule ($\Box$-rule), where l denotes a literal, because it will be necessary in MaxSAT; in the literature, applying this rule is usually referred to as closing the branch. Note that the uniform notation allows one to concisely define tableau rules for arbitrary propositional formulas.

The tableau method is used to determine the satisfiability of a given set of propositional formulas [22,27,60]. It starts creating an initial tableau composed of a single branch that has a node for each formula in the input set of formulas. Then, it applies the expansion rules of Table 2 until a contradiction is derived in each branch (in this case, the input set of formulas is unsatisfiable) or a branch is saturated without deriving a contradiction (in this case, the input set of formulas is satisfiable). A branch is saturated in a SAT tableau when all the possible applications of the expansion rules have been applied in that branch.

Table 2. Tableau expansion rules for SAT

$$\begin{array}{cccc} \dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} & \dfrac{\beta}{\beta_1 \mid \beta_2} & \dfrac{\overline{\overline{A}}}{A} & \dfrac{\begin{array}{c} l\\ \bar{l}\end{array}}{\Box} \\ \alpha\text{-rule} & \beta\text{-rule} & \neg\text{-rule} & \Box\text{-rule}\end{array}$$

4.2 Tableau Calculi for Clausal MaxSAT

The first tableau calculus for MaxSAT defined in the literature [41] limits the input to multisets of clauses; i.e., it is a clausal MaxSAT tableau calculus. Thus, it does not contain the α- and $\neg$-rule. It consists of the β- and $\Box$-rule. In fact, as all the formulas in the tableau are clauses and the formulas of type β are always disjunctions of literals of the form $l_1 \vee l_2 \vee \cdots \vee l_n$, the previous β-rule is replaced with the following n-ary β-rule:

$$\begin{array}{c} \dfrac{l_1 \vee l_2 \vee \cdots \vee l_n}{l_1 \mid l_2 \mid \cdots \mid l_n} \\ n\text{-ary } \beta\text{-rule} \end{array}$$

Note that the n-ary β-rule collapses $n-1$ applications of the β-rule over the clause $l_1 \vee l_2 \vee \cdots \vee l_n$.

The expansion rules are applied differently in MaxSAT: Firstly, the application of expansion rules in a branch cannot stop once a contradiction is detected. Since the aim of MaxSAT is to derive all the possible contradictions, the application of rules in a branch must continue until no more expansion rules can be applied. Thus, we will say that a tableau is completed when all the branches are saturated. Secondly, the application of rules in SAT leads to accumulate the newly added unit clauses in the branch in such a way that satisfiability is preserved in at least one branch when the input set of clauses is satisfiable. However, the addition of unit clauses can lead to wrong MaxSAT solutions. In clause MaxSAT tableaux, the goal should be to keep the minimum number of unsatisfied clauses in at least one branch and not to decrease that number in the rest of branches. To this end, we must maintain active and inactive clauses in a branch. Once a clause has been used as a premise of a rule in a branch, it cannot be used again in that branch and becomes inactive. For example, thanks to distinguishing between active and inactive clauses, we will detect one contradiction in the multiset of unit clauses $\{x_1, \neg x_1, \neg x_1\}$ and two in $\{x_1, x_1, \neg x_1, \neg x_1\}$. Without that, we could detect two contradictions in the first case, obtaining a wrong answer. Indeed, the inference rules for MaxSAT can be seen as rewriting rules.

Figure 3 shows the differences between clausal SAT and clausal MaxSAT tableaux using the multiset of clauses is $\phi = \{\overline{x}_1, \overline{x}_2, \overline{x}_3, x_1 \vee x_2, \overline{x}_1 \vee x_3\}$. In the SAT case, it is enough with applying the β-rule to $x_1 \vee x_2$. Since a contradiction is detected in each branch, the input multiset of formulas is declared unsatisfiable. However, in the

MaxSAT case, the β-rule must also be applied to $\overline{x}_1 \vee x_3$ and all the possible contradictions must be detected to complete the tableau. Note that in the leftmost branch of the clausal MaxSAT tableau there is just one contradiction because we have just one occurrence of x_1, which became inactive after detecting the first contradiction. In other words, every literal can only be used once to detect a contradiction.

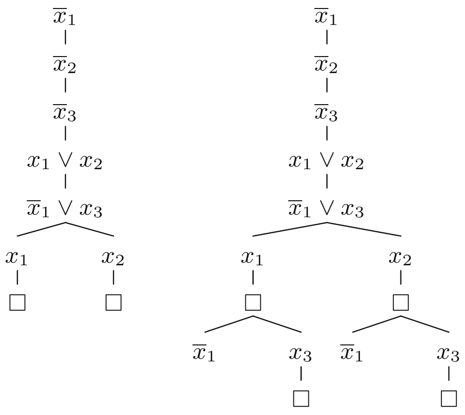


Fig. 3. Completed clausal SAT tableau (left) and completed clausal MaxSAT tableau (right) when the input multiset of clauses is $\phi = \{\overline{x}_1, \overline{x}_2, \overline{x}_3, x_1 \vee x_2, \overline{x}_1 \vee x_3\}$. The left tableau proves that ϕ is unsatisfiable and the right tableau proves that the minimum number of unsatisfied clauses in ϕ is 1.

The soundness of the previous clausal MaxSAT tableau calculus states that the β- and $\Box$-rule preserve the minimum number of unsatisfied clauses between a tableau and its extension; in particular, the β-rule preserves that number in at least one branch and does not decrease it in the rest of branches. So, once all branches have been saturated, the minimum number of contradictions derived among the branches of a completed tableau is the minimum number of unsatisfied clauses in the input multiset of clauses. The completeness states that any completed tableau for a multiset of clauses ϕ, whose minimum number of clauses that can be unsatisfied in it is k, has a branch with k contradictions and the rest of branches contain at least k contradictions [41]. Thus, in Fig. 3, the right tableau proves that the minimum number of unsatisfied clauses in ϕ is 1. This tableau calculus inspired the creation of a complete natural deduction calculus for clausal MaxSAT [21].

4.3 Tableau Calculi for Non-clausal MaxSAT

If we move to deal with arbitrary propositional formulas (i.e., non-clausal MaxSAT), the first drawback is that the α-rule does not preserve the minimum number of unsatisfied formulas as the β-rule does for clauses. Assume that we want to solve the non-clausal MaxSAT instance $\{x_1, x_2, \overline{x}_1 \wedge \overline{x}_2\}$, whose single optimal assignment is the one that sets x_1 and x_2 to true and only unsatisfies $\overline{x}_1 \wedge \overline{x}_2$. If we apply the α-rule to

$\overline{x}_1 \wedge \overline{x}_2$, we add two nodes, labelled with $\overline{x}_1$ and $\overline{x}_2$, to the initial tableau. Then, we can derive two contradictions by applying the $\Box$-rule to $\{x_1, \overline{x}_1\}$ and $\{x_2, \overline{x}_2\}$, but the minimum number of formulas unsatisfied by the optimal assignment is just one. This counterexample shows that the α-rule is unsound in MaxSAT. So, we need to define a new and sound α-rule as a first step towards getting a sound and complete non-clausal MaxSAT calculus.

Table 3. Tableau expansion rules for non-clausal MaxSAT

$$\begin{array}{cccc}
\begin{array}{c|c} \multicolumn{2}{c}{\alpha} \\ \hline \Box & \alpha_1 \\ & \alpha_2 \end{array}
&
\begin{array}{c|c} \multicolumn{2}{c}{\beta} \\ \hline \beta_1 & \beta_2 \end{array}
&
\begin{array}{c} \overline{\overline{A}} \\ \hline A \end{array}
&
\begin{array}{c} l \\ \bar{l} \\ \hline \Box \end{array}
\\
\alpha\text{-rule} & \beta\text{-rule} & \neg\text{-rule} & \Box\text{-rule}
\end{array}$$

The first defined complete non-clausal MaxSAT tableau calculus is formed by the expansion rules in Table 3 [43]. Note that all the rules preserve the number of premises unsatisfied by an assignment I in at least one branch and do not decrease that number in the other branch (if any). In particular, in the α-rule, we have that if I unsatisfies α, the left branch contains one contradiction and α_1 and α_2 cannot be used to derive any other contradiction in that branch because they are not expanded; moreover, I unsatisfies α_1 or α_2 (or both) on the right branch. On the other hand, if I satisfies α, then I also satisfies α_1 and α_2 on the right branch. A similar non-clausal MaxSAT tableau calculus was later proposed by Fiorino [24].

Example 3. We can determine the minimum number of unsatisfied formulas in the multiset $\phi = \{x_1, x_2, \overline{x}_1 \wedge \overline{x}_2\}$ using the previous tableau calculus. Figure 4 displays how the tableau is constructed. We start by constructing the initial tableau (the leftmost tableau) and then apply the α-rule to $\overline{x}_1 \wedge \overline{x}_2$, getting as a result the second tableau in the figure. The leftmost branch is saturated and we apply the $\Box$-rule to $\{x_1, \overline{x}_1\}$ on the rightmost branch, getting as a result the third tableau. Finally, we apply the $\Box$-rule to $\{x_2, \overline{x}_2\}$ on the same branch and get the rightmost tableau in the figure. Since the minimum number of boxes among the branches of the last tableau is 1, the minimum number of formulas that can be unsatisfied in ϕ is 1.

Table 4 displays the expansion rules for weighted partial MaxSAT formulas. In all the rules, the premises with weight $\top$ remain active after the application of a rule because hard formulas must be satisfied by any optimal assignment. For hard clauses, the α-, β- and $\neg$-rule are the same as in SAT. The $\Box$-rule derives $\blacksquare$ when a contradiction is derived from two hard formulas; in this case, an unfeasible solution has been detected and the search is stopped in the current branch. For weighted soft clauses, the α-, β- and $\neg$-rule have just one premise and the weight associated with the premise is transferred to the conclusions. If the $\Box$-rule has two weighted soft premises, the contradiction takes as weight the minimum of the weights associated with the premises $(\min(w_1, w_2))$. If the $\Box$-rule has a soft premise (A, w) and a hard premise $(\overline{A}, \top)$, then

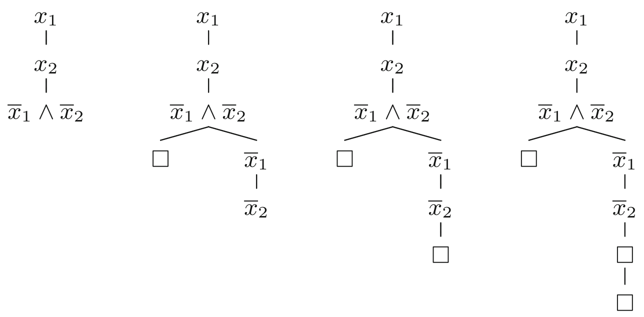


Fig. 4. A tableaux for the non-clausal MaxSAT instance $\{x_1, x_2, \overline{x}_1 \wedge \overline{x}_2\}$.

Table 4. Tableau expansion rules for weighted partial MaxSAT

$$\frac{(\alpha,\top)}{\begin{array}{c}(\alpha_1,\top)\\(\alpha_2,\top)\end{array}} \qquad \frac{(\alpha,w)}{\begin{array}{c|c}(\Box,w) & (\alpha_1,w)\\ & (\alpha_2,w)\end{array}} \qquad\qquad \frac{(\beta,\top)}{\begin{array}{c|c}(\beta_1,\top) & (\beta_2,\top)\end{array}} \qquad \frac{(\beta,w)}{\begin{array}{c|c}(\beta_1,w) & (\beta_2,w)\end{array}}$$

α-rule $\qquad\qquad$ β-rule

$$\frac{(\overline{\overline{A}},\top)}{(A,\top)} \qquad \frac{(\overline{\overline{A}},w)}{(A,w)} \qquad\qquad \frac{\begin{array}{c}(A,\top)\\(\overline{A},\top)\end{array}}{\blacksquare} \qquad \frac{\begin{array}{c}(A,\top)\\(\overline{A},w)\end{array}}{\begin{array}{c}(\Box,w)\\(A,\top)\end{array}} \qquad \frac{\begin{array}{c}(A,w_1)\\(\overline{A},w_2)\end{array}}{\begin{array}{c}(\Box,\min(w_1,w_2))\\(A,w_1-\min(w_1,w_2))\\(\overline{A},w_2-\min(w_1,w_2))\end{array}}$$

$\neg$-rule $\qquad\qquad$ $\Box$-rule

$(\Box, w)$ is derived, (A, w) becomes inactive and $(\overline{A}, \top)$ remains active. In the weighted case, when a branch has repeated occurrences of a formula A, say $(A, w_1), \ldots, (A, w_s)$, such occurrences can be replaced with the single formula $(A, w_1 + \cdots + w_s)$. Moreover, the cost of a saturated weighted branch is the sum of weights of the boxes that appear in the branch, and the cost of a completed weighted tableau is the minimum cost among all its branches.

Example 4. Let $\phi = \mathcal{H} \cup \mathcal{S}$ be a non-clausal partial MaxSAT instance, where $\mathcal{H}$ is the multiset of hard formulas and $\mathcal{S}$ is the multiset of soft formulas. Given the multiset of propositional formulas $\{x_1 \wedge x_2 \wedge x_3, \overline{x}_1, \overline{x}_2, \overline{x}_3\}$, we analyze the different tableaux obtained when we vary the formulas declared as hard and soft.

The first tableau of Fig. 5 displays a completed tableau when all the formulas are soft; in this case $\phi = \mathcal{H} \cup \mathcal{S} = \emptyset \cup \{x_1 \wedge x_2 \wedge x_3, \overline{x}_1, \overline{x}_2, \overline{x}_3\}$.

The second tableau displays a completed tableau when $x_1 \wedge x_2 \wedge x_3$ is hard and the rest of formulas are soft; in this case $\phi = \mathcal{H} \cup \mathcal{S} = \{x_1 \wedge x_2 \wedge x_3\} \cup \{\overline{x}_1, \overline{x}_2, \overline{x}_3\}$. We applied the α-rule of Table 2 because the premise is hard.

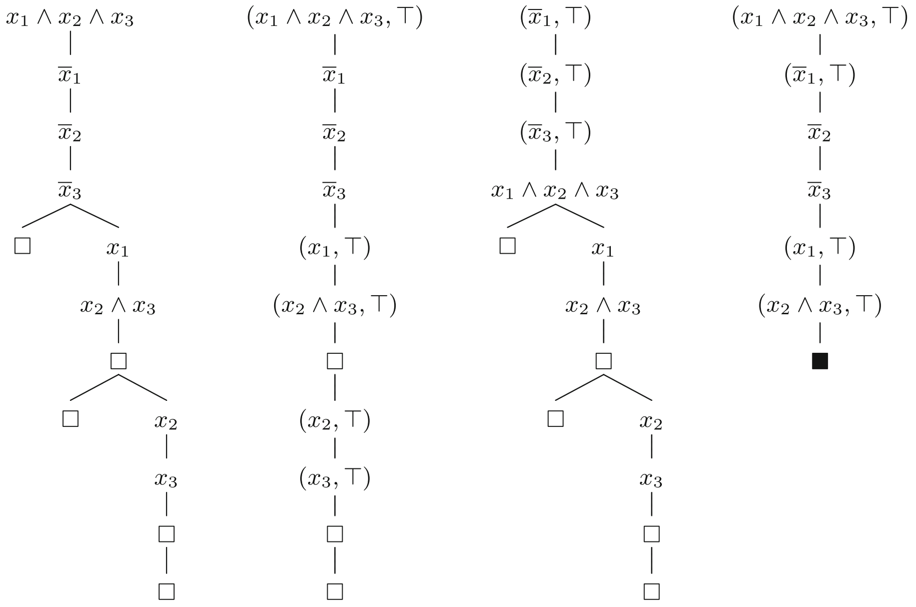


Fig. 5. Examples of non-clausal partial MaxSAT tableaux.

The third tableau displays a completed tableau when $\overline{x}_1$, $\overline{x}_2$ and $\overline{x}_3$ are hard, and $x_1 \land x_2 \land x_3$ is soft; in this case $\phi = \mathcal{H} \cup \mathcal{S} = \{\overline{x}_1, \overline{x}_2, \overline{x}_3\} \cup \{x_1 \land x_2 \land x_3\}$. We applied the α-rule of Table 3 because the premise is soft.

The fourth tableau displays a completed tableau when $x_1 \land x_2 \land x_3$ and $\overline{x}_1$ are hard, and $\overline{x}_2$ and $\overline{x}_3$ are soft; in this case $\phi = \mathcal{H} \cup \mathcal{S} = \{x_1 \land x_2 \land x_3, \overline{x}_1\} \cup \{\overline{x}_2, \overline{x}_3\}$. Note that the single branch of the tableau is pruned as soon as the $\square$-rule has two hard premises ($\overline{x}_1$ and x_1). We use a filled box to denote that there is no feasible solution.

In the first case, the minimum number of unsatisfied soft formulas is 1. In the second case, the minimum number of unsatisfied soft formulas among the assignments that satisfy the hard formulas is 3. In the third case, the minimum number of unsatisfied soft formulas among the assignments that satisfy the hard formulas is 1. In the fourth case, there is no optimal solution because the subset of hard formulas is unsatisfiable.

Example 5. Let $\phi = \{(x_1 \land x_3, \top), (\overline{x}_1 \rightarrow x_2, 3), (\overline{x}_1, 5), (\overline{x}_2, 1), (\overline{x}_3, 2)\}$ be a non-clausal weighted partial MaxSAT instance, where the first formula is hard and the rest of formulas are soft. Figure 6 displays a completed tableau for ϕ. This tableau has been obtained by applying the expansion rules for non-clausal weighted partial MaxSAT explained above. The cost of the left branch is 10 and the cost of the right branch is 8. Thus, the minimum sum of weights of unsatisfied soft formulas among the assignments that satisfy the hard formula is 8.

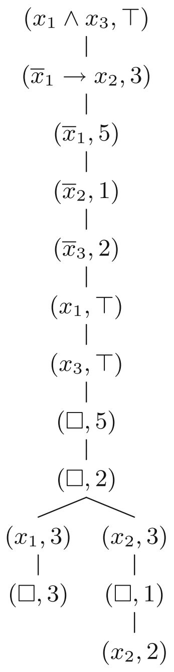


Fig. 6. Example of non-clausal weighted partial MaxSAT tableau.

4.4 Tableau Calculi for MinSAT

Non-clausal MaxSAT tableaux can be used to solve MinSAT taking into account the following property: Any optimal MaxSAT assignment of the multiset of propositional formulas $\phi = \{\overline{\phi}_1, \ldots, \overline{\phi}_m\}$ is an optimal MinSAT assignment of the multiset $\phi' = \{\phi_1, \ldots, \phi_m\}$. Indeed, if an optimal MaxSAT assignment I of $\phi = \{\overline{\phi}_1, \ldots, \overline{\phi}_m\}$ unsatisfies k formulas, then I is an optimal MinSAT assignment of $\phi' = \{\phi_1, \ldots, \phi_m\}$ that satisfies k formulas. Thus, to find an optimal MinSAT solution of $\phi' = \{\phi_1, \ldots, \phi_m\}$, we must build a completed non-clausal MaxSAT tableau of $\phi = \{\overline{\phi}_1, \ldots, \overline{\phi}_m\}$. If the cost of the non-clausal MaxSAT tableau is k, the maximum number of formulas that can be unsatisfied in ϕ' is $m - k$.

Example 6. Let $\phi = \{\overline{x}_1 \wedge \overline{x}_2, \overline{x}_3, x_1 \vee x_2 \vee x_3\}$ be a non-clausal MinSAT instance. Figure 7 displays a completed non-clausal MaxSAT tableau for $\{x_1 \vee x_2, x_3, \overline{(x_1 \vee x_2 \vee x_3)}\}$. Since the cost of the non-clausal MaxSAT tableau is 1, the maximum number of clauses that can be unsatisfied in ϕ is $3 - 1 = 2$.

Fiorino [25] defined a non-clausal MinSAT tableau calculus that derives the minimum number of satisfied formulas instead of the maximum number of unsatisfied formulas. Interestingly, his α-rule for MinSAT is similar to our β-rule for MaxSAT and his β-rule for MinSAT is similar to our α-rule for MaxSAT.

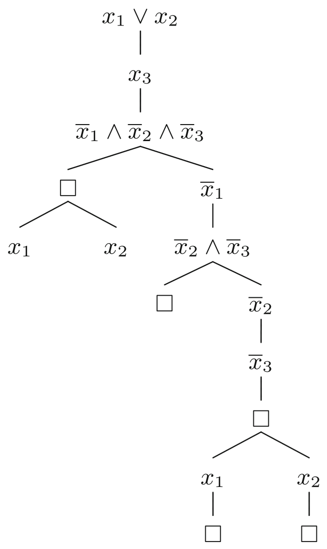


Fig. 7. A tableau for the non-clausal MinSAT instance $\phi = \{\overline{x}_1 \wedge \overline{x}_2, \overline{x}_3, x_1 \vee x_2 \vee x_3\}$.

5 Conclusions

We presented an overview of the existing complete calculi for MaxSAT and MinSAT, as well as for their variants with hard and (weighted) soft clauses. All these results allow us to better understand the logic behind MaxSAT and MinSAT, and provide a new approach to investigating these problems from a different perspective. The proof complexity results achieved suggest that MaxSAT and MinSAT calculi might be useful to create a SAT solver that exploits a proof system stronger than resolution.

A reviewer pointed out a curious reduction from the decision version of MaxSAT to the SAT problem of the Łukasiewicz infinite-valued logic [53]. This reduction opens the door to solve MaxSAT with solvers for the Łukasiewicz infinite-valued logic based on resolution and tableau methods such as [54,56,61].

Other future lines of work are the extension to first-order logic of the described calculi, and the implementation and empirical comparison of the described inference methods to assess their performance in solving specific combinatorial optimization problems.

Acknowledgments. We thank the reviewers for their valuable comments and suggestions, which greatly improved this manuscript. We also want to congratulate Reiner Hähnle on his 60th birthday, as well as thank Bernhard Beckert, Einar Broch Johnsen, Richard Bubel, and Wolfgang Ahrendt for organizing a symposium and Festschrift in honor of Reiner Hähnle.

References

1. Abramé, A., Habet, D.: Ahmaxsat: description and evaluation of a branch and bound Max-SAT solver. J. Satisfiability Boolean Modeling Comput. **9**, 89–128 (2014)
2. Abramé, A., Habet, D.: Local search algorithm for the partial minimum satisfiability problem. In: Proceedings of the 27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI, Vietri sul Mare, Italy, pp. 821–827 (2015)
3. Ansótegui, C., Bonet, M.L., Levy, J., Manyà, F.: Inference rules for high-order consistency in weighted CSP. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, Vancouver, Canada, pp. 167–172 (2007)
4. Ansótegui, C., Bonet, M.L., Levy, J., Manyà, F.: The logic behind weighted CSP. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI-2007, Hyderabad, India, pp. 32–37 (2007)
5. Ansótegui, C., Bonet, M.L., Levy, J., Manyà, F.: Resolution procedures for multiple-valued optimization. Inf. Sci. **227**, 43–59 (2013)
6. Ansótegui, C., Izquierdo, I., Manyà, F., Jiménez, J.T.: A Max-SAT-based approach to constructing optimal covering arrays. In: Proceedings of the 16th International Conference of the Catalan Association for Artificial Intelligence, CCIA 2013, Vic, Spain. Frontiers in Artificial Intelligence and Applications, vol. 256, pp. 51–59. IOS Press (2013)
7. Ansótegui, C., Levy, J.: Reducing SAT to Max2SAT. In: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-2021, Montreal, Canada, pp. 193–198 (2021)
8. Ansótegui, C., Li, C.M., Manyà, F., Zhu, Z.: A SAT-based approach to MinSAT. In: Proceedings of the 15th International Conference of the Catalan Association for Artificial Intelligence, CCIA-2012, Alacant, Spain, pp. 185–189. IOS Press (2012)
9. Ansótegui, C., Manyà, F., Ojeda, J., Salvia, J.M., Torres, E.: Incomplete MaxSAT approaches for combinatorial testing. J. Heuristics (2022, in press)
10. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The first and second Max-SAT evaluations. J. Satisfiability Boolean Modeling Comput. **4**(2–4), 251–278 (2008)
11. Argelich, J., Li, C.M., Manyà, F., Soler, J.R.: Clause tableaux for maximum and minimum satisfiability. Logic J. IGPL **29**(1), 7–27 (2021)
12. Bacchus, F., Berg, J., Järvisalo, M., Martins, R.: MaxSAT Evaluation 2020: Solver and Benchmark Descriptions. University of Helsinki, Department of Computer Science (2020)
13. Bacchus, F., Järvisalo, M., Ruben, M.: Maximum satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 929–991. IOS Press, Amsterdam (2021)
14. Beckert, B., Hähnle, R., Manyà, F.: The SAT problem of signed CNF formulas. In: Basin, D., D'Agostino, M., Gabbay, D., Matthews, S., Viganò, L. (eds.) Labelled Deduction. Applied Logic Series, vol. 17, pp. 61–82. Kluwer, Dordrecht (2000)
15. Bofill, M., Garcia, M., Suy, J., Villaret, M.: MaxSAT-based scheduling of B2B meetings. In: Proceedings of the12th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR, Barcelona, Spain, pp. 65–73 (2015)
16. Bonet, M.L., Buss, S., Ignatiev, A., Marques-Silva, J., Morgado, A.: MaxSAT resolution with the dual rail encoding. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI, New Orleans, Louisiana, USA, pp. 6565–6572 (2018)
17. Bonet, M.L., Levy, J.: Equivalence between systems stronger than resolution. In: Pulina, L., Seidl, M. (eds.) SAT 2020. LNCS, vol. 12178, pp. 166–181. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_13
18. Bonet, M.L., Levy, J., Manyà, F.: A complete calculus for Max-SAT. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 240–251. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_24

19. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. Artif. Intell. **171**(8–9), 240–251 (2007)
20. Cai, S., Lei, Z.: Old techniques in new ways: clause weighting, unit propagation and hybridization for maximum satisfiability. Artif. Intell. **287**, 103354 (2020)
21. Casas-Roma, J., Huertas, A., Manyà, F.: Solving MaxSAT with natural deduction. In: Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Spain. Frontiers in Artificial Intelligence and Applications, vol. 300, pp. 186–195. IOS Press (2017)
22. D'Agostino, M.: Tableaux methods for classical propositional logic. In: D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 45–123. Kluwer (1999)
23. D'Almeida, D., Grégoire, É.: Model-based diagnosis with default information implemented through MAX-SAT technology. In: Proceedings of the IEEE 13th International Conference on Information Reuse & Integration, IRI, Las Vegas, NV, USA, pp. 33–36 (2012)
24. Fiorino, G.: New tableau characterizations for non-clausal MaxSAT problem. Logic J. IGPL (2021). https://doi.org/10.1093/jigpal/jzab012
25. Fiorino, G.: A non-clausal tableau calculus for MinSAT. Inf. Process. Lett. **173**, 106167 (2022)
26. Guerra, J., Lynce, I.: Reasoning over biological networks using maximum satisfiability. In: Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP, Québec City, QC, Canada, pp. 941–956 (2012)
27. Hähnle, R.: Tableaux and related methods. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 100–178. Elsevier and MIT Press (2001)
28. Haken, A.: The intractability of resolution. Theoret. Comput. Sci. **39**, 297–308 (1985)
29. Heras, F., Larrosa, J.: New inference rules for efficient Max-SAT solving. In: Proceedings of the National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA, pp. 68–73 (2006)
30. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSAT: an efficient weighted Max-SAT solver. J. Artif. Intell. Res. **31**, 1–32 (2008)
31. Ignatiev, A., Morgado, A., Marques-Silva, J.: On tackling the limits of resolution in SAT solving. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 164–183. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_11
32. Jabbour, S., Mhadhbi, N., Raddaoui, B., Sais, L.: A SAT-based framework for overlapping community detection in networks. In: Proceedings of the 21st Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Part II, PAKDD, Jeju, South Korea, pp. 786–798 (2017)
33. Kuegel, A.: Improved exact solver for the weighted MAX-SAT problem. In: Proceedings of Workshop Pragmatics of SAT, POS-10, Edinburgh, UK, pp. 15–27 (2010)
34. Larrosa, J., Heras, F.: Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-2005, Edinburgh, Scotland, pp. 193–198. Morgan Kaufmann (2005)
35. Larrosa, J., Rollon, E.: Towards a better understanding of (partial weighted) MaxSAT proof systems. In: Pulina, L., Seidl, M. (eds.) SAT 2020. LNCS, vol. 12178, pp. 218–232. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_16
36. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 903–927. IOS Press (2021)
37. Li, C.M., Manyà, F.: An exact inference scheme for MinSAT. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI-2015, Buenos Aires, Argentina, pp. 1959–1965 (2015)

38. Li, C.M., Manyà, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound max-SAT solvers. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 403–414. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_31
39. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: Proceedings of the 21st National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA, pp. 86–91 (2006)
40. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. J. Artif. Intell. Res. **30**, 321–359 (2007)
41. Li, C.M., Manyà, F., Soler, J.R.: A clause tableaux calculus for MaxSAT. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI-2016, New York, USA, pp. 766–772 (2016)
42. Li, C.M., Manyà, F., Soler, J.R.: Clausal form transformation in MaxSAT. In: Proceedings of the 49th IEEE International Symposium on Multiple-Valued Logic, ISMVL, Fredericton, Canada, pp. 132–137 (2019)
43. Li, C.M., Manyà, F., Soler, J.R.: A tableau calculus for non-clausal maximum satisfiability. In: Cerrito, S., Popescu, A. (eds.) TABLEAUX 2019. LNCS (LNAI), vol. 11714, pp. 58–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29026-9_4
44. Li, C.M., Manyà, F., Soler, J.R., Vidal, A.: From non-clausal to clausal MinSAT. In: Proceedings of the 23rd International Conference of the Catalan Association for Artificial Intelligence, CCIA, Lleida, Spain, pp. 27–36. IOS Press (2021)
45. Li, C.M., Xiao, F., Manyà, F.: A resolution calculus for MinSAT. Logic J. IGPL **29**(1), 28–44 (2021)
46. Li, C.-M., Zhenxing, X., Coll, J., Manyà, F., Habet, D., He, K.: Boosting branch-and-bound MaxSAT solvers with clause learning. AI Commun. (2021). https://doi.org/10.3233/AIC-210178
47. Li, C.-M., Xu, Z., Coll, J., Manyà, F., Habet, D., He, K.: Combining clause learning and branch and bound for MaxSAT. In: Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming, CP, Montpellier, France. LIPIcs, vol. 210, pp. 38:1–38:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
48. Li, C.M., Zhu, Z., Manyà, F., Simon, L.: Minimum satisfiability and its applications. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI-2011, Barcelona, Spain, pp. 605–610 (2011)
49. Li, C.M., Zhu, Z., Manyà, F., Simon, L.: Optimizing with minimum satisfiability. Artif. Intell. **190**, 32–44 (2012)
50. Manyà, F., Negrete, S., Roig, C., Soler, J.R.: A MaxSAT-based approach to the team composition problem in a classroom. In: Sukthankar, G., Rodriguez-Aguilar, J.A. (eds.) AAMAS 2017. LNCS (LNAI), vol. 10643, pp. 164–173. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71679-4_11
51. Manyà, F., Negrete, S., Roig, C., Soler, J.R.: Solving the team composition problem in a classroom. Fundamamenta Informaticae **174**(1), 83–101 (2020)
52. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Ann. Math. Artif. Intell. **62**(3–4), 317–343 (2011)
53. Mundici, D.: Ulam game, the logic of MaxSAT, and many-valued partitions. In: Prade, H., Dubois, D., Klement, E.P. (eds.) Logics and Reasoning about Knowledge, pp. 121–137. Kluwer (1999)
54. Mundici, D., Olivetti, N.: Resolution and model building in the infinitely-valued calculus of Łukasiewicz. Theoret. Comput. Sci. **200**(1–2), 335–366 (1998)
55. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided MaxSAT resolution. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence, Québec City, Canada, pp. 2717–2723 (2014)

56. Olivetti, N.: Tableaux for Łukasiewicz infinite-valued logic. Stud. Logica. **73**(1), 81–111 (2003)
57. Py, M., Cherif, M.S., Habet, D.: A proof builder for max-SAT. In: Li, C.-M., Manyà, F. (eds.) SAT 2021. LNCS, vol. 12831, pp. 488–498. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_33
58. Robinson, J.A.: A machine oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
59. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, Texas, USA, pp. 13–19 (2007)
60. Smullyan, R.: First-Order Logic. Dover Publications, New York, second corrected edition (1995). First published 1968 by Springer-Verlag
61. Warner, H.: A new resolution calculus for the infinite-valued propositional logic of Łukasiewicz. In: Proceedings of the International Workshop on First order Theorem Proving, pp. 234–243 (1998)
62. Xu, H., Rutenbar, R.A., Sakallah, K.A.: sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing. IEEE Trans. CAD Integr. Circuits Syst. **22**(6), 814–820 (2003)
63. Zhang, L., Bacchus, F.: MAXSAT heuristics for cost optimal planning. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence, Toronto, Ontario, Canada, pp. 1846–1852 (2012)

Implications of Deductive Verification on Research Quality
Field Study

Wojciech Mostowski(✉)

Computing and Electronics for Real-Time Embedded Systems, School of Information Technology, Halmstad University, Halmstad, Sweden
wojciech.mostowski@hh.se

Abstract. This short paper discusses a handful of perhaps obvious, but important observations about KeY, the state-of-the-art deductive verification tool for Java programs. Two light research ideas surface out during the admittedly divergent discussion, both of which seem to be little explored, at least in the given context. Not all projects survive for as long as KeY does, it takes a good idea and dedicated people for that to happen. Hence, the paper also contributes with a formally proved correspondence between using KeY and being a good researcher. Apart from that, considering the occasion to which this paper is dedicated, a handful of memories about Prof. Hähnle are also shared.

1 Introduction

When we were invited to contribute to this *Lecture Notes in Computer Science* volume, there was absolutely no hesitation as to accept it or not. The problem of what to contribute with did, however, produce itself immediately. A requirement that it should be technical enough was given, but a relatively lightweight nature was also considered for an angle. Namely, it should be something to read over a glass of (obviously not any!) wine rather than in the office, library, or in front of a whiteboard with a marker in hand. Still, an urge to wake up[1] the computer and check the validity of presented propositions should be triggered too.

As one of the meeting points for us with Prof. Hähnle is the KeY verification system, so it made perfect sense to begin there and put KeY on the stage. Since the amount of research and technical contributions directly involving KeY is overwhelming[2] [2], we focus on more high-level insights. Namely, what makes KeY so special in our humble opinion and what are some implications of that.

In this paper we use the common method of plural personification of thoughts and claims, it should be noted, however, that these are mostly the author's

[1] The fact that computers are hardly ever turned off these days, and hence the phrase "turn on the computer" by now seems obsolete, would be one other observation indicating the passing time.

[2] The temptation to cite estimated 100 or more papers at this point was very strong, but a decision was made to stick with the most prominent one.

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 370–381, 2022.
https://doi.org/10.1007/978-3-031-08166-8_17

personal experiences, opinions, and observations, so we do not impose that all readers, normally implicitly included in the plural first person style, should agree with these.

2 The KeY System

For the unfamiliar reader, let us quickly recap what KeY is. A more detailed history of KeY, its assumptions and design principles are accounted for in [9]. From the technical standpoint, a deductive verifier for Java programs specified with Java Modelling Language (JML) [2, Chap. 7] would be one characterisation of KeY. An interactive First-Order Logic and Dynamic Logic theorem prover would be another. A general software verification platform particularly suitable for scientific experimentation would be yet another. Indeed, the core design principle of KeY is a powerful, user friendly, interactive theorem prover that explicitly works with the program source code through proof trees and visually presented logic sequents. The underlying logic, Dynamic Logic for Java [2, Chap. 3] that inherently includes plain First-Order Logic [2, Chap. 2], targets the verification of a large subset of Java programs. Only a small part of this logic (essentially its signature) is hardwired into the tool, while the actual semantics is defined externally through user alterable logic rules called taclets [2, Chap. 4]. This particular feature allows for the mentioned experimentation with a wide degree of freedom[3]. Supporting only a certain subset of Java is more of a necessity dictated by the open-language character of Java, namely, most limitations come from the Java reflection mechanism and associated concepts. Such limitations are common to all verification tools, however, dialects of Java that are fully verifiable by KeY do exist [2, Chap. 10]. Most other Java features are supported to one degree or another, sometimes by using a suitable workaround[4].

3 The Specialised Research Tool

The first observation about KeY is that it is somewhat of a specialised tool by design. At the beginning times of KeY, building a verification tool from scratch rather than adapting an existing general purpose theorem prover [8] triggered scepticism in the community and was probably considered counterproductive. Indeed, building something dedicated and specialised might be a little against the scientific way of finding or reusing well established, generic solutions, and it is also time consuming. Yes, it is a fact that now more than 20 years after the first lines of code for KeY have been written the system is still going through active development. But by now, KeY should be considered a generic and reusable solution, as other projects were successfully built on it in the meantime [4].

[3] The careful reader should immediately suspect the fact, that because of this "freedom" it is also relatively easy to introduce unsoundness at user will. Fortunately, the tool provides means to detect this too, but unsoundness issues may go unnoticed with careless use.

[4] See e.g. https://git.key-project.org/key-public/key/-/tree/stable/Stubby.

As a side personal note, it is important to say that there is nothing wrong with developing specialised tools. We occasionally repair cars in our free time. Figure 1 shows a picture of pliers to release or fasten a certain kind of a reusable hose clamp (on the side of the figure) used by some car makers. With this tool, each operation on the clamp is a couple of seconds, totally effortless procedure. Without it, using generic pliers and a screwdriver, each clamp takes several minutes (if one is lucky and acquainted with the procedure) to treat, it typically results in damaged clamps, cut fingers, and other inconveniences. And, the shown pliers do not serve any other purpose whatsoever.

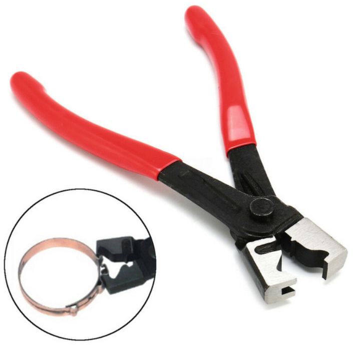


Fig. 1. So called Click-R pliers, picture found on the Wish website.

In retrospect, for KeY, we think the culprit was not in whether a general purpose theorem prover should be used or a dedicated one designed from scratch. The challenge and difficulty lied in formalising the details of the semantics of the programming language in question, and transforming these details into a working tool, either by writing a complete set of suitable theories for a general purpose theorem prover, or designing a suitable prover to begin with. We actually see a strong resemblance in the challenges between developing a verification framework of any kind to the process of developing a faithful emulator for a computer system. Not looking too far and just sticking to entertainment[5], the RetroPie project[6] struggles to emulate as many as possible legacy entertainment systems, like arcade cabinets or home gaming consoles of the past century, on a modern, but very affordable hardware. As in program verification, also here some details of truly faithful emulation have to be traded off for efficiency in most part, but also usability and maintainability of the underlying software. The core of the resemblance between these two seemingly very distant worlds, is that in verification one develops a reasoning system *embracing* another computing system, in emulation one develops an implementation embracing another implementation.

[5] Which for many is or was the first contact with computing machinery!
[6] See https://retropie.org.uk.

4 New Directions for KeY?

For us, this analogy and seeing where the emulation technology currently heads (see down below) has actually sparked some ideas closer to research besides spending time playing old computer games. We always had certain interest in emulation technology, but this was never seriously pursued. But, close to this, reverse engineering automotive embedded systems is an actual on-going side interest, with very satisfying results[7]. The typical approach to reverse engineering such systems is using a disassembler and reconstructing the meaning of binary code, in a manual, very slow, and very tedious process. In our opinion, this might have been the actual reason why Diesel-gate[8] managed to stay conceived for relatively long time. Typically, the most interesting part of this process are the actual calculation formulae used by the system, i.e., how exactly (not only if) the outputs depend on the inputs. The other interesting parts are the dependency and sequencing of procedures and events. Apart from viewing the source code, static analysis, and symbolic execution in particular, can provide similar answers, i.e., *symbolic outputs.* While typically applied to source code, nothing prevents one to apply symbolic execution on the binary machine code. Why not do it during the actual execution, as a side task? Obviously, it would not be exhaustive, which is the whole point of static analysis, but by providing sufficiently rich set of inputs, one could get quite high coverage of symbolic execution traces this way. In fact, in reverse engineering one is not always concerned with all possible scenarios, but the most common ones, or alternatively, some very specific ones, neither of which require exhaustive analysis. There is an obvious resemblance to concolic testing [10] or proof based test generation [2, Chap. 12] in this idea, however, the direction is the total opposite. In these existing methods, the results of symbolic execution or proving are used to select inputs to control actual execution or testing, while in our idea symbolic execution is done in parallel to actual execution. As a result one would get, apart from concrete program outputs, symbolic execution traces formalising the output based on the given inputs, a very useful artefact for reversing engineers. One other view of this would be execution-based de-compilation working on similar principles as automata learning [6].

As embedded systems in their production form are not really that penetrable to apply a technique like this directly, one would have to rely on an adequate emulator complemented with a suitable symbolic execution engine. Lifting this idea up to Java, however, should be more straightforward to realise. Namely, imagine an instrumented Java Virtual Machine that aside from executing bytecode keeps track of symbolic execution traces. There are obviously a lot of technical details and limitations that would have to be solved on the way, nevertheless, the open character of Java and its tooling should make this perfectly achievable.

[7] Check this for a sample: http://nefariousmotorsports.com/forum/index.php?topic=14417.0.
[8] See https://lwn.net/Articles/670488/.

For the other, perhaps more worthwhile, research idea, let us take a step back to the retro gaming again. In a project like the mentioned RetroPie, all emulations are based in software, and by clever programming a lot of hardware details of the emulated systems can be abstracted away, or in any other way solved with tricks. For example, faithful emulation of a sound chip (even an old one), is not that trivial. But, if one emulates a single game implemented on an arcade cabinet hardware, and the amount of sounds generated by that game is strictly bounded and actually small, why not pre-record these sounds on the original hardware and replay them during simulation by simply detecting that the original program attempts to play the given sound by trivial pattern matching on the executing code? In fact, this is exactly how earlier versions of the M.A.M.E. emulator[9] solved the problem for some systems before the emulator reached sophistication required to actually emulate the according sound hardware faithfully (enough).

The existing knowledge base of old systems and huge number of enthusiasts brought the number of emulated systems to thousands allowing one to essentially replay ones whole young-hood. However, despite all those tricks and shortcuts one can take when emulating hardware in software, software emulation inevitably has its limitations. Each step of the way one approximation or another has to be made or precise timing sacrificed due to synchronization with the host system and its devices. The lack of real-time guarantees of the off-the-shelf host operating system adds to this. In the end, all this results in visible imperfections, only giving the feeling of the real thing rather than the real thing.[10]

There is a solution to this though, and one that has reached maturity in recent years. Rather than to emulate, with the advent of high-performance Field Programmable Gate Array (FPGA) boards it is now possible to *replicate* the old hardware. For retro gaming, the MiSTer FPGA project[11] brought this to a smaller, but still quite large and comparable scale as RetroPie did for software emulations. This approach is praised by many as the “real thing”, even though the word *replicate* used earlier is also a bit on the too strong side. Practically, however, many issues caused by software emulation are eliminated. Such systems are effectively real-time, clock precise, internally synchronised as the original hardware, and totally immune to any unwanted interactions from the host operating system.

One may ask what does all this have to do with program verification. Well, it is a well-known fact that hardware solutions can be much more efficient than software ones, and can have a whole different actual complexity class when some application bounds can be established and some inputs are pre-compiled into hardware [11]. Hence, the obvious question is why not employ FPGAs for program verification? For generic SAT solving, the idea is certainly not new [1] and very much alive with promising results [3,14]. Given the availability and low

[9] See https://www.mamedev.org.

[10] Individuals investing serious resources in faithful music reproduction can certainly relate here.

[11] See https://github.com/MiSTer-devel/Main_MiSTer/wiki.

price of very accessible FPGA boards one wonders why at least some program verifiers are not equipped with hardware compilers to solve the difficult subproblems in hardware? That would probably be another example of a highly specialised tool, but as we noted earlier, specialised tools are invaluable and can go a very long way in some cases.

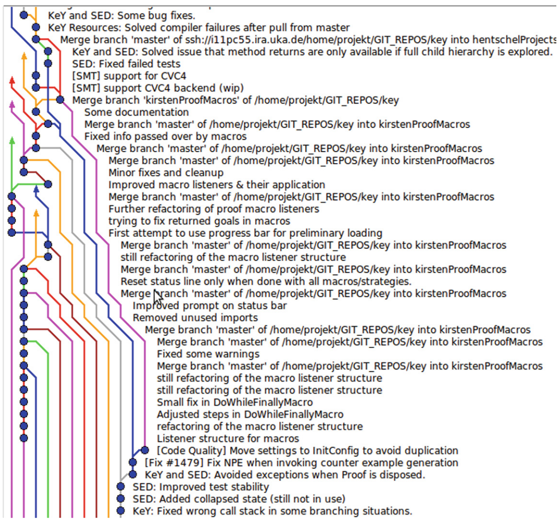
5 Actual Industrial Grade Software

KeY in itself is a complex software project. The implementation of KeY[12] is full of good programming practices, including very non-trivial applications of the object-oriented design patterns. The maintenance of the KeY software base has been a time consuming task, including the lengthy process of choosing the right version control system in the past. Today it may seem astonishing that this was an actual non-research related, problematic choice that had to be made and one could wonder why it was something that had to be discussed at length.[13] Nevertheless, the fact is, that among the other choices that were ever made for KeY, choosing GIT (which was in its infancy at the time) for version control was one of the very right ones, so the discussion time was well spent. To this day, we use a snapshot of the KeY GIT branching history in one of our lectures on object-oriented programming as an example of (brutal) reality of software development, see Fig. 2.

For us, one implication of working with KeY as a piece of quite complex, yet very good software is that after a while all other software validation tools and methods seem somewhat inadequate and one cannot really spark a proper interest in them, a kind of feeling "it is nice and all, but it is never going to be as good and cool as KeY is". This is very subjective, we admit, and despite the fact that some of the other techniques, contrary to KeY, are much easier applicable to industrial software and are not limited to a subset of one programming language. Notable examples here are all kinds of Model-Based Testing [12] tools that simply interact with an actual (not an abstraction or any other abbreviated representation) system being validated through the external interface that it provides. As with deductive verification, there are cases of these methods discovering subtle bugs [5,13]. Nevertheless, being in general black-box techniques, they deprive the user of the *close contact* with the validated program, which gives the feeling of actually understanding the process and the experience "Not only I verified some properties, I also looked under the hood of the process and I properly witnessed it." For some people looking under the hood is an itch that has to be scratched, and probably has something to do with limited trust. The limited trust, we believe, is one of defining characteristics of a good researcher.

[12] See https://git.key-project.org/key-public/key/-/tree/stable.

[13] The older developers of KeY would confirm that this was a very much heated discussion.



```
KeY and SED: Some bug fixes.
KeY Resources: Solved compiler failures after pull from master
Merge branch 'master' of ssh://i11pc55.ira.uka.de/home/projekt/GIT_REPOS/key into hentschelProjects
KeY and SED: Solved issue that method returns are only available if full child hierarchy is explored.
SED: Fixed failed tests
[SMT] support for CVC4
[SMT] support CVC4 backend (wip)
Merge branch 'kirstenProofMacros' of /home/projekt/GIT_REPOS/key
Some documentation
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Fixed info passed over by macros
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Minor fixes and cleanup
Improved macro listeners & their application
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Further refactoring of proof macro listeners
trying to fix returned goals in macros
First attempt to use progress bar for preliminary loading
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
still refactoring of the macro listener structure
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Reset status line only when done with all macros/strategies.
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Improved prompt on status bar
Removed unused imports
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
Fixed some warnings
Merge branch 'master' of /home/projekt/GIT_REPOS/key into kirstenProofMacros
still refactoring of the macro listener structure
still refactoring of the macro listener structure
Small fix in DoWhileFinallyMacro
Adjusted steps in DoWhileFinallyMacro
refactoring of the macro listener structure
Listener structure for macros
[Code Quality] Move settings to InitConfig to avoid duplication
[Fix #1479] Fix NPE when invoking counter example generation
KeY and SED: Avoided exceptions when Proof is disposed.
SED: Improved test stability
SED: Added collapsed state (still not in use)
KeY: Fixed wrong call stack in some branching situations.
```

Fig. 2. A historic snapshot of KeY GIT side branch strucutre.

6 Formal Relation Between KeY and Good Research

The last paragraph above actually suggests the following logical consequence. Namely, that KeY allows one to be a good researcher! In fact, let us turn a bit technical, formalise this (in KeY, what else) and prove it. Since this is not provable in the general case (it would be too simple of a recipe for good research), we limit ourselves to a small case study with two persons of different nationalities. Thus, we need the corresponding sorts, then the said two persons `R` and `W`, and their nationalities:

```
\sorts {
    Person;
    Nationality;
}

\functions {
    Person R;
    Person W;
    Nationality German;
    Nationality Polish;
}
```

We then need a couple of predicates binding persons to their nationalities and expressing some traits of character and habits, as follows:

```
\predicates {
    nationality(Person, Nationality);
    organised(Person);
```

```
        limitedTrust(Person);
        goodResearcher(Person);
        usesKeY(Person);
        usesDeductiveVerification(Person);
    }
```

The first predicate should be self explanatory, the second one establishes if the given person is a well organised one. The third predicate tells us if the given person shows signs of having limited trust towards the surrounding world, and the fourth predicate is actually our target property stating that someone is in fact a good researcher. The fifth and sixth predicates establish if the given person uses KeY regularly, and if not, perhaps at least uses other non-KeY deductive verification technique in their work.

To continue with the formalisation we need some axiomatisation of this simplified reality. This is typically the moment in formal verification where everything can go wrong and absurd facts could be derived, but let us give it a try. First of, any person of German nationality should be perceived as organised:

```
    \forall Person p; (nationality(p, German) -> organised(p))
```

Then, any person that uses KeY also uses deductive verification by definition:

```
    \forall Person p;
        (usesKeY(p) -> usesDeductiveVerification(p))
```

Then, people working with deductive verification are the ones that have trust issues:

```
    \forall Person p;
        (usesDeductiveVerification(p) -> limitedTrust(p))
```

And, finally, for any person, trust issues coupled with being organised define a good researcher:

```
    \forall Person p;
        (limitedTrust(p) & organised(p) -> goodResearcher(p))
```

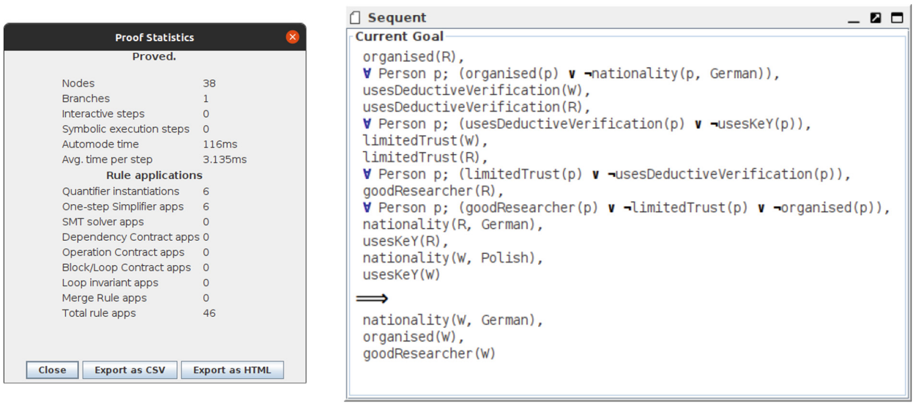
We can now ask the question if `R` is a good researcher, knowing some facts about `R`:

```
    \problem {
        nationality(R, German) & usesKeY(R) -> goodResearcher(R)
    }
```

Asking KeY gives a very quick proof that this is indeed the case, see the left side of Fig. 3. Asking if `W` is a good researcher under corresponding assumptions does not give a successful proof though. The missing required fact for `W` to be a good researcher, is that either `W` is German (which `W` is not) or that `W` can be proven to be organised, see the right side of Fig. 3. For `W` being Polish, this may not be a straightforward fact to prove,[14] so let us leave this part for future research.

Apart from showing the great versatility of KeY by essentially being developed on-spot, this simple example also underlines some well-known facts about

[14] In fact, the opposite may as well hold.



Proved.

Nodes	38
Branches	1
Interactive steps	0
Symbolic execution steps	0
Automode time	116ms
Avg. time per step	3.135ms
Rule applications	
Quantifier instantiations	6
One-step Simplifier apps	6
SMT solver apps	0
Dependency Contract apps	0
Operation Contract apps	0
Block/Loop Contract apps	0
Loop invariant apps	0
Merge Rule apps	0
Total rule apps	46

```
organised(R),
∀ Person p; (organised(p) ∨ ¬nationality(p, German)),
usesDeductiveVerification(W),
usesDeductiveVerification(R),
∀ Person p; (usesDeductiveVerification(p) ∨ ¬usesKeY(p)),
limitedTrust(W),
limitedTrust(R),
∀ Person p; (limitedTrust(p) ∨ ¬usesDeductiveVerification(p)),
goodResearcher(R),
∀ Person p; (goodResearcher(p) ∨ ¬limitedTrust(p) ∨ ¬organised(p)),
nationality(R, German),
usesKeY(R),
nationality(W, Polish),
usesKeY(W)
⟹
nationality(W, German),
organised(W),
goodResearcher(W)
```

Fig. 3. Main proof results.

logical inference. For one, provability very much depends on the axiomatisation, and the latter is the difficult part. The level of detail of the axiomatisation also plays a role. At the same time, too much detail, or too large of an axiom base, could easily lead to unsoundness issues. In our case, the axiom base could be, e.g., expanded and twisted sufficiently in a covert way to prove that all good researchers should use KeY, before anybody would notice:

```
\forall Person p; (goodResearcher(p) -> usesKeY(p))
```

In reality this is obviously not true, and it would be extremely controversial to even try to claim this.

7 The Educational Tool

The last, but certainly not least, thing that we want to express here about KeY is that it is a brilliant education tool and subject. The structure of how one layer of the KeY method and design builds on the other is very straightforward. This gives a very natural teaching sequence. Furthermore, very many concepts that build up KeY are generic (most notably the type system lattice, or the First Order Logic with its calculus) and are educationally very useful outside of the KeY context. These are topics that should be taught to every single computer science, or related subject, student. Moreover, the "open hood" character of the KeY method and design allows one to vividly demonstrate the workings of all the concepts, but without any additional hassle, all that can and should be shown to students is just there. This is certainly not the case with many other automated and opaque software validation techniques or tools. Perhaps a little bit of an inappropriate comparison, but it could be considered the same way as an open body dissection that medical students have to practice. With the main difference that in case of KeY it is actually not unpleasant.

Back in 2011, we had the chance to teach one of the earlier editions of the *Software Engineering using Formal Methods* course that Prof. Hähnle along with

Wolfgang Ahrendt and Richard Bubel developed at the time.[15] This was taught to a large group of Master students at the Chalmers University of Technology in Göteborg, Sweden. So it happens that this experience was one of the first larger steps to our current university teaching profession. Thus, the educational aspect of KeY is personally invaluable to us.

8 Informal Methods

And now for something completely different^{TM}, i.e., the less formal part of the paper. I always found it little bit difficult to spell, or perhaps just type in, the last name,[16] hence let us switch to Reiner. For transparency and better context for the unfamiliar readers, Reiner was my PhD supervisor at Chalmers University of Technology in Sweden between 2000 and 2005.

Reiner was the person that encouraged me to author papers on my own, obviously selectively and when appropriate. Only later I found out that this is not too common in PhD supervision. I counted 11 papers of this kind until now (this one is the twelfth), and interestingly only one of them does not involve KeY in any way. I think this is due to the fact that Reiner figured out early that I prefer to work on my own. I am not exactly sure of why I have this preference, but one of the reasons could be aversion to explaining my intermediate thoughts while I develop something into shape. For me, explaining comes once things are worked out to a sufficient degree (i.e., when they are effectively *finished*). Unfortunately, this "aversion" (Reiner once diplomatically described it as *healthy scepticism*) is also one of the reasons for having troubles in writing exciting project proposals. Nevertheless, we worked together well and I finished my PhD in good time, this certainly says something about Reiner and his judge of character.

For some reason, I always remember two specific events from my PhD studies at Chalmers involving Reiner, both having very little to do with the actual research. One of them is from the very beginning when Reiner walked into my office and said that we should start the whole supervision thing by having dinner at Stage Door (a pub in Göteborg). And I remember this because of his reply to my question about the time we should meet for that – "7 or later, one can't eat too early". For someone that just finished undergraduate studies in Poland with rather early days – classes from 8 a.m. to 3 p.m. straight with no lunch break and the main meal around 5 p.m. – pushing dinner towards what I considered almost night was somewhat unusual. Or perhaps the real puzzlement for me was that I did not really know what I was supposed to be doing in the office all the way until 7 p.m., not sure any more.

The other memory is from the Fall of 2004 when I still had almost a full year until the formal end of my studies. I was either in progress of finishing my "big" FASE paper [7], or I just submitted it. I bumped into Reiner in our top-floor

[15] There were surely also other teachers from the KeY group involved in this work, and the development continued later with many others involved, we have no means to name them all, unfortunately.

[16] The experience is surely mutual.

lunch room, we were both getting coffee or equivalent (tea most probably, I was off coffee at that time), and he said in passing "By the way, with this FASE paper out of the way, I think you should wrap it up." Then a short discussion of what exactly he meant by that followed. Long story short, one introduction chapter and 5 months later I had my PhD defence, and that was even before I got the chance to actually present the said FASE paper at the ETAPS conference two weeks later.

In conclusion, with this paper I would like to express my gratitude to Reiner for everything he has done for me and wish him a Very Happy 60th Birthday! And, I would also like to thank the editors for the invitation.

References

1. Abramovici, M., Sousa, J.: A SAT solver using reconfigurable hardware and virtual logic. J. Autom. Reasoning **24**(12), 5–36 (2000). https://doi.org/10.1023/A:1006310219368
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, vol. 10001, LNCS. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Bousmar, K., Monteiro, F., Habbas, Z., Dellagi, S., Dandache, A.: A pure hardware k-SAT solver architecture for FPGA based on generic tree-search. In: 2017 29th International Conference on Microelectronics, pp. 1–5, (2017)
4. Hähnle, R.: HATS: highly adaptable and trustworthy software using formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 3–8. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_2
5. Hughes, J., Pierce, B.C., Arts, T., Norell, U.: Mysteries of DropBox: property-based testing of a distributed synchronization service. In: 2016 IEEE International Conference on Software Testing, Verification and Validation, pp. 135–145. IEEE (2016)
6. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation learnlib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_18
7. Mostowski, W.: Formalisation and verification of Java Card security properties in dynamic logic. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 357–371. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_27
8. Nawaz, M.S., Malik, M., Li, Y., Sun, M., Lali, M.: A survey on theorem provers in formal methods. CoRR. https://arxiv.org/abs/1912.03028 (2019)
9. Schmitt, P.H.: A short history of KeY. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY, vol. 12345, LNCS, pp. 3–18. Springer International Publishing, Cham (2020) https://doi.org/10.1007/978-3-030-64354-6
10. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_38
11. Sourdis, I., Bispo, J., Cardoso, J.M.P., Vassiliadis, S.: Regular expression matching in reconfigurable hardware. J. Signal Process. Syst. **51**, 99–121 (2008). https://doi.org/10.1007/s11265-007-0131-0

12. Tretmans, J.: Model-based testing and some steps towards test-based modelling. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 297–326. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_9
13. Tretmans, J., van de Laar, P.: Model-based testing with torxakis: the mysteries of dropbox revisited. In: Strahonja, V. (ed.) 30th Central European Conference on Information and Intelligent Systems, October 2–4, 2019, pp. 247–258. Croatia, Varazdin (2019)
14. Yuan, Z., Ma, Y., Bian, J.: SMPP: generic SAT solver over reconfigurable hardware accelerator. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, pp. 443–448. IEEE (2012)

Computing in Łukasiewicz Logic and AF-Algebras

Daniele Mundici(✉)

Department of Mathematics and Computer Science "Ulisse Dini", University of Florence, Viale Morgagni 67/A, 50134 Florence, Italy
daniele.mundici@unifi.it

Abstract. Since the beginning of his career as a computer scientist, Reiner Hähnle has made important contributions to the proof theory of Łukasiewicz logic $Ł_\infty$, e.g., applying Mixed Integer Programming techniques to the satisfiability problem for the Łukasiewicz calculus. His work in this area culminated in a monograph on automated deduction in many-valued logic and other key contributions on this vibrating field of research. The present paper discusses recent developments in Łukasiewicz logic $Ł_\infty$, its associated algebraic semantics given by C.C.Chang MV-algebras, and related computational issues concerning the approximately finite-dimensional (AF) C*-algebras of quantum statistical mechanics.

Keywords: Łukasiewicz logic · MV-algebra · Γ functor · implication function · Elliott classification · C*-algebra · AFℓ-algebra · decision problem

1 Introduction

Since the beginning of his career as a computer scientist, Reiner has made important contributions to many-valued (notably Łukasiewicz) logic and its applications. His work in this area fits into the general context of nonclassical reasoning, discussed in his monograph [18] and handbook chapter [31]. While being a fragment of his copious production in computer science, Reiner's papers on many-valued logic deal with a variety of applications in automated deduction, Artificial Intelligence and proof theory. See e.g., [15–34].

The present paper in honor of his sixtieth birthday is a concise survey of the following recent developments of Łukasiewicz logic:

– The derivation of the Łukasiewicz axioms for infinite-valued logic $Ł_\infty$ from the *continuity* of any $[0, 1]$-valued implication operation defined on $[0, 1]^2$. See Theorem 1 in Sect. 2. By Theorem 5, $Ł_\infty$-formulas code continuous $[0, 1]$-valued random variables on any compact Hausdorff space—just as boolean formulas code (automatically continuous) $\{0, 1\}$-valued random variables on any totally disconnected compact Hausdorff space.

- The interpretation of Ł$_\infty$-formulas as codes for Murray-von Neumann equivalence classes of projections in any AF-algebra $\mathfrak{A}$ whose Grothendieck group $K_0(\mathfrak{A})$ is lattice ordered, AFℓ-algebra for short. This application of Elliott's classification is the subject matter of Sect. 3.
- Applications of the deductive machinery of Ł$_\infty$ to the algorithmic theory of AFℓ-algebras. See Sect. 4.

The first part of the next section is aimed at readers who may have come across nonclassical logics, but are not familiar with the most recent developments of Łukasiewicz infinite-valued logic and MV-algebras. The pace is slower than in later sections, which are written in the standard definition-theorem style, in line with Goethe's popular aphorism about mathematicians and the French.

Due to lack of space, proofs are omitted, but references are given where the interested reader can find all details.

2 A Characterization of Łukasiewicz Logic and Its Algebras

Let Q be an NP-problem. Cook and, independently, Levin constructed a polytime reduction of Q to the boolean satisfiability problem SAT. Since Q is arbitrary, this makes SAT an NP-complete problem, like Integer Programming, Knapsack, Traveling Salesman, Clique, and many others. Unlike all these problems, the boolean formulas describing the computation steps of a nondeterministic Turing machine T running on an instance x of Q speak our (mathematical) language. These formulas pertain to boolean "logic".

The question whether x belongs to Q has the two possible answers, *no* or *yes*. The random variable "T recognizes x as a member of Q" is a boolean function f of the elementary random variables "T is in state s at time t", "T is on square q at time t", and "the symbol a is printed on square q at time t". Then x belongs to Q iff 1 belongs to the range of f iff the Cook-Levin boolean formula coding f is satisfiable.

The set $\{no, yes\} = \{0,1\}$ of "truth values" comes equipped with a rich structure. The (truth-)*functionality* property of boolean logic states that the truth-value $v(\phi)$ assigned to a boolean formula ϕ only depends on the truth-values assigned to the immediate subformulas of ϕ. This gives syntax a key role in logic. Thus for instance, for any two boolean formulas ϕ and ψ, $v(\phi \to \psi) = 0$ iff $v(\phi) = 1$ and $v(\psi) = 0$. A moment's reflection then shows:

$$v(\phi \to \psi) = 1 \text{ iff } v(\phi) \leq v(\psi). \text{ Also, } v(\phi \to (\psi \to \rho)) = v(\psi \to (\phi \to \rho)). \quad (1)$$

Replacing $\{0,1\}$-valued by $[0,1]$-valued random variables, one may look for (propositional) "logics" whose truth-values lie in the unit real interval $[0,1]$. Probability does not help, because functionality fails: The probability of a disjunction of two events is the sum of the probabilities of the events—so long as the two events are incompatible. Consequently, syntax plays virtually no role in probability theory.

But what is a "logic"? Following (the essentials of) the Polish tradition, [49, Part C], let A be an algebra on a universe U (of *truth-values*) with finitely many operations and constants, coded by a set S of *symbols*. The constants of A are supposed to include a special (distinguished) element 1 for "true", sometimes also denoted $\top$. Given a set V of *variable* symbols, a nonambiguous *syntax* defines the algebra $F = F_{S,V}$ of formulas on S and V by a familiar inductive procedure. The *semantics* of the logic $L = L_{A,1}$ is next introduced by stipulating that a formula ϕ is a *tautology* if $v(\phi) = 1$ for every homomorphism v of the algebra F into the algebra A. Two formulas are *logically equivalent* if every homomorphism gives them the same value[1].

2.1 The Logic of a Continuous $[0, 1]$-Valued Implication

Having set up the general framework for our exploration of "logics", recalling (1), let $\rightsquigarrow$ be a $[0, 1]$-valued operation on $[0, 1]^2$ satisfying the following minimalist conditions for any reasonable "implication" operation on an ordered set of truth-values:

$$\text{(i) } x \rightsquigarrow y = 1 \text{ iff } x \leq y \qquad \text{and} \qquad \text{(ii) } x \rightsquigarrow (y \rightsquigarrow z) = y \rightsquigarrow (x \rightsquigarrow z) \tag{2}$$

for all $x, y, z \in [0, 1]$. Let us further assume that the function $\rightsquigarrow$ is continuous. This is to ensure that small errors or perturbations in the evaluation of the variable symbols have small repercussions on the truth-value of composite formulas[2]. Then our search for $[0, 1]$-valued logics is rewarded with the following uniqueness theorem:

Theorem 1. *With the stipulations (2) for the continuous map $\rightsquigarrow$, for all $x \in [0, 1]$ let $\neg x$ be shorthand for $x \rightsquigarrow 0$. Then*

(a) The algebra $W = W_{\rightsquigarrow} = ([0, 1], 1, \neg, \rightsquigarrow)$ satisfies the following equations:

$$\begin{aligned} 1 \rightsquigarrow x &= x \\ (x \rightsquigarrow y) \rightsquigarrow ((y \rightsquigarrow z) \rightsquigarrow (x \rightsquigarrow z)) &= 1 \\ (\neg x \rightsquigarrow \neg y) \rightsquigarrow (y \rightsquigarrow x) &= 1 \\ (x \rightsquigarrow y) \rightsquigarrow y &= (y \rightsquigarrow x) \rightsquigarrow x. \end{aligned}$$

For short, W is a Wajsberg algebra.

[1] The attentive reader will have noted the key role of (truth-)functionality in this definition, hinging upon the nonambiguity of the syntax of F.

[2] Once $\{0, 1\}$ is equipped with the discrete topology, boolean implication is trivially continuous.

(b) For all $x, y \in [0,1]$ let $x \oplus y$ be shorthand for $\neg x \rightsquigarrow y$. Then the algebra $A = A_{\rightsquigarrow} = ([0,1], 0, \neg, \oplus)$ satisfies the following equations:

$$\begin{aligned}
x \oplus (y \oplus z) &= (x \oplus y) \oplus z \\
x \oplus 0 &= x \\
x \oplus \neg 0 &= \neg 0 \\
\neg\neg x &= x \\
\neg(\neg x \oplus y) \oplus y &= \neg(\neg y \oplus x) \oplus x.
\end{aligned}$$

For short, A is an MV-algebra.

Proof. See [43, Theorem 2.3]. For Wajsberg algebras see [8, 4.2.1]. In [6, p.468] C.C. Chang gave a redundant list of equations for MV-algebras. His list was reduced to six equivalent equations in [8, p.7]. Then Kolařík [35] proved that the commutativity of $\oplus$ follows from the remaining five equations—the equations listed in (b). □

Remark 1. The defining equations of Wajsberg algebras in Theorem 1(a) are the counterpart of the time-honored Łukasiewicz axioms for infinite-valued logic Ł$_\infty$. See [8, 4.2]. Since the first three equations are satisfied by any implication, this theorem is to the effect that

$$\boxed{\text{the equation } (x \rightsquigarrow y) \rightsquigarrow y = (y \rightsquigarrow x) \rightsquigarrow x \;\text{ stands for the continuity of } \rightsquigarrow}$$

Corollary 1. ([6, 1.16–1.17]) *Adding to the five equations in Theorem 1(b) the idempotence equation $x \oplus x = x$ we recover an equational definition of boolean algebras.*

Definition 1. ([8]) The *standard* MV-algebra $[0,1] = ([0,1], 0, \neg, \oplus)$ is defined by: $\neg x = 1 - x$, $x \oplus y = \min(1, x+y)$, together with the derived constant $1 = \neg 0$ and the derived operation $x \odot y = \max(0, x+y-1) = \neg(\neg x \oplus \neg y)$.

From Theorem 1(b) one obtains:

Corollary 2. ([43] and references therein) *For $\rightsquigarrow$ a continuous function satisfying conditions (i)-(ii) in (2), let $A = A_{\rightsquigarrow} = ([0,1], 0, \neg, \oplus)$ be the MV-algebra of Theorem 1(b). Then A is uniquely isomorphic to the standard MV-algebra. The logic $L = L_{A,1}$ defined by A with the distinguished constant 1 has the same tautologies as Ł$_\infty$.*

Theorem 2. *If an equation holds in the standard MV-algebra then it holds in all MV-algebras.*

Proof. This is Chang's completeness theorem [7]. See [8, §2] for a self-contained proof. □

2.2 Ideals, Spectral Spaces, and Quotients

The following elementary notions are the special case for MV-algebras of general algebraic/topological concepts.

Definition 2. For all MV-algebras A, B and homomorphism $\epsilon\colon A \to B$, the *kernel* $\mathsf{ker}(\epsilon)$ is the set of elements $x \in A$ such that $\epsilon(x) = 0$. *Ideals* are kernels of homomorphisms. We denote by $\mu(A)$ the set of maximal ideals of A. An MV-algebra is *semisimple* if the intersection of all its maximal ideals is $\{0\}$.

The set $\mu(A)$ comes equipped with the *spectral (*also known as the *hull-kernel) topology.* A subbasis of open sets is given by letting a range over all elements of A and defining O_a as the set of maximal ideals $\mathfrak{m}$ of A such that $a \notin \mathfrak{m}$. We keep the notation $\mu(A)$ for the resulting topological space, and call it the *maximal spectral space of* A. When A is a boolean algebra the maximal spectral topology of A is the Stone topology of A.

The *radical* $\mathsf{Rad}(A)$ of A is the intersection of all maximal ideals of A.

Proposition 1. ([8, Proposition 1.2.10]) *For every MV-algebra A, the quotient map yields a homeomorphism of the maximal spectral space of A onto the maximal spectral space of the semisimple MV-algebra $A/\mathsf{Rad}(A)$.*

Theorem 3. ([41, §4]) *Let A be an MV-algebra. For any $\mathfrak{m} \in \mu(A)$ there is a unique pair $(\omega_{\mathfrak{m}}, \Omega_{\mathfrak{m}})$ with $\Omega_{\mathfrak{m}}$ an MV-subalgebra of the standard MV-algebra* $[0,1]$*, and $\omega_{\mathfrak{m}}$ an isomorphism of the quotient MV-algebra $A/\mathfrak{m}$ onto $\Omega_{\mathfrak{m}}$.*

As a consequence, for every $\mathfrak{m} \in \mu(A)$ the quotient MV-algebra $A/\mathfrak{m}$ may be identified with its uniquely isomorphic copy $\Omega_{\mathfrak{m}} \subseteq [0,1]$. This enables us to define the evaluation map $^*\colon A \to [0,1]^{\mu(A)}$ by the identification

$$a^*(\mathfrak{m}) = a/\mathfrak{m} \in [0,1], \text{ for every } a \in A \text{ and } \mathfrak{m} \in \mu(A). \tag{3}$$

Definition 3. ([8, p.66]) Let B an MV-algebra of $[0,1]$-valued functions on a set $X \neq \emptyset$, with the pointwise operations of the standard MV-algebra $[0,1]$. B is said to be *separating*, if for any $x \neq y \in X$ there is $f \in B$ such that $f(x) \neq f(y)$.

Theorem 4. ([41, §4]) *(i) For any semisimple MV-algebra A the evaluation map * in (3) is an isomorphism of A onto a separating MV-subalgebra of the MV-algebra $C(\mu(A), [0,1])$ of* $[0,1]$*-valued continuous functions on the maximal spectral space $\mu(A)$, with the pointwise operations of the standard MV-algebra* $[0,1]$.

(ii) Let $X \neq \emptyset$ be a nonempty compact Hausdorff space and B a separating subalgebra of the MV-algebra $C(X, [0,1])$. Then B is semisimple. Further, the map $\mathfrak{J}\colon x \in X \mapsto \mathfrak{J}(x) = \{f \in B \mid f(x) = 0\}$ is a homeomorphism of X onto the maximal spectral space $\mu(B)$. The inverse map $\mathcal{V} = \mathfrak{J}^{-1}$ sends each $\mathfrak{m} \in \mu(B)$ to the only element $\mathcal{V}(\mathfrak{m}) \in X$ of the set $\bigcap\{f^{-1}(0) \mid f \in \mathfrak{m}\}$.

(iii) As A ranges over all MV-algebras, the maximal spectral space $\mu(A)$ ranges over all nonempty compact Hausdorff spaces.

2.3 The Kroupa-Panti Theorem

Definition 4. ([41, and references therein]) For any elements x, y of an MV-algebra A, we write $x \odot y$ shorthand for $\neg(\neg x \oplus \neg y)$.

We let $\mathsf{hom}(A, [0,1])$ denote the set of homomorphisms of A into the standard MV-algebra.

A *state* of an MV-algebra A is a map $\sigma\colon A \to [0,1]$ with $\sigma(1) = 1$, such that $\sigma(x \oplus y) = \sigma(x) + \sigma(y)$ whenever $x \odot y = 0$. We let $\mathsf{S}(A)$ denote the set of states of A with the restriction topology of the Tychonoff cube $[0,1]^A$ [3].

We refer to [47] for all unexplained notions of probability and measure theory. The intuition that formulas in Łukasiewicz logic code $[0,1]$-valued random variables is made precise by the following MV-algebraic counterpart of the boolean algebraic Carathéodory extension theorem[4]:

Theorem 5. (See [36,46], [41, 10.2]) *For any MV-algebra A let $\mathsf{P}(A)$ denote the space of regular Borel probability measures on the maximal spectral space $\mu(A)$. Let us equip $\mathsf{P}(A)$ with the weak topology. We then have:*

(i) The map $\eta \in \mathsf{hom}(A, [0,1]) \mapsto$ Dirac point mass at $\mathsf{ker}(\eta)$ uniquely extends to an affine homeomorphism γ_A of the state space $\mathsf{S}(A)$ onto $\mathsf{P}(A)$.

(ii) For every nonempty compact Hausdorff space Ω and Kolmogorov probability space $(\Omega, \mathcal{F}_\Omega, P)$ with $\mathcal{F}_\Omega$ the sigma-algebra of Borel sets of Ω and P a regular probability measure on $\mathcal{F}_\Omega$, there is an MV-algebra A and a state σ of A such that $(\Omega, \mathcal{F}_\Omega, P) \cong (\mu(A), \mathcal{F}_{\mu(A)}, \gamma_A(\sigma))$.

2.4 Analogies Between Analogies

We have shown that $\{0,1\}$-valued propositions are for boolean logic what continuous $[0,1]$-valued propositions are for infinite-valued Łukasiewicz logic. In symbols,

$$\frac{\{0,1\}\text{-valued propositions}}{\text{boolean logic}} = \frac{\text{continuous } [0,1]\text{-valued propositions}}{\text{Łukasiewicz logic}}$$

Specifically, Theorems 3–5 show:

$$\frac{\{0,1\}\text{-valued random variables}}{\text{boolean algebras}} = \frac{\text{continuous } [0,1]\text{-valued random variables}}{\text{MV-algebras}}$$

In the next section we will show:

$$\frac{\text{Commutative AF-algebras}}{\text{Countable boolean algebras}} = \frac{\text{AF-algebras with lattice-ordered } K_0}{\text{Countable MV-algebras}}$$

[3] When A is a boolean algebra, its states are also known as "finitely additive probability measures".

[4] see, e.g., Corollary 1 in T.Tao, https://terrytao.wordpress.com/2009/01/03/.

3 AF-Algebras and MV-Algebras

In this section, isomorphism classes of countable MV-algebras will be shown to be in a functorial one-one correspondence with a class of C*-algebras currently used in the mathematical physics of quantum statistical systems. We refer to [2] for partially ordered groups.

We prepare:

Definition 5. ([2]) By a *unital ℓ-group* we mean a lattice-ordered abelian group equipped with a distinguished strong unit.

Theorem 6. ([38, Theorem 3.9]) *There is a categorical equivalence Γ transforming every unital ℓ-group (G, u) into the MV-algebra $([0, 1], 0, \neg, \oplus)$ given by:*

$$1 = u, \quad \neg x = u - x, \quad \textit{and} \quad x \oplus y = (x + y) \wedge u \quad \textit{for all } x, y \in [0, u].$$

Furthermore, for every unital ℓ-homomorphism $\theta\colon (G, u) \to (H, v)$, $\Gamma(\theta) = \theta \restriction [0, u] =$ the restriction of θ to the unit interval of (G, u).

Definition 6. ([5,10,11]) A **-algebra* is an algebra B over $\mathbb{C}$ with a linear map $\star$ satisfying $x^{\star\star} = x$, $(zy)^\star = y^\star z^\star$ and $(\lambda z)^\star = \overline{\lambda} z^\star$ for all $\lambda \in \mathbb{C}$ and $x, y, z \in B$. A *C*-algebra* is a *-algebra B with a norm making B into a Banach space such that $||1_A|| = 1$, $||yz| \leq ||y||\,||z||$ and $||zz^\star|| = ||z||^2$. An *AF-algebra* is the norm closure of the union of an ascending sequence of finite-dimensional C^*-algebras, all with the same unit.

3.1 Elliott Classification, [13]

Definition 7. ([10,11]) A *projection* p in a C*-algebra $\mathfrak{A}$ is a self-adjoint idempotent $p^\star = p = p^2$. Projections p, q of $\mathfrak{A}$ are said to be (Murray-von Neumann) *equivalent*, in symbols $p \sim q$, if $p = x^\star x$ and $q = xx^\star$ for some $x \in \mathfrak{A}$. We write $p \preceq q$ if $p \sim r$ for some projection r of $\mathfrak{A}$ with $rq = qr = r$.

Proposition 2. ([10, Theorem IV.2.3]) *For any AF-algebra $\mathfrak{A}$, $\sim$ is an equivalence relation on the set of projections of $\mathfrak{A}$. For any projection $p \in \mathfrak{A}$ we let $[p]$ denote the equivalence class of p, and $L(\mathfrak{A})$ be the set of these equivalence classes. The $\preceq$-relation in $\mathfrak{A}$ is reflexive and transitive and is preserved under equivalence. Since $\mathfrak{A}$ is stably finite, $\preceq$ is antisymmetric: $p \preceq q \preceq p \Rightarrow p \sim q$.*

Definition 8. ([10,12,13]) The resulting partial order relation on $L(\mathfrak{A})$, also denoted $\preceq$, is known as the *Murray-von Neumann order* of $\mathfrak{A}$. One next equips $L(\mathfrak{A})$ with *Elliott's partial addition* by setting $[p] + [q] = [p + q]$ whenever projections p and q are orthogonal.

Theorem 7. ([10,12,13]) *$L(\mathfrak{A})$ has the partial structure a countable partially ordered semigroup, known as* Elliott's "local" semigroup. *$L(\mathfrak{A})$ is a complete classifier of AF-algebras: $\mathfrak{A}_1 \cong \mathfrak{A}_2$ iff $L(\mathfrak{A}_1) \cong L(\mathfrak{A}_2)$. The $\preceq$-relation equips $L(\mathfrak{A})$ with a partial order such that Elliott's partial addition is monotone.*

3.2 AFℓ-Algebras

Definition 9. ([42,44]) An *AFℓ-algebra* is an AF-algebra $\mathfrak{A}$ whose Murray-von Neumann order is a lattice. Equivalently, [10,12], the Grothendieck K_0-group of $\mathfrak{A}$ is lattice-ordered.

Table 1. Some classes of AFℓ-algebras and their associated MV-algebras.

AFℓ-ALGEBRA $\mathfrak{A}$	COUNTABLE MV ALGEBRA $\Gamma(K_0(\mathfrak{A}))$
$\mathbb{C}$	the two element boolean algebra
$B(\mathbb{C}_n)$, the $n \times n$ complex matrices	Łukasiewicz chain $\{0,\ 1/n,\ 2/n, \ldots, 1\}$
finite dimensional	finite
commutative	boolean
$C(2^\omega)$, $2^\omega =$ Cantor cube, [12, p.13]	atomless boolean
with comparability of projections, [14]	totally ordered
Glimm's UHF algebra, [48, p.16]	subalgebra of $\mathbb{Q} \cap [0,1]$
CAR algebra, [10, III.2.4], [48, 1.2.6]	dyadic rationals in $[0,1]$
Glimm's universal algebra, [48, p.13]	$\mathbb{Q} \cap [0,1]$
simple with comparability, [14]	subalgebra of [0,1]
Effros-Shen algebra $\mathfrak{F}_\xi$, [12, p.65]	generated in $[0,1]$ by $\xi \in [0,1] \setminus \mathbb{Q}$
Blackadar algebra B, [3, p. 504]	real algebraic numbers in [0,1]
Behncke-Leptin, two-point dual, [1]	with two ideals
Behncke-Leptin $\mathcal{A}_{0,1}$, [1]	Chang algebra C, [6, p.474]
liminary, T_2 spectrum, [9]	every prime quotient is finite
subhomogeneous, T_2 spectrum, [9]	finite-valued, [8, §8.2]
homogeneous of order n, [9]	Post algebra of order $n+1$, [8, p.198]
the universal AFℓ-algebra $\mathfrak{M}$, [38, §8]	free on countably many generators [8]
the "Farey" algebra $\mathfrak{M}_1$, [4,39,40]	free on one generator, [37], [8, §3.2]
$\mathfrak{M}_n$, [44]	free on n generators, [8,37,41]
finitely presentable, [42]	finitely presentable, [41, §6.2]

Proposition 3. (Special case of Gelfand duality, [11]) *Every commutative AF-algebra $\mathfrak{B}$ has the form $C(X, \mathbb{C})$ for X a separable totally disconnected nonempty compact Hausdorff space. It follows that $\mathfrak{B}$ is an AFℓ-algebra. Moreover, commutative AF-algebras are categorically equivalent to countable boolean algebras.*

AFℓ-algebras are the AF-algebras most frequently found in the literature. (See Table 1.) They are virtually all AF-algebras having an algorithmic theory, [38, 42,44]. This depends on the following strengthening of Proposition 3, (we refer to [10] for the K_0-theory of AF-algebras):

Theorem 8. *Let $\mathfrak{A}$ be an AF-algebra.*

(i) ([45]) *Elliott's partially defined addition $+$ in $L(\mathfrak{A})$ has* at most one *extension to an associative, commutative, monotone operation $\oplus\colon L(\mathfrak{A})^2 \to L(\mathfrak{A})$ such that for each projection $p \in \mathfrak{A}$, $[1_{\mathfrak{A}} - p]$ is the smallest element $[q] \in L(\mathfrak{A})$ with $[p] \oplus [q] = [1_{\mathfrak{A}}]$. The uniquely determined semigroup $(S(\mathfrak{A}), \oplus)$ expanding $L(\mathfrak{A})$ exists iff $\mathfrak{A}$ is an AFℓ-algebra.*

(ii) (From (i) as a special case of Elliott's classification [13]) *Let $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be AFℓ-algebras. For each $j = 1, 2$ let $\oplus_j$ be the extension of Elliott's addition given by (i). Then the semigroups $(S(\mathfrak{A}_1), \oplus_1)$ and $(S(\mathfrak{A}_2), \oplus_2)$ are isomorphic iff so are $\mathfrak{A}_1$ and $\mathfrak{A}_2$.*

(iii) (From (i) as a special case of the K_0-theoretic reformulation of Elliott's classification; see [14] and [12]) *For any AFℓ-algebra $\mathfrak{A}$ the partially ordered Grothendieck group $K_0(\mathfrak{A})$ (which is shorthand for $(K_0(\mathfrak{A}), K_0(\mathfrak{A})^+, [1_{\mathfrak{A}}])$) is a countable unital ℓ-group. All countable unital ℓ-groups arise in this way. Let $\mathfrak{A}$ and $\mathfrak{A}'$ be AFℓ-algebras. Then $K_0(\mathfrak{A})$ and $K_0(\mathfrak{A}')$ are isomorphic as unital ℓ-groups iff $\mathfrak{A}$ and $\mathfrak{A}'$ are isomorphic.*

(iv) ([45]) *For any AFℓ-algebra $\mathfrak{A}$ the semigroup $(S(\mathfrak{A}), \oplus)$ has the structure of a monoid $(E(\mathfrak{A}), 0, \neg, \oplus)$ with an involution operation $\neg[p] = [1_{\mathfrak{A}} - p]$. The Murray-von Neumann lattice order of equivalence classes of projections $[p], [q]$ is definable by the involutive monoidal operations of $E(\mathfrak{A})$, upon setting $[p] \vee [q] = \neg(\neg[p] \oplus [q]) \oplus [q]$ and $[p] \wedge [q] = \neg(\neg[p] \vee \neg[q])$ for all $[p], [q] \in E(\mathfrak{A})$. $(E(\mathfrak{A}), 0, \neg, \oplus)$ is a countable MV-algebra.*

(v) (From (ii) and (iii), see [38, 3.9]) *Up to isomorphism, the map $\mathfrak{A} \mapsto (E(\mathfrak{A}), 0, \neg, \oplus)$ is a bijection of AFℓ-algebras onto countable MV-algebras. Furthermore, with Γ the categorical equivalence of Theorem 6, $(E(\mathfrak{A}), 0, \neg, \oplus)$ is isomorphic to $\Gamma(K_0(\mathfrak{A}))$.*

From this theorem and the fundamentals of the K_0-theory of AF-algebras, [10, 12], we obtain:

Corollary 3. ([39, and references therein]) *In any AFℓ-algebra $\mathfrak{A}$ we have:*

(i) K_0 induces an isomorphism $\eta : \mathfrak{i} \mapsto K_0(\mathfrak{i}) \cap E(\mathfrak{A})$ between the lattice of ideals of $\mathfrak{A}$ and the lattice of ideals of the MV-algebra $E(\mathfrak{A})$.

(ii) Suppose I is an ideal of the countable MV-algebra B. In view of Theorem 8(v) let the AFℓ-algebra $\mathfrak{A}$ be defined by $E(\mathfrak{A}) = B$. Let $\mathfrak{i}$ be the ideal of $\mathfrak{A}$ defined by $\eta(\mathfrak{i}) = I$. Then B/I is isomorphic to $E(\mathfrak{A}/\mathfrak{i})$.

(iii) For every ideal $\mathfrak{i}$ of $\mathfrak{A}$, the map $[p/\mathfrak{i}] \mapsto [p]/\eta(\mathfrak{i})$, ($p$ a projection of $\mathfrak{A}$), is an isomorphism of $E(\mathfrak{A}/\mathfrak{i})$ onto $E(\mathfrak{A})/\eta(\mathfrak{i})$.

(iv) The map $J \mapsto J \cap \Gamma(K_0(\mathfrak{A}))$ is an isomorphism of the lattice of ℓ-ideals *of $K_0(\mathfrak{A})$ (i.e., kernels of unit preserving ℓ-homomorphisms of $K_0(\mathfrak{A})$ into unital ℓ-groups) onto the lattice of ideals of $E(\mathfrak{A})$. Further,*

$$\Gamma\left(\frac{K_0(\mathfrak{A})}{J}\right) \cong \frac{\Gamma(K_0(\mathfrak{A}))}{J \cap \Gamma(K_0(\mathfrak{A}))}\,.$$

4 Computing on AFℓ-Algebras

The results of the foregoing section are applied in this section to compute on equivalence classes of projections in AFℓ-algebras. The algorithmic-deductive machinery of Łukasiewicz logic will play a fundamental role.

4.1 Ł$_\infty$-Coding of Projections of AFℓ-Algebras

As usual, the set $\mathcal{A}^*$ of *strings* over the *alphabet* $\mathcal{A} = \{0, \neg, \oplus, X_1, X_2, \dots,), (, \}$ is defined by $\mathcal{A}^* = \{(s_1, s_2, ..., s_l) \in \mathcal{A}^l \mid l = 0, 1, 2, \dots\}$.

Definition 10. ([8, 3.1]) A *term in the variables* $X_1, \dots, X_n$, is a string $\phi \in \mathcal{A}^*$ obtainable by the following inductive definition: 0 and $X_1, X_2, \dots, X_n$ are terms; if α and β are terms, then so are $\neg\alpha$ and $(\alpha \oplus \beta)$. We let TERM_n denote the set of terms in the variables $X_1, X_2, \dots, X_n$. Each $\phi \in \mathsf{TERM}_n$ is said to be an n-variable Ł$_\infty$-*formula.* We let $\mathsf{TERM}_\omega = \bigcup_n \mathsf{TERM}_n$.

Free MV-algebras, [8, §§3–4], are algebras of logically equivalent formulas. They are given a concrete geometrical realization by the following result. Here the adjective "linear" is understood in the affine sense:

Proposition 4. (McNaughton's representation theorem, ([37], [8, 9.1.5]) *Fix* $n = 1, 2, \dots$ *and let* $\mathcal{M}([0, 1]^n)$ *denote the MV-algebra of* n*-variable* McNaughton functions*, i.e., continuous piecewise linear functions* $f \colon [0, 1]^n \to [0, 1]$ *(with a finite number of linear pieces) such that every piece of* f *agrees with a linear polynomial with integer coefficients. Let* π_i, $(i = 1, \dots, n)$*, denote the ith coordinate function on* $[0, 1]^n$*. With the pointwise operations of the standard MV-algebra,* $\mathcal{M}([0, 1]^n)$ *is the free MV-algebra over the free generating set* $\{\pi_1, \dots, \pi_n\}$.

Ł$_\infty$-Coding of Projections of $\mathfrak{M}_n$. Recalling Theorem 8(v), the AFℓ-algebra $\mathfrak{M}_n$ is defined by $E(\mathfrak{M}_n) \cong \mathcal{M}([0, 1]^n)$. Let us fix, once and for all, projections $\mathsf{p}_1, \dots, \mathsf{p}_n \in \mathfrak{M}_n$ such that the equivalence classes $[\mathsf{p}_1], \dots, [\mathsf{p}_n] \in E(\mathfrak{M}_n)$ correspond to the free generators $\pi_1, \dots, \pi_n$ via Elliott classification. We then say that the variable symbol X_i *codes* both the coordinate function $\pi_i \in \mathcal{M}([0, 1]^n)$ and its isomorphic image given by the equivalence class $[\mathsf{p}_i] \in E(\mathfrak{M}_n)$. The coordinate function π_i is said to be the *interpretation of* X_i *in (the Elliott monoid* $\mathcal{M}([0, 1]^n)$ *of)* $\mathfrak{M}_n$*, in symbols,* $X_i^{\mathfrak{M}_n} = \pi_i$. Next for every $\phi \in \mathsf{TERM}_n$, the *interpretation* $\phi^{\mathfrak{M}_n}$ *of* ϕ *in* $\mathfrak{M}_n$ has the usual inductive definition: $0^{\mathfrak{M}_n} =$ the constant zero function over $[0, 1]^n$, and $(\neg\psi)^{\mathfrak{M}_n} = \neg(\psi^{\mathfrak{M}_n})$, $(\alpha \oplus \beta)^{\mathfrak{M}_n} = (\alpha^{\mathfrak{M}_n} \oplus \beta^{\mathfrak{M}_n})$. We also say that ϕ *codes* $\phi^{\mathfrak{M}_n}$[5].

Ł$_\infty$-Coding of Projections of $\mathfrak{M}_n/\mathfrak{I}$. More generally, let $\mathfrak{A} = \mathfrak{M}_n/\mathfrak{I}$ be an AFℓ-algebra, for some ideal $\mathfrak{I}$ of $\mathfrak{M}_n$. Let $\mathfrak{i} = K_0(\mathfrak{I}) \cap E(\mathfrak{M}_n)$. be the ideal of $\mathcal{M}([0, 1]^n)$ corresponding to $\mathfrak{I}$ by Corollary 3(iv). Identifying $E(\mathfrak{M}_n/\mathfrak{I})$ and $\mathcal{M}([0, 1]^n)/\mathfrak{i}$, the *interpretation* $\phi^{\mathfrak{A}}$ *of* ϕ *in* $\mathfrak{A}$ is defined by: $0^{\mathfrak{A}} = \{0\} \subseteq \mathfrak{A}$, $X_i^{\mathfrak{A}} =$

[5] By a traditional abuse of notation, the MV-algebraic *operation symbols* also denote their corresponding *operations.*

$X_i^{\mathfrak{M}_n/\mathfrak{I}} = X_i^{\mathfrak{M}_n}/\mathfrak{i} = \pi_i/\mathfrak{i}$, and inductively, $(\neg\psi)^{\mathfrak{A}} = \neg(\psi^{\mathfrak{A}})$, $(\alpha \oplus \beta)^{\mathfrak{A}} = (\alpha^{\mathfrak{A}} \oplus \beta^{\mathfrak{A}})$. The following identities are easily verified by induction on the number of symbols occurring in each term:

$$(\neg\psi)^{\mathfrak{A}} = \frac{(\neg\psi)^{\mathfrak{M}_n}}{\mathfrak{i}} = \neg\left(\frac{\psi^{\mathfrak{M}_n}}{\mathfrak{i}}\right) = \frac{\neg(\psi^{\mathfrak{M}_n})}{\mathfrak{i}}$$

and

$$(\alpha \oplus \beta)^{\mathfrak{A}} = \frac{(\alpha \oplus \beta)^{\mathfrak{M}_n}}{\mathfrak{i}} = \frac{\alpha^{\mathfrak{M}_n}}{\mathfrak{i}} \oplus \frac{\beta^{\mathfrak{M}_n}}{\mathfrak{i}} = \frac{\alpha^{\mathfrak{M}_n} \oplus \beta^{\mathfrak{M}_n}}{\mathfrak{i}}.$$

We also say that ϕ *codes* $\phi^{\mathfrak{A}}$ *in* $\mathfrak{A}$.

Ł$_\infty$-Coding of Projections of $\mathfrak{M}$. In [38, §8], the universal AFℓ-algebra $\mathfrak{M}$ is defined by $E(\mathfrak{M}) \cong \mathcal{M}([0,1]^\omega)$ = the free MV-algebra over countably many generators = the direct limit of the free MV-algebras $\mathcal{M}([0,1]^n)$. Then one defines the *interpretation* $\phi^{\mathfrak{M}}$ of $\phi \in \mathsf{TERM}_\omega$ in $\mathfrak{M}$ mimicking the definition of $\phi^{\mathfrak{M}_n}$. A Bratteli diagram of $\mathfrak{M}$ is constructed in [44]. By Corollary 3, every AFℓ-algebra $\mathfrak{A}$ is a quotient of $\mathfrak{M}$. The Bratteli diagram of $\mathfrak{A}$ can be obtained as a subdiagram of the diagram of $\mathfrak{M}$, [44].

4.2 Decision Problems for AFℓ-Algebras and Their Complexity

Fix a cardinal $\kappa = 1, 2, \ldots, \omega$. Let $\mathfrak{A} = \mathfrak{M}_\kappa/\mathfrak{I}$ for some ideal $\mathfrak{I}$ of $\mathfrak{M}_\kappa$. (Here $\mathfrak{M}_\omega = \mathfrak{M}$.)

- The *word problem* P_{word} *of* $\mathfrak{A}$ *is defined by* $\mathsf{P}_{\text{word}} = \{(\phi, \psi) \in \mathcal{A}^* \mid (\phi, \psi) \in \mathsf{TERM}_\kappa^2 \text{ and } \phi^{\mathfrak{A}} = \psi^{\mathfrak{A}}\}$. In down to earth terms, on input strings ϕ and ψ, P_{word} checks if the strings ϕ and ψ are elements of TERM_κ that code the same equivalence class of projections of $\mathfrak{A}$.
- The *order problem* $\mathsf{P}_{\text{order}}$ of $\mathfrak{A}$ checks if ϕ codes an equivalence class of projections $\phi^{\mathfrak{A}}$ in $\mathfrak{A}$ that precedes $\psi^{\mathfrak{A}}$ in the Murray-von Neumann order $\preceq$ of projections in $\mathfrak{A}$.
- The *zero problem* $\mathsf{P}_{\text{zero}} = \{\phi \in \mathsf{TERM}_\kappa \mid \phi^{\mathfrak{A}} = 0\}$ of $\mathfrak{A}$ checks if $\phi^{\mathfrak{A}} = 0$.
- The *central projection problem* $\mathsf{P}_{\text{central}}$ of $\mathfrak{A}$ checks if $\phi^{\mathfrak{A}}$ is an equivalence class of a *central* projection $p \in \mathfrak{A}$, i.e., a projections such that $pa = ap$ for all $a \in \mathfrak{A}$.
- The *nontrivial projection problem* $\mathsf{P}_{\text{nontrivial}}$ of $\mathfrak{A}$ checks if $\phi^{\mathfrak{A}}$ different from 0 and 1.
- The *nontrivial central projection problem* $\mathsf{P}_{\text{nontrivialcentral}}$ of $\mathfrak{A}$ checks if $\phi^{\mathfrak{A}}$ is an equivalence class of central projections of $\mathfrak{A}$ other than 0 or 1.

We collect here a number of results on the complexity of computations on (equivalence classes of) projections in many AFℓ-algebras listed in Table 1.

The "Farey" AFℓ-algebra $\mathfrak{M}_1$ was first introduced in [39] and rediscovered in [4] (see [40]). As already noted, in view of Theorem 8(v), $\mathfrak{M}_1$ can be defined by $E(\mathfrak{M}_1) \cong \mathcal{M}([0,1]) =$ the free one-generator MV-algebra.

We first record a result on *Gödel incomplete* (i.e., recursively enumerable undecidable) problems.

Theorem 9. (See [42, §8]) *There exists a quotient of* $\mathfrak{M}_1$ *having a* Gödel incomplete *zero problem. No primitive quotient of* $\mathfrak{M}_1$*, whence a fortiori, no simple quotient of* $\mathfrak{M}_1$*, has a Gödel incomplete zero problem. The same holds for all problems* P_{word}, $\mathsf{P}_{\text{order}}$, P_{zero}, $\mathsf{P}_{\text{nontrivial}}$.

We refer to [12] for the Effros-Shen algebras $\mathfrak{F}_\theta$, to [1] for the Behncke-Leptin algebras $\mathcal{A}_{m,n}$, to [48, p.16] for Glimm's universal UHF algebra, and to [10, III.2.4, III.5.4] or [48, 1.2.6] for the CAR algebra.

Extending results in [42] we have:

Theorem 10. P_{word}, $\mathsf{P}_{\text{order}}$, P_{zero}, $\mathsf{P}_{\text{central}}$, $\mathsf{P}_{\text{nontrivial}}$, $\mathsf{P}_{\text{central nontrivial}}$ *are decidable in polynomial time for the following AFℓ-algebras:*

(i) The Effros-Shen algebra $\mathfrak{F}_\theta$ *for* θ *a quadratic irrational, or* $\theta = 1/\mathrm{e}$, *(with* e *Euler's constant) or* $\theta \in [0,1] \setminus \mathbb{Q}$ *a real algebraic integer.*

(ii) The Effros-Shen algebra $\mathfrak{F}_\theta$ *for any irrational* $\theta \in [0,1]$ *having the following property: There is a real* $\kappa > 0$ *such that for every* $n = 0, 1, \ldots$ *the sequence* $[a_0, \ldots, a_n]$ *of partial quotients of* θ *is computable (as a finite list of binary integers) in less than* $2^{\kappa n}$ *steps.*

(iii) Each Behncke-Leptin algebra $\mathcal{A}_{m,n}$.

(iv) Glimm's universal UHF algebra.

(v) The CAR algebra.

The proof of the following result will appear elsewhere:

Theorem 11. *(i) For the AFℓ-algebra* $\mathfrak{M}$*, each problem* P_{word}, $\mathsf{P}_{\text{order}}$, P_{zero}, $\mathsf{P}_{\text{central}}$ *is coNP-complete. On the other hand,* $\mathsf{P}_{\text{nontrivial}}$ *is NP-complete.*

(ii) Each problem P_{word}, $\mathsf{P}_{\text{order}}$, P_{zero}, $\mathsf{P}_{\text{central}}$, $\mathsf{P}_{\text{nontrivial}}$, $\mathsf{P}_{\text{central nontrivial}}$ *for the AFℓ-algebra* $\mathfrak{M}_n$ *is decidable in polynomial time,* $(n = 1, 2, \ldots)$.

5 Final Remarks: Closing a Circle of Ideas

As is well known, if SAT turns out to be decidable in polynomial time, many other important computational problems will also be. No less importantly, the very nature of mathematical proofs will change dramatically, as all proof methods for boolean logic known today take exponential time.

Similar considerations apply to $Ł_\infty$-satisfiability, another NP-complete problem, [8, §9.3 and references therein]. The results of this paper, in combination with earlier work by the present author and by Reiner, show the wide scope of $Ł_\infty$-satisfiability, from Mixed Integer Programming to the algorithmic theory of a class of C*-algebras currently used in quantum statistical mechanics.

References

1. Behncke, H., Leptin, H.: C*-algebras with a two-point dual. J. Funct. Anal. **10**(3), 330–335 (1972). https://doi.org/10.1016/0022-1236(72)90031-6
2. Bigard, A., Wolfenstein, S., Keimel, K.: Groupes et Anneaux Réticulés. LNM, vol. 608. Springer, Heidelberg (1977). https://doi.org/10.1007/BFb0067004
3. Blackadar, B.: A simple C*-algebra with no nontrivial projections. Proc. Amer. Math. Soc. **78**, 504–508 (1980). https://www.jstor.org/stable/2042420
4. Boca, F.: An AF algebra associated with the Farey tessellation. Canad. J. Math. **60**, 975–1000 (2008). https://doi.org/10.4153/CJM-2008-043-1
5. Bratteli, O.: Inductive limits of finite-dimensional C*-algebras. Trans. Amer. Math. Soc. **171** , 195–234 (1972). https://www.jstor.org/stable/1996380
6. Chang, C.C.: Algebraic analysis of many-valued logics. Trans. Amer. Math. Soc. **88**, 467–490 (1958). https://www.jstor.org/stable/1993227
7. Chang, C.C.: A new proof of the completeness of the Łukasiewicz axioms. Trans. Amer. Math. Soc. **93**, 74–90 (1959). https://www.jstor.org/stable/1993423
8. Cignoli, R., D'Ottaviano, I.M.L., Mundici, D.: Algebraic foundations of many-valued reasoning. Trends in Logic, vol. 7, Kluwer, Dordrecht (2000). Reprinted, Springer (2013). https://doi.org/10.1007/978-94-015-9480-6
9. Cignoli, R., Elliott, G.A., Mundici, D.: Reconstructing C*-algebras from their Murray von Neumann order. Adv. Math. **101**, 166–179 (1993). https://doi.org/10.1016/j.jpaa.2003.10.021
10. Davidson, K.R.: C*-Algebras by Example, Fields Institute Monographs. American Mathematical Society, Providence, vol. 6 (1996). https://doi.org/10.1090/fim/006
11. Dixmier, J.: C*-algebras, North-Holland Mathematical Library. North-Holland, Amsterdam, vol. 15(1977). https://www.sciencedirect.com/bookseries/north-holland-mathematical-library/vol/15/suppl/C
12. Effros, E.G.: Dimensions and C*-algebras, CBMS Regional Conference Series in Mathematics. American Mathematical Society, Providence, vol. 46 (1981). https://bookstore.ams.org/cbms-46/20ISBN:78-0-8218-1697-4
13. Elliott, G.A.: On the classification of inductive limits of sequences of semisimple finite-dimensional algebras. J. Algebra **38**, 29–44 (1976). https://doi.org/10.1016/0021-8693(76)90242-8
14. Elliott, G.A.: On totally ordered groups, and K_0. In: Handelman, D., Lawrence, J. (eds.) Ring Theory Waterloo 1978 Proceedings, University of Waterloo, Canada, 12–16 June 1978. LNM, vol. 734, pp. 1–49. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0103152
15. Hähnle, R.: Tableaux-based theorem proving in multiple-valued logics, Ph.D. thesis, University of Karlsruhe, Department of Computer Science (1992). http://tubiblio.ulb.tu-darmstadt.de/101360/
16. Beckert, B., Hähnle, R., Gerberding, S., Kernig, W.: The tableau-based theorem prover ${}_3\mathrm{T}^A\mathrm{P}$ for multiple-valued logics. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 758–760. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_219
17. Hähnle, R.: Short CNF in finitely-valued logics. In: Komorowski, J., Raś, Z.W. (eds.) ISMIS 1993. LNCS, vol. 689, pp. 49–58. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56804-2_5
18. Hähnle, R.: Automated Deduction in Multiple-Valued Logics. Oxford University Press (1994). https://global.oup.com/academic/product/automated-deduction-in-multiple-valued-logics-9780198539896?lang=en&cc=nl ISBN: 9780198539896

19. Hähnle, R.: Many-valued logic and mixed integer programming. Ann. Math. Artif. Intell. **12**, 231–263 (1994). https://doi.org/10.1007/BF01530787
20. Hähnle, R.: Automated deduction and integer programming. In: Collegium Logicum (Annals of the Kurt-Gödel-Society), vol. 1, pp. 67–77. Springer, Vienna (1995). https://doi.org/10.1007/978-3-7091-9394-5_6
21. Hähnle, R.: Exploiting data dependencies in many-valued logics. J. Appl. Non-Classical Logics **6**(1), 49–69 (1996). https://doi.org/10.1080/11663081.1996.10510866
22. Hähnle, R.: Proof theory of many-valued logic-linear optimization-logic design: connections and interactions. Soft Comput. **1**(3), pp. 107–119 (1997). https://link.springer.com/content/pdf/10.1007%2Fs005000050012.pdf
23. Hähnle, R.: Commodious axiomatization of quantifiers in multiple-valued logic. Stud. Logica **61**(1), 101–121 (1998). https://doi.org/10.1023/A:1005086415447
24. Hähnle, R., Bernhard, B., Escalada-Imaz, G.: Simplification of many-valued logic formulas using anti-links. J. Logic Comput. **8**(4), 569–588 (1998). https://doi.org/10.1093/logcom/8.4.569
25. Hähnle, R.: Tableaux for many-valued logics. In: D'Agostino, M., et al. (eds.) Handbook of Tableau Methods, pp. 529–580. Kluwer, Dordrecht (1999). https://link.springer.com/book/10.1007%2F978-94-017-1754-0
26. Hähnle, R., Beckert, B., Manyá, F.: Transformations between signed and classical clause logic. In: Proceedings, 29th IEEE International Symposium on Multiple-Valued Logic (ISMVL 1999), pp. 248–255. https://doi.org/10.1109/ISMVL.1999.779724
27. Hähnle, R., Beckert, B., Manyá, F.: The 2-SAT problem of regular signed CNF formulas. In: Proceedings 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000), pp. 331–336. https://doi.org/10.1109/ISMVL.2000.848640
28. Hähnle, R., Beckert, B., Manyá, F.: The SAT problem of signed CNF formulas. In: Basin, D., et al. (eds.) Labelled Deduction. Applied Logic Series, vol. 17. Springer, Dordrecht (2000). https://doi.org/10.1007/978-94-011-4040-9_3
29. Hähnle, R.: Short conjunctive normal forms in finitely valued logics. J. Logic Comput. **4**, 905–927 (2000). https://doi.org/10.1093/logcom/4.6.905
30. Hähnle, R.: Proof theory of many-valued logic and linear optimization. In: B. Reusch et al. (eds.) Advances in Soft Computing. Computational Intelligence in Theory and Practice. Advances in Soft Computing. Physica, Heidelberg, vol. 8, pp. 15–33 (2001). https://doi.org/10.1007/978-3-7908-1831-4_2
31. Hähnle, R.: Advanced Many-Valued Logics. In: Gabbay, D.M., et al. (eds.) Handbook of Philosophical Logic, vol. 2, pp. 297–395. Springer, Dordrecht (2001). https://doi.org/10.1007/978-94-017-0452-6_5
32. Hähnle, R.: Tableaux and related methods. In: Handbook of Automated Reasoning 1, Elsevier Science Publishers B.V., pp. 101–178 (2001). https://doi.org/10.1016/B978-044450813-3/50005-9
33. Hähnle, R.: Complexity of many-valued logics. In: Fitting, M., et al. (eds.) Beyond Two: Theory and Applications of Multiple-Valued Logic, Studies in Fuzziness and Soft Computing. Physica, Heidelberg, vol. 114, pp. 211–233 (2003). https://doi.org/10.1007/978-3-7908-1769-0_9
34. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Logic J. IPGL **13**(4), 415–433 (2005). https://doi.org/10.1093/jigpal/jzi032
35. Kolařík, M.: Independence of the axiomatic system for MV-algebras. Math. Slovaca **63**, 1–4 (2013). https://doi.org/10.2478/s12175-012-0076-z

36. Kroupa, T.: Every state on a semisimple MV-algebra is integral. Fuzzy Sets Syst. **157**, 2771–2782 (2006). https://doi.org/10.1016/j.fss.2006.06.015
37. McNaughton, R.: A theorem about infinite-valued sentential logic. J. Symbolic Logic, **16**, 1–13 (1951). https://www.jstor.org/stable/2268660
38. Mundici, D.: Interpretation of AF C*-algebras in Łukasiewicz sentential calculus. J. Funct. Anal. **65**, 15–63 (1986). https://core.ac.uk/download/pdf/81941332.pdf
39. Mundici, D.: Farey stellar subdivisions, ultrasimplicial groups, and K_0 of AF C*-algebras. Adv. Math. **68**, 23–39 (1988). https://doi.org/10.1016/0001-8708(88)90006-0
40. Mundici, D.: Recognizing the Farey-Stern-Brocot AF algebra. Rendiconti Lincei Mat. Appl., **20**, 327–338 (2009). https://ems.press/journals/rlm/articles/2994, Dedicated to the memory of Renato Caccioppoli
41. Mundici, D.: Advanced Łukasiewicz calculus and MV-algebras. Trends in Logic, vol. 35. Springer, Berlin (2011). https://link.springer.com/book/10.1007%2F978-94-007-0840-2
42. Mundici, D.: Word problems in Elliott monoids. Adv. Math. **335**, 343–371 (2018). https://doi.org/10.1016/j.aim.2018.07.015
43. Mundici, D.: What the Łukasiewicz axioms mean. J. Symbolic Logic **85**, 906–917 (2020). https://doi.org/10.1017/jsl.2020.74
44. Mundici, D.: Bratteli diagrams via the De Concini-Procesi theorem. Commun. Contemp. Math. **23**(07), 2050073 (2021). https://doi.org/10.1142/S021919972050073X
45. Mundici, D., Panti, G.: Extending addition in Elliott's local semigroup. J. Funct. Anal. **117**, 461–471 (1993). https://doi.org/10.1006/jfan.1993.1134
46. Panti, G.: Invariant measures in free MV-algebras. Commun. Algebra **36**, 2849–2861 (2009). https://doi.org/10.1080/00927870802104394
47. Parthasarathy, K.R.: Probability Measures on Metric Spaces. Academic Press, New York (1967). https://doi.org/10.1016/C2013-0-08107-8
48. Rørdam, M., Størmer, E.: Classification of Nuclear C*-Algebras, Entropy in Operator Algebras, Encyclopaedia of Mathematical Sciences, Operator Algebras and Non-Commutative Geometry, vol. 126. Springer-Verlag, Berlin, Heidelberg (2002). https://doi.org/10.1007/978-3-662-04825-2 ISBN 978-3-540-42305-8
49. Wójcicki, R.: Lectures on Propositional Calculi. Publishing House of the Polish Academy of Sciences, Ossolineum (1984). http://sl.fr.pl/wojcicki/Wojcicki-Lectures.pdf

Speaking About Wine: Another Case Study in Bridging the Gap Between Formal and Informal Knowledge

Aarne Ranta(✉)

Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
aarne@chalmers.se

Abstract. This paper presents WineSpeak, a system that uses controlled natural language for translation and information retrieval about the topic of wine. WineSpeakcombines some recent work on natural language interfaces to databases with other recent work on information extraction from heterogeneous sources. It supports database queries about wine in natural language, as well as the translation of questions, comments, and short articles about wine in human-to-human communication. WineSpeakwill be available in English, German, French, Italian, and Spanish, but also readily portable to other languages via the use of Grammatical Framework and its Resource Grammar Library (The system will be released in connection to the publication of this volume at https://www.grammaticalframework.org/gf-winespeak/. The final name of the system can be different, to avoid confusion with other similar names).

Keywords: Grammatical Framework · Natural language queries · Wine information

1 Introduction

Speaking about wine can be at least as rigorous as speaking about software specifications. Unlike software, wine has thousands of years of tradition of description, classification, and legislation, much of it expressed in standardized vocabulary. The first traces of wine cultivation are found in Georgia around 7000–5000 BC. Ancient Roman literature contains several systematic treatises of viticulture and wines, including classifications of grape varieties and growing areas, by authors such as Cato the Elder, Columella, Pliny the Elder, and Galen [7]. For example, one of the 37 books of Pliny's encyclopedic work *Naturalis Historia*, book 14 on fruit trees, is mostly about vines and wine. Its chapters has titles such as "Fifty kinds of generous wines", "Thirty-eight varieties of foreign wines"[9]. Today, the

To Reiner, my elder brother in wine and in science.

body of knowledge about grape varieties, regions, classifications, producers, and vintages is orders of magnitude larger and in constant need of extensions and updates.

Even real experts, who know how to speak rigorously about wine in their own language and a couple of others, may feel unsure when trying to communicate in other languages, as they want to be sure to make the right choices and pose the right questions when communicating with wine growers, merchants, and waiters in different countries. They might find it challenging to be as rigorous when speaking about wine as when using their own language.

Getting accurate information about wine and communicating it across languages is obviously a task where computers could help. It covers several subtasks within the field of Natural Language Processing (NLP):

- translation: talking and writing about wine across language boundaries;
- queries: posing questions and getting answers in different languages;
- documentation: publishing product descriptions and encyclopaedic articles about wines;
- reviews: giving comments on wine and sharing them with others;
- updates: collecting new information about both new and old wines from written documents, databases, etc.

Each of these tasks is growingly popular as an issue in Artificial Intelligence (AI). The most common techniques used in current AI are machine learning, heuristic rules, and crowd sourcing. These techniques, however, are not the most ideal ones for rigorous tasks such as speaking about wine. They could play a useful role in gathering information and suggesting solutions, but what we want in the end is **reliable** solutions that are **explainable** and **controllable**.

In NLP, as in much of AI, rigorous methods and reliable solutions are currently less popular than in the past, since they are regarded not to be scalable. More of these techniques can be seen, for instance, in older volumes of Springer Lecture Notes in Artificial Intelligence: logic, theorem proving, databases, formal grammars, software models. In this paper, we will take a step backward—and thereby probably also forward, as the development of AI has largely been cyclic—and use some of the "old" techniques to build a system for speaking about wine. We demonstrate the techniques by building a system called WineSpeak. Its main components are

- a database to answer queries, store information, and control the soundness of the information,
- a formal grammar to enable access to the information in natural languages (English, German, Italian, French, Spanish) and to translate between these languages,
- a web interface that enables tasks such as translation, queries, and document generation, as well as updates by authorized users.

The concrete, practical purpose of WineSpeakis to help its users to speak about wine, rigorously and across language barriers. But we also have a more general purpose in mind: to experiment with an architecture and a reference implementation that can be adapted to other domains of knowledge.

2 Informal Specification: System Functionalities

WineSpeakis aimed to deal with accurate, objective, true information about wines. It can answer both detailed and general questions:

- *What grapes and in what proportions is Tignanello made of?*
- *Show me all information about Tignanello.*
- *Show me some white wines from Rhône.*
- *What red wines are recommended for lobster?*
- *What is the best DOC region of Piemonte?*

The first three of these questions are clearly objective: they can be answered on the basis of verifiable information. The last two sound more subjective, but even they can get objective answers, if the answers are qualified with their sources:

- *The Guide Orange recommends red Saumur for lobster.*
- *According to John Johnson, Pinerolese is the best DOC of Piemonte.*

So much about queries. A wine information system can never be complete, and it must therefore be modifiable by users, at least authorized users. We can think of four levels of authorized users:

1. **managers**, who are allowed to change any information, including addition and removal of lower level users;
2. **experts**, who are allowed to enter both objective and subjective information;
3. **registered users**, who are allowed to enter comments of the subjective kind, such as recommendations;
4. **unregistered users**, who are only allowed to pose questions.

Since even managers and experts can make mistakes, the system needs to have **control mechanisms** to verify the consistency of the objective information added. For example,

- A red wine cannot be made from white grapes only.
- A wine from Bordeaux cannot be listed as as a Spanish wine, but only as French.
- If a wine is given new properties, they must be consistent with the already given ones.

The last-mentioned behaviour must still allow **destructive updates**, where the old information is corrected. Since updates are only permitted from expert users (level 2), **conflicts** between them can be solved by managers (level 1). An interim solution is to store both versions of the information, with a mention of what sources they come from.

With all of the above functionalities, the system works much like well-known applications such as the Vivino discussion forum[1]. One difference is that ours

[1] https://www.vivino.com/.

is completely open source, and all its design and implementation details will be documented. But the main difference, from the user's point of view, is the role of natural language: we maintain the same information in English, French, German, Italian, Spanish, and any other language that might be added.

The main language functionalities of WineSpeakare thus:

– ask questions and get answers in a natural language,
– add information and make comments in a natural language,
– translate questions and comments from any language to any other.

The natural language component is an integral part of the information system: everything that is said is **interpreted** in terms of the data via a **formal semantics**. This means that the use of "informal" language inherits the rigour of the underlying formalized information.

As the data is open to new information, the language component must also be open for new words and syntactic forms. The users (of appropriate levels) must be able to add expressions to the language. For instance, the ground data about Italian wines might use Italian names of the regions, such as *Toscana*, which is initially used in all languages. But as the system develops, users may add *Tuscany* to be used in English, *Toskana* in German, *Toscane* in French. This is clearly a part of the objective information in the system.

A slightly different example is words that are used to describe taste. The descriptions themselves might be subjective, but their translations can be objective information based on the standard terminology established among wine experts in each language.

3 Database

WineSpeakuses a **relational database**, with several **tables** and associated **constraints** to eliminate redundancy and guarantee consistency. The **schemas** of the tables specify what information is to be found where. Here is a list of the most important tables and their **attributes**, with **primary keys** (uniquely identifying attributes) in italics and **foreign keys** (attributes referring to other tables) given with information about what other tables they are keys of:

– **Countries**: *name*, continent.
– **Regions**: *name*, country (from Countries), status (e.g. DOC).
– **SubRegions**: *subregion, main region* (both from Regions).
– **Grapes**: *name*, colour (red or white).
– **Brands**: *name, colour* (red, white, rosé, orange), region, producer.
– **BrandGrapes**: *brand, colour* (from Brands), *year*, grape (from Grapes), percentage, alcohol percentage.

We have here used **natural keys** instead of artificial keys such as product numbers used by wine shops (e.g., the Swedish monopoly shop Systembolaget). This gives certain guarantees of consistency, but does not of course guard against everything, such as spelling variations. There are also situations where the keys do not really uniquely determine the other attributes—for instance, the continent where Turkey lies. As long as such situations are reasonably rare, they are easy to solve by adding information to the names, e.g. *Turkey (Europe).*

More importantly, since the system is meant to be used via a natural language interface, the parser can detect ambiguous references and ask for more information, e.g. *do you mean the European or Asian part of Turkey?* It remains to be seen how common such conflicts are in larger categories, in particular brand names. We have already anticipated some of this by including both brand name and colour in the primary key in Brands.

Adding the vintage year to brand name and colour is obviously needed in BrandGrapes, as the percentages of grapes and alcohol by volume change from year to year. A problem with this is that years are not always known, which is not acceptable for a primary key attribute in a relational database. Hence "unspecified" must be accepted as a value of "year".

Notice that the word *wine* itself is too ambiguous to be used as a name of any of the formalized types of data. Sometimes it stands for a whole region (*Barolo is a red wine from Piemonte*), sometimes for a brand, sometimes for a vintage of a brand or a region. But obviously it cannot be avoided in the natural language interface. A query like *Show me white wines from Rhône* should not be replied by a disambiguation question (*do you mean regions or brands or vintages*), but a reasonable list that might mix objects from different formalized categories.

The database schemas above are used for storing *objective information.* Subjective information, such as users' comments, are also convenient to store in the relational database, because they contain references to the objective data. A subjective comment is typically included in the system because it is *about* some wine in the objective database. In addition, it comes with a reference to some user—gathered in a table of users and their privileges—and has a time stamp.

But how are comment texts stored? Free text is the most immediate alternative, chosen in most message boards. But we will provide an alternative: **abstract syntax trees**, which support automatic translation and a formal structure supporting fine-grained queries via a formal semantics. This enables questions such as

– *Which German wines does Hugh Hughson never want to drink again?*

based on his subjective comments in the database[2]. This leads us naturally to the most special feature of WineSpeak: the grammar.

4 Grammar

The grammar takes care of parsing user input to abstract representations, which are converted to queries to the database. It also takes care of the opposite direction: generating answers to the user from abstract representations obtained from database response.

[2] It also makes sense to store the original free text comments, in particular, if they have typos or are otherwise not fully covered by the grammar.

In order to guarantee support for these functionalities, the grammar design starts from abstract representations—with an **abstract syntax**. An abstract syntax is a system of algebraic datatypes, consisting of **constructor functions** of each datatype. Here are some examples of constructor functions:

```
QWhatValue : Attribute -> Object -> Query
AColour : Attribute
OGrape  : Grape -> Object
GMalbec : Grape
```

From these functions, one can construct the abstract syntax tree

```
QWhatValue AColour (OGrape GMalbec)
```

representing the natural language query

What is the colour of Malbec?

The representation is abstract in the sense that it specifies the semantic components of the query (the colour, the grape, and the question form "what value") but leaves out the tokens "is", "the", "of", "?" in the same way as the abstract syntax of a programming languages ignores its keywords and punctuation marks. Even more importantly, the constructor functions themselves, `QWhatValue` etc., are abstract entities that can be realized in different ways in different languages. The realization is performad by **linearization functions**, which convert abstract syntax trees into linear strings of tokens in concrete languages.

A linearization function, at its simplest, is just a **template**, in which the linearizations of subtrees are inserted in strings with "holes". Here is a simple linearization function for the constructor `QWhatValue`:

```
QWhatValue <attr> <obj> = What is the <attr> of <obj>?
```

A set of such templates defines a context-free grammar, which, in addition to linearization, can be used for parsing natural language input into abstract syntax trees.

Different languages can have different templates for the same constructor functions:

```
QWhatValue <attr> <obj> = Was ist der <attr> von <obj>?
QWhatValue <attr> <obj> = Quel est le <attr> de <obj>?
```

for German and French, respectively. This means that one and the same abstract syntax works as an interface towards the database for any number of languages. However, as readers familiar with German or French will notice, the simple-minded templates do not work in all situations. The attribute can have different genders, requiring different articles in both languages. The object may need a conversion to the dative form in German, and the preposition *de* may need to be converted to *d'*, *du* or *des* depending on the object.

In Grammatical Framework (GF) [11], string templates are generalized to linearization functions that produce **records** and **tables**. Thus for instance the German word for wine is a record with an inflection table (*Wein*, *Weines*, *Weine*, etc., mapping combinations of case and number to a form of the word) and a gender (masculine). The GF code for this is

```
{s = table {
   Sg => table {Gen => "Weines" ; _ => "Wein"} ;
   Pl => table {Dat => "Weinen" ; _ => "Weine"}
   } ;
 g = Masc
}
```

Linearization rules for functions that take arguments combine the components of these data structures in different ways to guarantee grammatical correctness. For instance, the piece of code corresponding to the English "of the X" for a noun X in German is

```
von ++ defArt ! X.g ! Sg ! Dat ++ X.s ! Sg ! Dat
```

where the form of the definite article is selected from a table to match the gender of X in the dative singular, and the noun form is also the dative singular. A more sophisticated rule would also produce the contraction "vom" when the gender is masculine or neuter.

While GF makes it possible to describe these structures and their combinations accurately, they easily get complex and subtle, and therefore require detailed linguistic knowledge to get right. Therefore, a huge community effort has been made in GF to create a Resource Grammar Library (RGL, [10]), which hides low-level linguistic details behind a high-level API. Thus the inflection of the noun *Wein* can be given by the expression

```
mkN "Wein" "Weine" masculine
```

The RGL function `mkN` expands this expression to the record shown above. This expression is easier to write than the full record, but it can also be extracted automatically from some dictionary that contains nouns with their plural forms and genders.

Going to the level of complete queries, we can use the syntactic functions of the RGL API. The linearization function for `QWhatValue` can then be defined as

```
mkQCl (mkIComp what_IP) (mkNP the_Det (possessNP attr obj))
```

which hides all details about genders, cases, and prepositions. In fact, this expression is the same for all languages implementing the resource grammar API. Thus the grammar can be written as a **functor** over the resource grammar API used as **interface**, which makes the creation of multilingual grammars in GF and RGL productive: a grammar written for one language can be (almost) automatically ported to other ones.

Linearization functions can generate **variants**, that is, alternative ways of expressing the same abstract syntax. In a query language, it can be practical to allow both full sentences and their shorthands for queries. Thus the constructor `QWhatValue` can also allow a linearization dropping the words "what is", the definite article, the question mark, or any combination of these:

[What is] [the] colour of Malbec [?]

The parser returns the same abstract syntax tree for all of the eight resulting variants. Variants make it easier for the users to write input that can be interpreted by the system. Input is moreover helped by the predictive parsing functionality of the GF runtime, showing word completions compatible with the grammar.

The library and its language-independent API make grammar writing both productive and feasible for non-linguist programmers. A base grammar, with syntactic combination rules, is initially implemented and later completed by linguist programmers familiar with GF (on the manager level of WineSpeak). New expressions can later be added by expert users who know how to express wine concepts properly in their language and give minimal grammatical information, such as the gender of nouns.

5 Semantics

The semantics of natural language directed to a relational database can be expressed by a direct interpretation in relational algebra or, often more conveniently, as a translation to SQL. A common practice in semantics is to make the translation **compositional**. This means that the translation of a syntax tree is composed from the translations of its immediate subtrees—there is no need to look deeper into the subtrees.

As natural language and SQL are, at least superficially, very different, compositionality is an interesting challenge. The simplest way to implement compositional translation is by using GF [4]. In GF, linearization functions are compositional by design, as forced by the very syntax of the formalism. Just like with text templates, using simple context-free rules would not be powerful enough. A more scalable way is to use records that have separate fields corresponding to fields in SQL queries: `SELECT`, `FROM`, and `WHERE`. Then, for example, adding the adjective *Italian* to a wine description means adding a condition to the `WHERE` part, which may already contain other conditions.

At the time of writing, the semantics of WineSpeakis entirely based on linearization in GF. The advantage of the approach is that the overall system is simple and easy to extend, by only touching the grammar. But this requires rather complicated encoding and may be changed to, or replaced by, an external translation in a general purpose programming language. This will also make non-compositional translation possible, by using techniques studied in [12].

6 Data Sources

WineSpeakis intended to be a living source of data built and extended by the community. However, as a proof of concept, and also to attract users, we don't want to start with an empty shell. Thus we have used different sources to populate WineSpeakwith available open-source data. We have not used—and do not aim to use—commercial or otherwise proprietary data.

One obvious source is Wikipedia, which has comprehensive lists of established facts such as European wine regions and their *appellations* or *denominations*. The main challenge is to extract the data in a precise way from HTML-formatted text. There is no standard format even without single regions, let alone between countries, so heuristic algorithms followed by manual editing is the obvious method. The data is too scarce to support machine learning, but the scarcity also makes it feasible to use manual methods.

Translations of wine terms and names of regions into different languages is another task where Wikipedia helps, although in an even more fragmentary way than the base data, because different languages in Wikipedia are seldom in a clear correspondance. Thus we have also looked at specific terminology databases meant for translators. In both cases, the terms that are found must be equipped with grammatical information, so that they can be used correctly in all syntactic contexts. This is a task where the GF Resource Grammar Library is helpful—not as an automatic method but together with human interaction that we want to minimize. One promising technique is **Concept Alignment**, which uses machine-learned parsers together with GF-RGL to extract translation rules from parallel texts [8].

Concepts and facts collected in WineSpeakare a general resource, which can be used in other applications as well. A particular example we have in mind is the **Abstract Wikipedia**, an initiative to generate Wikipedia articles from formalized data [13]. The data is formalized as RDF triples (subject-predicate-object formulas) in WikiData [14], from where it can be organized into more complex sentences and texts by using abstract syntax functions of GF. It has turned out that the current coverage of wine data is very fragmentary in WikiData, and one of the intended by-products of the WineSpeakproject is to help this situation by making both data, abstract syntax functions, and linearization rules available for Abstract Wikipedia.

7 Related Work

Natural language interfaces to databases are an old topic, dating back at least to the LUNAR system about moon stones [15]. The goal is to bridge between formal and informal language, where WineSpeakuses essentially an approach similar to [6], which became a part of the KeY system [2]. Similar approaches have been used for information systems in different domains and formalisms, for instance, [1,3–5].

8 Conclusion

What we have presented above is the general structure of a system, for which we have built a basic implementation in which all parts are in place: database, grammar, and web interface. We are continuously extending it with more data, with the goal of launching the system and publishing its source code will be in connection to the publication of this volume.

Acknowledgements. I am grateful to the editors for their cooperative spirit and to the anonymous referees for insightful suggestions.

References

1. Angelov, K., Enache, R.: Typeful ontologies with direct multilingual verbalization. In: Rosner, M., Fuchs, N.E. (eds.) CNL 2010. LNCS (LNAI), vol. 7175, pp. 1–20. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31175-8_1
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
3. Dannélls, D., Damova, M., Enache, R., Chechev, M.: Multilingual online generation from semantic web ontologies. In: Proceedings of the 21st International Conference on World Wide Web, pp. 239–242. ACM (2012)
4. Davallius, D.: Natural-SQL translator a general natural language interface to SQL using the grammatical framework programming language. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden (2021). https://odr.chalmers.se/handle/20.500.12380/303909
5. Davis, B., Enache, R., van Grondelle, J., Pretorius, L.: Multilingual verbalisation of modular ontologies using GF and *lemon*. In: Kuhn, T., Fuchs, N.E. (eds.) CNL 2012. LNCS (LNAI), vol. 7427, pp. 167–184. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32612-7_12
6. Hähnle, R., Johannisson, K., Ranta, A.: An authoring tool for informal and formal requirements specifications. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 233–248. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_16
7. Johnson, H.: The Story of Wine: From Noah to Now. Academie du Vin Library, Ascot, Berkshire (1989)
8. Masciolini, A., Ranta, A.: Grammar-based concept alignment for domain-specific machine translation. In: Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21) (2021). https://aclanthology.org/2021.cnl-1.2.pdf
9. Bostock, J., Riley, H.T.: Pliny the Elder: The Natural History of Pliny, vol. III. H. G. Bohn, London (1855). https://www.gutenberg.org/files/59131/59131-h/59131-h.htm
10. Ranta, A.: The GF resource grammar library. Linguistic Issues in Language Technology 2 (2009)
11. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011)

12. Ranta, A.: Translating between language and logic: what is easy and what is difficult. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23, pp. 5–25. Springer, Berlin Heidelberg (2011)
13. Vrandečić, D.: Building a multilingual wikipedia, 64(4), 38–41 (2021). https://cacm.acm.org/magazines/2021/4/251343-building-a-multilingual-wikipedia/fulltext
14. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489
15. Woods, W.A.: Progress in natural language understanding: an application to lunar geology. In: Proceedings of the June 4–8, 1973, National Computer Conference and Exposition, pp. 441–450. AFIPS 1973, ACM (1973)

Software & System Verification with KIV

Gerhard Schellhorn, Stefan Bodenmüller(✉), Martin Bitterlich, and Wolfgang Reif

Institute for Software and Systems Engineering, University of Augsburg, Augsburg, Germany
{schellhorn,stefan.bodenmueller,martin.bitterlich, reif}@informatik.uni-augsburg.de

Abstract. This paper gives an overview of the KIV system, which in its long history has evolved from a prover for sequential programs using Dynamic Logic to a general purpose theorem prover. Today, KIV's main focus is the refinement-based development of sequential and concurrent software systems. In this paper we describe KIV's logic, highlighting recent developments such as support for polymorphism and for exceptions in programs. We show its proof engineering support that uses a graphical user interface and explicit proof trees, as well as KIV's support for the development of large-scale software systems using modular components and for the verification of concurrent algorithms using a rely-guarantee calculus. Finally, we give a short survey over the case studies that have been conducted with KIV.

Keywords: Formal Methods · Interactive Theorem Proving · Polymorphic Higher-Order Logic · wp Calculus · Rely Guarantee Calculus

1 Introduction

KIV was originally developed in the 80's by Maritta Heisel, Wolfgang Reif and Werner Stephan at the chair of Prof. Menzel in Karlsruhe [25]. The original focus was on developing proof support for the verification and synthesis of sequential programs using Dynamic Logic [17]. Underlying the work was the development of a specific functional "proof programming language" (PPL), that replaced LISP's basic data structure of s-expressions with proof trees.

Reiner Hähnle was one of the first students involved in the implementation of PPL (that used an instance of Cardelli's SECD machine [9] to interpret PPL) and the realization of first deduction concepts. He co-authored [19]. This work later influenced the design of the Key System [1], which also uses a version of Dynamic Logic to verify Java programs.

Since then KIV has evolved to a general-purpose theorem prover, however, still with a focus on developing verified software. KIV supports the refinement-based development of sequential as well as concurrent software systems. The logic

Partly supported by the Deutsche Forschungsgemeinschaft (DFG), "Verifikation von Flash-Dateisystemen" (grants RE828/13-1 and RE828/13-2).

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 408–436, 2022.
https://doi.org/10.1007/978-3-031-08166-8_20

has been extended to a higher-order temporal logic and recently polymorphism has been added. The programming language now has exceptions and a code generator supports generating Scala as well as C-Code.

The implementation language of KIV also changed a few years ago: KIV is now programmed entirely in Scala [43].

This paper gives an overview of the current concepts supported in KIV. It is organized as follows. Section 2 introduces polymorphic higher-order logic, which is the basis of our specification language. Exemplary specifications of the free data type of lists and the non-free data type of heaps with separation formulas are given.

Section 3 describes the core features of KIV's proof engineering. Explicit proof trees are used, that can be saved and manipulated, AC rewriting and heuristics are used to automate proving theorems. Interaction takes place via a graphical interface that allows context-sensitive rewriting by clicking on subexpressions.

Section 4 introduces KIV's program logic for sequential, abstract programs. In contrast to many other theorem provers, which use (some variant of) higher-order logic only and embed programs as data structures, KIV always had programs as a native concept together with a calculus for symbolic execution of Dynamic Logic and wp-calculus formulas. The section then introduces KIV's concept of components, subcomponents, and refinement. Components are also called abstract state machines (ASMs) since they are close to ASMs as defined in [8]. Abstractly, a component can be viewed as an instance of an abstract data type, i.e. a collection of operations working on a common state. When operations are called sequentially, then refinement is essentially data refinement with the contract approach [12].

Section Sect. 5 introduces KIV's basic concepts for the verification of concurrent systems. This calculus has evolved over time. It started as a calculus that views programs as formulas of interval temporal logic (ITL; an extension of LTL) and is able to prove arbitrary temporal logic properties. Rely-Guarantee (RG) formulas were initially defined as abbreviations [55]. They became native concepts later on, with calculus rules that are stream-lined to the verification of partial and total correctness of concurrent programs.

KIV also implements extensions of the component concept to concurrency. Such components may either have concurrent internal threads (e.g. Garbage Collection) or offer a thread-safe interface such that operations can be called by several threads in parallel. They are proved to be linearizable [26] and deadlock-free using proof obligations from RG calculus. A discussion of concurrent components and the proof obligations that are necessary for their correctness is beyond the scope of this paper, the interested reader should look at [4,52].

Section 6 gives an overview of some medium- to large-sized case studies that have been verified with KIV. Finally, Sect. 7 gives some ongoing work and concludes.

2 Basic Logic and Structured Specifications

The basic logic of the KIV system is higher order logic (HOL), recently extended from monomorphic to polymorphic types. The definition of the set of types Ty is based on a finite set of type constructors Tc with fixed arity l (we write $tc{:}l \in Tc$) and a countable set of type variables $'a \in Tv$. It is assumed that the type constant $bool$ is predefined, i.e. $bool{:}0 \in Tc$. We will usually leave the arity of type constructors implicit when writing types. Hence, a type $ty \in Ty$ is an application of a type constructor, a type variable, a tuple type, or a n-nary function type

$$ty \ := tc{:}l(\underline{ty}) \mid {'a} \mid (\underline{ty}) \mid \underline{ty} \to ty'$$

where $\underline{ty}$ denotes a sequence $ty_1 \times \ldots \times ty_n$ of types (we will use underlining to denote sequences in general). The sequence must have l elements in the application of a type constructor, at least two elements for tuples, and at least one element for function types.

Expressions $e \in Expr$ are defined over a set of (typed) variables $x{:}ty \in X$ and a signature $\Sigma = (Tc, Op)$ which in addition to type constructors contains (typed) operations $\mathtt{op}{:}ty \in Op$. Op always includes the usual boolean operations like $\mathtt{true} : bool$, $\neg\ . : bool \to bool$ (written prefix), or $. \wedge . : bool \times bool \to bool$ (written infix), equality $. = . :\ 'a \times\ 'a \to bool$, an if-then-else-operator $\supset$: $bool \times\ 'a \times\ 'a \to\ 'a$, as well as tuple constructors (written $(e_1, \ldots, e_n)$) and tuple selectors (written e.g. $e.\_3$). The basic set of higher-order expressions, which will be extended in Sect. 4 and Sect. 5, is defined by the grammar

$$e \ := x \ \mid \ op \mid \ e_0(\underline{e}) \ \mid \ \lambda\ \underline{x}.\ e \ \mid \ \forall\ \underline{x}.\ \varphi \ \mid \ \exists\ \underline{x}.\ \varphi$$

Here, φ denotes a formula, i.e. an expression of type bool, and the variables of $\underline{x}$ must be pairwise disjoint. In the following we will also use t to denote terms, which are expressions without quantifiers, and ε to denote quantifier-free formulas, which are used e.g. for conditions of programs (see Sect. 4.1). The typing rules are standard, e.g. in an application the type of e_0 must be a function type, where the argument types are equal to those of $\underline{e}$. Operations are allowed to be used with an instantiated type in theorems, but not in axioms (definitions). Most types can be inferred by type inference, so we will leave the types of variables and the instance types of operations implicit in formulas. Application of the if-then-else-operator is written as $(\varphi \supset e_1; e_2)$. Its result is e_1 if φ is true and e_2 otherwise.

The semantics of an expression $[\![e]\!]$ essentially follows the semantics of HOL defined in [18]. It is based on algebras $\mathcal{A} = (\mathcal{U}, \{tc{:}l^{\mathcal{A}}\}, \{\mathtt{op}{:}ty^{\mathcal{A}}\})$.

The first component of an algebra is the universe $\mathcal{U}$, which is a set of non-empty (potential) carrier sets. The semantics $tc{:}l^{\mathcal{A}} : \mathcal{U}^l \to \mathcal{U}$ of a type constructor maps the carrier sets of its argument types to the one of the full type. The semantics of booleans, functions and tuples is standard, i.e. $\mathcal{U}$ is assumed to contain the set $\{\mathbf{tt}, \mathbf{ff}\}$ that interprets booleans, and to be closed against forming cartesian products and functions to interpret function and tuple types. Given a type valuation $w : Tv \to \mathcal{U}$, that maps each type variable to a carrier set, an algebra fixes the semantics $[\![ty]\!](\mathcal{A}, w)$ of a type as one of the carrier sets in $\mathcal{U}$.

```
data specification
  using nat

data types
  list('a) = [] | . + . (. .head : 'a; . .tail : list('a)) ;

variables
  x, y, z : list('a);
  a : 'a;

size functions # . : list('a) → nat;
order predicates . < . : list('a) × list('a);

end data specification
```

Fig. 1. KIV specification of the generic free data type *list*.

The interpretation $\mathtt{op}{:}ty^{\mathcal{A}}$ of an operation over an algebra then yields an element of $[\![ty]\!](\mathcal{A}, w)$ for every possible type valuation w. Finally, the semantics of an expression $[\![e]\!](\mathcal{A}, w, v)$ refers to an algebra $\mathcal{A}$, to interpret type constructors and operations, a type valuation w, and a valuation v (compatible with w) that maps each variable $x : ty$ to an element of $[\![ty]\!](\mathcal{A}, w)$. The result of $[\![e]\!](\mathcal{A}, w, v)$ is an element of the carrier set $[\![ty]\!](\mathcal{A}, w)$. We write $\mathcal{A}, w, v \models \varphi$ when the semantics of a formula evaluates to **tt**. An algebra is a model of an axiom φ ($\mathcal{A} \models \varphi$), iff the formula evaluates to true for all w and v.

In the following we are interested in valuations v that are used as (the changing) states of programs, while algebra $\mathcal{A}$ and type valuation w are fixed. In this case we often drop these arguments and just write $[\![e]\!](v)$.

2.1 Structured Specifications of Algebraic Data Types

In KIV we use structured algebraic specifications to build a hierarchy of data type definitions, which may be generated freely or non-freely. Such data type definition specifications can be augmented by additional functions and can be combined using the usual structuring operations like enrichment, union, and renaming. KIV also supports a generic instantiation concept. It allows to replace an arbitrary subspecification P (the "parameter") of a generic specification G with an actual specification A using a *mapping*. A mapping is a generalized morphism that renames types and operations of P to types and expressions over A. Such an instantiation generates proof obligations that require to prove that the axioms of P instantiated by the mapping are theorems over A.

An example of a free data type specification is given in Fig. 1 for lists. A list *list*('a) is defined using a constant constructor `[]` (representing the empty list) and a non-constant infix constructor `+`. Non-empty lists consist of an head element of generic type $'a$ and and a remaining tail list. These fields can be accessed via the selector functions `.head` and `.tail`, respectively.

For a free data type specification KIV generates all necessary axioms: the constructor functions are injective, different constructor functions yield different results, selectors and update functions (written e.g. x`.head:=` *newhead*) get definitions. Selector (and update) functions are not given axioms for all arguments: `[].head` is left unspecified (as is `[].tail`). The semantic function in a model then is still a total function, and `[].head` may be any value, following the standard loose approach to semantics. However, for use in programs, KIV attaches a *domain* to the function, here given as $\lambda\ x.\ x \neq$ `[]`. Calling `.head` outside of its domain in a program (here: with `[]`, where it is "undefined") will raise an exception, explained in detail in Sect. 4.2.

A *size function* (`#` x counts the number of non-constant constructors in x, i.e. it calculates list length) and an *order predicate* ($x < y$ iff x is a suffix of y) can be specified, for which axioms are generated as well. Note that we abbreviate functions with result type *bool* by omitting the result type and declaring them as *predicates*.

2.2 Modelling the Heap and Separation Logic

The specification of non-free data types requires more effort as axioms cannot be generated automatically. For example, we use the polymorphic non-free data type shown in Fig. 2 to reason about pointer structures in the heap. A heap can be considered as a partial function mapping references r to objects *obj* of a generic type $'a$, where allocation of references is explicit. As stated by the "induction" clause, the $heap('a)$ data type is inductively generated by the constant $\emptyset$ representing the empty heap, by allocating an new reference r (written h `++` r), or by updating an allocated location r with a new object *obj* (written $h[r, obj]$).

For a non-free data type one has to give an *extensionality axiom*: two heaps are considered to be equal if they have allocated the same locations and they store the same objects under their allocated locations. This definition requires two additional operations that also have to be axiomatized. A predicate $r \in h$ is defined for checking whether a reference is allocated in a heap and a function $h[r]$ is used for looking up objects in the heap (this corresponds to dereferencing a pointer). References can also be deallocated by the function h `--` r.

The constructor functions as well as lookup and deallocation are declared as partial functions in order to specify valid accesses to the heap. This requires to give domains for the functions (see lambda clauses in Fig. 2): Accesses to the heap with the `null` reference are always undefined ($r \neq$ `null`) and allocation is only allowed for new references ($\neg\ r \in h$). Lookups, updates, and deallocations are defined only for allocated references ($r \in h$).

```
generic specification
   parameter ref
   using nat
target

sorts heap('a);

constants ∅ : heap('a);

predicates
    . ∈ . : ref × heap('a);

variables
    h : heap('a);
    r : ref;
    obj : 'a;

partial functions
    . ++ .   : heap('a) × ref → heap('a)
               with λ h, r. r ≠ null ∧ ¬ r ∈ h;
    . -- .   : heap('a) × ref → heap('a)
               with λ h, r. r ≠ null ∧ r ∈ h;
    . [ . ] : heap('a) × ref → 'a
               with λ h, r. r ≠ null ∧ r ∈ h;
    . [ . ] : heap('a) × ref × 'a → heap('a)
               with λ h, r, obj. r ≠ null ∧ r ∈ h;

induction
    heap('a) generated by ∅, ++, [ ];

axioms

extensionality:
⊢ h0 = h1 ↔ ∀ r.   (r ∈ h0 ↔ r ∈ h1)
                 ∧ (r ∈ h0 → h0[r] = h1[r]);

;; ... definitions of ∈, [ ], -- ...

end generic specification
```

Fig. 2. KIV specification of the polymorphic non-free data type $heap('a)$.

This explicit specification of the heap is necessary since in KIV all parameters of procedures are explicit. Hence, when reasoning about pointer-based programs, like a pointer-based implementation of red-black trees, the heap must be an explicit parameter of the program as well. To facilitate the verification of such programs we built a library for Separation Logic (SL) [48] in KIV. SL formulas are encoded using heap predicates $hP : heap('a) \rightarrow bool$. A heap predicate describes the structure of a heap h. At its simplest, h is the empty heap `emp`:

$$\vdash \texttt{emp}(h) \leftrightarrow h = \emptyset$$

The maplet $r \mapsto obj$ describes a singleton heap, containing only one reference r mapping to an object obj. It is defined as a higher-order function of type $(ref \times 'a) \rightarrow heap('a) \rightarrow bool$:

$$\vdash (r \mapsto obj)(h) \leftrightarrow h = (\emptyset \texttt{ ++ } r)[r, obj] \wedge r \neq \texttt{null}$$

More complex heaps can be described using the separating conjunction $hP_0 * hP_1$ asserting that the heap consists of two disjoint parts, one satisfying hP_0 and one satisfying hP_1, respectively. Since it connects two heap predicates, it is defined as a function with type $(heap('a) \rightarrow bool) \times (heap('a) \rightarrow bool) \rightarrow (heap('a) \rightarrow bool)$:

$$\vdash (hP_0 * hP_1)(h) \leftrightarrow \exists\, h_0, h_1.\ h_0 \perp h_1 \wedge h = h_0 \cup h_1 \wedge hP_0(h_0) \wedge hP_1(h_1)$$

Besides the basic SL definitions, the KIV library contains various abstractions of commonly used pointer data structures like singly-/doubly-linked list or binary trees. These abstractions allow to prove the functional correctness (incl. memory safety) of algorithms on pointer structures against their algebraic counterparts.

3 Proof Engineering

Proof engineering (in analogy to software engineering) is the process of developing a verified software system. Since the goal is mechanized verification, the tool support a verification system can provide for speeding up the process is very important. The process includes various complex and time-consuming tasks. Structured specifications containing axioms and definitions must be set up or (better) reused from a library, properties must be formalized, and finally proved. Many revisions are necessary when proofs fail or definitions are found to be inadequate. Only a small part of the effort is verifying the final theorems with the correct axioms, where flexible features to automate proofs are crucial to avoid repeating the same interactive proof steps over and over. Most of the effort is spent restructuring specifications, revising axioms and theorems, and then particularly correcting proofs that become invalidated by these changes, so that maintaining a large lemma base is a critical factor in developing a verified system.

One key building block of KIV to address these challenges is the Graphical User Interface (GUI), which provides intuitive and interactive support for the proof engineer. Algebraic specifications and theorems can be managed via a graphical representation of the specification hierarchy (see Sect. 3.1). Theorems are proved semi-interactively in the GUI, where proof automation techniques like heuristics or the automatic rewriting of expressions support the user (see Sect. 3.2). Proofs are visualized explicitly as *proof trees* that give insight into every proof step and offer direct manipulation of the proof (see Sect. 3.3).

The KIV system is publicly available as an IDE plugin developed in Scala. More information about the setup can be found at [28].

3.1 Management of Specifications and Proofs

The basis of the development is a hierarchy of specifications that can contain data type definitions, components (see Sect. 4), and structuring operations (union, actualization, enrichment, etc.). This hierarchy is called the *development graph* and shown graphically. It is the starting point when developing a software project in KIV. A specification (a node in the graph) has a *theorem base* attached that contains axioms and theorems (some of them may be proof obligations) together with their proofs, as far as they have been done.

A specification can be in different *states*, depending on whether it has just been *created*, is *imported* from a library, or is currently *valid*. Changing a subspecification may yield an *invalid* specification that must be fixed, e.g. when some signature symbols used in axioms has changed. When all theorems over a specification have been successfully proved, it can be put in *proved state*, which asserts that all lemmas used in proofs that are from subspecifications are still present in *some* subspecification. Deferring this check for entering proved state allows to avoid invalidating proofs on restructuring specifications. It also avoids the need to enforce a bottom-up strategy for proving theorems. In general, the *correctness management* of KIV always minimizes the number of proofs that are invalidated when specifications or theorems are changed. When all specifications are in proved state, the correctness management guarantees that all theorems are proved and do not have cyclic dependencies.

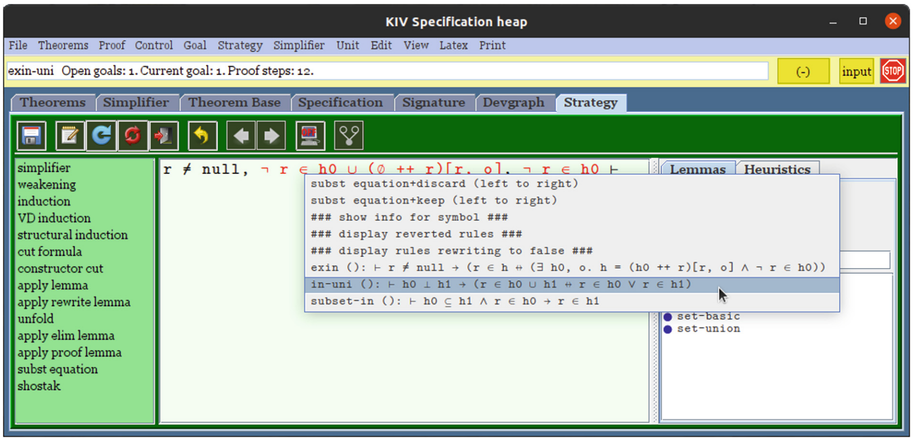
The development graph also offers access to work on a particular specification. When selecting a specification, the theorem base is loaded and an overview is shown that gives information about its axioms, and its proved, unproved, and invalid lemmas. At this level, theorems can be added, edited, and deleted. Axioms and theorems can be declared as *simplifier rules*, which are then used automatically in proofs (see the following section). For the theorems of the specification, new proofs can be started, existing proofs can be deleted, and partial proofs can be continued. KIV projects can be combined by importing (parts of) other projects and using them as a library. The standard library included in the KIV plugin provides specifications for common data structures like lists, arrays, sets, and maps. Other libraries define support for separation logic (see Sect. 2.2), or provide basic locking constructs (mutexes, reader-writer locks, and conditions) to support concurrency. Providing an exhaustive library also has the benefit of adding many useful simplifier rules to the theorem base, such that verification effort will be reduced drastically.

Finally, the KIV application offers general features, like exporting projects to an HTML representation (presentations of many projects can be found at [30]) or viewing proof statistics for projects or individual specifications.

3.2 Proving Theorems

Proving theorems in KIV is done using a sequent-based calculus. A sequent $\Gamma \vdash \Delta$ abbreviates the formula $\forall \underline{x}. \bigwedge \Gamma \rightarrow \bigvee \Delta$ where Γ (the antecedent) and Δ (the succedent) are lists of formulas and $\underline{x}$ is the list of all free variables in Δ and Γ. The rules of the calculus follow the structure of the formulas and are applied backwards in order to reduce the conclusion to simpler premises, until they can be closed using axioms.

KIV facilitates this process for proving theorems by using extensive automation combined with interactive steps performed by a proof engineer. Therefore the interface must help the proof engineer to understand the automated steps and to identify possible actions. The interface for proving consists of two parts: one for the proof tree (see Sect. 3.3) and another for performing proof interactions. The latter is the most important part of the GUI and is shown in Fig. 3. The large area in the middle contains the sequent of the current goal, i.e. what currently has to be proven. The left-hand side shows the list of calculus rules that can be applied to the goal. A status bar under the menu bar provides information about the proof: the name of the lemma, the number of open goals, the number of the current goal, and the total number of proof steps.



exin-uni Open goals: 1. Current goal: 1. Proof steps: 12.

```
r ≠ null, ¬ r ∈ h0 ∪ (∅ ++ r)[r, o], ¬ r ∈ h0 ⊢
subst equation+discard (left to right)
subst equation+keep (left to right)
### show info for symbol ###
### display reverted rules ###
### display rules rewriting to false ###
exin (): ⊢ r ≠ null → (r ∈ h ↔ (∃ h0, o. h = (h0 ++ r)[r, o] ∧ ¬ r ∈ h0))
in-uni (): ⊢ h0 ⊥ h1 → (r ∈ h0 ∪ h1 ↔ r ∈ h0 ∨ r ∈ h1)
subset-in (): ⊢ h0 ⊆ h1 ∧ r ∈ h0 → r ∈ h1
```

Fig. 3. Sequent of a proof goal with context sensitive choice.

The proof engineer must choose the next proof step, if no rules are applicable by the automation. This is usually done context-sensitively, as shown in Fig. 3. When the user moves the mouse over the sequent, the KIV system permanently determines the smallest (sub-)expression and highlights it. Right-clicking on a formula opens a context menu with applicable rules. If the leading symbol is a logical connective, the rules for dealing with this connective are offered, e.g. "quantifier instantiation" for quantifiers. If the leading symbol is a predicate, a list of applicable rules is shown.

Figure 3 shows the result of a click on the predicate $\in$ in the expression $\neg\ r \in h0 \cup (\emptyset \mathrel{++} r)[r, obj]$ in the antecedent. It shows the three rewrite lemmas matching on the selected part of the expression out of the hundreds of lemmas in the theorem base. When the user selects the rewrite lemma *in-uni*: $\vdash h0 \perp h1 \rightarrow (r \in h0 \cup h1 \leftrightarrow r \in h0 \vee r \in h1)$ from the context menu, all occurrences of the selected expression are rewritten to $\neg r \in h0 \vee r \in (\emptyset \mathrel{++} r)[r, obj]$. The precondition generates a side goal where one has to prove $h0 \perp (\emptyset \mathrel{++} r)[r, obj]$. The context-sensitive computation of applicable lemmas restricts the theorem base to the relevant cases, so that even very large verification projects remain manageable.

Besides applying rules manually, KIV offers different levels of automation: the user can switch between different sets of heuristics that automatically apply calculus rules. The base heuristic that is used in every proof is the *simplifier* The simplifier rewrites the current goal by applying rules and lemmas that simplify the sequent. This includes applying all the propositional rules of sequent calculus with at most one premise as well as the two rules that eliminate quantifiers (universal quantifier right and existential quantifier left). The simplifier also applies simplifier rules that are lemmas following the pattern of *conditional rewrite rules* and of *forward rules* if enabled by the user: Rewrite rules replace terms with simpler terms, forward rules are typically used to add transitive information to the goal. Application of these rules is performed modulo associativity and commutativity, the strategy is currently enhanced to include matches modulo neutral elements as well. Additionally, specialized heuristics are available, depending on the content of the sequent. E.g., for sequents containing programs (see Sect. 4), different heuristics performing *symbolic execution* can be selected. Furthermore, KIV offers heuristics that call the SMT solvers Z3 [10] and CVC4 [2].

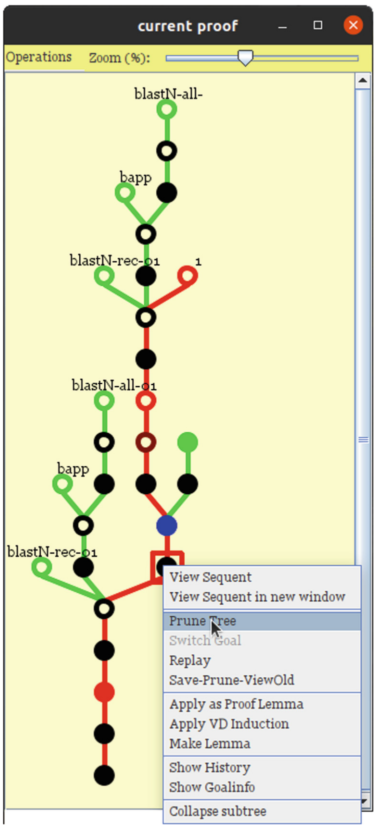


Fig. 4. Example of a proof tree.

3.3 Proof Trees

A proof tree like the one shown in Fig. 4 is a graphical representation that shows the applied rules in a particular proof step. Typically a proof tree consists of 20 up to 300 steps. For every step of the proof, the current sequent, the applied proof rule, used lemmas and simplifier rules, the used substitution for the variable instantiation, the active heuristics, and a lot of internal information are stored and accessible. Characteristic proof steps are easy to identify, similarities between different parts of the proof can be recognized, and the proof can be saved in any state and continued later. If a premise of a proof cannot be closed, the proof tree allows to easily trace back the problem to its cause, e.g. a problematic branch of a program present in the original conclusion.

The nodes of the proof tree have different forms and colors. Filled circles represent steps performed by a heuristic; open circles represent interactive steps. The color indicates the different kinds of steps. For instance, induction steps are red. The edges of the proof also have different colors. Edges leading to an incomplete subtree are red, the edges of completely proved subtrees are green. At the nodes representing the interactive usage of a lemma, the name of the lemma is added to the node.

Clicking on nodes allows to view information about the goal or to inspect which simplifier rules have been used by the simplifier. When a premise is not provable, tracing through the branch is regularly used to identify where or why something went wrong.

Incorrect decisions can be undone by standard chronological backtracking. But since the incorrect step is usually not the last one, it is typically done by pruning the proof tree, i.e. removing the complete subtree above a selected node. Often fixing the incorrect step allows to successfully reapply the steps of the subtree (without its incorrect first step) that has been pruned away, since the approach for proving the branch is not affected by the fix.

Such a reapplication is directly supported in KIV with the *replay* functionality. The mechanism allows the user to reuse (parts of) proofs, even if the goal has changed, using a sophisticated strategy to adjust proof steps. Thus, replaying is also a powerful tool for re-validating invalid proofs: if a proof becomes invalid due to a minor change in one (or some) of its used theorems, most of the time, the old proof can be replayed automatically (the same applies if the theorem goal itself has changed).

All in all, KIV offers helpful features to support the proof engineer in the process of developing specifications and proofs. Among them are a direct interaction with the proof structure through proof trees, assistance for automation using heuristics, and context-sensitive rule selection. Finally, KIV's correctness management supports maintaining many proofs split into theorem bases for many specifications. More detailed information on the KIV GUI can be found in [20].

4 Components, Refinement and Program Logic

4.1 Sequential Programs

KIV features an imperative programming language with recursive procedures. Recently, the programming language was extended by constructs for exception handling. The syntax of sequential programs in KIV is given by the grammar

$$\begin{aligned}\alpha := \quad & \textbf{skip} \mid \textbf{abort} \mid \underline{x} := \underline{t} \mid \alpha_1; \alpha_2 \mid \textbf{if } \varepsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \\ & \mid \textbf{let } \underline{x} = \underline{t} \textbf{ in } \alpha \mid \textbf{choose } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha_1 \textbf{ ifnone } \alpha_2 \mid \alpha_1 \textbf{ or } \alpha_2 \\ & \mid \textbf{while } \varepsilon \textbf{ do } \alpha \mid \textbf{proc\#}(\underline{t}; \underline{x}; \underline{y}) \mid \textbf{throw } \mathtt{op} \mid \textbf{try } \alpha \textbf{ catch } \underline{\eta}\end{aligned}$$

The program **skip** does nothing and **abort** is the program that never terminates. Assignments $\underline{x} := \underline{t}$ assign each variable x_i the value of term t_i simultaneously. This allows for swaps of the form $x, y := y, x$ in a single statement with no additional variables necessary. Two programs α_1 and α_2 can be executed in order by sequential composition (;). The **if-then-else** and **while** programs function as usual.

The **let** program is used to introduce new variables $\underline{x}$ initialized with values $\underline{t}$. **choose** also introduces new local variables $\underline{x}$ but with a random valuation that satisfies a condition φ. If such a valuation can be found, the program α_1 is executed using these variables. If no values that satisfy φ exist, the program α_2 is executed. The **or** program makes an nondeterministic choice between two programs α_1 and α_2.

For **if**-programs the **else**-branch can be omitted if α_2 is the program **skip**. Similarly, for **choose**-programs α_2 can be dropped if it is the program **abort**. A '**with** true' clause in a **choose** can be omitted too.

A procedure **proc#** can be called with input arguments $\underline{t}$, reference arguments $\underline{x}$, and output arguments $\underline{y}$ by the statement **proc#**$(\underline{t}; \underline{x}; \underline{y})$, where the different argument types are separated by semicolons. Input arguments may be arbitrary terms while only variables are allowed to be passed as reference and output arguments. Typically, a procedure has a declaration of the form **proc#**$(\underline{x}; \underline{y}; \underline{z})\{\alpha\}$ with disjoint formal parameters $\underline{x}$, $\underline{y}$, and $\underline{z}$. α must set all output parameters $\underline{z}$, only the input parameters $\underline{x}$ and the reference parameters $\underline{y}$ can be read in α. Updates to $\underline{x}$ and $\underline{y}$ are allowed, however, updates to $\underline{x}$ are invisible to the caller of **proc#**.

The recently added **try-catch** and **throw** constructs enable exception handling in KIV programs. A program statement may now raise an exception if a partial function (see Sect. 2) is applied to arguments outside of its domain. Each exception is thus coupled with an operation `op`. For example, the subtraction operation $-$ on natural numbers throws its exception in the program $m := n_0 - n_1$ when $n_0 < n_1$. Additionally, the exception of operation `op` can be thrown explicitly by the **throw** program. **try-catch** blocks can be used to catch exceptions within a program α by giving exception handlers

$$\underline{\eta} \equiv \textbf{case } \mathtt{op}_1 :: \alpha_1; \ \ldots\ ; \ \textbf{case } \mathtt{op}_n :: \alpha_n; \ \textbf{default} :: \alpha_{default}$$

$\underline{\eta}$ can contain exception handlers for any number of operations $\texttt{op}_1, \ldots, \texttt{op}_n$ as well as an **default** exception handler (both are optional). If the leading statement of α throws an exception $\texttt{op}_i$ for which a matching handler **case** $\texttt{op}_i :: \alpha_i \in \underline{\eta}$ exists, the remaining statements of α are discarded and α_i is executed. If no matching handler exists but there is a **default** handler, then $\alpha_{default}$ is executed, otherwise the exception is not caught.

A small-step semantics of concurrent programs is explained in Sect. 5. For sequential programs the semantics can be abstracted to a relation $[\![\alpha]\!] \subseteq ST \times ST \times (Op \cup \{\top\})$ between states ST (valuation of program variables), augmented with Op and $\top$ to express termination with or without an exception. $(v, v', \zeta) \in [\![\alpha]\!]$ iff there is a terminating execution of the program α starting in state $v \in ST$ and finishing in state $v' \in ST$, without raising an exception (if $\zeta = \top$) or finishing with an exception $\zeta \in Op$.

4.2 Weakest Precondition Calculus with Exceptions in KIV

Reasoning about sequential programs in KIV is done with a weakest-precondition calculus, borrowing notation from Dynamic Logic (DL) [24] including its two standard modalities: formula $[\alpha]\varphi$ (*box*) denotes that for every terminating run the final state must satisfy φ, corresponding to the weakest liberal precondition $wlp(\alpha, \varphi)$. The formula $\langle\alpha\rangle\varphi$ (*diamond*) guarantees that there is a terminating execution of α that establishes φ. Finally, the formula $\langle|\alpha|\rangle\ \varphi$ (*strong diamond*) states that the program α is guaranteed to terminate and that all final states reached satisfy φ. This is equivalent to the weakest precondition $wp(\alpha, \varphi)$. For deterministic programs the two formulas $\langle\alpha\rangle\varphi$ and $\langle|\alpha|\rangle\ \varphi$ are equivalent. As a sequent, partial and total correctness of α with respect to pre-/post-conditions $Pre/Post$ are written as $Pre \vdash [\alpha]Post$ and $Pre \vdash \langle|\alpha|\rangle\ Post$. To handle exceptions the modalities are extended by exception specifications

$$\underline{\xi} \equiv \texttt{op}_1 :: \varphi_1,\ \ldots\ ,\ \texttt{op}_k :: \varphi_k,\ \textbf{default} :: \varphi_{default}$$

which yields program formulas of the form $\langle|\alpha|\rangle\ (\varphi\ ;\ \underline{\xi})$, $\langle\alpha\rangle(\varphi\ ;\ \underline{\xi})$, and $[\alpha](\varphi\ ;\ \underline{\xi})$, respectively. The exception specifications allow to give additional postconditions for executions that terminate with a specific exception, e.g. φ_1 must hold if α terminates with exception $\texttt{op}_1$, or to give a generic exception postcondition $\varphi_{default}$ for executions of α that terminate with an exception $\texttt{op} \notin \texttt{op}_1, \ldots, \texttt{op}_k$. If one wants to show the absence of exceptions, the exception specifications $\underline{\xi} \equiv$ **default** :: $\texttt{false}$ can be chosen, which is the default and omitted from program formulas.

The main proof technique for verifying program correctness in KIV is *symbolic execution*. Basically, a symbolic execution proof step executes the first statement of the program and calculates the strongest postconditions from the preconditions. When the symbolic execution of the program is completed, the proof of the postcondition is performed purely in predicate logic. In the following we will present an excerpt of the calculus rules for program formulas in KIV, with a focus on the recent addition of exceptions and exception handling.

$$\frac{\Gamma_{\underline{x}}^{\underline{x}'},\ \delta(\underline{t}_{\underline{x}}^{\underline{x}'}),\ \underline{x} = \underline{t}_{\underline{x}}^{\underline{x}'} \vdash \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta_{\underline{x}}^{\underline{x}'} \quad \text{where } \underline{x}' \text{ fresh} \qquad Exc(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})}{\Gamma\ \vdash \langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 5. Calculus rule for assignments.

Figure 5 shows the rule for parallel assignments for total correctness (the rule is identical for the other modalities). The rule uses a vector $\underline{x}'$ of fresh variables to store the values of $\underline{x}$ before the assignment. The assignment is removed and instead the formula $\underline{x} = \underline{t}_{\underline{x}}^{\underline{x}'}$ is added to the antecedent ($\underline{t}_{\underline{x}}^{\underline{x}'}$ denotes the renaming of $\underline{x}$ to $\underline{x}'$ in $\underline{t}$). Note that renaming is possible on all program formulas, while substitution of $\underline{x}$ by general terms $\underline{t}$ in $\langle\!|\alpha|\!\rangle\,\varphi$ is not possible and just yields $\langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,\varphi$. Only when the assignment is the last statement of the program and α is missing, the standard premise of Hoare calculus, which replaces the program formula in the premise with $\varphi_{\underline{x}}^{\underline{t}}$, can be used.

Since the expressions $\underline{t}$ can contain partial operations, this is correct only when the evaluation of $\underline{t}$ does not raise any exception: we use $\delta(\underline{e})$ to describe the condition that the expressions $\underline{e}$ are defined, i.e. that all partial operations in $\underline{e}$ are applied to arguments within their respective domain. For example, a heap constructor term $(h \texttt{ ++ } r_0)[r_1, obj]$ (see Sect. 2.2) produces the definedness condition

$$\delta((h \texttt{ ++ } r_0)[r_1, obj]) \ \equiv\ (r_0 \neq \texttt{null} \wedge \neg\, r_0 \in h) \ \wedge\ (r_1 \neq \texttt{null} \wedge r_1 \in (h \texttt{ ++ } r_0))$$

Exception premises $Exc(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})$ need to be shown for all potential violations of $\delta(\underline{t})$, i.e. for all partial operations $\texttt{op}_1, \ldots, \texttt{op}_m$ in $\underline{t}$, *throw premises* are generated. For each partial operation an exception condition is calculated following a bottom-up approach: the exception is thrown if and only if an application of $\texttt{op}_i$ violates the domain of $\texttt{op}_i$ and the evaluation of its arguments does not throw an exception. E.g. the assignment $h := (h \texttt{ ++ } r_0)[r_1, obj]$ would yield the two exception premises

$$\begin{aligned}
&Exc(\Gamma, \Delta, (h \texttt{ ++ } r_0)[r_1, obj], \varphi, \underline{\xi}) \equiv\\
&\quad (1)\ \Gamma \vdash r_0 = \texttt{null} \ \vee\ r_0 \in h \rightarrow \langle\!|\textbf{throw}\texttt{ ++ }|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta\\
&\quad (2)\ \Gamma \vdash (r_0 \neq \texttt{null} \ \wedge\ \neg\, r_0 \in h) \ \wedge\ (r_1 = \texttt{null} \ \vee\ \neg\, r_1 \in (h \texttt{ ++ } r_0))\\
&\qquad\qquad \rightarrow \langle\!|\textbf{throw}\texttt{ [ ]}|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta
\end{aligned}$$

Computing exception clauses uses the standard left-to-right strategy for evaluating arguments, and a shortcut semantics for boolean connectives (like Java and Scala). The test $y \neq 0 \wedge x/y = 1$ of a conditional will never throw an exception, while switching the conjunction will throw the division exception, when $y = 0$.

The rules for **throw** are quite simple. If an operation is thrown for which a specific exception specification $\texttt{op} :: \varphi_\xi$ is given in $\underline{\xi}$, the program formula is discarded and replaced by the exception postcondition φ_ξ. If there is no exception specification in $\underline{\xi}$ for the thrown operation $\texttt{op}$, the default exception postcondition $\varphi_{default}$ must hold.

$$\frac{\Gamma \vdash \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}'),\ \Delta}{\Gamma \vdash \langle\!|\mathbf{try}\ \alpha\ \mathbf{catch}\ \{\ \underline{\eta}\ \}|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 6. Calculus rule for try-catch programs.

$$\frac{\begin{array}{l}(1)\ \Gamma \vdash INV,\ \Delta\\ (2)\ Exc(INV, \langle\rangle, \varepsilon, \varphi, \underline{\xi})\\ (3)\ INV,\ \varepsilon,\ \delta(\varepsilon),\ z = t \vdash \langle\!|\alpha|\!\rangle\,(INV \wedge t \ll z;\ \underline{\xi})\\ (4)\ INV,\ \neg\,\varepsilon,\ \delta(\varepsilon) \vdash \varphi\end{array}}{\Gamma \vdash \langle\!|\mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 7. Invariant rule for while loops.

The **try-catch**-rule shown in Fig. 6 takes advantage of the fact that exception postconditions can also contain program formulas. The program α can be executed symbolically with adjusted exception specifications $\underline{\xi}'$ that include the exception handling of $\underline{\eta}$. $\underline{\xi}'$ contains exception specifications for all operations that have either already a specification in $\underline{\xi}$ or have a case in $\underline{\eta}$. For an operation `op` the adjusted exception postcondition φ'_ξ is defined as

$$\varphi'_\xi = \begin{cases} \langle\!|\alpha'|\!\rangle\ (\varphi\ ;\ \underline{\xi}) & \text{if } \mathbf{case}\ \mathtt{op} :: \alpha' \in \underline{\eta} \\ \varphi_\xi & \text{if } \mathtt{op} \notin \underline{\eta} \wedge \mathtt{op} :: \varphi_\xi \in \underline{\xi} \end{cases}$$

If an exception handler was given for `op`, φ'_ξ is set to a program formula with the exception handler program α' and the original postcondition φ and exception specifications $\underline{\xi}$. This has exactly the desired effect: if an `op` exception is thrown within α, symbolic execution continues with α' while the postcondition is still φ. For operations without a case in $\underline{\eta}$, the exception postcondition remains unchanged ($\varphi'_\xi = \varphi_\xi$). The adjusted default exception postcondition $\varphi'_{default}$ is constructed similarly. If there is a default exception handler **default** $:: \alpha_{default} \in \underline{\eta}$, a program formula with $\alpha_{default}$ is built, otherwise the default postcondition $\varphi_{default}$ from $\underline{\xi}$ is used.

Proofs about recursive procedures are typically done using (well-founded) induction. For **while**-loops, typically the invariant rule shown in Fig. 7 is used, though the more general induction rule occasionally leads to simpler proofs (see e.g. the proof online at [30] for Challenge 3 at the VerifyThis2012 competition).

The rule requires an invariant formula INV as an input from the user from which multiple premises need to be proven: the invariant must hold at the beginning of the loop (1), INV must be stable over the loop body α (3), and INV must be strong enough to prove the postcondition φ after the loop was exited (4). Similar to the assignment rule, exception premises are generated for the loop condition ε (2). These cannot include Γ or Δ as ε is evaluated again after each iteration. Instead, the invariant INV can be assumed as well as $\delta(\varepsilon)$ for premises (3) and (4). The figure shows the rule for total correctness which requires to give a variant t that decreases with every iteration of the loop ($t \ll z$ in premise (3)). The rule functions analogously for diamond formulas, for partial correctness no decreasing variant is necessary.

4.3 Hierarchical Components and Refinement

For the development of complex software systems in KIV we use the concept of hierarchical components combined with the contract approach to data refinement [12]. A component is an abstract data type $(ST, \mathtt{Init}, (\mathtt{Op}_j)_{j \in J})$ consisting of a set of states ST, a set of initial states $\mathtt{Init} \subseteq ST$, and a set of operations $\mathtt{Op}_j \subseteq In_j \times ST \times ST \times Out_j$. An operation $\mathtt{Op}_j$ takes inputs In_i and outputs Out_j and modifies the state of the component. Operations are specified with contracts using the operational approach of ASMs [6]: for an operation $\mathtt{Op}_j$, we give a precondition pre_j and a program α_j (that establishes the postcondition of the operation) in the form of a procedure declaration $\mathbf{op}_j\#(in_j; st; out_j)$ **pre** pre_j $\{\alpha_j\}$. Instead of defining initial states directly, we also give a procedure declaration $\mathbf{init}\#(in_{init}; st; out_{init})$ $\{\alpha_{init}\}$ where in_{init} are initialization parameters, that determine the intial state, and out_{init} is an error code, that may indicate that initialization with these parameters failed.

Components are distinguished between specifications and implementations. The former are used to model the functional requirements of a (sub-)system and are typically kept as simple as possible by heavily utilizing algebraic functions and nondeterminism. The approach is as general as specifying with pre- and postconditions, since **choose** st', out' **with** $post(st', out')$ **in** $st, out := st', out'$ can be used to establish any postcondition $post$ over state st and output out. Implementations are typically deterministic and only use constructs that allow to generate executable Scala- or C-code from them with our code generator.

The functional correctness of implementation components is then proven by a data refinement of the corresponding specification components (we write $\mathtt{C} \leq \mathtt{A}$ if $\mathtt{C} = (ST^{\mathtt{C}}, \mathtt{Init}^{\mathtt{C}}, (\mathtt{Op}_i^{\mathtt{C}})_{j \in J})$ is a refinement of $\mathtt{A} = (ST^{\mathtt{A}}, \mathtt{Init}^{\mathtt{A}}, (\mathtt{Op}_i^{\mathtt{A}})_{j \in J})$ where $\mathtt{C}$ and $\mathtt{A}$ have the same set of operations J). Proofs for such a refinement are done with a forward simulation $R \subseteq ST^{\mathtt{A}} \times ST^{\mathtt{C}}$ using commuting diagrams. This results in correctness proof obligations for all $j \in J$ (an extra obligation ensures that $\mathtt{Init}^{\mathtt{A}}$ and $\mathtt{Init}^{\mathtt{C}}$ establish matching states).

$$\begin{aligned}&R(st^{\mathtt{A}}, st^{\mathtt{C}}),\ pre_j^{\mathtt{A}}(st^{\mathtt{A}})\\ &\vdash \langle\!|\mathbf{op}_j^{\mathtt{C}}\#(in_j; st^{\mathtt{C}}; out_j)|\!\rangle\ \langle \mathbf{op}_j^{\mathtt{A}}\#(in_j; st^{\mathtt{A}}; out_j')\rangle(R(st^{\mathtt{A}}, st^{\mathtt{C}}) \wedge out_j = out_j')\end{aligned}$$

Informally, one has to prove that, when starting in R-related states, for each run of an operation $\mathbf{op}_j^{\mathtt{C}}\#$ of $\mathtt{C}$ there must be a matching run of $\mathbf{op}_j^{\mathtt{A}}\#$ of $\mathtt{A}$ that maintains $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ with the same inputs and outputs. The obligations also require to show that the precondition $pre_j^{\mathtt{A}}(st^{\mathtt{A}})$ is strong enough to establish the precondition $pre_j^{\mathtt{C}}(st^{\mathtt{C}})$ if $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ holds. This obligation is implicit as the call rule creates this premise for a procedure with a precondition.

For each component invariant formulas $inv(st)$ over the state st can be given, which must be maintained by all $(\mathtt{Op}_j)_{j \in J}$. This simplifies (or even makes it possible in the first place) to prove the correctness proof obligations of a refinement as invariants $inv^{\mathtt{A}}(st^{\mathtt{A}})$ and $inv^{\mathtt{C}}(st^{\mathtt{C}})$ are added as assumptions. If an invariant is given for a component, additional proof obligations for all its operations are generated that ensure that the invariant holds. Additionally, one can give an individual postcondition $post_j(st)$ for an operation which extends its invariant contract.

$$pre_j(st),\ inv(st) \vdash \langle\!|\mathbf{op}_j\#(in_j; st; out_j)|\!\rangle\ (inv(st) \wedge post_j(st))$$

These invariant contracts can be applied when proving the refinement proof obligations and may further simplify the proofs since symbolic execution of the operation can be avoided. The theory has also been extended with proof obligations for crash-safety, see [16] for more details.

To facilitate the development of larger systems, we introduced a concept of modularization in the form of *subcomponents*. A component (usually an implementation) can use one or more components as subcomponents (usually specifications). The client component cannot access the state of its subcomponents directly but only via calls to the interface operations of the subcomponents. Using subcomponents, a refinement hierarchy is composed by multiple refinements like the one shown in Fig. 8.

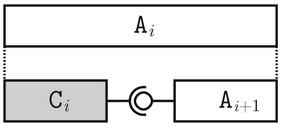


Fig. 8. Data refinement with subcomponents.

A specification component $\mathtt{A}_i$ is refined by an implementation $\mathtt{C}_i$ (dotted lines in Fig. 8) that uses a specification $\mathtt{A}_{i+1}$ as a subcomponent (—◎— in Fig. 8, we write $\mathtt{C}_i(\mathtt{A}_{i+1})$ for this subcomponent relation). This pattern then repeats in the sense that $\mathtt{A}_{i+1}$ is refined further by an implementation $\mathtt{C}_{i+1}$ that again uses a subcomponent $\mathtt{A}_{i+2}$ and so on. $\mathtt{A}_i$ may also be used as a subcomponent of an implementation $\mathtt{C}_{i-1}$ if it is not the toplevel specification. The complete implementation of the system then results from composing all individual implementation components $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...)))$. In [16] we have shown that $\mathtt{C} \leq \mathtt{A}$ implies $\mathtt{M(C)} \leq \mathtt{M(A)}$ for a client component $\mathtt{M}$ which ensures that the composed implementation is in fact a correct refinement of its toplevel specification $\mathtt{A}_0$, i.e. $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...))) \leq \mathtt{A}_0$. This allows us to divide a complex refinement task into multiple, more manageable ones.

5 Concurrency, Temporal Logic and Rely-Guarantee Calculus

5.1 Concurrent Programs and Their Semantics

KIV supports concurrency in the form of weak fair and non-fair interleaving of sequential programs. Concurrent programs β extend the syntax of sequential ones (α) by the following constructs

$$\begin{aligned}\beta := \quad & \alpha \mid \mathbf{if}*\ \varepsilon\ \mathbf{then}\ \beta_1\ \mathbf{else}\ \beta_2 \mid \mathbf{atomic}\ \varepsilon\ \{\beta\} \mid \beta_1 \parallel_{\text{nf}} \beta_2 \mid \beta_1 \parallel \beta_2 \\ & \mid \mathbf{forall} \parallel \underline{x}\ \mathbf{with}\ \varphi\ \mathbf{do}\ \beta \mid \mathbf{forall} \parallel_{\text{nf}} \underline{x}\ \mathbf{with}\ \varphi\ \mathbf{do}\ \beta \mid \psi\end{aligned}$$

Programs execute atomic steps, consisting of one assignment, the evaluation of the test of a condition, binding a variable with **let** or calling/returning from a procedure. To enable thread-local reasoning, they are now assumed to execute their steps in an environment that may modify the state in between program steps. The environment may consist of other interleaved programs or a global environment, e.g. a physical environment that changes sensor values read by the program.

There are several new constructs that may be freely mixed with the sequential constructs. The first is the variant **if**$*$ of a conditional, which evaluates the test together with the first step of the branch taken in one atomic step. It is used to model test-and-set, or CAS (compare-and-swap) instructions.

The construct **atomic** ε $\{\beta\}$ is assumed to (passively) wait for the environment to make the test ε true. While ε is false, the program is *blocked*. A program where the environment never enables the test is deadlocked. When the test becomes true, the program β is executed in a single step. The typical use of the construct is to model locking with **atomic** $\mathit{lock} = \texttt{free}$ $\{\mathit{lock} := \texttt{locked}\}$. Atomic programs are also used in Lipton's reduction strategy [14,36], which proves that program instructions can be combined to larger atomic blocks when specific commutativity conditions hold.

Both $\beta_1 \parallel_{\mathrm{nf}} \beta_2$ and $\beta_1 \parallel \beta_2$ interleave steps of β_1 and β_2 non-deterministically. The interleaving is blocked only, when both programs are blocked. The first assumes no fairness, so when β_1 does not terminate and is never blocked, one possible run executes steps of β_1 only. The second $\beta_1 \parallel \beta_2$ has a weak fairness constraint: if β_2 is enabled continuously (i.e. is never blocked) then it will eventually execute a step, even if β_1 is always enabled. Weak fairness is typically assumed for programs which use locks while CAS-based programs often ensure the progress condition of lock-freedom that does not assume fairness.

The programs **forall**$\parallel$ and **forall**$\parallel_{\mathrm{nf}}$ generalize binary interleaving to interleaving instances of β for all values that satisfy φ bound to local variables $\underline{x}$. The typical use would be **forall**$\parallel$ n **with** $n < m$ **do** β where the body β uses variable n as the thread identifier. Earlier versions of RG calculus in KIV used an equivalent recursive **spawn#**-program (called with m and the variables $\underline{x}$ used in β) which is defined as

$$\textbf{spawn\#}(n; \underline{x})\{\ \textbf{if}*\ n = 0\ \textbf{then skip else}\ \{\ \beta \parallel \textbf{spawn\#}(n-1; \underline{x})\ \}\ \}$$

They imported a theory which verified the correctness of the forall-rule given at the end of the next section.

Finally, since the semantics of programs which will be explained next and the semantics of temporal formulas ψ explained in Sect. 5.2 agree—both are a set of intervals—it is possible to use a formula in place of a program.

Formally, the semantics of programs $[\![\beta]\!]$ is defined as a set of finite or infinite state sequences also called *intervals* following the terminology of interval temporal logic (ITL) [41]. Formally an interval is of the form

$$I = (I(0), I(0)_b, I'(0), I(1), I(1)_b, I'(1), I(2), \ldots, \zeta)$$

where every state $I(k)$ and $I'(k)$ is a valuation which maps variables to values. A finite interval ends with an unprimed state. The transitions from $I(k)$ to $I'(k)$

are program transitions and the transitions from $I'(k)$ to $I(k+1)$ from a primed to the subsequent unprimed state are environment transitions. Hence, intervals alternate between program and environment transitions, similar to the reactive sequence semantics in [11].

To model passive waiting, the boolean flag $I(k)_b$ denotes whether the program transition from $I(k)$ to $I'(k)$ is *blocked*, i.e. the program waits to continue its execution. In this case, when $I(k)_b = \mathbf{tt}$, then $I'(k) = I(k)$.

To model exceptions when running a program, the final state of a finite run carries information ζ whether an exception has happened. This may be either $\top$ to indicate regular termination without exception or the information that an operation $op \in OP$ has thrown its exception. To have uniform notation, we assume that $\zeta = \infty$ for infinite (non-terminating) runs, so $\zeta \in OP \cup \{\top, \infty\}$.

The semantics of programs is *compositional*, i.e. the semantics of complex programs can be constructed by combining intervals that are members of the semantics of its parts. This has been explained in detail for programs without exceptions in [55]. Adding exceptions for the sequential case is straightforward, for interleaving the combined run ends with an exception if one of the interleaved intervals does. Unlike in programming languages, where exceptions are thread-local, we therefore have *global* exceptions, which abort the whole interleaved program. If necessary, the global effect must be avoided by exception handlers in the interleaved programs.

5.2 Temporal and Program Formulas

To verify concurrent programs, KIV's logic is based on the idea of having *programs as formulas*. To do this, the semantics of expressions e is generalized from $[\![e]\!](v)$ to $[\![e]\!](I)$ using an interval I instead of a single state v. The expressions considered so far refer to the initial valuation $I(0)$ of the interval only. The extended semantics makes it possible to view a program β with free variables $\underline{z}$ as a formula $[: \underline{z} \mid \beta]$ (expression with boolean result) which returns $\mathbf{tt}$ iff the semantics of β includes the interval I. That β has a temporal property ψ is then simply expressed as the implication $[: \underline{z} \mid \beta] \rightarrow \varphi$.

The resulting calculus is again based on symbolic execution of programs as well as of temporal formulas. The resulting calculus has been described in detail in [55]. The calculus is strong enough to define rely-guarantee formulas as abbreviations of temporal logic formulas and to formally derive the rules of rely-guarantee calculus. Since rely-guarantee calculus is the one we predominantly use in the practical verification of programs, we have explicitly added rely-guarantee formulas and derived rules for them.

The extended expression language partitions variables into flexible variables, that may be modified by concurrent programs, and static ones, that always have the same value in all states $I(0)$, $I'(0)$, ... of an interval. In KIV, flexible variables start with uppercase letters, all others are static. The abstract syntax definition uses y to denote a static and z for a flexible variable. Flexible variables are allowed in quantifiers, but not as parameters of λ-expressions.

The extended language allows to use primed and double primed flexible variables y' and y'' in predicate logic expressions. $[\![y']\!](I)$ and $[\![y'']\!](I)$ are defined as $I'(0)(y)$ and in $I(1)(y)$, except for the case where the interval consists of a single state. For such an *empty* interval the value of both is $I(0)(y)$. Formulas like $y' = y$ or $y'' \geq y'$ therefore talk about the relation of the first program step (y is not changed) and about the first environment step (y is not decremented). They are used as guarantee and rely formulas that constrain program and environment steps. We use G to denote a predicate logic formula over unprimed and primed variables and R to denote one over primed and double primed variables. We write φ' and φ'' for predicate logic formulas where one resp. two extra primes are added to every free variable that is flexible.

The extended syntax then extends higher-order and wp-calculus formulas that may use any primed or double primed variables written χ (φ still denotes a formula without primed variables) to temporal expressions ψ, including program expressions. These have the following syntax:

$$\begin{aligned}\psi := \chi \mid\ [: \underline{z} \mid \beta] \mid\ \Box\, \psi \mid \Diamond\, \psi \mid \psi_1 \ \mathbf{until}\ \psi_2 \mid\ \mathbf{last} \mid\ \mathbf{lastexc} \mid\ \mathbf{lastexc}(\mathbf{op})\\ \mid\ \mathbf{blocked} \mid\ [: \underline{z} \mid R, G, \varphi, \beta](\psi; \underline{\xi}) \mid\ \langle: \underline{z} \mid R, G, \mathit{Runs}, \varphi, \beta\rangle(\psi; \underline{\xi})\end{aligned}$$

A formal semantics is given in [55], here we just give an informal explanation.

- $[: \underline{z} \mid \beta]$ is a program formula, where the used variables of β are required to be a subset of $\underline{z}$. The formula holds over an interval (i.e. the semantics evaluates to **tt**) if the program steps are steps of the program. The environment steps between program steps are not constrained: they may arbitrarily change the values of variables. The frame assumption $\underline{z}$ indicates that non-program variables can change arbitrarily in program steps.
- $\Box\, \psi$, $\Diamond\, \psi$ and ψ_1 **until** ψ_2 are the standard formulas of linear temporal logic.
- **last** is true, if the interval is empty, i.e. consists of just the state $I(0)$.
- **lastexc(op)** resp. **lastexc** is true on an empty interval where $\zeta = \mathit{op}$ resp. $\zeta \neq \top$. The formula $\Diamond(\mathbf{last} \wedge \neg\ \mathbf{lastexc})$ therefore states, that the interval is finite (terminates) without exception.
- **blocked** is true for non-empty intervals, where the first step is blocked, i.e. $I(0)_b = \mathbf{tt}$. The formula $\Box$ **blocked** is used to express a deadlock.

The last two formulas are used to express rely-guarantee (RG) properties of programs, and we assume the reader to be familiar with the basic ideas. In [60] such an RG assertion would be written as β $\underline{\textbf{sat}}$ $\{\mathit{Pre}, R, G, \mathit{Post}\}$, another common notation is as an extended Hoare-quintuple $\{\mathit{Pre}, R\}\ \beta\ \{G, \mathit{Post}\}$. Often, whether partial or total correctness is intended, depends on the context. In KIV, partial and total correctness are expressed as a sequents

$$\mathit{Pre} \vdash [: \underline{z}\ |, R, G, \beta]\mathit{Post} \qquad\qquad \mathit{Pre} \vdash \langle: \underline{z} \mid R, G, \mathit{Runs}, \alpha\rangle \mathit{Post}$$

Like for the wp-formulas we leave away the default exception condition $\xi \equiv$ **default** :: `false` that forbids any exceptions for the final state.

Partial correctness asserts that if precondition Pre holds in the initial state, then program steps will not violate the guarantee G and the final state will not violate $Post$ (and have no exception) unless an earlier environment step violated R. The formula implies that partial correctness holds: if all environment steps are rely steps then all program steps will be guarantee steps and in final states the postcondition will hold.

Total correctness guarantees two additional properties. First, when the rely is never violated, then the program is guaranteed to terminate. Second, in all states, where predicate $Runs$ holds, the next program step is guaranteed not to be blocked. The additional $Runs$-predicate is used to verify deadlock-freedom: when an interleaved program satisfies total correctness with $Runs = \texttt{true}$, then the program is deadlock-free: there is always at least one of its threads that is not currently waiting to acquire a lock.

At the end of this section we want to note that the interval semantics of programs can be used to define the semantics of wp-calculus formulas as well by abstracting to initial and final states of the interval and assuming an "empty" environment that does not change program variables via the equivalences

$$[\alpha]\varphi \equiv [: \underline{z} \mid \underline{z}'' = \underline{z}', \texttt{true}, \alpha]\varphi \qquad \langle\!|\alpha|\!\rangle\, \varphi \equiv \langle : \underline{z} \mid \underline{z}'' = \underline{z}', \texttt{true}, \alpha\rangle\varphi$$

In the formulas, $\underline{z}$ are the flexible variables that occur free in α (all variables, except those bound by **let**, **choose** or **forall**) or free in φ.

5.3 Rely-Guarantee Calculus

Since rely-guarantee formulas can be viewed as abbreviations of temporal formulas, we could just use the calculus defined in [55] for deduction (and earlier case studies have done so). Since rely-guarantee formulas are now available directly, it is unnecessary for a user to get familiar with temporal logic formulas. Knowledge of the calculus given in this section is sufficient to do partial or total correctness proofs for concurrent programs. The embedding into temporal logic is useful however to prove results about progress conditions such as lock-freedom [58] or starvation-freedom [57], which would not be possible with pure RG calculus only.

Symbolic execution using the rely-guarantee calculus resembles symbolic execution in wp-calculus. The extra effort needed when proving RG formulas can be seen when looking at the rule for assignment:

$$\frac{\begin{array}{c} Pre(\underline{y}_0), \underline{y}_1 = \underline{t}(\underline{y}_0) \vdash G(\underline{y}_0, \underline{y}_1) \\ Pre(\underline{y}_0), \underline{y}_1 = \underline{t}(\underline{y}_0), R(\underline{y}_1, \underline{z}) \vdash \langle : \underline{z} \mid R, G, \beta\rangle Post \end{array}}{Pre(\underline{z}) \vdash \langle : \underline{z} \mid R(\underline{z}', \underline{z}''), G(\underline{z}, \underline{z}''), \underline{z} := \underline{t}(\underline{z}); \beta\rangle Post}$$

For easier notation the rule assumes that all variables of the frame assumption are assigned. We also leave away the side conditions for possible exceptions when evaluating $\underline{t}$. These are the same as in wp-calculus. The rule makes explicit that Pre and t may contain variables from $\underline{z}$ by writing them as arguments in the conclusion. Analogously, R and G may depend on $\underline{z}$, $\underline{z'}$, and $\underline{z''}$.

In the semantics, symbolic execution of the assignment reduces the interval $I = (I(0), I(0)b, I'(0), I(1), \ldots)$ to the one shorter interval $(I(1), \ldots)$. The values of variables $\underline{z}$ in states $I(0)$ and $I'(0)$ are now stored in two vectors $\underline{y}_0$ and $\underline{y}_1$ of fresh static variables, the remaining program β again starts with the values of the variables in $\underline{z}$. The old precondition now holds for $\underline{y}_0$, the values stored in $\underline{y}_1$ are equal to the ones of the terms $\underline{t}(\underline{y}_0)$. As the assignment rule shows, the RG calculus has two differences to wp-calculus. First there is an additonal premise that asserts that executing the assignment satisfies the guarantee (which is simple usually). Second, the main premise needs two vectors of fresh variables instead of one to store old values: one before the assignment and one after the assignment but before the environment step.

The other rules of RG-calculus (e.g. the invariant rules for partial and total correctness) look very similar to wp-calculus. The only difference is again that a premise is generated that ensures that the step is a guarantee step. Finally, we need a rule for parallel composition. We give the rule for one flexible variable u, the generalization to several variables should be obvious.

The rule assumes that a lemma for the body β is available, that is applied by the rule and shown as its first premise. The precondition Pre of the goal can depend on $\underline{z}$, and we write Pre($\underline{y}$) for the assertion, where $\underline{z}$ has been replaced by $\underline{y}$. Similar conventions apply for the other formulas. E.g. Rely R_0 can depend on $u', \underline{z'}, u'', \underline{z''}$, so we write $R_0(u, \underline{y}_0, u, \underline{y}_1)$ for an instance. Formula φ may use u and $\underline{z}$. Since u is a local variable, the environment steps of β can not change its value, so the lemma can (and should) include the rely condition $u'' = u'$. Since the program is also prohibited from modifying variable u (otherwise it could not be used to identify the thread executing), instances of R_0 (and similarly: G_0) always use the same static variable v to instantiate u' and u''. The possible values for u are chosen in the inital state before the **forall** executes, using new static variables $\underline{y}$ for this initial state and static variables v, v_1, v_2 for the values that all satisfy $Pre(\underline{u}) \wedge \varphi(v, \underline{u})$.

$$\frac{\begin{array}{l}
(1)\ Pre_0 \vdash \langle : u, \underline{z} \mid R_0, G_0, Runs_0, \beta \rangle Post_0 \\
(2)\ Pre(\underline{y}), \varphi(v, \underline{y}) \vdash G_0(v, \underline{y}_0, v, \underline{y}_0) \\
(3)\ Pre(\underline{y}), \varphi(v, \underline{y}) \vdash R_0(v, \underline{y}_0, v, \underline{y}_1) \wedge R_0(v, \underline{y}_1, v, \underline{y}_2) \rightarrow R_0(v, \underline{y}_0, v, \underline{y}_2) \\
(4)\ Pre(\underline{y}), \varphi(v_1, \underline{y}), \varphi(v_2, \underline{y}), v_1 \neq v_2 \vdash G_0(v_1, \underline{y}_0, v_1, \underline{y}_1) \rightarrow R_0(v_2, \underline{y}_0, v_2, \underline{y}_1) \\
(5)\ Pre(\underline{y}), \varphi(v, \underline{y}) \vdash Pre_0(v, \underline{y}_0) \wedge R_0(v, \underline{y}_0, v, \underline{y}_1) \rightarrow Pre_0(v, \underline{y}_1) \\
(6)\ Pre(\underline{y}), \varphi(v, \underline{y}) \vdash Post_0(v, \underline{y}_0) \wedge R_0(v, \underline{y}_0, v, \underline{y}_1) \rightarrow Post_0(v, \underline{y}_1) \\
(7)\ Pre(\underline{y}) \vdash APre(\underline{y}) \\
(8)\ Pre(\underline{y}), Rely(y_0, y_1) \vdash AR(\underline{y}, \underline{y}_0, \underline{y}_1) \\
(9)\ Pre(\underline{y}), EG(\underline{y}, \underline{y}_0, \underline{y}_1) \vdash Guar(\underline{y}_0, \underline{y}_1) \\
(10)\ Pre(\underline{y}), Runs(\underline{y}_0) \vdash\ ERuns(\underline{y}, \underline{y}_0) \\
(11)\ Pre(\underline{y}), APost(\underline{y}, \underline{z}) \vdash \langle : \underline{z} \mid R, G, Runs, \beta_0 \rangle Post
\end{array}}{Pre \vdash \langle : \underline{z} \mid R, G, Runs, \{\textbf{forall} \| \ u \ \textbf{with} \ \varphi \ \textbf{do} \ \beta\}; \beta_0 \rangle Post}$$

Conditions (1) to (6) ensure that the forall satisfies

$$Pre(\underline{z}), APre(\underline{z}) \vdash \langle : \underline{z} \mid AR, EG, ERuns, \textbf{forall} \ldots \rangle APost$$

where

$$\begin{aligned}
APre(\underline{y}) &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow Pre_0(v, \underline{y}) \\
APost(\underline{y}, \underline{z}) &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow Post_0(v, \underline{z}) \\
EG(\underline{y}, \underline{z}, \underline{z'}) &\equiv \exists\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \wedge G_0(v, \underline{z}, v, \underline{z}') \\
ERuns(\underline{y}, \underline{z}) &\equiv \exists\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \wedge Runs_0(v, \underline{z}) \\
AR(\underline{y}, \underline{z'}, \underline{z}'') &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow R_0(v, \underline{z}', v, \underline{z}'')
\end{aligned}$$

The remaining conditions (7) to (11) ensure that the given predicates used in the conclusion are weaker (for rely, precondition, and runs) resp. stronger (for guarantee) than the most general ones defined above, and that the remaining program β_0 is correct with the established postcondition *APost* of the **forall**.

The rule for interleaving two programs $\beta_1 \parallel \beta_2$ is essentially the special case we get from the equivalent program **forall** $\parallel$ b **with** `true` **do if** $*$ b **then** β_1 **else** β_2 where b is a boolean variable. Two lemmas are now required, one for β_1 and one for β_2 (corresponding to the two boolean values).

6 Applications

This section gives a short overview over applications that have been modeled and verified using KIV. We start with a brief overview over historically important case studies on sequential systems, described in the next subsection. Case studies on concurrency are described in Sect. 6.2. Finally, Sect. 6.3 gives a brief overview over concepts used in the development of a verified file system for flash memory, developed in the Flashix project, which is by far the largest project we have tackled so far.

6.1 Overview

Initially, KIV had a focus on verifying single programs with Dynamic Logic, which is more flexible than just generating verification conditions from a standard Hoare-calculus [23] (see Challenge 3 of the VerifyThis competition 2012 in [15] for a recent example).

The focus however shifted early on to the development of modular, sequential software. An early concept there were algebraic modules [46], which were used to e.g. verify the correctness of Dynamic Hashtables or AVL trees [47]. Algebraic modules are stateless, and are nowadays integrated with the instantiation concept: a parameter P may be instantiated by placing a restriction and a congruence on the instance A. A simple example would be to use duplicate-free (restriction) lists, where the order of elements is ignored (congruence), to implement sets. This concept can be used to formally verify the consistency of non-free data type specifications.

Later on the focus shifted to the verification of components with state. The biggest case study in this area was the verification [51] of a compiler for Prolog, that compiles to the Warren Abstract Machine (WAM), formalizing the development in [7] as a hierarchy of a dozen refinements. The case study uses the complete theory of ASM refinement [50], which generalizes the theory of data refinement: the main correctness condition for a 1:1 diagram involving one abstract and one concrete operation (cf. Sect. 4.3) is generalized to using m:n diagrams, which are useful in particular for compiler verification.

Another area where component-based development was important, was the Mondex case study, which initially was about mechanizing a proof of a protocol for the secure transfer of money between Mondex smartcards [59]. This case study became a part of the development of a strategy of model-based development for security protocols in general, starting with a UML model and ending with verified Java Code. For Mondex the original single refinement was extended to three refinements that end with verified Java code, see [21] for an overview.

6.2 Case Studies on Concurrency

Research in this area has focused on the verification of efficient low-level implementations of components, in particular on CAS-based implementations, which are harder to verify and need different concepts for progress than standard lock-based implementations. The most complex proof using extensions of RG calculus to the thread-local verification of linearizability [26] and lock-freedom was a proof of Treiber's stack that used Hazard pointer with non-atomic reading and writing [39,56,58]. A number of other examples, e.g. one with fine-grained locking and formal proof obligations for starvation-freedom [57] can be found on [29]. The Web page also shows proofs for two of Cliff Jones' original examples that demonstrate the use of RG calculus: FINDP [22] (a parallel search over an array) and SIEVE [34] (parallel version of the Sieve of Erathostenes).

As part of the VerifyThis Competition 2021 [45], we used RG reasoning to verify ShearSort, a parallel sorting algorithm for matrices. The case study uses the **forall**$\|$ construct presented in Sect. 5.3 and also requires to solve the tricky problem of how to formalize the 0–1-principle, which is often used in informal proofs. A web presentation of the case study can be found at [32].

For some implementations thread-local verification of linearizability is difficult (including Herlihy and Wing's simple, but hard to verify queue implementation [26]), since their linearization points (LPs) are not fixed to specific steps of the thread itself, but are in steps of other threads and "potential": whether an LP has been passed depends on future execution. With colleagues we developed a complete approach [53] that uses a formalization of IO Automata [37] (using HOL only) in KIV that has recently been extended to an automatic translation of programs as steps of an IO automaton together with proof obligations generated from assertions [13].

The formalization of IO Automata provided one of the main motivations for adding polymorphism to KIV, IO-Automata are specified as data structures, with states and actions as parameters. Though the content of proofs does not change, using states and actions as type parameters instead of instantiating specifications (with e.g. “concrete” and “abstract” states for refinement, more instances are necessary to prove transitivity of refinement, or to define product automata) roughly halved the number of specifications required.

The most complex case study verified in this setting was the Elimination queue [40], an extension of the Michael-Scott queue [38] with an elimination array, that has very complex potential LPs. This case study, several others and a number of papers in this area can be found again online at [30].

Recently we also have looked at opacity [35] as the correctness criterion for STMs (implementations of software transactional memory, formalizing the theory as IO automaton refinement [3].

6.3 The Flashix Project

Flashix [4,54] has been proposed as a pilot project for Hoare's Grand Challenge [27] with the goal to develop and verify a realistic file system for flash memory. The project motivated many enhancements in KIV, especially in component modularization and refinement methodology (cf. Sect. 4.3). To tackle the complexity we decomposed the file system into a deep hierarchy of components connected by 11 refinements. We generate C and Scala code from this hierarchy (we currently work on the generation of Rust code), the generated C code is about 18 kLOC and can be integrated into the Linux kernel or run via the FUSE interface.

A major focus in the project was to ensure *crash-safety*, i.e. the dealing with arbitrary power cuts. In [16] we added crash behavior to the semantics of our components and proposed a verification methodology to prove crash-safety for modular refinement hierarchies.

The project required the use of our Separation Logic library (see Sect. 2.2) on multiple occasions. In the lower levels of Flashix, a verified pointer-based implementation of red-black trees is used (the KIV code and proofs can be found online [31]). There, we combined Separation Logic with KIV's concept of components and refinement (see Sect. 4.3) to “separate Separation Logic”: the proof is split into the verification of a version based on algebraic trees, where the properties of red-black trees are verified, and a separate refinement that shows that copying branches on modifications can be avoided by replacing the algebraic tree with a pointer structure, that is updated in place. Only the latter, simple refinement has to deal with pointer structures, aliasing, and the avoidance of memory leaks by using Separation Logic formulas. Similarly, this concept was applied to the top-level refinement of Flashix, where the abstract representation of the file system as an algebraic tree is broken down to linked file and directory nodes.

The last phase of the project was mainly about introducing performance-oriented features like caching and concurrency. We added different caches to the file system, proposed novel crash-safety criteria for cached file systems, extended our refinement approach with corresponding proof obligations, and proved that the crash-safety criteria apply to the Flashix file system (see [44] and [5]). To add concurrency to the existing (purely sequential) refinement hierarchy, another kind of refinement was introduced: *atomicity refinement* [52] allows to incrementally reduce lock-based concurrent implementations to their sequential counterparts. This allowed to reuse large parts of the sequential proof work, in particular complex data refinements. The generation of proof obligations for atomicity refinements was implemented in KIV, they are based on rely-guarantee formulas (cf. Sect. 5.3), ownerships, and Lipton reductions [14,36]. With this methodology, Flashix now allows concurrent calls to its toplevel interface and features concurrent Wear Leveling [52] as well as concurrent Garbage Collection [4].

7 Conclusion

This paper has given an overview of the concepts that are used in KIV to model and verify software systems. A lot of the support for software development was overhauled since [15] was published in 2014, starting with switching to Scala as KIV's implementation language. The basic logic has been extended to support polymorphism and exceptions in programs, rely-guarantee calculus has been revised to have native formulas and rules in favor of using abbreviations and general temporal rules. Support for components and for proof obligations has been significantly enhanced, and a code generator has been added that supports generation of Scala- and C-code.

Many of these developments have been motivated by the requirements to develop a realistic, concurrent file system for flash memory in the Flashix project. As an example, adding exceptions has uncovered several errors, that would have gone unnoticed with the standard unspecified values semantics that is used in HOL, and that would also have been likely escaped testing, as they happened only on certain combinations of rare hardware errors (one was e.g. in formatting a flash device). Others, like adding assertions to programs or the forall-rule, have been motivated by the wish to be able to do smaller experiments, e.g. the challenges of the VerifyThis competition series, more quickly than before.

There is still lots of room for improvement though. Support for automating separation logic proofs is still far less developed than in native separation logic provers like Verifast [33] or Viper [42]. There is still lots of potential to optimize code generation from KIVs programs, using a careful dataflow analysis, where destructive updates and aliasing can be allowed in the generated C-code. The programming language is still (like Java) based on having programs ("statements") and expressions separately. We currently work on extending the language to have program expressions, where programs are the special case of program expressions of type unit. In the semantics this will generalize the void result $\zeta = \top$ on regular termination to a value of any type. This will allow methods with results and returns, and the use of Scala syntax for KIV programs (using Scalameta [49] for parsing). This will hopefully lead to increased expressiveness as well as an easier learning curve.

Acknowledgement. We would like to thank our student Kilian Kotelewski who has added the forall-construct and its rules to the calculus.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Bila, E., Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Defining and verifying durable opacity: correctness for persistent software transactional memory. In: Gotsman, A., Sokolova, A. (eds.) FORTE 2020. LNCS, vol. 12136, pp. 39–58. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50086-3_3
4. Bodenmüller, S., Schellhorn, G., Bitterlich, M., Reif, W.: Flashix: modular verification of a concurrent and crash-safe flash file system. In: Raschke, A., Riccobene, E., Schewe, K.-D. (eds.) Logic, Computation and Rigorous Methods. LNCS, vol. 12750, pp. 239–265. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_14
5. Bodenmüller, S., Schellhorn, G., Reif, W.: Modular integration of crashsafe caching into a verified virtual file system switch. In: Dongol, B., Troubitsyna, E. (eds.) IFM 2020. LNCS, vol. 12546, pp. 218–236. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_12
6. Börger, E.: The ASM refinement method. Formal Aspects Comput. **15**(1–2), 237–257 (2003)
7. Börger, E., Rosenzweig, D.: The WAM–definition and compiler correctness. In: Logic Programming: Formal Methods and Practical Applications. Studies in Computer Science and Artificial Intelligence, vol. 11, pp. 20–90. Elsevier (1995)
8. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis, Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
9. Cardelli, L.: The functional abstract machine. AT&T Bell Laboratories Technical Report. Technical report, TR-107 (1983)
10. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
11. de Roever, W.P., et al.:. Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press (2001)
12. Derrick, J., Boiten, E.: Refinement in Z and in Object-Z: Foundations and Advanced Applications. FACIT. Springer, Cham (2001). Second, revised edition 2014
13. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures: a sound and complete method. Formal Aspects Comput. **33**(4), 547–573 (2021)
14. Elmas, T., Qadeer, S., Tasiran, S.: A Calculus of atomic actions. In: Proceeding POPL 2009, pp. 2–15. ACM (2009)
15. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: overview and VerifyThis competition. Int. J. Softw. Tools Technol. Transf. **17**(6), 677–694 (2015)

16. Ernst, G., Pfähler, J., Schellhorn, G., Reif, W.: Modular, crash-safe refinement for ASMs with submachines. Sci. Comput. Program. **131**, 3–21 (2016). Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014)
17. Goldblatt, R.: Axiomatising the Logic of Computer Programming. LNCS, vol. 130. Springer, Berlin (1982). https://doi.org/10.1007/BFb0022481
18. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
19. Hähnle, R., Heisel, M., Reif, W., Stephan, W.: An interactive verification system based on dynamic logic. In: Siekmann, J.H. (ed.) CADE 1986. LNCS, vol. 230, pp. 306–315. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16780-3_99
20. Haneberg, D., et al.: The user interface of the KIV verification system – a system description. In: Proceedings of UITP 2005 (2005)
21. Haneberg, D., Moebius, N., Reif, W., Schellhorn, G., Stenzel, K.: Mondex: engineering a provable secure electronic purse. Int. J. Softw. Inform. **5**(1), 159–184 (2011). http://www.ijsi.org
22. Hayes, I.J., Jones, C.B., Colvin, R.J.: Laws and semantics for rely-guarantee refinement. Technical report CS-TR-1425, Newcastle University (2014)
23. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In: Morik, K. (ed.) GWAI-87 11th German Workshop on Artifical Intelligence, vol. 152, pp. 201–210. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-642-73005-4_22
24. Heisel, M., Reif, W., Stephan, W.: A dynamic logic for program verification. In: Meyer, A.R., Taitslin, M.A. (eds.) Logic at Botik 1989. LNCS, vol. 363, pp. 134–145. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51237-3_12
25. Heisel, M., Reif, W., Stephan, W.: Tactical theorem proving in program verification. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 117–131. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52885-7_83
26. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
27. Hoare, T.: The verifying compiler: a grand challenge for computing research. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 262–272. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36579-6_19
28. Institut für Software & Systems Engineering - Universität Augsburg. Introduction and Setup of KIV. https://www.uni-augsburg.de/en/fakultaet/fai/isse/software/kiv/
29. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of Starvation Freedom. https://kiv.isse.de/projects/Starvation-Free.html
30. Institut für Software & Systems Engineering - Universität Augsburg. Web Presentation of KIV Projects. https://kiv.isse.de/projects/
31. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of Red-Black Trees (2021). https://kiv.isse.de/projects/RBtree.html
32. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of ShearSort (2021). https://kiv.isse.de/projects/shearsort.html
33. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. NASA Formal Methods **6617**, 41–55 (2011)
34. Jones, C.B., Hayes, I.J., Colvin, R.J.: Balancing expressiveness in formal approaches to concurrency. Formal Aspects Comput. **27**(3), 465–497 (2015)
35. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place (2012). http://www.cs.ucr.edu/~lesani/downloads/Papers/WTTM12.pdf

36. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
37. Lynch, N., Vaandrager, F.: Forward and backward simulations. Inf. Comput. **121**(2), 214–233 (1995)
38. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
39. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
40. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2005, pp. 253–262. ACM (2005)
41. Moszkowski, B., Manna, Z.: Reasoning in interval temporal logic. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 371–382. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12896-4_374
42. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
43. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, 3rd edn. Artima Incorporation (2016)
44. Pfähler, J., Ernst, G., Bodenmüller, S., Schellhorn, G., Reif, W.: Modular verification of order-preserving write-back caches. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 375–390. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_25
45. Programming Methodology Group - ETH Zürich. VerifyThis (2021). https://www.pm.inf.ethz.ch/research/verifythis/Archive/20191.html
46. Reif, W.: Correctness of generic modules. In: Nerode, A., Taitslin, M. (eds.) LFCS 1992. LNCS, vol. 620, pp. 406–417. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0023893
47. Reif, W., Schellhorn, G., Stenzel, K.: Interactive correctness proofs for software modules using KIV. In: COMPASS 1995 - Tenth Annual Conference on Computer Assurance. IEEE Press (1995)
48. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 2002 Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002)
49. scalameta - Library to Read, Analyze, Rransform and Generate Scala Programs. https://scalameta.org/
50. Schellhorn, G.: Completeness of ASM refinement. Electron. Notes Theor. Comput. Sci. **214**, 25–49 (2008)
51. Schellhorn, G., Ahrendt, W.: The WAM case study: verifying compiler correctness for prolog with KIV. In: Automated Deduction – A Basis for Applications, volume III: Applications, Chapter 3: Automated Theorem Proving in Software Engineering, pp. 165–194. Kluwer Academic Publishers (1998)
52. Schellhorn, G., Bodenmüller, S., Pfähler, J., Reif, W.: Adding concurrency to a sequential refinement tower. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 6–23. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_2
53. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. **15**(4), 31:1–31:37 (2014)

54. Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., Reif, W.: Development of a verified flash file system. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 9–24. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_2. Invited Paper
55. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: a temporal logic framework for compositional reasoning about interleaved programs. Ann. Math. Artif. Intell. **71**, 131–174 (2014)
56. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved Programs and rely-guarantee reasoning with ITL. In: Proceedings of the 18th International Symposium on Temporal Representation and Reasoning (TIME), pp. 99–106. IEEE Computer Society Press (2011)
57. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 193–209. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_13
58. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23283-1_16
59. Woodcock, J., Stepney, S., Cooper, D., Clark, J., Jacob, J.: The certification of the Mondex electronic purse to ITSEC level E6. Formal Aspects Comput. **20**, 5–19 (2008)
60. Xu, Q., de Roever, W.-P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997)

A Note on Idleness Detection of Actor Systems

Rudolf Schlatte(✉)

Institute for Informatics, University of Oslo, Oslo, Norway
rudi@ifi.uio.no

Abstract. In an actor system, detecting global idleness is trivial if the global state can be inspected synchronously. A distributed implementation, especially with asynchronous messages, poses more problems. This paper discusses detecting idleness in actor systems with various characteristics, among them topology and message passing semantics. We present a TLA+ specification of a simple termination detection algorithm, add necessary conditions missing in the original description, and establish a correctness invariant to model-check the specification against. We further show via model-checking that the algorithm only works with synchronous messages but is robust against message reordering. We then model and discuss a simplified version of the idleness detection implementation in the Erlang backend of the ABS modeling language.

Keywords: Actor systems · Termination detection · Model checking

1 Introduction

Detecting global idleness or termination is necessary to detect when it is safe to shupt down a distributed actor system. Idleness detection is also used to initiate other actions, such as advancing a logical clock during simulation. Termination detection algorithms should be precise (i.e., give no false positives) and efficient (work with a minimum of messages).

Following the taxonomy in [8], we define a distributed actor system as follows:

- A set of actors that share no state and are either running or idle. An actor can change its state from running to idle at any time, but a change from idle to running requires a message from another, running actor.
- A set of communication channels between actors, among which both activation messages and control messages are sent. Communication takes arbitrary but finite time, and can be synchronous or asynchronous.

Any termination detection algorithm relies on certain properties of the underlying infrastructure: synchronous versus asynchronous message sending, the possibility of message loss, message duplication and reordering, fixed or arbitrary channel topologies etc. An algorithm can only be correct with respect to these basic assumptions.

One method of validating a termination algorithm is to formally specify it and use a theorem prover or a model checker to find violations of correctness invariants. TLA+ [7] is a specification language designed for distributed concurrent systems; this paper uses TLA+ to formally specify algorithms. Section 2 presents a simple termination detection system, then Sect. 3 discusses quiescence detection for a simulator of the ABS modeling language. Section 4 concludes the paper.

2 A Formal Model of Dijkstra's Termination Detection Algorithm

E.W. Dijkstra presented an easy-to-understand termination detection algorithm for actor systems [3]. This algorithm relies on synchronous message passing among an ordered sequence of actors $a_0 \mathrel{..} a_{n-1}$, with each actor knowing which actor is their predecessor. Any active actor can send a message to any other actor; the sending actor a_i colors itself black ($\mathit{color}_i = \mathit{black}$) and the receiving actor a_j becomes active ($\mathit{state}_j = \mathit{active}$).

The actor a_0 has no predecessor but knows a_{n-1}; a_0 is responsible for initiating a round of the algorithm by sending a white token t to the last actor ($t = 0 \wedge t' = n-1 \wedge c' = \mathit{white}$). During a round of the algorithm, the token t is passed down from actor to actor ($t = i \wedge t' = i-1$). An inactive actor ($\mathit{state}_i = \mathit{idle}$) passes on the token. If that actor sent an activation message to another actor after the previous round (i.e., $\mathit{color}_i = \mathit{black}$), it colors the token black ($c = \mathit{black}$) and itself white ($\mathit{color}_i = \mathit{white}$) before passing it on; otherwise it leaves the token color unchanged.

The algorithm terminates when a white token arrives back at the initiating actor (i.e., $t = 0 \wedge c = \mathit{white}$) and the initiating actor is itself inactive and white ($\mathit{color}_0 = \mathit{white} \wedge \mathit{state}_0 = \mathit{idle}$). When reaching this state, all actors are deemed to be idle. In other words, the correctness invariant is:

$$\begin{aligned} & t = 0 \wedge\ c = \mathit{white} \wedge \mathit{state}_0 = \mathit{idle} \wedge \mathit{color}_0 = \mathit{white} \\ \implies & \forall i \in 0..n-1 : \mathit{state}_i = \mathit{idle} \wedge \mathit{color}_i = \mathit{white} \end{aligned} \tag{1}$$

We modeled Dijkstra's algorithm in TLA+ (see Appendix A) and model-checked the correctness invariant for five actors. During that process, we discovered that the description of the algorithm in [3] is incomplete: the paper does not specify the initial state and location of the token. Via model checking, we can show that an initial state with $t = 0 \wedge c = \mathit{black}$ is safe, while starting with a white token ($c = \mathit{white}$) can lead to a faulty termination detection.

We also briefly investigated whether the algorithm depends on synchronous and/or ordered message delivery. With a modified model that implements activation message sending as a two-step process of first placing the message into an actor's inbox and then handling it in a second step (see Appendix B for the changes), model-checking produces traces with erroneous termination detection.

A further modification of the model whereby actors only pass on the token when their mailbox is empty (making message passing synchronous again since an actor "knows" that it has incoming messages) does not lead to erroneous termination detection, meaning that Dijkstra's algorithm is robust against message reordering.

3 Quiescence Detection in the Erlang Backend of ABS

ABS [5] is an actor-based modeling language with executable semantics. ABS processes are contained in and scheduled by cogs, which in turn are contained in deployment components that distribute resources. Thus, the runtime state hierarchy of a running ABS model forms a spanning tree that can be used for idleness detection.

Models are executed via transpiling into Erlang [1], an actor-based language with asynchronous messages. The Erlang backend uses a component called `cog_monitor` that communicates with all running cogs, and forms the root of the spanning tree. This component was in the past subject to a lot of race conditions; after vigorous debugging, an unknown number of those remain.

The original version of the Erlang backend, which did not include the timed semantics of Real-Time ABS, implemented termination detection as follows [4, p. 58]: "[a]fter all COGs stayed idle for configurable time span (default 1 s), [. . .] the model is assumed to be terminated." Each cog reported its status to the `cog_monitor` component, which had a (potentially outdated) overview of each cog's status. The spurious global idle conditions detected resulted from insufficient coordination between cogs and processes. Specifically, the following invariants were broken:

– A cog must not signal idleness when it has processes that are ready to execute.
– A process must synchronize with all processes that wait for its return value before terminating.
– A process must synchronize with the cog of a new process it created before terminating.

These conditions, taken together, form an invariant for the cog monitor: all "active" signals must arrive before any "idle" signal that caused the "active" signals. Figure 1 shows the original and fixed message sequence for a process that creates a process on another cog and then terminates itself.

All these conditions were informally discovered via test cases, and the resulting buggy spurious clock advances fixed, while implementing the semantics of Real-Time ABS, as described e.g. in [6]. A simplified version of the updated protocol, with one process per actor and the cog monitor expressed implicitly as part of the global state, is shown in Appendix C.

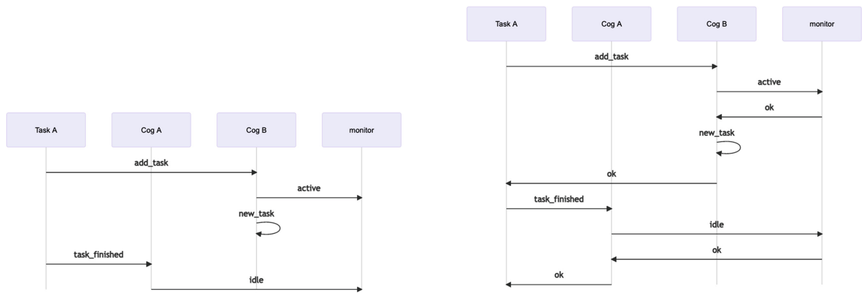


Fig. 1. Process creation followed by process termination in the Erlang backend: original (left) and revised (right, slightly simplified). Note that in the original, Task A can send both its messages without waiting, so the idle message can overtake the active message.

4 Conclusion and Future Work

As shown for example in [9], formal methods are a suitable tool for exhaustively specifying distributed algorithms. This paper demonstrated how to specify a simple termination-detection algorithm, validate its correctness via model-checking, and check its robustness against various changes in its operating environment.

We do want to apply these techniques against the Erlang-based actor implementation of the ABS modeling language simulator, which has been exhaustively tested but never formally specified. The current quiescence detection implementation might be message-optimal, since the lower bound for M activation messages is $\Omega(M)$ control messages for actor systems with asynchronous message passing [2]. But the `cog_monitor` component must handle all activation and deactivation messages synchronously, thereby forming a bottleneck, so a less synchronous algorithm could lead to large performance improvements in practice with the same number of messages. The author would not attempt to implement such an algorithm without a formal specification, and the fact that ABS features unbounded actor and process creation will pose interesting challenges for any chosen formal approach.

In conclusion, the author wishes that the work presented and proposed in this paper had been undertaken at the time the ABS simulator was first implemented.

Acknowledgements. The development of ABS was supported by the EU projects HATS and Envisage. Reiner Hähnle participated in Envisage and coordinated HATS; his influence on the development and developer of ABS is gratefully acknowledged.

A Dijkstra's Algorithm

MODULE *TDijkstra*

EXTENDS *Naturals*

CONSTANT N The number of participating actors; each actor is identified by its number

VARIABLE $state$ state[i] is the state of actor i
VARIABLE $color$ white/black state of actor color[i]
VARIABLE $tokenpos$ position of the token: 0..N-1
VARIABLE $tokencolor$ white or black

$Vars \triangleq \langle state, color, tokenpos, tokencolor \rangle$

$AI \triangleq 0\,..\,N-1$ The set of participating actors

$TTypeOK \triangleq$
- $\land\ state \in [AI \rightarrow \{\text{“active”}, \text{“idle”}\}]$
- $\land\ color \in [AI \rightarrow \{\text{“white”}, \text{“black”}\}]$
- $\land\ tokenpos \in AI$
- $\land\ tokencolor \in \{\text{“white”}, \text{“black”}\}$

$TInit \triangleq$
- $\land\ state = [i \in AI \mapsto \text{“active”}]$
- $\land\ color = [i \in AI \mapsto \text{“white”}]$
- $\land\ tokenpos = 0$
- $\land\ tokencolor = \text{“black”}$ start with a probe ()

$P0 \triangleq \forall\, i \in (tokenpos + 1)\,..\,(N-1) : state[i] = \text{“idle”}$

$P1 \triangleq \exists\, j \in 0\,..\,tokenpos : color[j] = \text{“black”}$

$P2 \triangleq tokencolor = \text{“black”}$

$SystemTerminatedWF \triangleq$
LET $terminated \triangleq$
- $\land\ tokenpos = 0$
- $\land\ tokencolor = \text{“white”}$ denotes a successful probe
- $\land\ state[0] = \text{“idle”}$
- $\land\ color[0] = \text{“white”}$

IN $terminated \implies \forall\, i \in AI : state[i] = \text{“idle”} \land color[i] = \text{“white”}$

$TInv \triangleq$
- $\lor\ P0$ Taken from Dijkstra’s paper
- $\lor\ P1$ ditto
- $\lor\ P2$ ditto

$becomeIdle(i) \triangleq$
- $\land\ state[i] = \text{“active”}$
- $\land\ state' = [state \text{ EXCEPT } ![i] = \text{“idle”}]$
- $\land\ \text{UNCHANGED } \langle color, tokenpos, tokencolor \rangle$

$sendMsg(i, j) \triangleq$
- $\land\ state[i] = \text{“active”}$
- $\land\ state' = [state \text{ EXCEPT } ![j] = \text{“active”}]$
- $\land\ color' = [color \text{ EXCEPT } ![i] = \text{“black”}]$ Rule 1’
- $\land\ \text{UNCHANGED } \langle tokenpos, tokencolor \rangle$

$passToken \triangleq$

```
  $\land\ tokenpos > 0$
  $\land\ state[tokenpos] = \text{“idle”}$    Rule 0
  $\land\ tokenpos' = tokenpos - 1$
  $\land\ tokencolor' = \text{IF } color[tokenpos] = \text{“black” THEN “black” ELSE } tokencolor$  Rule 2
  $\land\ color' = [color \text{ EXCEPT } ![tokenpos] = \text{“white”}]$  Rule 5
  $\land\ \text{UNCHANGED } \langle state \rangle$

$initiateProbe \triangleq$
  $\land\ tokenpos = 0$
  $\land\ state[0] = \text{“idle”}$
  $\land\ \lor\ color[0] = \text{“black”}$      Rule 3 (unsuccessful probe)
      $\lor\ tokencolor = \text{“black”}$
  $\land\ tokenpos' = N - 1$            Rule 4
  $\land\ tokencolor' = \text{“white”}$        Rule 4
  $\land\ color' = [color \text{ EXCEPT } ![0] = \text{“white”}]$
  $\land\ \text{UNCHANGED } \langle state \rangle$

$TNext \triangleq$
  $\lor\ \exists\, i \in AI : becomeIdle(i)$
  $\lor\ \exists\, i, j \in AI : sendMsg(i, j)$
  $\lor\ initiateProbe$
  $\lor\ passToken$
```

B Asynchronous Message Sending in the Dijkstra Algorithm Model

Add the *inbox* variable and replace the rule $sendMsg(i, j)$ of Appendix A with the two rules *sendMsg*, *becomeActive* as follows:

```
VARIABLE $inbox$                For each actor, a set of messages

$TTypeOK \triangleq$
  ...
  $\land\ inbox \in [AI \rightarrow \text{SUBSET } AI]$

$TInit \triangleq$
  ...
  $\land\ inbox = [i \in AI \mapsto \{\}]$

$sendMsg(i, j) \triangleq$
  $\land\ state[i] = \text{“active”}$
  $\land\ inbox' = [inbox \text{ EXCEPT } ![j] = @ \cup \{i\}]$
  $\land\ color' = [color \text{ EXCEPT } ![i] = \text{“black”}]$  Rule 1’
  $\land\ \text{UNCHANGED } \langle state, tokenpos, tokencolor \rangle$

$becomeActive(i) \triangleq$
  $\land\ state[i] = \text{“idle”}$
  $\land\ inbox[i] \neq \{\}$
```

```
∧ $state' = [state$ EXCEPT $![i] =$ "active"$]$
∧ $inbox' = [inbox$ EXCEPT $![i] = @ \cap \{$CHOOSE $j \in @ :$ TRUE$\}]$
∧ UNCHANGED $\langle color, tokenpos, tokencolor \rangle$
```

C Specification of the Erlang Backend Behavior

```
MODULE TErlangBackend

EXTENDS Naturals

CONSTANT N                  the number of Cogs

VARIABLE CogState           the state of each Cog
VARIABLE TaskState          The state of each cog's task (one per cog)

VARIABLE Messages           The current in-flight messages
```

```
$Vars \triangleq \langle CogState, TaskState, Messages \rangle$

$AC \triangleq 1 \,..\, N$              All Cog and task identifiers (one task per cog)

$AllMessages \triangleq \{$"invoke", "invoke_ok"$\} \times AC \times AC$

$TypeOK \triangleq$
  ∧ $CogState \in [AC \rightarrow \{$"active", "idle"$\}]$
  ∧ $TaskState \in [AC \rightarrow \{$"none", "running", "blocked"$\}]$
  ∧ $Messages \in$ SUBSET $AllMessages$

$Init \triangleq$
  ∧ $CogState = [i \in AC \mapsto$ IF $i = 1$ THEN "active" ELSE "idle"$]$
  ∧ $TaskState = [i \in AC \mapsto$ IF $i = 1$ THEN "running" ELSE "none"$]$
  ∧ $Messages = \{\}$

$sendInvocation(i, j) \triangleq$
  ∧ $TaskState[i] =$ "running"
  ∧ $Messages' = Messages \cup \{\langle$"invoke", $i, j\rangle\}$
  ∧ $TaskState' = [TaskState$ EXCEPT $![i] =$ "blocked"$]$
  ∧ UNCHANGED $CogState$

$acceptInvocation(j) \triangleq$
  ∧ $TaskState[j] =$ "none"
  ∧ $\exists i \in AC :$
    ∧ $\langle$"invoke", $i, j\rangle \in Messages$
    ∧ $Messages' = (Messages \setminus \{\langle$"invoke", $i, j\rangle\}) \cup \{\langle$"invoke_ok", $i, j\rangle\}$
    ∧ $CogState' = [CogState$ EXCEPT $![j] =$ "active"$]$
    ∧ $TaskState' = [TaskState$ EXCEPT $![j] =$ "running"$]$
```

```
$acceptConfirmation(i) \triangleq$
  $\land\ TaskState[i] = \text{“blocked”}$
  $\land\ \exists j \in AC :$
      $\land\ \langle \text{“invoke\_ok”}, i, j \rangle \in Messages$
      $\land\ Messages' = Messages \setminus \{\langle \text{“invoke\_ok”}, i, j \rangle\}$
      $\land\ TaskState' = [TaskState\ \text{EXCEPT}\ ![i] = \text{“running”}]$
      $\land\ \text{UNCHANGED}\ CogState$

$endTask(i) \triangleq$
  $\land\ TaskState[i] = \text{“running”}$
  $\land\ TaskState' = [TaskState\ \text{EXCEPT}\ ![i] = \text{“none”}]$
  $\land\ CogState' = [CogState\ \text{EXCEPT}\ ![i] = \text{“idle”}]$
  $\land\ \text{UNCHANGED}\ Messages$

$Next \triangleq$
  $\lor\ \exists i, j \in AC : sendInvocation(i, j)$
  $\lor\ \exists j \in AC : acceptInvocation(j)$
  $\lor\ \exists i \in AC : acceptConfirmation(i)$
  $\lor\ \exists i \in AC : endTask(i)$

$SystemTerminatedWF \triangleq$
  $\text{LET}\ allIdle \triangleq \forall i \in AC : CogState[i] = \text{“idle”}$
  $\text{IN}\ \ allIdle \implies$
        $\land\ \forall i \in AC : TaskState[i] = \text{“none”}$
        $\land\ Messages = \{\}$
```

References

1. Armstrong, J.: Erlang. Commun. ACM **53**(9), 68–75 (2010). https://doi.org/10.1145/1810891.1810910
2. Chandy, K.M., Misra, J.: How processes learn. Distrib. Comput. **1**(1), 40–52 (1986). https://doi.org/10.1007/BF01843569
3. Dijkstra, E.W., Feijen, W., van Gasteren, A.: Derivation of a termination detection algorithm for distributed computations. Inf. Process. Lett. **16**(5), 217–219 (1983). https://doi.org/10.1016/0020-0190(83)90092-3
4. Göri, G.: Erlang-based Execution and Error Handling for Abstract Behavioural Specifications. Master's thesis, Graz University of Technology, Institute for Software Technology (2015). https://diglib.tugraz.at/download.php?id=576a741586a71&location=search
5. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8

6. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Logical Algebraic Methods Program. **84**(1), 67–91 (2015). https://doi.org/10.1016/j.jlamp.2014.07.001
7. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
8. Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. J. Syst. Softw. **43**(3), 207–221 (1998). https://doi.org/10.1016/S0164-1212(98)10034-1
9. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417

Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives

Dominic Steinhöfel(✉)

CISPA Helmholtz Center for Information Security,
Stuhlsatzenhaus 5, Saarbrücken, Germany
dominic.steinhoefel@cispa.de

Abstract. Symbolic Execution (SE) enables a precise, deep program exploration by executing programs with symbolic inputs. Traditionally, the SE community is divided into the rarely interacting sub-communities of *bug finders* and *program provers*. This has led to independent developments of related techniques, and biased surveys and foundational papers. As both communities focused on their specific problems, the *foundations* of SE *as a whole* were not sufficiently studied. We attempt an unbiased account on the foundations, central techniques, current applications, and future perspectives of SE. We first describe essential *design elements* of symbolic executors, supported by *implementations in a digital companion volume*. We recap a *semantic framework*, and derive from it a—yet unpublished—*automatic testing approach* for SE engines. Second, we introduce SE *techniques* ranging from concolic execution over compositional SE to state merging. Third, we discuss *applications* of SE, including test generation, program verification, and symbolic debugging. Finally, we address the *future*. Google's OSS-Fuzz project routinely detects *thousands of bugs* in hundreds of major open source projects. What can symbolic execution contribute to future software verification in the presence of such competition?

This chapter comes with a digital companion volume [84] in form of a Jupyter notebook including additional examples, visualizations, and the complete code of all presented implementations. The companion volume will be updated also after this chapter has been published.

1 Introduction

It is no secret that every non-trivial software product contains bugs, and not just a few: A data analytics company reported in 2015 [7] that on average, a developer creates 70 bugs per 100 lines of code, of which 15 survive until the software is shipped to customers. In a recent, global survey among 950 developers [74], 88% of the participants stated that bugs and errors are frequently detected and reported by the actual *users* of the product, rather than being detected by tests or monitoring tools. In the same survey, more than a third of developers (37%) declared that they spend more than quarter of their time fixing bugs instead of "doing their job."

These numbers indicate that testing with manually written test cases alone is insufficient for *effective* and *sustainable* software verification: First, it is notoriously hard to come up with the right inputs to ensure well-enough coverage of all *semantic* features implemented in code. The fact that a test suite achieves a high *syntactic* code coverage, which is hard enough to accomplish, does unfortunately not imply that all bugs are found. Second, developers already spend much of their time testing and fixing bugs. In the above survey, 43% of the developers complained that testing is one of the major "pain points" in software development.

Clearly, there is a need for *automated* bug finding strategies. A simple, yet surprisingly effective, idea is to run programs with *random inputs*. When following this idea on the system level, it is called (blackbox) *fuzz testing* (or *fuzzing*) [65], while on the unit level, the label *Property-Based Testing (PBT)* [24] is customary.

Random approaches, however, struggle with covering parts of programs that are only reachable by few inputs only. For example, it is hard to randomly produce a structured input (such as an XML file or a C program), a magic value, or a value for a given checksum. Furthermore, even if code is covered, we might miss a bug: The expression `a // (b + c)` only raises a `ZeroDivisionError` if *both* `b` and `c` are 0. Generally, it can be difficult to choose "suitable" failure-inducing values, even when massively generating random inputs.

Symbolic Execution (SE) [2,8,19,58] provides a solution by executing programs with *symbolic inputs*. Since a (potentially constrained) symbolic value represents many values from the concrete domain, this allows to explore the program for *any possible input*. Whenever the execution depends on the concrete value of a symbolic input (e.g., when executing an **`if`** statement), SE follows all or only a subset of possible paths, each of which is identified by a unique *path condition*.

The integration of SE into *fuzzing* techniques yields so-called *white-box*, or *constraint-based*, fuzzers [20,38–40,69,87,93]. Especially at the unit level, SE can even exhaustively explore *all possible paths* (which usually requires auxiliary specifications), and *prove* that a property holds for *all possible inputs* [2,52].

Since its formation, the SE community has been split into two distinct sub communities dedicating their work either to test generation *or* program proving. The term "Symbolic Execution" has been coined by King [58] in 1976, who applied it to program testing. Independently, Burstall [19] proposed a program proving technique in 1974, which is based on "hand simulation" of programs—essentially, nothing different than SE. This community separation still persists today. In this chapter, we attempt a holistic approach to Symbolic Execution, explaining foundations, technical aspects, applications, and the future of SE from *both* perspectives.

This chapter combines aspects of a survey paper with a guide to implementing a symbolic interpreter. Moreover, especially Sect. 6 on the future of SE is based on personal opinions and judgments, which is more characteristic of an essay than of a research paper. We hope to provide useful theoretical and practical insights into the nature of SE, and to inspire impactful discussions.

We begin in Sect. 3 by addressing central design choices in symbolic executors. Especially for this chapter, we implemented a symbolic interpreter for `minipy`, a Python subset. Section 2 introduces the `minipy` language itself. When suitable, we enrich our explanations with implementation details. We continue with a frequently neglected aspect: The *semantic foundations* of SE. Special attention is paid to the properties an SE engine has to satisfy to be useful for its intended application (testing vs. proving). We introduce all four existing works on the topic, one of which we discuss in-depth. In the course of this, we derive a novel technique for *automated testing of SE engines* which has not been published before.

We discuss selected SE techniques in Sect. 4. For example, we derive a *concolic* interpreter from the baseline symbolic interpreter in just eight lines of code. We implemented most techniques as extensions of our symbolic interpreter.

In Sect. 5, we describe current trends in four application scenarios. Apart from the most popular ones, namely test generation and program proving, we also cover *symbolic debugging* and *model checking of abstract programs*.

Finally, we take a look at the future of SE. The probabilistic analysis in [13] indicates that systematic testing approaches like SE need to become *significantly faster* to compete with randomized approaches such as coverage-guided fuzzers. Otherwise, we can expect to reach the same level of *confidence* about a program's correctness more quickly when using random test generators instead of symbolic executors. What does this imply for the role of SE in future software verification?

2 Minipy

All our examples and implementations target the programming language `minipy`, a statically typed, imperative, proper subset of the Python language. It supports Booleans, integers, and integer tuples, first-order functions, assignments, and **`pass`**, **`if`**, **`while`**, **`return`**, **`assert`**, **`try-except`**, **`break`**, and **`continue`** statements. Excluded are, e.g., classes and objects, strings, floats, nested function definitions and lambdas, comprehensions and generators, **`for`** loops, and the **`raise`** statement. Expressions are *pure* in `minipy` (without side effects other than raised exceptions), since we have no heap and omitted Python's **`global`** keyword.

An example `minipy` program is the linear search routine in Listing 1. The values of `x` and `y` after execution are 2 and -1, respectively. The implementation uses an **`else`** block after the **`while`** loop, which is executed whenever the loop completes *normally*—i.e., not due to the **`break`** statement in Line 5, executed if

Listing (minipy) 1 Linear search program.

```
 1 def find(needle: int, haystack: tuple) -> int:
 2     i = 0
 3     while i < len(haystack):
 4         if haystack[i] == needle:
 5             break
 6
 7         i = i + 1
 8     else:
 9         return -1
10
11     return i
12
13 t = (1, 2, 3, 4, )
14 x = find(3, t)
15 y = find(5, t)
```

`needle` has been found. The type annotations in Line 1 are mandatory in minipy; thus, the type of a variable can always be determined either from the type of the right-hand side of an initial assignment, or from the annotations in function signatures.

We constructed a concrete interpreter for minipy. It consists of functions of the shape `evaluate_`*exprType*`(expr, environment)` for evaluating expressions, and `execute_`*stmtType*`(stmt, environment)` for executing statements. The `environment` consists of a *store* mapping variables to values and a repository of function implementations (e.g., `len`). The evaluation functions return a value and leave the environment unchanged; the execution functions *only* have side effects: They may change the environment, and complete abruptly. Abrupt completion due to **`return`**s, **`break`**s, and **`continue`**s is signaled by special exception types.

3 Foundations

We can focus on two aspects when studying the principles of SE. First, we consider how SE engines are *implemented*. We distinguish *static* symbolic *interpreters*, e.g., angr [80], KeY [2], KLEE [20], and S^2E [23], and approaches *dynamically executing* the program.[1] The PEF tool [10], e.g., extracts symbolic constraints using *proxy objects*; QSYM [93] by a *runtime instrumentation*, and

[1] This distinction is a simplification, as many *dynamic* SE tools belong two both categories. KLEE, for instance, statically interprets LLVM instructions and maintains multiple branches in memory; yet, it also integrates elements of *dynamic* execution, e.g., when interacting with external code such as the Linux kernel. We discuss this style of *selective* SE in Sect. 4.

SYMCC [69] by *compiling* directives maintaining path constraints directly into the target program.

The second aspect addresses the *semantics* of SE, i.e., *what does and should SE compute?* Test generators compute an *underapproximation* of all possible final program states. In case one of these is an error state, e.g., crashes or does not satisfy an assertion, there is always a corresponding concrete, fault-inducing input (a *test case*). *Program proving tools*, on the other hand, *overapproximate* the state space. Thus, the absence of any erroneous state in the analysis result implies the absence of such states in the reachable state space, which results in a *program proof*.

We begin this section by providing a scheme to characterize SE engines. At the same time, we describe how to implement a relatively simple symbolic interpreter for minipy (Sect. 3.1). We decided on implementing a *static* executor since this allows investigating both over- and underapproximating SE variants (e.g., in Sect. 4 we integrate (overapproximating) loop invariants, and turn the interpreter into an underapproximating *concolic* interpreter with only a few lines of code). In Sect. 3.2, we then introduce a semantic framework for Symbolic Execution. Finally, in Sect. 3.3, we derive an *automatic testing technique* for SE engines from this formal framework. To the best of our knowledge, this is only the second approach addressing the verification of SE engines using automated testing, and the first which can address multi-path and overapproximating SE in a meaningful way.

3.1 Designing a Symbolic Interpreter

To describe implementation aspects of an SE engine *in a structured way*, we extracted characteristics for distinguishing them (displayed in Table 1) by comparing different kinds of engines from the literature. This catalog is definitely incomplete. Yet, we think that it is sufficiently precise to contextualize most engines; and we did not find any satisfying alternative in the literature.

In the following, we step through the catalog and briefly explain the individual characteristics. We describe how our implemented *baseline symbolic interpreter* fits into this scheme, and provide chosen implementation details.

Implementation Type. We distinguish SE engines that *statically interpret* programs from those that *dynamically execute* them. Among the interpretation-based approaches, we distinguish those that retain multiple paths in memory and those that only keep a single path. An example for the latter would be an interpretation-based *concolic* executor.

As a baseline for further studies, we implemented a *multi-path* symbolic minipy *interpreter*, with the concrete interpreter serving as a reference. The interpreter keeps *all* execution tree leaves discovered so far in memory (which is not necessarily required from a multi-path engine). What is more, it also retains all *intermediate* execution states, such that the output is a full *Symbolic Execution Tree (SET)*. An example SET for the linear search program in Listing 1, automatically produced by our implemented framework, is shown in Fig. 1. The nodes

Table 1. Characteristics of SE engines.

Implementation Type	
(1)	Interpretation-Based
(1.1) / (1.2)	Multi / Single-Path
(2)	Execution-Based
(2.1)	Compilation-Based
(2.2)	Runtime Instr.-Based
(2.3)	Using Proxy Objects

Loop / Recursion Treatment	
(1)	Bounded Unrolling
(2)	Invariants
(3)	Concolic

Constraint & Value Representation	
(1)	External Theories
(1.1) / (1.2)	Shallow / Deep Embedding
(2)	Internal Theories

Call Treatment	
(1)	Inlining
(2)	Summaries / Contracts
(3)	On-Demand Concretization

Constraint Solving	
(1)	Off-the-shelf Solver
(1.1)	With Reduction / Reuse
(1.2)	Non-exhaustive Techniques
(2)	Special Solver

Path Explosion Countermeasures	
(1)	Summaries / Contracts
(2)	Subsumption
(3)	State Merging

of the tree are *Symbolic Execution States (SESs)* consisting of (1) a *path condition*, which is a set of closed formulas (*path constraints*) over program variables, (2) a *symbolic store*, a mapping of program variables to symbolic expressions over program variables, and (3) a *program counter* pointing to the next statement to execute. Assignments update the store, while case distinctions (such as **`while`** and **`if`** statements) update path constraints. Together, path condition, store, and program counter determine the concrete states represented by an SES. We formalize this semantics in Sect. 3.2.

The following definition assumes sets $PVars$ of *program variables*, $Expr$ of (arithmetic, boolean, or sequence) *expressions*, and Fml of *formulas* over program variables. We formalize symbolic stores as partial mappings $PVars \hookrightarrow Expr$ and use the shorthand $SymStores$ for the set of all these mappings.

Definition 1 (Symbolic Execution State). *A* Symbolic Execution State (SES) *is a triple* $(Constr, Store, PC)$ *of (1) a set of* path constraints $Constr \subseteq Fml$*, the* path condition*, (2) a mapping* $Store \in SymStores$ *of program variables to symbolic expressions, the* symbolic store*, and (3) a* program counter *PC pointing to the next statement to execute. We omit PC if it is empty. SESs is the set of all SEStates.*

The structure of our symbolic interpreter aligns with the concrete `minipy` interpreter. Environments are now *symbolic* environments, consisting of a set

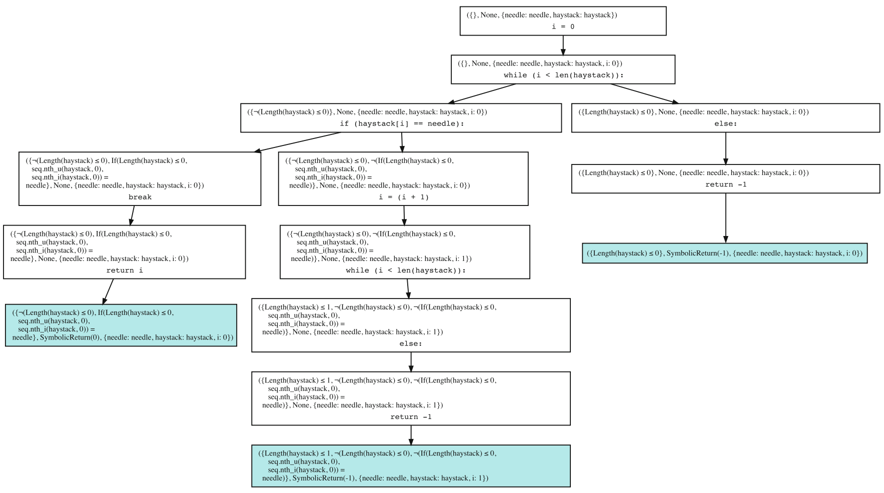

({}, None, {needle: needle, haystack: haystack})
i = 0
({}, None, {needle: needle, haystack: haystack, i: 0})
while (i < len(haystack)):
({¬(Length(haystack) ≤ 0)}, None, {needle: needle, haystack: haystack, i: 0})
if (haystack[i] == needle):
({Length(haystack) ≤ 0}, None, {needle: needle, haystack: haystack, i: 0})
else:
({¬(Length(haystack) ≤ 0), If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle}, None, {needle: needle, haystack: haystack, i: 0})
break
({¬(Length(haystack) ≤ 0), ¬(If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle)}, None, {needle: needle, haystack: haystack, i: 0})
i = (i + 1)
({Length(haystack) ≤ 0}, None, {needle: needle, haystack: haystack, i: 0})
return -1
({¬(Length(haystack) ≤ 0), If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle}, None, {needle: needle, haystack: haystack, i: 0})
return i
({¬(Length(haystack) ≤ 0), ¬(If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle)}, None, {needle: needle, haystack: haystack, i: 1})
while (i < len(haystack)):
({Length(haystack) ≤ 0}, SymbolicReturn(-1), {needle: needle, haystack: haystack, i: 0})
({¬(Length(haystack) ≤ 0), If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle}, SymbolicReturn(0), {needle: needle, haystack: haystack, i: 0})
({Length(haystack) ≤ 1, ¬(Length(haystack) ≤ 0), ¬(If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle)}, None, {needle: needle, haystack: haystack, i: 1})
else:
({Length(haystack) ≤ 1, ¬(Length(haystack) ≤ 0), ¬(If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle)}, None, {needle: needle, haystack: haystack, i: 1})
return -1
({Length(haystack) ≤ 1, ¬(Length(haystack) ≤ 0), ¬(If(Length(haystack) ≤ 0, seq.nth_u(haystack, 0), seq.nth_i(haystack, 0)) = needle)}, SymbolicReturn(-1), {needle: needle, haystack: haystack, i: 1})

Fig. 1. SET of the linear search program in Listing 1 with one loop unrolling.

of path constraints in addition to the (symbolic) store and built-in function repository. Execution functions are side-effect free and produce SETs instead of manipulating environments. Expressions evaluate to *conditioned* symbolic values, since the evaluation of the same expression, e.g., `x // y`, may result in a value (e.g., 4 for `x` = 4 and `y` = 2) or in an exception (e.g., for `y` = 0). In Sect. 3.1, we explain how we concretely represent symbolic expressions and constraints.

As an example, we discuss the symbolic execution of **`if`** statements. As usual, the concrete code is available at [84]. First, we evaluate the guard expression. Since this can result in multiple conditioned values, we loop over all values and attached constraints. If an evaluation result is unsatisfiable with the current path condition, it is not considered; if the evaluation resulted in an exceptional result, we set the "abrupt completion" flag of the symbolic environment to the returned exception.

If evaluating the guard resulted in a value, we compute the symbolic environments for the then and else branch. We only add the subtrees for these branches if they are satisfiable to avoid the execution of infeasible paths. Finally, the then and else blocks are executed and added to the result SET; if there is no **`else`** branch, we add the corresponding environment without program counter.

An alternative to implementing symbolic transitions in code, as we did in our symbolic interpreter, is encoding them as a set of small-step *rules* in a domain-specific language. This is the approach followed by the KeY SE engine [2].

In Sect. 3.1, we discuss satisfiability checking for symbolic environments. Next, we focus on the representation of constraints and symbolic values.

Representation of Constraints and Values. The choice of how to express symbolic values and constraints in an SE engine usually goes hand in hand with the choice of the used constraint solver (which we discuss in the next section). When using an off-the-shelf solver, one is ultimately bound to its available theories. However, there are two styles of *embedding* the language of symbolic values and constraints, namely a *deep* and a *shallow* approach [35,88]. In a deep embedding, one defines dedicated Abstract Syntax Trees (ASTs) for values and constraints, and formalizes specialized operations on those in terms of the solver theories. Shallow embeddings directly encode the embedded language in terms of already available data structures. Both approaches have advantages and disadvantages: Deep embeddings offer more flexibility, but require the definition of new theories, which can be non-trivial and give rise to inconsistencies. Shallow embeddings, on the other hand, are simpler and allow re-using pre-defined theories, but can come at the cost of a reduced expressiveness. A different approach followed, e.g., by the KeY [2] engine, is the development of a specialized solver with home-grown theories. This approach offers most flexibility, but is costly to implement and decouples the engine from advances of general-purpose solvers.

Our symbolic interpreter is based on a shallow embedding into the Z3 SMT solver [66]. Integer types are mapped to `z3.ArithRef` expressions, constraints and Booleans to `z3.BoolRef`, and tuples to sequences of sort `z3.SeqRef`. This—easy to implement—approach restricts us to *non-nested* tuples, since nested sequences are not supported in Z3. For a complete support of `minipy`'s language features, we would have to resort to a deeper embedding into Z3.

Constraint Solving. Constraint solving plays a crucial role in SE engines [8]: Constraints are checked when *evaluating path feasibility*, *simplifying symbolic stores*, or *verifying assertions*. Usually, path constraints are eagerly checked to rule out the exploration of infeasible paths. However, this can be expensive in the presence of many complex constraints (e.g., nonlinear arithmetics). Engler and Dunbar [30] propose a lazier approach to constraint solving. If a constraint cannot be quickly solved, they defer its evaluation to a later point (e.g., when an error has been found). Only then, feasibility is checked with higher timeout thresholds.

Few systems (e.g., KeY) implement own solvers. Most symbolic executors, however, use off-the-shelf solvers like Z3. To alleviate the overhead imposed by constraint solving, some systems preprocess constraints before querying the solver. KLEE [20], e.g., *reduces* expressions with rewriting optimizations, and *re-uses* previous solutions by caching counterexamples. Systems like QSYM [93] resort to "optimistic" solving, which is an unsound technique only considering *partial* path conditions. Since QSYM is a hybrid fuzzer combining a random generator and a symbolic executor, generating "unsound" inputs (which do not conform to the path they are generated for) is not a big problem, since the fuzzing component will quickly detect whether the input is worthwhile or not by concrete execution (and, e.g., collecting coverage data). The Fuzzy-Sat system [15] analyzes constraints collected during SE and, based on that, performs *smart mutations* (a concept known from the fuzzing domain) of known solutions for partial constraints to create satisfying inputs faster.

Frequently, SMT solvers time out, and fail to provide a definitive answer at all. We use Z3 as a default solver, and discovered that its behavior can be highly non-deterministic: For a given unsatisfiable formula, we either received a timeout *or* a quick, correct answer, depending on the order of the constraints in a formula, or the value of an initial seed value, or whether Z3 is run in parallel mode.[2]

We follow a pragmatic approach to deal with Z3's incompleteness: Whenever a Z3 call returns "timeout," we query another solver, in this case KeY, which is run inside a small service to reduce bootstrapping costs. KeY is significantly slower than Z3 in the average case (fractions of seconds vs. multiple seconds), but is *stable* (behaves deterministically), and it features a powerful theory of sequences, which behaves well together with arithmetic constraints.

Treatment of Loops and Recursion. Loops and recursion require special attention in SE. Consider, e.g., the execution of the body of the `find` function in Listing 1. If `haystack` is a symbolic value such that we do not know its length, the number of times an engine should execute the body of the **`while`** loop until the loop guard becomes unsatisfiable is unknown. There are three main ways to address this issue (which also apply to recursive functions). First, we can impose a fixed bound on the number of loop executions and *unroll* the loop that many times. This is the procedure implemented in our baseline symbolic interpreter. For example, in Fig. 1, we only unrolled the loop one time. If we increase the threshold by one, we obtain two additional leaves in the SET, one for the case where `needle` has been found, and one for the case where it has not. Second, one can use *loop invariants* [49]. A loop invariant is a summary of the loop's behavior that holds independently from the number of loop iterations. For example, an invariant for Listing 1 is that `i` does not grow beyond the size of `haystack`. Instead of executing the actual loop body, one can then step over a loop and add its invariant to the path condition. Although much research has been conducted in the area of automatic *loop invariant inference* [17,31,33,60,81], those specifications are mostly manually annotated, turning specification into a main bottleneck of invariant-based approaches [2]. In Sect. 3.1, we show how to integrate invariants into the baseline symbolic interpreter, turning it into a tool that can be used for *program proving*. Finally, loops are naturally addressed by *concolic* SE engines, where the execution is guided by concrete inputs. Then, the loop is executed as many times as it would have been under concrete execution. In Sect. 4.1, we derive a concolic tester from the baseline interpreter with minimal effort.

Treatment of Calls. Analogously to loops, there are two ways of executing calls in SE: Either, one can *inline* and symbolically execute the function body,

[2] See https://github.com/Z3Prover/z3/issues/5559 for an issue we reported to the Z3 team. According to the answer of Z3's main developer Nikolaj Bjørner, the "sequence solver is rather unstable, especially when you mix integer constraints and sequence constraints".

or take a *summary* (depending on the context also called *contract*) of the function's behavior and add this abstraction to the path condition. If their code is unavailable, we cannot inline function bodies (e.g., for a system call). Usually, specifying a contract is one option. Yet, this is problematic when analyzing of systems with many calls to unspecified libraries. Furthermore, it might not even be possible. Godefroid [36] names as a simple example an **`if`** statement with guard `x == `**`hash`**`(y)` containing an error location in its then branch. To reach that location, we have to find two symbolic values `x` and `y` such that `x` is the hash value of `y`. If **`hash`** is a *cryptographic* hash function, it has been designed exactly to prevent such reasoning, and we cannot expect to come up with a contract for **`hash`** (and even less to obtain a usable path constraint from SE without contracts). Concolic execution does not provide a direct solution this time, either: While we *can* concretely execute any function call, we will not obtain a *constraint* from it.

A pragmatic approach in *dynamic* (i.e., integrating elements from concrete execution) SE is to *switch* between concrete and symbolic execution whenever adequate (see, e.g., [23,38]). When reaching the **`if`** statement of the program above, for example, it is easy to decide whether the equation holds by choosing random concrete values for `x` and `y` satisfying the above precondition, and continue symbolically executing and collecting constraints from then on. What is more, we can fix the value of `y` only, compute its hash, and choose the value of `x` such that it does (not) satisfy the comparison. We discuss this approach in Sect. 4.2.

The baseline symbolic interpreter inlines function calls. It does so in a "non-transparent" way (inspired by the concrete `minipy` interpreter): Whenever we reach a function *definition*, we add a *continuation* to the functions repository. When evaluating a call, the continuation is retrieved and passed the current symbolic environment and symbolic arguments, and returns an `EvalExprResult`. In the computed SET, the execution steps inside the function body are thus not communicated. In the digital companion volume [84], we extend the baseline interpreter with a technique for *transparent inlining*; in Sect. 3.1, we show how to integrate *function summaries*.

Path Explosion Countermeasures. *Path explosion* is a major, if not *the* most serious, obstacle for SE [8,22,91]. It is caused by an exponential growth of feasible paths in particular in the presence of loops, but also of more innocuous constructs like **`if`** statements (e.g., if they occur right at the beginning of a substantially sized routine). *Auxiliary specifications*, i.e., loop invariants and function summaries which we already mentioned before, effectively reduce the state space. While loop invariants and full functional contracts generally have to be annotated manually, there do exist approaches inferring function summaries from cached previous executions in the context of dynamic SE (e.g., [5]).

Subsumption techniques drop paths that are *similar* (possibly after an abstraction step) to previously visited paths. Anand et al. [6], e.g., summarize heap objects (e.g., linked lists and arrays) with techniques known from shape analysis to decide whether two states are to be considered equal. Another line of work (see, e.g., [63]) distinguishes a subset of possible program locations considered "interesting," e.g., because of the annotation with an **`assert`** statement. When an execution does *not* reach an interesting location, intermediate locations are tagged with path constraints. Whenever such a label is visited another time, there are two options: Either, the current path condition is implied by the label. In that case, this execution path is dropped, since we can be sure that it will not reach an interesting location. In the other case, either an interesting location is eventually reached, or the label is *refined* using an interpolation technique summarizing previous unsuccessful paths at a position.

State merging [46,61,76,78,82] is a flexible and powerful technique for mitigating path explosion. The idea is to bring together SESs with the same program counter (e.g., after the execution of an **`if`** statement) by computing a summary of the path constraints and stores of the input states. This summary can be fully precise, e.g., using *If-Then-Else* terms, underapproximating (omitting one input state in the most extreme case), or overapproximating (e.g., using an abstract domain).

Our baseline symbolic interpreter does not implement any countermeasure to path explosion. However, we extend it with *contracts* and *state merging* in Sect. 4.

3.2 Semantic Foundations of Symbolic Execution

Despite the popularity of SE as a program analysis technique, there are only few works dedicated to the *semantics* and *correctness* of SE. This could be because most SE approaches focus on test generation, and deep formal definitions and proofs are less prevalent in that area than in formal verification. Furthermore, every experienced user of SE has a solid *intuition* about the intended working of the symbolic analysis, and thus might not have felt the need to make it formal.

We know of four works on the semantic foundations of SE: One from the 90s [59] and three relatively recent ones [12,62,82], published between 2017 to 2020.

Kneuper [59] distinguishes *fully precise* SE, which *exactly* captures the set of *all* execution paths, and *weak* SE, which *overapproximates* it. Intuitively, the weak variant can be used in program proving, and the fully precise one in testing. Yet, fully precise SE is generally out of reach; Kneuper does not consider underapproximation. The frameworks by Lucanu et al. [62] and de Boer and Bonsangue [12] relate symbolic and concrete execution via simulation relations; they do not consider the semantics of individual SESs. Lucanu et al. [62] define two properties of SE. *Coverage* is the property that for every concrete execution, there is a corresponding feasible symbolic one. *Precision* means that for every feasible symbolic execution, there is a corresponding concrete one. De Boer and Bonsangue argue from a program proving point of view. Their property corresponding to "coverage" of [62] is named *completeness*, and *soundness* for "precision."

In [82], we provided a framework based on the semantics of individual SESs, which represent many concrete states. A transition is *precise* if the output SESs represent *at most* the concrete states represented by the input SESs, and *exhaustive* if the outputs represent *at least* the states represented by the inputs. Other than [12,62], this is a big-step system not considering paths and intermediate states. We think that coverage/completeness from [12,62] imply exhaustiveness, and precision/soundness "our" precision. Kneuper's weak SE is exhaustive/complete/has full coverage, while fully precise SE is *additionally* precise/sound.

In the following, we present a simplified account of the framework from [82]. Apart from personal taste, the focus on the input-output behavior of symbolic transitions allows us to derive a novel technique for *automatically testing SE engines* in Sect. 3.3. The only other work we know of on testing symbolic executors [56] only tests precision, and struggles (resorts to comparatively weak oracles) with testing multi-path engines. We think that the focus on *paths*, and not the semantics of *states*, binds such approaches to precision and single-path scenarios; the state-based big step semantics allows addressing these shortcomings. The framework from [82] is based on the concept of *concretizations* of symbolic stores and SESs. Intuitively, a symbolic store represents up to infinitely many concrete states. For example, the store *Store* mapping the variable `x` to $2 \cdot \texttt{y}$ represents all concrete states where `x` is even. Given any *concrete* input, we can concretize *Store* to a concrete state by interpreting variables in the range of *Store* within the concrete state. If $\sigma(\texttt{y}) = -3$, e.g., the concretization of *Store* w.r.t. σ maps `x` to -6.

Definition 2 (Concretization of Symbolic Stores). *Let ConcrStates denote all concrete execution states (sets of pairs of variables and concrete values). The* symbolic store concretization function $concr_{store} : SymStores \times ConcrStates \rightarrow ConcrStates$ *maps a symbolic store Store and a concrete state* σ *to a concrete state* $\sigma' \in ConcrStates$ *such that (1) for all* $\texttt{x} \in PVars$ *in the domain of Store,* $\sigma'(\texttt{x})$ *equals the right-hand side of* `x` *in Store when evaluating all occurring program variables in* σ*, and (2)* $\sigma(\texttt{y}) = \sigma'(\texttt{y})$ *for all other program variables* `y` *not in the domain of Store.*

The concretization of symbolic stores is extended to SESs by first checking whether the given concrete store satisfies the path condition; if this is not the case, the concretization is empty. Otherwise, it equals the concretization of the store. Consider the constraint $\texttt{y} > 0$. Then, the concretization of $(\{\texttt{y} > 0\}, \mathit{Store})$ w.r.t. σ (where *Store* and σ are as before) is $\emptyset$. For $\sigma'(\texttt{y}) = 3$, on the other hand, we obtain a singleton set with a concrete state mapping `x` to 6. Additionally, we can take into account *program counters* by executing the program at the indicated location starting in the concretization of the store. The execution result is then the concretization.

Definition 3 (Concretization of SESs). *Let, for every* minipy *program p,* $\rho(p)$ *be a (concrete) transition relation relating all pairs* σ, σ' *such that executing p in the state* $\sigma \in ConcrStates$ *results in the state* $\sigma' \in ConcrStates$*. Then, the* concretization function $concr : SEStates \times ConcrStates \rightarrow 2^{ConcrStates}$ *maps* *an SES* $(Constr, Store, PC)$ *and a concrete state* $\sigma \in ConcrStates$ *(1) to the empty set* $\emptyset$ *if either* $Constr$ *does not hold in* σ*, or there is no* σ' *such that* $(concr_{store}(Store, \sigma), \sigma') \in \rho(PC)$*, or otherwise (2) the singleton set* $\{\sigma'\}$ *such that* $(concr_{store}(Store, \sigma), \sigma') \in \rho(PC)$*.*

Consider the SES $s = (\{\mathtt{1} \in \mathtt{t}\}, (\mathtt{n} \mapsto 1), r = \mathtt{find(n,\ t)})$, where `n` and `t` are variables of integer and tuple type, $(\mathtt{n} \mapsto 1)$ a symbolic store which maps `n` to 1 and is undefined on all other variables, and `find` the linear search function from Listing 1. We write $\mathtt{1} \in \mathtt{t}$ to express that the value 1 is contained in `t`. For any σ where 1 is not contained in $\sigma(\mathtt{t})$, we have $concr(s, \sigma) = \emptyset$. For all other states σ', $concr_{store}((\mathtt{n} \mapsto 1), \sigma') = \sigma'[\mathtt{n} \mapsto 1]$ (i.e., σ', but with `n` mapped to `1`). The concretization $concr(s, \sigma')$ is then a state resulting from running `find` with arguments 1 and $\sigma'(\mathtt{t})$ (i.e., the values of `n` and `t` in $concr_{store}((\mathtt{n} \mapsto 1), \sigma')$) and assigning the result to `r`: $concr(s, \sigma')(\mathtt{r})$ is the index of the first 1 in $\sigma'(\mathtt{t})$.

By considering all possible concrete states as initial states for concretization, we obtain the *semantics*, i.e., the set of all represented states, of an SES.

Definition 4 (Semantics of SESs). *The semantics* $[\![s]\!]$ *of an SES* $s \in SEStates$ *is defined as the union of its concretizations:* $[\![s]\!] := \bigcup_{\sigma \in ConcrStates} concr(s, \sigma)$*.*

Usually, SE systems take one input SES to at least one output. Systems with *state merging*, however, also transition from *several inputs states* (the merged states) to one output state. The notion of SE transition relation defined in [82] goes one step further and permits m-to-n transition relations for *arbitrary* m and n. In principle, this allows for merging techniques producing *more than one output state.*

Definition 5 (SE Configuration and Transition Relation). *An* SE Configuration *is a set* $Cnf \subseteq SEStates$*. An* SE Transition Relation *is a relation* $\delta \subseteq 2^{SEStates} \times (2^{SEStates} \times 2^{SEStates})$ *associating to a configuration* Cnf *transitions* $t = (I, O)$ *of* input states $I \subseteq Cnf$ *and* output states $O \subseteq 2^{SEStates}$*. We call* $Cnf \setminus I \cup O$ *the* successor configuration *of the transition* t *for* Cnf*. The relation* δ *is called SE Transition Relation* with (without) State Merging *if there is a (there is no) transition with more than one input state, i.e.,* $|I| > 1$*. We write* $Cnf \xrightarrow{t}_{\delta} Cnf'$ *if* $(Cnf, t) \in \delta$ *and* Cnf' *is the successor configuration of* t *in* Cnf*.*

The major contribution of the SE framework from [82] are the notions of *exhaustiveness* and *precision* defined subsequently.

Definition 6 (Exhaustive SE Transition Relations). *An SE transition relation* $\delta \subseteq 2^{SEStates} \times (2^{SEStates} \times 2^{SEStates})$ *is called* exhaustive *iff for each transition* (I, O) *in the range of* δ*,* $i \in I$ *and concrete states* $\sigma, \sigma' \in ConcrStates$*, it holds that* $\sigma' \in concr(i, \sigma)$ *implies that there is an SES* $o \in O$ *s.t.* $\sigma' \in concr(o, \sigma)$*.*

Definition 7 (Precise SE Transition Relations). *An SE transition relation* $\delta \subseteq 2^{SEStates} \times (2^{SEStates} \times 2^{SEStates})$ *is called* precise *iff for each transition* (I, O) *in the range of* δ*,* $o \in O$ *and concrete states* $\sigma, \sigma' \in \mathit{ConcrStates}$*, it holds that* $\sigma' \in \mathit{concr}(o, \sigma)$ *implies that there is an SES* $i \in I$ *s.t.* $\sigma' \in \mathit{concr}(i, \sigma)$*.*

The following lemmas (proved in [82]) connect exhaustiveness and precision with practice. *Test generation* requires precise SE to make sure that discovered failure states can be lifted to concrete, fault-inducing test inputs. Conversely, *program proving* requires exhaustive SE, s.t. a proof of the absence of assertion violations in the output SESs corresponds to a proof of the absence of errors in the inputs.

Lemma 1 (Bug Feasibility in Precise SE). *Let* δ *be a precise SE transition relation and* $\mathit{Cnf} \xrightarrow{(I,O)}_{\delta} \mathit{Cnf}'$*. If an assertion* $\varphi \in \mathit{Fml}$ *does not hold in* some *state* $o \in \mathit{Cnf}'$*, it follows that* there is *an* $i \in \mathit{Cnf}$ *s.t.* φ *does not hold in* i*.*

Lemma 2 (Validity of Assertions Proved in Exhaustive SE). *Let* δ *be an exhaustive SE transition relation and* $\mathit{Cnf} \xrightarrow{(I,O)}_{\delta} \mathit{Cnf}'$*. If an assertion* $\varphi \in \mathit{Fml}$ *holds* in all *states* $o \in \mathit{Cnf}'$*, it follows that* φ *holds in* all $i \in \mathit{Cnf}$*.*

The nice feature of these definitions is that they can be turned into a powerful *automatic testing procedure* for SE engines, as demonstrated subsequently.

3.3 An Oracle for Automatic Testing of SE Engines

From Definitions 6 and 7, we can derive an automated testing procedure for precision and exhaustiveness. Listing 2 shows the code of our testing routine for exhaustiveness; the version for precision works analogously. The algorithm specializes Definition 6: It only considers a finite number of initial states for concretization, does not account for state merging (i.e., only considers 1-to-n transitions), and is not robust against diverging programs (which could be mitigated by setting a timeout). We consider the most general input SES i (Line 5) for a given program counter `test_program`, i.e., one with an empty path condition and simple assignments `x` $\mapsto$ `x` for each variable in the set `variables`, which typically are the "free" program variables in `test_program`. Then, we take `num_runs` concrete states σ (Lines 12 to 16), compute $\{\mathit{concr}(i, \sigma)\}$ (Lines 22 and 23) and $\{\mathit{concr}(o, \sigma) | o \in O\}$ (Lines 18 to 20), where O are all output states produced by the symbolic interpreter (computed in Lines 7 to 10). We verify that there is an output state o satisfying the condition in Definition 6 by asserting that the former set is a subset of the latter one (Line 25). If this is not the case, we return the concrete input state used for concretization as a counterexample (Line 26). If no counterexample was found, we return **`None`** (Line 28).

Using the counterexample, we can examine the bug by comparing the outputs of the concrete and the symbolic interpreter (the symbolic interpreter produces only a single path since we start in a concrete state). Consider a simple **`while`**

Listing 2 Automatic search for counterexamples to exhaustiveness.

```
def find_exhaustiveness_counterexample(
        symbolic_interpreter,
        variables, test_program, num_runs=100):
    input_state = SymbolicEnvironment(SymbolicStore(
        {variable: variable.to_z3() for variable in variables}))

    output_states = [
        leaf.environment
        for leaf, _ in get_leaves(symbolic_interpreter.execute(
            test_program, input_state))]

    for _ in range(num_runs):
        sigma = Store({
            variable: random_val(variable.type)
            for variable in variables
        })

        concr_outputs = ConcrResultSet([
            concr(output_state, None, sigma)
            for output_state in output_states])

        concr_input = ConcrResultSet([
            concr(symbolic_input_state, test_program, sigma)])

        if not concr_input.subset_of(concr_outputs)
            return sigma  # Counterexample found

    return None  # No counterexample found
```

loop decrementing a variable `idx` by one as long as `idx >= x`. The exhaustiveness testing routine, for the baseline interpreter with a *loop unrolling* threshold of 2, produces an output like `{x: -36, idx: 93}`. Running the program in the concrete interpreter yields a final value of $-37 \neq 93$ for `idx`: They are indeed different! This is because the symbolic interpreter unrolled the loop only two times, and not the necessary `idx` − `x` = 130 times. Note that loop unrolling is precise, since there are always input states for which two times unrolling is sufficient. If the lack of exhaustiveness in the baseline interpreter was unexpected, a technique like [56] which only can detect *precision* problems would never have been able to find the problem.

Using the precision check, we discovered two real bugs in the interpreter. First, we did not consider negative array indices. In Python, `t[-i]` is equivalent to `t[len(t)-i]`. The second bug was more subtle. Integer division in Python is implemented as a "floor division," such that `1 // -2` evaluates to -1, because results are always floored. In Z3 and languages like Java, the result of the division is 0. We thus had to encode floor division in our mapping to Z3 expressions.

This approach has certain advantages over the technique proposed by Kapus & Cadar [56], apart from also supporting exhaustiveness checking. They distinguish runs of the SE engine in *single-path* and *multi-path* modes. For single-path, they uniquely constrain symbolic inputs to chosen concrete inputs (such that only a single program path is followed). However, they *obfuscate* these bindings by encoding them in sufficiently complicated ways to prevent the solver from *inferring* those concrete values. This is to prevent the executor from falling back to *concrete* execution, such that the actual SE engine would not be tested. For the multi-path mode, they cannot use the test oracle comparing outputs, and resort to the crash and "function call chain" oracles only. Our approach does not require outwitting the solver, and naturally handles the multi-path mode. Kapus & Cadar automatically generate test programs (program counters) using the CSmith tool [92]. This should be integrated into our approach; otherwise, test quality still depends on human judgment.

To the best of our knowledge, the technique we presented is only the second approach to automatic testing of SE engines, and the first to test exhaustiveness and apply an output-based oracle to multi-path executions.

4 Techniques

Previously, we described the characteristics of a baseline symbolic interpreter without much fuzz, and introduced the central notions of exhaustiveness and precision by which one can judge whether an SE engine is suitable for test generation or program proving. Now, we shed some light on different design alternatives and advanced techniques listed in Table 1. We consider both exhaustive (loop invariants) and precise techniques (concolic and selective SE) as well as orthogonal techniques (compositional SE, state merging). More technical details are discussed in the SE surveys [8,91] (which almost exclusively focus on such details). In particular, we omit areas such as the symbolic execution of *concurrent* programs and *memory models* for heap-manipulating programs. Concurrency adds to the path explosion problem, since different *interleaving* executions have to be considered [67]. The challenge is therefore to reduce the search space. This can be done, for instance, by restricting the class of checked properties (e.g., specifically to concurrency bugs [32,89] or regressions [43]), or by excluding irrelevant interleavings in the absence of potential data races [54,55]. Interesting topics related to symbolic memory modeling include the integration of Separation Logic [52,73], and Dynamic Frames [57] and the Theory of Arrays [2,34] into SE.

However, we think that the mentioned techniques are important design elements one should know and consider when analyzing and designing a symbolic executor.

In some cases, we extend the minipy language with new statement types to support a technique. Then, we also extend the concrete minipy interpreter to allow for an automatic cross-validation using the method described in Sect. 3.3.

4.1 Advanced Loop Treatment

Any symbolic executor has to ensure termination for programs with loops (and recursion). Our baseline interpreter implements bounded unrolling. This simple measure is precise, but not exhaustive; even when using SE for test generation and not program proving, that can be problematic if a bug hides beyond the set threshold.

Concolic Execution. *Concolic execution* (short for "*conc*rete and symb*olic* execution," coined in [77]) gracefully ensures termination. The idea is to let a *concrete input* steer the symbolic execution, collecting constraints along the way. Thus, the symbolic analysis terminates if, and only if, the *concrete execution terminates* for the given inputs. This can be implemented in an interpretation-based or execution-based way; for efficiency, most concolic engines are *execution-based.* To extract constraints from the program under test, those engines usually use runtime [93] or static *instrumentation* [21,69,77]. Another alternative is to run the tested program with *proxy objects* [10].

The baseline symbolic interpreter can be turned into an interpretation-based concolic executor *in only eight lines of code* (cf. [84]). We inherit from the baseline interpreter, and override the method `constraint_unsatisfiable` which is called, e.g., by the functions executing **`if`** statements and, in particular, loops, to check whether a path is feasible. Instead of directly calling Z3, we first *instantiate* the passed constraint to a variable-free formula, using a concrete state passed to the interpreter's constructor. Consequently, the choice of which execution branch to follow is uniquely determined. It is also much faster to check concrete than symbolic constraints. This can be further optimized: The authors of [15], e.g., created an optimized Z3 fork for concrete constraints.

For the `find` function from Listing 1 and the concrete state setting `needle` to `2` and `haystack` to the tuple `(1,)`, the concolic interpreter outputs an SET with a single, linear path. The constraints in the leaf node are (1) $0 <$ `len(haystack)`, (2) `haystack[0]` $\neq$ `needle`, and (3) $1 \geq$ `len(haystack)`. Concolic execution, e.g., as implemented in SAGE [39], negates these constraints one by one, keeping the constraints occurring *before* the negated one as they are; the constraints occurring afterward are not considered. Negating the first constraint yields an empty `haystack`; negating the second one, and keeping the first, some tuple containing `needle` in its first element. If we negate the third constraint, we obtain a `haystack` with more than one element, the first of which is `needle`. With the initial input and the second new one, we already obtain full branch coverage in `find`.

Listing (minipy) 3 Incomplete loop invariant encoding of `find` (from Listing 1).

```
i = 0
assert Inv(needle, haystack, i)

havoc i
assume Inv(needle, haystack, i)


if i < len(haystack):
    if haystack[i] == needle:
        break  # ???

    i = i + 1
    assert Inv
    assume False
else:
    return -1


return i
```

Listing (minipy) 4 `find` method with loop scope.

```
i = 0
assert Inv(i, needle, haystack)

havoc i
assume Inv(i, needle, haystack)

loop-scope(inv=Inv(i, needle, haystack)):
    if (i < len(haystack)):
        if (haystack[i] == needle):
            break  # OK now!

        i = i + 1
        continue # Signal next iteration

    else:
        return -1
    break  # Signal loop left

return i
```

Loop Invariants. A loop invariant [49] is a summary of a loop's behavior that holds *at the beginning of any loop iteration*. Thus, we can replace a loop with its invariant, even if we do not know how many times a loop will be executed (for recursion, *recursive contracts* take a similar role). In principle, loop invariants can be fully precise, overapproximating, or underapproximating [82, Sect. 5.4.2]. In practice, however, the underapproximating variant is rarely used. Test case generation, which would be the use case for such a scenario, typically uses specification-free approaches (e.g., concolic testing) to deal with loops. Fully precise invariants are ideal, but frequently hard to come up with. Thus, loop invariants, as used in program proving, are usually sufficiently strong (w.r.t. the proof goal), but not necessarily precise, *abstractions*.

Loop invariants can be encoded using assertions, assumptions, and "**`havoc`**" statements. This approach is followed, e.g., by the Boogie verifier [9]. We implemented this by adding two new statement types to minipy: **`assume`** *expr* adds an assumption *expr* to the path condition, and **`havoc`** `x` assigns a fresh, symbolic value to variable `x`, effectively erasing all knowledge its previous value. Loop invariants are based on an inductive argument: If a loop invariant holds *initially* and is *preserved* by any loop iteration, is can be used to abstract the loop (*use case*).

We apply this idea to the `find` function (Listing 1). An invariant for the loop is that `i` stays positive and does never grow beyond **`len`**`(haystack)`, and that at all previously visited positions, `needle` was not found (otherwise, we would have **`break`**ed from the loop). We extend the symbolic interpreter to take a list of *predicate definitions* that can be used similarly to Boolean-valued functions in minipy code. A predicate maps its arguments to a formula of type `z3.BoolRef`. The definition of $\texttt{Inv}(n, h, i)$ is $0 \le i \le \texttt{len}(h) \land (\forall 0 \le k \le i : h[i] \neq n)$.

We adapt the body of `find` to invariant-based SE in Boogie style. Listing 3 shows the result. In Line 2, we assert that the loop invariant `Inv` holds initially. Then, to enable reasoning about an *arbitrary* loop iteration, we **`havoc`** all variables (here only `i`) that are assigned in the loop `body` (Line 4). We assume the validity of the loop invariant (the induction hypothesis) in Line 5. The original **`while`** statement is transformed to an **`if`** (Line 8). In Line 13, we show that the loop invariant is *preserved* and still holds in the next iteration. If this check was successful, we **`assume`** falsity (Line 14), which renders the path condition unsatisfiable and causes SE to stop here.

Abrupt completion, such as the **`break`** statement in Line 10, complicates applying this method. Since we eliminated the loop, the **`break`** is syntactically illegal now. Addressing this requires a potentially *non-trivial transformation* of the loop body.

A solution to this problem is provided by Steinhöfel and Wasser in [86]. They propose so-called *loop scope statements* for invariant-based SE; in Listing 4 you find the loop scope version of `find`. One replaces the **`while`** loop with an **`if`** statement similarly to Listing 3. The **`if`** is put this inside a new **`loop-scope`** statement, which is passed the loop invariant `Inv`. Note that the original **`continue`** and **`break`** statements are preserved as is; indeed, the original loop body is not touched at all. Instead, a new **`continue`** statement is *added* as a last statement of the loop body, and a **`break`** statement is added as a last statement of the loop scope. The additional **`continue`** and **`break`** statements ensure that the body of the loop scope *always completes abruptly.* If it does so because of a **`continue`**, `Inv` is asserted; if it completes because of a **`break`**, the loop scope completes normally and execution is resumed. If the body completes for any other reason, the loop scope completes for the same reason.

Our symbolic interpreter does not transform the executed program into loop scope form on-the-fly as in [86] (which does not conform to our "interpreter style"). Instead, we implemented a program transformer which automatically turns loops into loop scope statements *before* they are symbolically interpreted. The complete implementation of the transformer spans only 18 lines of code.

4.2 Advanced Call Treatment

Function calls can cause two different kinds of problems in SE: First, there are too many feasible program paths in large, realistic programs. This leads to immense SETs in the case of interpretation-based SE, and many long, complicated path conditions to be processed in the case of concolic testing, both instances of path explosion. Second, the code of called functions might be unavailable, as in the case of library functions or system calls. There are two ways to address these problems. One is to use *summaries* of function behavior. Those can be manually specified, but also, in particular for test generation, automatically inferred. The other one is a non-exhaustive solution which, however, also works in the rare cases where summarization is not possible (e.g., for cryptographic hash functions): One concretizes function arguments to concrete values and simply *executes* the function non-symbolically. Existing constraints on variables not affected by the execution are retained, and SE can resume.

Compositional SE. Compositional SE works by analyzing functions individually, as opposed to complete systems. This is accomplished by annotating functions with summaries, which conjoin "constraints on the function inputs observed during the exploration of a path (...) with constraints observed on the outputs" [8]. Instead of symbolically executing a called function, we use its summary to obtain the resulting symbolic state, which can drastically reduce the overall search space.

Function summaries seem to have arisen independently in the areas of SE-based test generation and program verification. In the former area, Godefroid [36] introduces the idea, building on existing similar principles form interprocedural static analysis (e.g., [72]). As is common in automated test case generation, summaries are expected to be computed automatically; they are means for re-using previously discovered analysis results. Anand et al. [5] extend the original idea of function summaries to a demand-driven approach allowing lazy expansion of incomplete summaries, which further improves the performance of the analysis.

We did not find an explicit reference to the first original work using function summaries for modular, symbolic execution in the context of *program verification*. However, function summaries are already mentioned as a means for modularization in the first paper reporting on the KeY project [1], which appeared in 2000. Usually, function summaries are called "contract" in the verification context, inspired by the "design-by-contract" methodology [64]. Contracts are not only used for scalability, but additionally *define the expected behavior* of functions.

We integrated the latter variant of compositional SE, based on manually specified contracts, into our system by implementing a code transformer. The transformer replaces function calls by assertions of preconditions, **havoc** statements for assignment targets of function calls, and assumptions of postconditions. This resembles the "Boogie-style" loop invariant transformation described in Sect. 4.1, only that we do not verify that a function respects its contract when calling it. The verification can be done separately, for instance by **assert** statements in function bodies. Recall that since we support calls to externally specified predicates in **assert** statements, we can also assert properties that are outside the minipy language.

On-Demand Concretization. *Selective* SE interleaves concrete and symbolic execution on-demand. The authors of the S^{2}E tool [23] motivate this with the observation that there might be critical parts in a system one wants to analyze symbolically, while not caring so much about other parts. Two directions of context switches are supported: One can switch from concrete to symbolic (and back) by making function arguments symbolic and, after returning from the call, again concrete; and analogously switch from symbolic to concrete (and back). In our interpreter, one can switch from concrete to symbolic by adding a **havoc**

statement, which makes a concrete assignment to a variable symbolic again. For the other direction, we add a **`concretize`** statement assigning to a variable a concrete value satisfying the current path condition by querying Z3.

To mitigate negative effects of concretization on exhaustiveness, S^2E marks concrete execution results as *soft*. Whenever a subsequent SE branch is made inactive due to a soft constraint, the system backtracks and chooses a different concretization. To increase the effect, constraints are also collected during concrete execution (as in concolic testing), allowing S^2E to infer concretizations triggering different paths. Note that these optimizations are not possible if the code of invoked functions is not available, or they cannot be symbolically executed for other reasons.

4.3 State Merging to Mitigate Path Explosion

Loop invariants, function summaries, and on-demand concretization are all instruments for mitigating the path explosion problem of SE. *State merging* is another instrument for that purpose. It can be used together with the aforementioned ones, and there are both exhaustive and precise variants. Furthermore, there are (even fully precise) state merging techniques that do not require additional specification and work fully automatically. The idea is to take multiple SESs arising from an SE step that caused a case distinction (e.g., guard evaluation, statements throwing exceptions, polymorphic method calls) to one summary state. Different merge techniques have been proposed in literature (e.g., [17,46,61,78]); a framework for (exhaustive) state merging techniques is presented in [76] and subsumed by the more general SE theory proposed in [82] and discussed in Sect. 3.2.

A popular state merging technique uses If-Then-Else terms to summarize symbolic stores (e.g., [46,61,76]). Consider a simple program inverting a number `i` if it is negative. It consists of an **`if`** statement with guard `i < 0` and body `i = i * -1`. Two SESs arise from the execution of this statement: $(\{\mathtt{i} < 0\}, (\mathtt{i} \mapsto -\mathtt{i}))$ and $(\{\mathtt{i} \geq 0\}, (\mathtt{i} \mapsto \mathtt{i}))$. Merging those two states with the If-Then-Else technique results in $(\emptyset, (\mathtt{i} \mapsto \mathit{ITE}(\mathtt{i} < 0, -\mathtt{i}, \mathtt{i})))$. The right-hand side in the symbolic store evaluates to $-\mathtt{i}$ if `i` was initially negative, and to the (nonnegative) initial value otherwise. Path constraints are merged by forming the disjunction of the inputs' constraints, which in this case results in the empty (true) path condition.

To support state merging with the If-Then-Else technique in our symbolic interpreter, we add a **`merge`** statement to `minipy`. It is used like a **`try`** statement: One writes "**`merge:`**" and puts the statements after which to merge inside the scope of that statement. Conveniently, Z3 offers If-Then-Else terms, reducing implementation effort. Otherwise, we could introduce a fresh constant for merged values instead and define its value(s) in the path condition.

A fact not usually discussed in literature is that If-Then-Else-based state merging can be *imprecise* if the path constraints in merged states are *not mutually exclusive* [83]. This can happen, e.g., if one tries to merge different states arising from unrolling a loop. An alternative are guarded value summaries as proposed by [78]. If we allow *overlapping* guards, fully precise loop state merging is possible.

Finally, one should consider that state merging generally increases the complexity of SESs and thus the solver load, which has to be weighed up against the benefits from saved symbolic branches. In our experience, state merging pays off if one merges *locally* (i.e., the programs inside the **`merge`** statements should be small) and *early* in SE. Kuznetsov et al. [61] systematically discuss this problem and devise a metric by which one can automatically decide whether or not to merge.

5 Applications

The strength of SE lies in its ability to *deterministically* explore many program paths with *high precision*. This is in contrast to *fuzzing* [65], including language-based [50,51] and coverage-based fuzzing [14], where it always depends on chance and time whether a program path is reached. Only the integration of SE into *whitebox* fuzzing approaches [38,39] enables fuzzers to *enforce* coverage of specific paths. On the other side of the spectrum are *static* analysis techniques such as *Abstract Interpretation (AI)* [25]. AI is fully automatic, but designed to operate on an *abstract domain*, which is why full precision is generally out of reach.

While one could, in principle, regard SE as a specialization of AI, there are striking differences. Amadini et al. [4] argue that SE is essentially a *dynamic* technique, as it *executes* programs in a forward manner and generally *under-approximates* the set of reachable states (unless supported by additional, costly specifications). Intrinsically *static* techniques, on the other hand, produce (possibly false) alarms and generally focus on smaller code regions. In their work, the authors provide a detailed discussion on the relation between the two techniques.

Consequently, SE is popular in the area of *test generation*, where high precision is vital. Yet, it *has* been successfully applied in *program proving*. Here, its precise, dynamic nature is a problem: When one *needs* abstraction, especially in the case of loops or recursive methods with symbolic guards or arguments, *manual specifications* are required. As pointed out in [2], specifications are the "new" bottleneck of SE-based program proving. On the other hand, SE-based proofs can address strong functional properties, while abstract interpreters like ASTRÉE [26] address coarser, general properties (e.g., division by zero or data races).

5.1 Test Generation

Precise SE is a strong tool for automatically generating high-coverage test cases. Frequently, the SE variant used to that end is referred to as *Dynamic Symbolic Execution (DSE)*. Baldoni et al. [8] define this term as an *interleaving* of concrete and symbolic execution. Thus, DSE subsumes *concolic* and *selective* SE, which we discussed in Sects. 4.1 and 4.2. Although each concrete input used for concolic execution corresponds to exactly one symbolic path, concolic SE still suffers from path explosion. A symbolic path is associated to a set of atomic path constraints; one has to pick and negate one constraint, which may in turn result in a new path explored and new constraints to be negated. DART [38], e.g., chooses a depth-first strategy; SAGE [40], a tool which "has saved Microsoft millions of dollars" [40], uses *coverage information* to rank new inputs generated from constraint negation. As in the case of mutation-based graybox fuzzers like AFL[3], the quality of the generated tests depends on the chosen initial input (which explains why concolic test generators are frequently referred to as *whitebox fuzzers* [8]). If this input, for example, is rejected as invalid by the program (e.g., by an input parser), it may take many rounds of negation and re-exploration before the actual program logic is reached. One way to address the problem of complex, structured input strings is to integrate *language specifications* with whitebox fuzzing [37].

SAGE's use of coverage information to rank candidate inputs can already be seen as an integration of classic graybox fuzzing à la AFL. Driller [87] is a *hybrid fuzzer* using selective SE to identify different program *compartments*, while inexpensive mutation-based fuzzing explores paths within those compartments. In the QSYM [93] and FUZZOLIC [15] systems, the concolic component runs in parallel with a coverage-guided fuzzer. Both systems loosen the usual soundness requirements of SE: QSYM by "optimistic" solving (ignoring some path constraints), and FUZZOLIC by approximate solving using *input mutations*. For their benchmarks, the respective authors showed that QSYM outperforms Driller in terms of generated test cases per time and achieved code coverage, and FUZZOLIC outperforms QSYM (but less clearly). Both QSYM and FUZZOLIC scale to real-world programs, such as `libpng` and `ffmpeg`, and QSYM found 13 new bugs in programs that have been heavily fuzzed by Google's OSS-Fuzz[4].

Interpretation-based SE engines can also be used for (unit) test generation. The KeYTestGen [2,29] tool, for example, executes programs with bounded loop unrolling, and obtains inputs satisfying the path conditions of the leaves in the resulting SET. With the right settings, the tool can achieve full MC/DC coverage [47] of the program under test [2]. As can be expected, however, this can lead to an explosion of the analysis costs and numbers of generated test cases.

5.2 Program Proving

The goal of program proving is not to demonstrate the presence, but the *absence* of bugs, for any possible input. In our impression, SE is less popular in program proving than in test generation. One possible reason might be that *exhaustive* SE, as required for program proofs (cf. Sect. 3.2), generally needs auxiliary specifications. Yet, program provers based on *Weakest Precondition (WP)* [28] reasoning, such as Boogie [9] and Frama-C [27], are closely related. A WP is a formula

[3] https://github.com/google/AFL.
[4] https://github.com/google/oss-fuzz.

implied by *any* precondition strong enough to demonstrate that a program satisfies a given postcondition. Traditionally, WP-based systems compute WPs by executing a program starting from the leaves of its Control Flow Graph (CFG), specializing the postcondition in each step. SE, in turn, computes WPs by *forward* execution. Both approaches require specifications of loops and recursive functions.

To our knowledge, there are two actively maintained SE-based program provers. VeriFast [52] is a verifier for single- and multithreaded C and Java programs specified with separation logic assertions. The tool has been used in four industrial case studies [68] to verify the absence of general errors such as memory leaks. The authors discovered bugs in each case study. KeY [2] is a program prover for single-threaded Java programs specified in the Java Modeling Language. Several sorting algorithms have been verified using KeY: Counting sort and Radix sort [42], the TimSort hybrid sorting algorithm [41], and Dual Pivot QuickSort [11]. For TimSort and Dual Pivot QuickSort, the actual implementations in the OpenJDK have been verified; TimSort (which is also used in Android and Python) is OpenJDK's default sorting algorithm, and Dual Pivot QuickSort the default for primitive arrays.

The TimSort case study gained particular attention. During the verification, the authors of [41] discovered a bug present in the TimSort implementations from Android's and Sun's, and the OpenJDK, as well as in the original Python version. When one asks the (unfixed) algorithm to sort an array with sufficiently many segments of consecutive elements, it raises an `ArrayOutOfBoundsException`.

Dual Pivot QuickSort was successfully proven correct; however, a loop invariant specified in natural language was shown to be wrong.

Testing tools like QSYM and FUZZOLIC use "unsound" techniques to find more bugs faster. SE-based program proving can go the other way and integrate *abstraction* techniques inspired by Abstract Interpretation to increase automation. In [2, Chapter 6], several abstract domains for SE of Java programs, especially for heaps and arrays, are introduced. This approach retains full precision in the absence of loops or recursion. Should it be necessary, abstraction is applied automatically to find, e.g., the fixed point of a loop. However, only the changed portion of the state inside the loop is abstracted. Integrating SE with reasoning in abstract domains thus yields a program proving approach with the potential of full automation and increased precision compared to Abstract Interpretation. Unfortunately, to our knowledge, there exists no mature implementation of this approach.

5.3 Symbolic Debugging

Several works independently introducing SE [16,19,58] were motivated by *debugging*. Indeed, *symbolic* debugging has several advantages: (1) Dynamic debugging requires setting up a failure-inducing state, which can be nontrivial especially if one aims to debug individual functions deeply nested in the program. Using SE, one can take an over-approximating *symbolic* state. (2) The SE variant we call *transparent*, which maintains full SETs at the granularity of individual statements, has the potential to implement an *omniscient* debugger, which is hard to implement efficiently for dynamic debugging [71]. This enables programmers to arbitrarily step *back and forth* during debugging as needed.

To implement a symbolic debugger, one does not necessarily need an exhaustive SE engine. However, concolic approaches are unsuitable, since they do not construct SETs and might not have a notion of symbolic stores. Thus, in our opinion, an interpretation-based approach is required to implement symbolic debugging.

The first implementations of symbolic debuggers were *independently* developed in 2010, in the context of the VeriFast and KeY program verifiers [44,53]. In [48], an improved implementation of KeY's Symbolic Execution Debugger (SED) was presented. Both tools allow forward and backward steps in execution paths and inspection of path constraints and symbolic memory. The VeriFast debugger supports the analysis of a single error path in the SET, while the SED shows full SETs. This is also motivated by different application scenarios: VeriFast offers debugging facilities for failed proof attempts, while the SED inventors explicitly address the scenario of debugging in absence of a proof attempt. The SED supports some extended features like visualization of different heap configurations or highlighting of subformulas in postconditions whose verification failed. In VeriFast, heaps are represented with separation logic assertions and not visualized.

Our minipy symbolic interpreter supports a limited degree of symbolic debugging. It visualizes SETs and explicitly represents path constraints and symbolic stores in nodes. Leaves are highlighted using the following color scheme: Red leaves represent raised exceptions (including failed assertions), green leaves represent nodes with unsatisfiable path conditions, and blue leaves all other cases.

5.4 Model Checking of Abstract Programs

Model checking usually abstracts the program under test into a graph-like structure and exhaustively searches this model to show the absence of (mostly generic) errors. Although the areas of model checking and formal software verification (including SE-based program proving) are slowly converging [79], one would usually not directly relate SE and model checking. Recently, however, a SE-based technique named Abstract Execution (AE) was proposed [82,85], which allows for a rigorous analysis of *abstract program models*. An abstract program model is a program containing *placeholder* statements or expressions. These placeholders represent arbitrary statements or expressions conforming to the placeholder's specifications. For example, one can restrict which locations a placeholder can read from or write to, define under which conditions instantiations complete abruptly, and impose postconditions for different (abrupt or normal completion) cases. AE is not intended to scale to big programs. Rather, it is used to *model* program verification problems that *universally quantify* over statements or expressions.

An example are *program transformations*, represented by *pairs* of abstract programs. In [82], we modeled nine Java *refactoring* techniques, and extracted preconditions for semantics-preserving transformations. Usually, the requirements for a behavior-preserving refactoring application are incompletely described in literature. A manual extraction of test cases from the models unveiled several bugs in the *Extract Method* refactoring implementations in IntelliJ IDEA and Eclipse. Our reports have been assigned "major" or "normal" priority and lead to bug fixes.[5] Other applications of AE include *cost analysis* of program transformations [3], *parallelization* of sequential code [45], the delta-based *verification of software product families* [75], and "correct-by-construction" program development [90].

We implemented AE within the KeY system. One noteworthy feature of KeY is that it *syntactically* represents state changes within its program logic. This made it easy to add *abstract updates* representing the abstract state changes caused by placeholder statements or expressions. Indeed, we discovered that it is not so straightforward to implement AE, especially when using *dynamic frames* to represent underspecified memory regions, on top of our minipy symbolic interpreter.

AE is a noteworthy showcase of interpretation-based, exhaustive SE: It does not *need* to scale to large programs, since the goal is not program verification but, e.g., the *verification of transformations.* Thus, it covers a niche that cannot be adequately addressed by concolic testing or fuzzing, which can only ever consider pairs of *concrete* programs resulting from the *application* of a transformation.

6 Future Perspectives

The success of modern automated testing techniques (most prominently coverage-guided mutation-based fuzzers) cannot be denied. When writing this sentence, Google's oss-fuzz had discovered 34,313 bugs in 500 open source projects. The 30 bugs in JavaScript engines discovered by the LangFuzz tool [51] translate to about 50,000 US$ in bug bounties. There are two key advantages of blackbox or graybox techniques over *systematic* testing approaches like SE: (1) They require no or only little code instrumentation and are applicable to any program for which a compiler or interpreter exists. (2) They are *fast*, with the only threshold being the execution time of the program under test. Considering Item (1), SE either crashes or outputs useless results if a program uses an unsupported statement or expression type. For that reason, many engines operate on Intermediate Languages (ILs) like LLVM, which commonly comprise a few dozen different instructions, while CPU instruction sets can easily reach hundreds to thousands [69]. For this paper, we substantially restricted the features of the minipy language to reduce the implementation effort. Not to mention that implementing a symbolic executor on *source level* for a language like Python

[5] For example, **IDEA-271736**: "'Extract Method' of 'if-else if' fragment with multiple returns yields uncompilable code," https://youtrack.jetbrains.com/issue/IDEA-271736.

with many high-level (e.g., functional and object-oriented) abstractions is highly nontrivial.

Let us assume that Item (1) has been adequately addressed (e.g., using a stable IL, or, as in the case of the SymQEMU system [70], a CPU emulation framework). Addressing the question of speed (Item (2)), it is tempting to say—if you are a supporter of SE—that analysis speed does not matter so much, since SE covers program paths *systematically*, while fuzzers rely mostly on chance. Instead of the *effectiveness* of a verification strategy, i.e., its ability to inspire a *maximum* degree of confidence in the correctness of a program, Böhme and Paul [13] suggest to put *efficiency* into the focus. In their words, an *efficient* verification technique (1) generates a sufficiently effective test suite in minimal time or (2) generates the most effective test suite in the given time budget. They address the efficient verification problem with a probabilistic analysis. Assume that the cost associated to the generation of one input by a random input generator R is 1. We compare R to a *systematic* input generator S (e.g., a concolic tester) sampling an input with cost c. Böhme and Paul prove that, for a desired *confidence* x, the time taken by S to sample an input *must not exceed* $(ex - ex^2)^{-1}$ times the time taken by R to sample an input. Otherwise, R is expected to *achieve confidence* x *earlier*. If R, e.g., needs 10 ms to generate one test input, and we aim to establish that a program works correctly for 90% of its inputs, then S must take *less than 41 ms* to come up with an input [13]. In the face of this observation, we must ask ourselves the question: *Is it worth investing in SE techniques, or should we simply concentrate on improving randomized automated testing approaches?*

Subsequently, we discuss scenarios where SE can *assist or outperform* randomized approaches, demonstrating that it has a place in future software verification.

Fast Sampling. As pointed out by Böhme and Paul, efficiency is key for a verification approach to be practically adopted. Recent work on compilation-based SE [69,70] and "fuzzy" constraint solving [15,93] address the *execution* and *constraint solving* components, which are the main bottlenecks of SE. Compromising soundness is justified if the analysis *discovers bugs fast*, and stays within the critical bound from [13].

Hybrid Fuzzing Tools like DRILLER [87], SAGE [40] and QSYM [93] combining SE with coverage-guided fuzzing showed promising results. In their paper [13], Böhme and Paul propose a hybrid approach switching from a random to a systematic tester when the expected time estimate of the random tester to detect the next *partition* exceeds a threshold. They proved, and demonstrated in simulations, that the hybrid tester is *at least* as efficient as the most efficient combination of the elementary testers. Finally, Bundt et al. [18] showed in a recent measurement study that "Hybrid fuzzers such as QSYM that integrate concolic execution to solve path constraints clearly outperform approaches that adopt a brute-force strategy."

Verification of Critical Routines. Coverage alone would not have sufficed to unveil the TimSort bug [41]: How should a concolic (not to mention a random) tester come up with an array of 67,108,864 elements with sufficiently many short consecutive sections to trigger the "out of bounds" exception in a method that has been *extensively* used in practice for years? Interpretation-based, exhaustive SE is still one of the best techniques to ensure that there is *really* no algorithmic bug left even after heavy testing. Yet, we think that this will always require much person-power: Abstraction techniques *can* help exhaustive SE to get more automatic. This, however, comes at the cost of precision, such that ground is lost to competing, (more) automatic static analysis techniques. The golden bullet of fully precise (loop) summarization will probably be forever out of reach for realistic programs. Thus, we think that the program proving community should invest into better tooling for writing and fixing specifications, as well as into communicating idioms and best practices for verification-friendly program development, than to develop the next incomplete loop invariant inference approach.

SE for Model Checking. One way to remove the scalability issues of, in particular, interpretation-based SE, is to focus on small, *yet meaningful*, problems. The work on Abstract Execution is such an example. AE builds a modeling language *on top* of Java to express, e.g., program transformations. Using strong contracts with a variable degree of abstraction, practically relevant correctness properties of transformations are *derived* and proven correct. There are two ways of applying these results. One is to *prove* that the derived properties hold for actual transformations. This means that one has to show that an input to, e.g., a refactoring technique, satisfies a set of non-trivial preconditions, which can require coming up with strong loop invariants. Instead, we suggest to automatically derive *test cases* or *assertions* from the abstract model that are tailored to the transformed, concrete program. This brings together strong "once-and-for-all" results obtained from a heavyweight technique requiring annotations with targeted automatic testing of realistic programs: The best of two worlds.

7 Conclusion

Symbolic Execution is a popular, precise program exploration technique that can be lifted to an exhaustive approach to program proving. The SE community is split into two rarely interacting sub-communities: One dedicated to *finding bugs*, and one aiming to *prove their absence*. In this paper, we attempted an application-agnostic analysis of the *foundations* of SE. We provided a framework for classifying symbolic engines, and showed how to design and extend a symbolic interpreter. For illustrative purposes and to foster a deeper understanding, we implemented most aspects inside a new SE framework for a Python subset. We recapitulated a semantics for SE applying to both test generation and program proving, and derived from it a *novel automated testing approach* for SE engines. Finally, we elaborated on chosen applications of SE, ranging from test generation over symbolic debugging to model checking of abstract programs, and discussed the role of SE in future software verification.

The digital companion volume including our implementations is available at

https://rindphi.github.io/se-book-festschrift-rh

References

1. Ahrendt, W., et al.: The approach: integrating object oriented design and formal verification. In: Ojeda-Aciego, M., de Guzmán, I.P., Brewka, G., Moniz Pereira, L. (eds.) JELIA 2000. LNCS (LNAI), vol. 1919, pp. 21–36. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40006-0_3
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Albert, E., Hähnle, R., Merayo, A., Steinhöfel, D.: Certified abstract cost analysis. In: FASE 2021. LNCS, vol. 12649, pp. 24–45. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_2
4. Amadini, R., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: Abstract interpretation, symbolic execution and constraints. In: de Boer, F.S., Mauro, J. (eds.) Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift. OASIcs, Bologna, Italy, 27 November 2020, vol. 86, pp. 7:1–7:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASIcs.Gabbrielli.7
5. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_28
6. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. Int. J. Softw. Tools Technol. Transf. **11**(1), 53–67 (2009). https://doi.org/10.1007/s10009-008-0090-1
7. Assaraf, A.: This is what your developers are doing 75% of the time, and this is the cost you pay (2015). https://tinyurl.com/coralogix. Accessed 08 Oct 2021
8. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657
9. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
10. Barsotti, D., Bordese, A.M., Hayes, T.: PEF: python error finder. In: Selected Papers of the XLIII Latin American Computer Conference (CLEI). Electronic Notes in Theoretical Computer Science, vol. 339, pp. 21–41. Elsevier (2017). https://doi.org/10.1016/j.entcs.2018.06.003
11. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 35–48. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
12. de Boer, F.S., Bonsangue, M.: On the nature of symbolic execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 64–80. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_6

13. Böhme, M., Paul, S.: A probabilistic analysis of the efficiency of automated software testing. IEEE Trans. Softw. Eng. **42**(4), 345–360 (2016). https://doi.org/10.1109/TSE.2015.2487274
14. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. **45**(5), 489–506 (2019). https://doi.org/10.1109/TSE.2017.2785841
15. Borzacchiello, L., Coppa, E., Demetrescu, C.: Fuzzing symbolic expressions. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE) 2021, pp. 711–722. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00071
16. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT - a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software, pp. 234–245. ACM, New York (1975)
17. Bubel, R., Hähnle, R., Weiß, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04167-9_13
18. Bundt, J., Fasano, A., Dolan-Gavitt, B., Robertson, W., Leek, T.: Evaluating synthetic bugs. In: Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS) (2021, to appear)
19. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Rosenfeld, J.L. (ed.) Proceedings of the 6th IFIP Congress 1974 on Information Processing, pp. 308–312. North-Holland (1974)
20. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI, pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
21. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) 13th ACM Conference on Computer and Communications Security, (CCS), pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
22. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
23. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: Gupta, R., Mowry, T.C. (eds.) Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 265–278. ACM (2011). https://doi.org/10.1145/1950365.1950396
24. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
25. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
26. Cousot, P., et al.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3

27. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
28. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
29. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73770-4_10
30. Engler, D.R., Dunbar, D.: Under-constrained execution: making automatic code destruction easy and scalable. In: Rosenblum, D.S., Elbaum, S.G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 1–4. ACM (2007). https://doi.org/10.1145/1273463.1273464
31. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
32. Farzan, A., Holzer, A., Razavi, N., Veith, H.: Con2colic testing. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013), pp. 37–47. ACM (2013). https://doi.org/10.1145/2491411.2491453
33. Furia, C.A., Meyer, B.: Inferring loop invariants using postconditions. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation. LNCS, vol. 6300, pp. 277–300. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15025-8_15
34. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
35. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, pp. 339–347. ACM (2014). https://doi.org/10.1145/2628136.2628138
36. Godefroid, P.: Compositional dynamic test generation. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 47–54. ACM (2007). https://doi.org/10.1145/1190216.1190226
37. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI), pp. 206–215. ACM (2008). https://doi.org/10.1145/1375581.1375607
38. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 213–223. ACM (2005). https://doi.org/10.1145/1065010.1065036
39. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, (NDSS) 2008. The Internet Society (2008). https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

40. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564
41. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reason. **62**(1), 93–126 (2017). https://doi.org/10.1007/s10817-017-9426-4
42. de Gouw, S., de Boer, F., Rot, J.: Proof pearl: the KeY to correct and stable sorting. J. Autom. Reason. **53**(2), 129–139 (2014). https://doi.org/10.1007/s10817-013-9300-y
43. Guo, S., Kusano, M., Wang, C.: Conc-iSE: incremental symbolic execution of concurrent software. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2016, pp. 531–542. ACM (2016). https://doi.org/10.1145/2970276.2970332
44. Hähnle, R., Baum, M., Bubel, R., Rothe, M.: A visual interactive debugger based on symbolic execution. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 143–146. ACM (2010). https://doi.org/10.1145/1858996.1859022
45. Hähnle, R., Heydari Tabar, A., Mazaheri, A., Norouzi, M., Steinhöfel, D., Wolf, F.: Safer parallelization. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12477, pp. 117–137. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_8
46. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 76–92. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_6
47. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. Technical report, TM-2001-0057789, NASA Technical Reports Server, May 2001. https://ntrs.nasa.gov/citations/20010057789
48. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 255–262. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_21
49. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
50. Hodován, R., Kiss, Á., Gyimóthy, T.: Grammarinator: a grammar-based open source fuzzer. In: Prasetya, W., Vos, T.E.J., Getir, S. (eds.) Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST@ESEC/SIGSOFT FSE), pp. 45–48. ACM (2018). https://doi.org/10.1145/3278186.3278193
51. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium. pp. 445–458. USENIX Association (2012). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
52. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4

53. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
54. Kähkönen, K., Saarikivi, O., Heljanko, K.: Using unfoldings in automated testing of multithreaded programs. In: Goedicke, M., Menzies, T., Saeki, M. (eds.) Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), pp. 150–159. ACM (2012). https://doi.org/10.1145/2351676.2351698
55. Kamburjan, E., Scaletta, M., Rollshausen, N.: Crowbar: behavioral symbolic execution for deductive verification of active objects. CoRR abs/2102.10127 (2021)
56. Kapus, T., Cadar, C.: Automatic testing of symbolic execution engines via program generation and differential testing. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 590–600. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115669
57. Kassios, I.T.: The dynamic frames theory. Formal Asp. Comput. **23**(3) (2011). https://doi.org/10.1007/s00165-010-0152-5
58. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
59. Kneuper, R.: Symbolic execution: a semantic approach. Sci. Comput. Program. **16**(3), 207–249 (1991). https://doi.org/10.1016/0167-6423(91)90008-L
60. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_33
61. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Vitek, J., Lin, H., Tip, F. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 193–204. ACM (2012). https://doi.org/10.1145/2254064.2254088
62. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: a coinductive approach. J. Symb. Comput. **80**, 125–163 (2017). https://doi.org/10.1016/j.jsc.2016.07.012
63. McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_10
64. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
65. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
66. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
67. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785
68. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: industrial case studies. Sci. Comput. Program. **82**, 77–97 (2014). https://doi.org/10.1016/j.scico.2013.01.006

69. Poeplau, S., Francillon, A.: Symbolic execution with SymCC: don't interpret, compile! In: Capkun, S., Roesner, F. (eds.) Proceedings of the 29th USENIX Security Symposium, pp. 181–198. USENIX Association (2020)
70. Poeplau, S., Francillon, A.: SymQEMU: compilation-based symbolic execution for binaries. In: Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2021). https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/
71. Pothier, G., Tanter, É., Piquer, J.M.: Scalable omniscient debugging. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 535–552. ACM (2007). https://doi.org/10.1145/1297027.1297067
72. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Cytron, R.K., Lee, P. (eds.) Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 49–61. ACM Press (1995). https://doi.org/10.1145/199448.199462
73. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS) 2002, pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
74. Rollbar: The State of Software Code Report (2021). https://content.rollbar.com/hubfs/State-of-Software-Code-Report.pdf. Accessed 08 Oct 2021
75. Scaletta, M., Hähnle, R., Steinhöfel, D., Bubel, R.: Delta-based verification of software product families. In: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2021, pp. 69–82. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3486609.3487200
76. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 57–73. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_5. The author Dominic Scheurer is the same person as the author of this paper
77. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
78. Sen, K., Necula, G.C., Gong, L., Choi, W.: MultiSE: multi-path symbolic execution using value summaries. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 842–853. ACM (2015). https://doi.org/10.1145/2786805.2786830
79. Shankar, N.: Combining model checking and deduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 651–684. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_20
80. Shoshitaishvili, Y., et al.: SOK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, 22–26 May 2016, pp. 138–157. IEEE Computer Society (2016). https://doi.org/10.1109/SP.2016.17

81. Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27** (2017). https://doi.org/10.1017/S0956796817000090
82. Steinhöfel, D.: Abstract execution: automatically proving infinitely many programs. Ph.D. thesis, TU Darmstadt, Department of Computer Science, Darmstadt, Germany (2020). https://doi.org/10.25534/tuprints-00008540
83. Steinhöfel, D.: Precise Symbolic State Merging (2020). https://www.dominic-steinhoefel.de/post/precise-symbolic-state-merging/. Accessed 25 Nov 2021
84. Steinhöfel, D.: Symbolic Execution: Foundations, Techniques, Applications and Future Perspective, Digital Companion Volume (2021). https://rindphi.github.io/se-book-festschrift-rh. Accessed 10 May 2022
85. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 319–336. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_20
86. Steinhöfel, D., Wasser, N.: A new invariant rule for the analysis of loops with non-standard control flows. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 279–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_18
87. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: 23rd Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2016)
88. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 21–36. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40447-4_2
89. Wang, C., Kundu, S., Limaye, R., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. Formal Aspects Comput. **23**(6), 781–805 (2011). https://doi.org/10.1007/s00165-011-0179-2
90. Winterland, D.: Abstract execution for correctness-by-construction. Master's thesis, Technische Universität Braunschweig (2020)
91. Yang, G., Filieri, A., Borges, M., Clun, D., Wen, J.: Advances in symbolic execution. In: Memon, A.M. (ed.) Advances in Computers, Advances in Computers, vol. 113, pp. 225–287. Elsevier (2019). https://doi.org/10.1016/bs.adcom.2018.10.002
92. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 283–294. ACM (2011). https://doi.org/10.1145/1993498.1993532
93. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: Enck, W., Felt, A.P. (eds.) Proceedings of the 27th USENIX Security Symposium 2018, pp. 745–761. USENIX Association (2018)

Locally Abstract Globally Concrete Semantics of Time and Resource Aware Active Objects

Silvia Lizeth Tapia Tarifa(✉)

Department of Informatics, University of Oslo, Oslo, Norway
sltarifa@ifi.uio.no

Abstract. Active objects provide a powerful conceptual model of distributed systems. This paper presents an extension of a locally abstract, globally concrete trace semantics of an active object language with time and resources. The proposed extension keeps the flexibility of the framework in which abstract traces are generated, one rule per each syntactic construct of the language, to later be composed in a concrete global context where symbolic values are resolved. The paper also includes a running example to show how abstract and concrete traces are constructed.

1 Introduction

Active objects provide a powerful conceptual model of distributed systems [10]. Active object languages combine the basic Actor model [1] with object-oriented concepts. Communication between active objects is realized by means of asynchronous method calls. The notion of method calls can further support return values while maintaining a loose coupling between actors; by means of future variables [9]. A future can be seen as a mailbox that will eventually contain the return value from a method call. Therefore, the calling method may proceed with its computation and pick up the reply later. Such synchronization mechanism can be combined with cooperative scheduling of the method activation of an object, using an explicit statement to release control, allowing interleaved execution inside an active object with its own thread of execution. To express *timed models*, this paper integrates a global clock and local timers in the different method calls to capture *explicit* passage of time [6]. To make a separation of concern between *cost* and *resource capacity*, this paper considers cost statements consuming restricted amount of resources that are available per time interval in concrete cost centres, to capture *implicit* passage of time, as described in [18].

To establish formal properties of a language as well as for formal verification, a formal semantics is required. This paper presents the trace semantics of an active object language which is time- and resource-sensitive [6,18]. This paper aims for the semantics to be locally abstract and globally concrete (LAGC), and therefore uses LAGC semantics [12,13], a semantics that is intended to align with contract-based deductive verification [16]. The semantics uses a generalized version of program trace [17], capturing a sequence of transitions, starting in

W. Ahrendt et al. (Eds.): The Logic of Software. A Tasting Menu of Formal Methods, LNCS 13360, pp. 481–499, 2022.
https://doi.org/10.1007/978-3-031-08166-8_23

some initial local symbolic state, e.g., *symbolic execution* [7,19]. This semantics is compositional and generates global traces from abstract (symbolic) local traces, using a hybrid approach that combines states and events, and is is well-suited to capture the concurrency model of active objects.

Symbolic states allow to evaluate local code independently of its call context. For example, when evaluating the semantics of the explicit passage of time in a global clock, executed locally in an object, such object cannot locally know the explicit value of the global clock, since it is in its context. In this case the local evaluation can be expressed in a trace with symbolic states and parameters. Both, the symbolic states and the parameters are *concretised* when the global context of the local computations (e.g., the global clock) is resolved, hence, the resulting global traces are concrete.

The extension of the semantics with time and resources builds on recent work with Reiner Hähnle, which proposes a framework for trace semantics [12,13], that scales flexibly to a range of concurrent, imperative programming paradigms, including active objects [10]. This paper keeps the *modularity* of this semantics in which modular extensions of the language are added without the need to revise the whole framework.

The paper is organised as follows: Sect. 2, introduces the syntax of a time- and resource-sensitive active object language. Section 3 summarises the run-time syntax of abstract traces. Sections 4–5 present the local and global semantics and the trace composition of the language. Related work is discussed in Sect. 6, while Sect. 7 summarises and concludes the paper. The paper also contains a running example to illustrate the main building blocks of the proposed semantics.

2 A Core Active Object Language with Time and Resources

Let us consider a mini active object language [10] which considers the actor paradigm [1] integrated with object-oriented concepts, that behaves autonomously and concurrently, and communicates with other actors without transfer of control through asynchronous method calls and futures [9]. The language in addition is time- and resource-sensitive. Resembling the Real-Time ABS language [18,22]. We call this language mini Real-Time ABS.

The formal syntax of the language is given in Fig. 1, which highlights in grey the statements and expressions related to time and resources and surrounds in a white box the statements that are added at run-time. A program P consists of a set of classes and a main block sc. The main block consists of variable declarations $\overline{x}$ and a method body with statements s. A class CL has a name C, a number of fields $\overline{x}$, passed as parameters to the class and a set of methods. A method M has a name m, a number of local variables $\overline{x}$, passed as parameters to the method, and a method body with statements s. Statements s are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, and **return** constructs. In addition to expressions e, the right hand side of an assignment $x{:=}rhs$ includes method calls $o!m(\overline{e})$, future dereferencing e.**get**, object creation **new** $C(\overline{e})$, and

Synt. categories.	*Definitions.*
C, m in Name	$P ::= \overline{CL}\ \mathrm{sc}$
s in Stmt	$CL ::= \textbf{class}\, C(\overline{x})\ \overline{M}$
r in ResId	$M ::= m(\overline{x})\{s; \textbf{return}\ e\}$
x in Var	$\mathrm{sc} ::= \{\overline{x}; s\}$
v in Val	$s ::= \textbf{skip} \mid x = rhs \mid \textbf{if}\ e\ \{\ s\ \} \mid s; s \mid \textbf{await}\ g \mid$
e in Expr	$\quad \textbf{duration}(\mathrm{e}) \mid \textbf{cost}(\mathrm{e,e}) \mid \boxed{\textbf{rtDur}(\mathrm{v,v})}$
g in Guard	$rhs ::= e \mid e.\textbf{get} \mid e!m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid \textbf{new res}\ \mathrm{r(e)}$
	$e ::= x \mid v \mid e\ op\ e \mid \textbf{this} \mid \textbf{now}$
	$g ::= e \mid e?$

Fig. 1. Syntax of mini-Real-Time ABS. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories.

resource creation **new res** $r(e)$. *Expressions* e include variables x, values v, standard operations op over expressions, the self-identifier for objects **this**, and the expression **now**, which evaluates to the current value of the global clock.

Cooperative scheduling is achieved by explicitly suspending the execution of the current active process, the statement **await** g conditionally suspends its execution, the guard g controls the suspension and consists of a Boolean condition e or the return tests $e?$ (explained in the next paragraph). The evaluation of a guard g is side-effect free. However, if g evaluates to false, the current executing process is *suspended*, letting other processes to be selected for execution from the pool of suspended processes by means of a default scheduling policy (e.g., random). *Communication* and *synchronization* are decoupled in active objects [9]. Communication is based on asynchronous method calls, denoted by assignments $f{=}e!m(\overline{e})$ to future variables f. Here, e is an object expression, m a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. After calling $f{=}o!m(\overline{e})$, the future variable f refers to the return value of the call, and the caller may proceed with its execution *without blocking*. The language includes two operations on future variables to control synchronization. First, the guard **await** $f?$ *suspends the active process* unless a return to the call associated with f has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all executions* in the object until the return value is available.

The local passage of time can be modelled both *explicitly* and *implicitly*. With explicit time, the modeller inserts duration statements **duration**(e) with an execution time e. This is the standard approach to model timed behaviour, well-known from, e.g., timed automata in UPPAAL [20]. A special statement is introduced at run-time **rtDur**(v, v) to capture ongoing executions of duration statements. The language also considers *cost centres* with *resources* [18], which are created via **new res** $r(e)$, with id r and a total amount of resources e available per time interval. One can understand these kind of cost centres as counting semaphores (cost statements) where the release is done implicitly when time advances, making a separation of concern between the *cost* of performing something (e.g., a computation), via **cost** statements, and the capacity of cost

```
class worker(r){
  request(c,start){
    cost(this.r,c);
    return (now−start);
  }
}
```

```
{ cpu, net, w, fut, resp;
  duration(1);
  cpu := new res r1(50);
  net := new res r2(10);
  w := new worker(cpu);
  cost(net,10);
  fut := w!request(40,now);
  await fut?;
  resp:= fut.get; }
```

Fig. 2. An example using mini-Real-Time ABS.

centres, which are renewed per time interval. This allows to capture observations with implicit time execution (no assumptions about explicit duration are given in the model), depending on the amount of resources available per time interval and the amount of competing processes accessing such cost centres. In many cases it is natural to use both explicit and implicit time in a model, so both are supported in this core active object language.

2.1 Example: Cost-Sensitive Requests

Let us consider an example where objects workers are associated to cost centres with computing capacity. Requests to such objects have a computing cost, and the execution time of the request will vary depending on how many requests are currently being sent to such objects. This is captured using the mini Real-Time ABS language in Fig. 2. For simplicity the main block captures an explicit passage of time for the creation of cost centres and objects, while a cost statement is used to capture the among of network resources needed to send a method invocation. The return value of the method call returns the amount of time that has elapsed while executing the request. We will use this example to illustrate the trace semantics used in this paper.

3 The Syntax of Abstract Traces

This section follows the syntax of conditional abstract traces as introduced in [12,13]. The syntax is given in Fig. 3. Abstract traces have a path condition pc, consisting of the conjunction of symbolic Boolean expressions sb or concrete values tt or ff, and a sequence τ of terms γ, that are either symbolic states σ or events $ev(\overline{e})$, followed by a continuation marker K which is empty, denoted by $\mathrm{K}^F(\odot)$, or contains the sequence of statements that has not yet been executed. Continuation markers K are parametric on the local future of the process, indicating where the returned value should be stored. Traces can be finite or infinite. For simplicity, let $\langle\sigma\rangle = \varepsilon \curvearrowright \sigma$ denote the *initial singleton trace*. Concatenation is denoted by $\tau_1 \cdot \tau_2$ and is only defined when τ_1 is finite. When concatenating

Syntactic categories.	*Definitions.*
sb in SymbBoolExp	$st ::= pc \triangleright \tau \cdot \mathrm{K}^F(s)$
	$s ::= \odot \mid \cdots$
	$pc ::= \mathtt{tt} \mid \mathtt{ff} \mid pc \wedge sb$
	$\tau ::= \varepsilon \mid \tau \curvearrowright \gamma$
	$\gamma ::= \sigma \mid ev(\overline{e})$
	$\sigma ::= x \mapsto se \mid \sigma \circ \sigma$
	$se ::= e \mid *$

Fig. 3. Syntax of symbolic traces

two traces τ_1 and τ_2 the first symbolic state of τ_2 should be an extension of the last symbolic state of τ_1.

A *symbolic state* σ is a partial mapping $x \mapsto se$. In a symbolic state, a *symbolic variable* is defined as a variable bound to an unknown value, represented by the starred expression $*$. Symbolic variables act as parameters, relative to which a local statement is evaluated. A symbolic state σ is well-formed if all variables are mapped to $*$, values or to expressions which only contains symbolic variables. This condition is captured as follows: $\mathrm{wf}(\sigma) = \mathtt{tt}$ if $\forall x \in (\mathrm{dom}(\sigma) \cap \mathrm{symb}(\sigma)), \mathrm{vars}(\sigma(x)) \subseteq \mathrm{symb}(\sigma)$, where $\mathrm{symb}(\sigma) = \{x \in \mathrm{dom}(\sigma) \mid \sigma(x) = *\}$.

Events record specific information in a trace and as such they do not update the values in a symbolic state, but they may extend a state with fresh symbolic variables. To achieve this, an event $ev(\overline{e})$ is inserted into a trace τ after a symbolic state σ that is augmented later with some fresh symbolic variables. To capture such actions, trace semantics introduces even traces $ev_\sigma^{\overline{x}}(\overline{e}) ::= \langle\sigma\rangle \curvearrowright ev(\overline{e}) \curvearrowright \sigma[\overline{x \mapsto *}]$ of length three. If event traces do not introduce fresh variables, the following notation is used $ev_\sigma(\overline{e})$, as a short form of $ev_\sigma^{\emptyset}(\overline{e})$.

Note that similar to well-formed states, the symbolic expressions in events and path conditions of a well-formed trace should only contain symbolic variables. This requires all states in a trace to agree upon which variables are symbolic. Further details of local abstract traces can be found in [12,13].

4 The Local Semantics of Mini Real-Time ABS

This section presents the *local* evaluation rules for expressions and for each syntax construct of mini Real-Time ABS, extending the trace semantics presented in [12,13] with time and resources. The local evaluation is parametric with respect to a symbolic state σ, and the execution context given by the future Id F, object id O and global time T.

The evaluation of expressions uses a function that reduces an expression as much as it is currently possible. In a parametric context, it reduces known variables to their values and only keeps symbolic variables inside the expressions. The evaluation function is given in Fig. 4.

The evaluation of statements uses a function that takes each syntax construct of mini Real-Time ABS in a parametric context, and returns a set of abstract

$$
\begin{aligned}
[\![v]\!]_\sigma^{O,F,T} &:= v \\
[\![x]\!]_\sigma^{O,F,T} &:= \begin{cases} x & \text{if } \sigma(x) := * \\ \sigma(x) & \text{otherwise} \end{cases} \\
[\![e_1 \; op \; e_2]\!]_\sigma^{O,F,T} &:= \begin{cases} [\![e_1]\!]_\sigma^{O,F} \; \underline{op} \; [\![e_2]\!]_\sigma^{O,F,T} & \text{if } [\![e_1]\!]_\sigma^{O,F,T}, [\![e_2]\!]_\sigma^{O,F,T} \in \mathsf{Val} \\ [\![e_1]\!]_\sigma^{O,F,T} \; op \; [\![e_2]\!]_\sigma^{O,F,T} & \text{otherwise} \end{cases} \\
[\![\textbf{this}]\!]_\sigma^{O,F,T} &:= O \\
[\![\textbf{now}]\!]_\sigma^{O,F,T} &:= T
\end{aligned}
$$

Fig. 4. Local evaluation of expressions in the language. Here, $\underline{op}$ means that expressions can be semantically reduced, while op means that expressions are syntactically distributed.

traces. The evaluation function is given in Fig. 5. The evaluation of a main block $\{\overline{x}; s\}$ generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ extends σ with the declared variables initialized with fresh symbolic variables, one per declared variable, the continuation K contains all the statements in the main block. The evaluation of **skip** generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ is a singleton trace, the continuation is empty. The evaluation of **return** e generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ is a *completion* trace event *comEv* with a future and a return value as parameters. This event denotes that value $[\![e]\!]_\sigma^{O,F,T}$ is available for retrieval in the future F. The evaluation of the conditional statement **if** e $\{ s \}$ generates two elements in the set, the first one for e evaluating to the value true and the second one when e evaluates to false. In both cases the trace τ is the singleton trace, if e is true then the continuation contains the rest of statements s, otherwise the continuation is empty. The evaluation of the sequential composition $s_1; s_2$ returns the set of elements in which s_1 is partially evaluated and therefore the continuation contains the rest of the statements in s_1, denoted as s_1' followed by s_2 and the set of elements in which s_1 evaluates completely, and therefore the continuation only contains s_2. Note that statements can pattern match more than one element and the sequential composition can be applied recursively. The evaluation of the suspension point **await** g generates one element in the set and it can only proceed if the guard g evaluates to true. There are two syntactic distinct cases. The first case is when the guard is an expression e, which is checked in the path condition. The second case is when the guard $e?$ is checking if the future $[\![e]\!]_\sigma^{O,F,T}$ has been resolved and introduces a *completion reaction* event *comREv* with a future and return value as parameters, which at the global level should match with a previous *comEv*, with the same future and value, generated by the return statement in another method, which will be ensured by well-formedness.

The evaluation of an assignment to an expression $x := e$ generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ extends the symbolic state σ with an update to the variable x. The evaluation of a blocking **get** statement $x := e.$**get** generates one element in the set and it can only proceed if the future evaluated from the expression e has been resolved. It introduces a

$$\begin{array}{rcl}
\llbracket\{\overline{x}; s\}\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \langle\sigma\rangle \curvearrowright \sigma[\overline{y \mapsto *}, \overline{x \mapsto y}] \cdot \mathrm{K}^F(s) \mid \overline{y} \notin \mathrm{dom}(\sigma)\} \\
\llbracket\textsf{skip}\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \langle\sigma\rangle \cdot \mathrm{K}^F(\odot)\} \\
\llbracket\textsf{return}\ e\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{comEv}_\sigma(F, \llbracket e\rrbracket_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\} \\
\llbracket\textsf{if}\ e\ \{\ s\ \}\rrbracket_\sigma^{O,F,T} & := & \{\llbracket e\rrbracket_\sigma^{O,F,T} = \mathrm{true} \triangleright \langle\sigma\rangle \cdot \mathrm{K}^F(s)\} \cup \\
 & & \{\llbracket e\rrbracket_\sigma^{O,F,T} = \mathrm{false} \triangleright \langle\sigma\rangle \cdot \mathrm{K}^F(\odot)\} \\
\llbracket s_1; s_2\rrbracket_\sigma^{O,F,T} & := & \{pc \triangleright \tau \cdot \mathrm{K}^F(s_1'; s_2) \mid pc \triangleright \tau \cdot \mathrm{K}^F(s_1') \in \llbracket s_1\rrbracket_\sigma^{O,F,T}\} \cup \\
 & & \{pc \triangleright \tau \cdot \mathrm{K}^F(s_2) \mid pc \triangleright \tau \cdot \mathrm{K}^F(\odot) \in \llbracket s_1\rrbracket_\sigma^{O,F,T}\} \\
\llbracket\textsf{await}\ e\rrbracket_\sigma^{O,F,T} & := & \{\llbracket e\rrbracket_\sigma^{O,F,T} = \mathrm{true} \triangleright \langle\sigma\rangle \cdot \mathrm{K}^F(\odot)\} \\
\llbracket\textsf{await}\ e?\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{comREv}_\sigma^{\{y\}}(\llbracket e\rrbracket_\sigma^{O,F,T}, y) \cdot \mathrm{K}^F(\odot) \mid y \notin \mathrm{dom}(\sigma)\} \\
\llbracket x := e\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto \llbracket e\rrbracket_\sigma^{O,F,T}] \cdot \mathrm{K}^F(\odot)\} \\
\llbracket x := e.\textsf{get}\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{comREv}_\sigma^{\{y\}}(\llbracket e\rrbracket_\sigma^{O,F,T}, y) \curvearrowright \sigma[y \mapsto *, x \mapsto y] \cdot \mathrm{K}^F(\odot) \mid \\
 & & y \notin \mathrm{dom}(\sigma)\} \\
\llbracket x := \textsf{new}\ C\,(\overline{e})\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{newObEv}_\sigma^{\{y\}}(y) \curvearrowright \sigma[y \mapsto *, x \mapsto y, \overline{y.z \mapsto v}] \cdot \mathrm{K}^F(\odot) \mid \\
 & & y \notin \mathrm{dom}(\sigma), \mathrm{class}(x) := C, \mathrm{fields}(C) := \overline{z}, \overline{v := \llbracket e\rrbracket_\sigma^{O,F,T}}\} \\
\llbracket x := \textsf{new}\ \ \textsf{res}\ r(e)\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{newResEv}_\sigma(r, \llbracket e\rrbracket_\sigma^{O,F,T}) \curvearrowright \sigma[x \mapsto r] \cdot \mathrm{K}^F(\odot)\} \\
\llbracket x := e_1!m(\overline{e_2})\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{invEv}_\sigma^{\{f\}}(\llbracket e_1\rrbracket_\sigma^{O,F,T}, m, f, \overline{\llbracket e_2\rrbracket_\sigma^{O,F,T}}) \curvearrowright \\
 & & \sigma[f \mapsto *, x \mapsto f] \cdot \mathrm{K}^F(\odot) \mid f \notin \mathrm{dom}(\sigma)\} \\
\llbracket m(\overline{x})\{s\}\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{invREv}_\sigma^{\{\overline{y}\}}(O, m, F, \overline{y}) \curvearrowright \sigma[\overline{y \mapsto *}, \overline{x_1 \mapsto y}] \cdot \\
 & & \mathrm{K}^F(\textsf{await}\ \mathrm{true}; s[\overline{x \leftarrow x_1}]) \mid \overline{y}, \overline{x_1} \notin \mathrm{dom}(\sigma)\} \\
\llbracket\textsf{duration}(e)\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{durEv}_\sigma(O, T, \llbracket e\rrbracket_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\} \\
\llbracket\textsf{rtDur}(v_1, v_2)\rrbracket_\sigma^{O,F,T} & := & \{v_2 = 0 \triangleright \mathit{durREv}_\sigma(O, v_1) \cdot \mathrm{K}^F(\odot)\} \\
\llbracket\textsf{cost}(e_1, e_2)\rrbracket_\sigma^{O,F,T} & := & \{\mathit{tt} \triangleright \mathit{costEv}_\sigma(\llbracket e_1\rrbracket_\sigma^{O,F,T}, \llbracket e_2\rrbracket_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\}
\end{array}$$

Fig. 5. Local evaluation of statements.

completion reaction event *comREv* with a future and return value as parameters, which at the global level should match with a previous *comEv*, with the same future and value, via well-formedness, it also updates the symbolic state σ with the assignment to the unknown return value, using a fresh symbolic variable. The evaluation of a new object creation $x := \textsf{new}\ C\,(\overline{e})$ generates one element in the set where the the path condition is the value tt and τ is a *new object* trace event *newObEv*, with an object id as parameter, and an extension to the trace, updating the symbolic state σ with a fresh symbolic variable, for the object id, and the fields of the new created object. Similarly, the evaluation of a new resource creation $x := \textsf{new}\ \ \textsf{res}\ r(e)$ generates one element in the set where the the path condition is the value tt and τ is a *new resource* trace event *newResEv*, with a resource id and total amount of available resources per time interval as parameters. The trace also has an update to the symbolic state with an assignment that points to the resource id. Note that for simplicity the statement includes a resource id, however the id can be created dynamically (as shown with object creation). The evaluation of an asynchronous call $x := e_1!m(\overline{e_2})$ generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ in an *invocation* trace event *invEv*, with object id, method name, future id and actual parameters to the method call. It also updates the symbolic state σ with a freshly symbolic variable for the future id f and the corresponding assignment from x to f. The evaluation of the body of a method $m(\overline{x})\{s\}$ generates one element in the set where the path condition is the value $\mathtt{tt}$ and τ starts with an *invocation reaction* trace event *invREv* with object id, method name, future id and actual parameters to the method. It also updates the symbolic state σ with a fresh symbolic variable for each parameter and the corresponding assignment from renamed parameter variables ($\overline{x \hookleftarrow x_1}$) to the symbolic variables. The continuation forces a suspension by artificially adding an **await** true; followed by the statements in the method body where parameter variables have been renamed. The reason to add this artificial suspension point will be clarified in Sect. 5.1. At the concrete global level, well-formedness conditions (over invocations and invocation reaction events) will guarantee that calls to method bodies are only instantiated if there is a previous corresponding method invocation.

The evaluation of a duration statement **duration**(e) generates one element in the set where the path condition is the value $\mathtt{tt}$ and τ is a *duration* trace event *durEv* with parameters object id, the global time and the duration value, that can be understood as a local timer that will be decreased as time advances in the global trace. At run-time there is an additional blocking statement **rtDur**(v_1, v_2) that works together with a duration statement and keeps track of the decrements in the local timer when time advances globally. Run-time duration statements can only proceed if $v_2 = 0$, which is checked in the path condition, and indicates that the local timer has reached zero. A run-time duration statement generates a *duration reaction* trace event *durREv* with an object id and a time as parameters. The evaluation of a cost statement **cost**(e_1, e_2) generates one element in the set where the the path condition is the value $\mathtt{tt}$ and τ is a *cost* trace event *costEv* with parameters resource id and the amount of resources to be consumed by the cost statement.

4.1 Example: Local Semantics of Cost Sensitive Requests in Mini-Real-Time ABS

This section shows the application of the abstract local trace semantics to the example in Fig. 2. They are summarised in Fig. 6. The figure is organised according to the code lines in Fig. 2, using the notation l_n. The statements with notation s_n abbreviates the code from Line n until the end of the method body. The figure shows first the local evaluation of the method request, starting in Line 2 and later it shows the evaluation of the main block, starting in Line 7.

The evaluation of the method request in Line 2 generates a trace with an invocation event, where the parameters cost and starting time are renamed and initialised with symbolic variables. The continuation marker contains a suspension point, followed by the statements in the method body starting in Line 3 where the local variables have been renamed. The rest of the method body is a cost statement and a return statement, which generate the corresponding

Local evaluation of statements in the method request:

l_2 : $[\![\mathsf{request(c,start)}\{s_3\}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{invREv}_\sigma^{\{c_1, st_1\}}(O, \mathit{request}, F, [c_1, st_1]) \curvearrowright \sigma[c_1 \mapsto *, st_1 \mapsto *, c' \mapsto c_1, \mathit{start'} \mapsto st_1] \cdot$
$\mathrm{K}^F(\textbf{await}\ \mathrm{true}; s_3[c \leftarrow c', \mathit{start} \leftarrow \mathit{start'}\,]) \mid c_1, st_1, c', \mathit{start'} \notin \mathrm{dom}(\sigma)\}$

l_3 : $[\![\textbf{cost}(\textbf{this}.\mathsf{r,c'})]\!]_\sigma^{O,F,T} := \{\mathtt{tt} \triangleright \mathit{costEv}_\sigma([\![\ \textbf{this}.\mathsf{r}]\!]_\sigma^{O,F,T}, [\![\mathsf{c'}]\!]_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\}$

l_4 : $[\![\textbf{return}\ (\textbf{now}-\mathsf{start'})]\!]_\sigma^{O,F,T} := \{\mathtt{tt} \triangleright \mathit{comEv}_\sigma(F, [\![\textbf{now}-\mathsf{start'}]\!]_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\}$

Local evaluation of statements in the main block:

l_7 : $[\![\{\mathsf{cpu,\ net,\ w,\ fut,\ resp};s_8\}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \langle\sigma\rangle \curvearrowright \sigma[c_0 \mapsto *, n_0 \mapsto *, w_0 \mapsto *, f_0 \mapsto *, \mathit{res}_0 \mapsto *, \mathit{cpu} \mapsto c_0, \mathit{net} \mapsto n_0,$
$w \mapsto w_0, \mathit{fut} \mapsto f_0, \mathit{resp} \mapsto \mathit{res}_0] \cdot \mathrm{K}^F(s_8) \mid c_0, n_0, w_0, f_0, \mathit{res}_0 \notin \mathrm{dom}(\sigma)\}$

l_8 : $[\![\textbf{duration}(1)]\!]_\sigma^{O,F,T} := \{\mathtt{tt} \triangleright \mathit{durEv}_\sigma(O, T, [\![1]\!]_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\}$

l_8 : $[\![\textbf{rtDur}(v_1, v_2)]\!]_\sigma^{O,F,T} := \{v_2 = 0 \triangleright \mathit{durREv}_\sigma(O, v_1) \cdot \mathrm{K}^F(\odot)\}$

l_9 : $[\![\mathsf{cpu} := \textbf{new res}\ \mathsf{r_1(50)}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{newResEv}_\sigma(r_1, [\![50]\!]_\sigma^{O,F,T}) \curvearrowright \sigma[\mathit{cpu} \mapsto r_1] \cdot \mathrm{K}^F(\odot)\}$

l_{10} : $[\![\mathsf{net} := \textbf{new res}\ \mathsf{r_2(10)}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{newResEv}_\sigma(r_2, [\![10]\!]_\sigma^{O,F,T}) \curvearrowright \sigma[\mathit{net} \mapsto r_2] \cdot \mathrm{K}^F(\odot)\}$

l_{11} : $[\![\mathsf{w} := \textbf{new}\ \mathsf{worker(cpu)}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{newObEv}_\sigma^{\{o_0\}}(o_0) \curvearrowright \sigma[o_0 \mapsto *, w \mapsto o_0, o_0.r \mapsto v] \cdot \mathrm{K}^F(\odot) \mid$
$o_0 \notin \mathrm{dom}(\sigma), \mathrm{class}(w) := \mathsf{worker}, \mathrm{fields}(\mathsf{worker}) := [r], [v := [\![\mathsf{cpu}]\!]_\sigma^{O,F,T}]\}$

l_{12} : $[\![\textbf{cost}(\mathsf{net,10})]\!]_\sigma^{O,F,T} := \{\mathtt{tt} \triangleright \mathit{costEv}_\sigma([\![\mathsf{net}]\!]_\sigma^{O,F,T}, [\![10]\!]_\sigma^{O,F,T}) \cdot \mathrm{K}^F(\odot)\}$

l_{13} : $[\![\mathsf{fut} := \mathsf{w!request(40,}\textbf{now}\mathsf{)}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{invEv}_\sigma^{\{f_1\}}([\![\mathsf{w}]\!]_\sigma^{O,F,T}, \mathsf{request}, f_1, [[\![40]\!]_\sigma^{O,F,T}, [\![\textbf{now}]\!]_\sigma^{O,F,T}]) \curvearrowright$
$\sigma[f_1 \mapsto *, \mathit{fut} \mapsto f_1] \cdot \mathrm{K}^F(\odot) \mid f_1 \notin \mathrm{dom}(\sigma)\}$

l_{14} : $[\![\textbf{await}\ \mathsf{fut?}]\!]_\sigma^{O,F,T} := \{\mathtt{tt} \triangleright \mathit{comREv}_\sigma^{\{v_0\}}([\![\mathsf{fut}]\!]_\sigma^{O,F,T}, y) \cdot \mathrm{K}^F(\odot) \mid v_0 \notin \mathrm{dom}(\sigma)\}$

l_{15} : $[\![\mathsf{resp} := \mathsf{fut}.\textbf{get}]\!]_\sigma^{O,F,T} :=$
$\{\mathtt{tt} \triangleright \mathit{comREv}_\sigma^{\{v_1\}}([\![\mathsf{fut}]\!]_\sigma^{O,F,T}, y) \curvearrowright \sigma[v_1 \mapsto *, \mathit{resp} \mapsto v_1] \cdot \mathrm{K}^F(\odot) \mid v_1 \notin \mathrm{dom}(\sigma)\}$

Fig. 6. Local evaluation of the example in Fig. 2.

abstract traces with events. The evaluation of the main block in Line 7 generates a trace where variables are updated in the symbolic state σ using symbolic variables, one per each variable. The continuation contains the rest of the body of the main block, starting in Line 8. The rest of the main block contains the abstract traces of the duration statement, the creation of the resources r_1 and r_2 for cpu and net, the creation of the object worker, the cost of calling a method, the asynchronous call to the method request, the awaiting of the method request to be completed and finally the retrieval of the returned value of the call, using a **get** statement.

$$
\begin{array}{ll}
\textit{Synt. cat.} & \textit{Definitions.} \\
o \text{ in } \mathsf{Oid} & \textit{Conf} ::= \widehat{\tau}, \Sigma \\
f \text{ in } \mathsf{Fid} & \quad \widehat{\tau} ::= \varepsilon \mid \widehat{\tau} \curvearrowright \widehat{\gamma} \\
t \text{ in } \mathsf{Time} & \quad \widehat{\gamma} ::= \widehat{\sigma} \mid \widehat{ev} \\
 & \quad \widehat{\sigma} ::= x \mapsto v \mid \widehat{\sigma} \circ \widehat{\sigma} \\
 & \quad \widehat{ev} ::= \textit{newObEv}(o) \mid \textit{invEv}(o, m, f, \overline{v}) \mid \textit{invREv}(o, m, f, \overline{v}) \mid \\
 & \qquad \textit{comEv}(f, v) \mid \textit{comREv}(f, v) \mid \textit{newResEv}(r, v) \mid \textit{avResEv}(r, v) \mid \\
 & \qquad \textit{costEv}(r, v) \mid \textit{durEv}(o, t, v) \mid \textit{durREv}(o, t) \mid \textit{timeAdvEv}(t) \\
 & \quad \Sigma ::= o \mapsto q \mid \Sigma \circ \Sigma \\
 & \quad q ::= Q \mid Q \cup \{\mathrm{K}^f(s_a)\} \\
 & \quad Q ::= \emptyset \mid Q \cup \{\mathrm{K}^f(\textsf{await } g; s)\} \\
 & \quad \widehat{\rho} ::= x \mapsto v \mid \widehat{\rho} \circ \widehat{\rho} \\
 & \quad \mathcal{G} ::= C \mapsto \langle \overline{fd}, \overline{m(\overline{x})\{s\}} \rangle \mid \mathcal{G} \circ \mathcal{G}
\end{array}
$$

Fig. 7. Runtime syntax of mini Real-Time ABS. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories. Here s_a (active statements) are statements that do not necessarily start with **await** g.

5 Global Trace Semantics of Mini Real-Time ABS

This section contains the run-time syntax of *global* configurations and the *global* composition rules for mini Real-Time ABS, extending the semantics presented in [12,13] with time advance and resource refill, following the time to completion operational semantics shown in [18]. The global *run-time syntax* is described in Fig. 7. A global configuration *Conf* consists of a concrete trace $\widehat{\tau}$ and a global map Σ. A concrete trace $\widehat{\tau}$ is sequence of concrete terms $\widehat{\gamma}$, that are either concrete states $\widehat{\sigma}$ or concrete events $\widehat{ev}$. A concrete state $\widehat{\sigma}$ is map from variables to values. Concrete events are events in which the parameters are values. A global map Σ maps objects id to a set of continuations q, representing the set of processes that are currently being executed in an active object. The pool of processes q consists of suspended processes Q, which start with an **await** statement and a process currently being executed, which do not necessarily start with **await**, e.g., if the object is idle, one of the suspended processes starting with await, where either the guard evaluates to true or the future is already resolved (checked by well-formedness), will be picked randomly. If the object is not idle, then one of the processes do not start with an **await** statement and it is currently the active process. Run-time configurations will use substitutions $\widehat{\rho}$, which are maps from variables to values, to concretise symbolic traces. Configurations also use a global table $\mathcal{G}$ mapping class names to fields and methods.

Auxiliary Functions Over Traces. Let us consider a concrete substitution $\widehat{\rho}$ such that for a symbolic $\sigma, \mathrm{symb}(\sigma) \subseteq \mathrm{dom}(\widehat{\rho})$. The concretisation of $pc \triangleright \tau$ written as $\widehat{\rho}(pc \triangleright \tau) := \widehat{\tau}$ if $[\![pc]\!]_{\widehat{\rho}} = \mathrm{true} \wedge \widehat{\tau} = \widehat{\rho}(\tau)$, where $\widehat{\rho}(\gamma_1 \cdots \gamma_n) := \widehat{\rho}(\gamma_1) \cdots \widehat{\rho}(\gamma_n)$, $\widehat{\rho}(\sigma) := \sigma \circ \widehat{\rho}$ and $\widehat{\rho}(ev(\overline{e})) := ev([\![\overline{e}]\!]_{\widehat{\rho}})$. The functions $\mathrm{last}(\widehat{\tau} \curvearrowright \widehat{\sigma}) := \widehat{\sigma}$ and $\mathrm{first}(\langle \widehat{\sigma} \rangle \cdot \widehat{\tau}) := \widehat{\sigma}$ return the last and first state of a trace, respectively (defined for both symbolic and concrete traces). The function $\mathrm{concrete}(\widehat{\rho}, \tau) :=$

$$\begin{aligned}
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{newObEv}(o)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{newObEv}(o) \not\in \widehat{\tau} \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{newResEv}(r,v)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{newResEv}(r,\_) \not\in \widehat{\tau} \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{invEv}(o,\_,f,\_)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{newObEv}(o) \in \widehat{\tau} \wedge \mathit{invEv}(\_,\_,f,\_) \not\in \widehat{\tau} \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{invREv}(o,m,f,\overline{v})) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{invEv}(o,m,f,\overline{v}) \in \widehat{\tau} \wedge \mathit{invREv}(\_,\_,f,\_) \not\in \widehat{\tau} \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{comREv}(f,v)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{comEv}(f,v) \in \widehat{\tau} \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{costEv}(r,v)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{newResEv}(r,\_) \in \widehat{\tau} \wedge \mathrm{avRes}(\widehat{\tau},r) \geq v \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{durREv}(o,t)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathit{durEv}(o,t,v) \in \widehat{\tau} \wedge \mathrm{time}(\widehat{\tau}) \geq t+v \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \mathit{durEv}(o,t,v)) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathrm{time}(\widehat{\tau}) == t \\
\mathrm{wf}(\widehat{\tau} \curvearrowright \widehat{\sigma}) &:= \mathrm{wf}(\widehat{\tau}) \wedge \mathrm{symb}(\widehat{\sigma}) = \emptyset \wedge \forall x \in \mathrm{dom}(\widehat{\sigma}), \widehat{\sigma}(x) = v
\end{aligned}$$

Fig. 8. Well-formedness of concrete traces

if symb(first(τ)) $\subseteq$ dom($\widehat{\rho}$) then tt else ff returns true if the domain of the substitution contains at least all the symbolic variables in a symbolic trace and false otherwise. The lookup function over a concrete state $\sigma \circ \widehat{\rho}$ is defined as follows:

$$\sigma \circ \widehat{\rho}(x) := \begin{cases} \widehat{\rho}(x) & \text{if } x \in \mathit{symb}(\sigma) \\ [\![\sigma(x)]\!]_{\widehat{\rho}} & \text{otherwise} \end{cases}$$

Let us also define the chop operator between two concrete traces, which is used to glue two traces: $\widehat{\tau}_1 ** \widehat{\tau}_2 := \widehat{\tau}_1 \cdot \widehat{\tau}' \wedge \mathrm{last}(\widehat{\tau}_1) = \widehat{\sigma} \wedge \widehat{\tau}_2 = \langle \widehat{\sigma}' \rangle \cdot \widehat{\tau}' \wedge \widehat{\sigma} \subseteq \widehat{\sigma}'$. The function is only defined when $\widehat{\tau}_1$ is finite.

Well-formedness over traces give certain ordering restrictions. Well-formedness is defined in Fig. 8. Events recording new object and new cost centres with resources should guarantee that the ids are fresh. Invocation events to an object should only occur after the object is created. An invocation reaction event with some parameters should match to a previous invocation event with the same parameters and the future should be fresh, indicating that one can only instantiate a method if there is an exiting call to it. A completion reaction even with some parameters should match to a previous completion events with the same parameters, indicating that one can only retrieve a return value that has been resolver with a completion event. A cost event to a cost centre with resources can only be issued if the cost centre has been previously created and should record consumptions that are less or equal to the current available resources in such cost centre. A duration reaction event should match with a previous duration event with the same object id and time stamp, and the time passed between these two events should be greater or equal to the time stamp recorded in the duration event. This will guarantee that a process is at least blocked as stated in the original duration statement. Duration events are issue according to a current global time. Finally, states in a well-formed trace are concrete.

Additional *auxiliary functions* used for time advance and resource refill are defines in Fig. 9. The pending function returns true if there exist open invocation events, which has not yet been instantiated. The function blocked is true if all the continuations in Σ cannot proceed, either because timers in duration statements are not yet 0, there is not resources to consume a cost statement, or futures are not yet resolved or conditions are not true in await statements. The function mte,

$$
\begin{aligned}
\text{pending}(\widehat{\tau}) &:= \#(\mathit{invEv}(\cdots),\widehat{\tau}) > \#(\mathit{invREv}(\cdots),\widehat{\tau}) \wedge \text{wf}(\widehat{\tau})\\[1ex]
\text{blocked}(\widehat{\tau},\Sigma_1 \circ \Sigma_2) &:= \text{blocked}(\widehat{\tau},\Sigma_1) \wedge \text{blocked}(\widehat{\tau},\Sigma_2)\\
\text{blocked}(\widehat{\tau}, o \mapsto q) &:= \text{blocked}(\widehat{\tau}, o, q)\\
\text{blocked}(\widehat{\tau}, o, Q \cup \{\mathrm{K}^f(s)\}) &:= \text{blocked}(\widehat{\tau}, o, Q) \wedge \text{blocked}(\widehat{\tau}, o, \mathrm{K}^f(s))\\
\text{blocked}(\widehat{\tau}, o, \emptyset) &:= \mathtt{tt}\\
\text{blocked}(\widehat{\tau}, o, \mathrm{K}^f(\textsf{\textbf{rtDur}}(t,v);s)) &:= v > 0\\
\text{blocked}(\widehat{\tau}, o, \mathrm{K}^f(\textsf{\textbf{cost}}(r,v);s)) &:= v > 0 \wedge \text{avRes}(\widehat{\tau}) == 0\\
\text{blocked}(\widehat{\tau}, o, \mathrm{K}^f(s)) &:= \mathtt{tt} \text{ if } \forall pc, \tau \text{ st. } (pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]^{o,f,\text{time}(\widehat{\tau})}_{\text{last}(\widehat{\tau})} \wedge\\
&\qquad \text{concrete}(\widehat{\rho},\tau) \wedge (\neg[\![pc]\!]_{\widehat{\rho}} \vee \neg\text{wf}(\widehat{\tau} ** \widehat{\rho}(\tau))))\\
\text{blocked}(\widehat{\tau}, o, \mathrm{K}^f(s)) &:= \mathtt{ff} \text{ otherwise}\\[1ex]
\text{mte}(\Sigma_1 \circ \Sigma_2) &:= \min(\text{mte}(\Sigma_1), \text{mte}(\Sigma_2))\\
\text{mte}(o \mapsto q) &:= \text{mte}(q)\\
\text{mte}(Q \cup \{\mathrm{K}^f(\textsf{\textbf{rtDur}}(t,v);s)\}) &:= \min(\text{mte}(Q), v)\\
\text{mte}(Q \cup \{\mathrm{K}^f(\textsf{\textbf{cost}}(r,v);s)\}) &:= \min(\text{mte}(Q), 1)\\
\text{mte}(Q \cup \{\mathrm{K}^f(s)\}) &:= \min(\text{mte}(Q), \infty)\\
\text{mte}(\emptyset) &:= \infty\\[1ex]
\text{timeAdv}(\Sigma_1 \circ \Sigma_2, v) &:= \text{timeAdv}(\Sigma_1, v) \circ \text{timeAdv}(\Sigma_2, v)\\
\text{timeAdv}(o \mapsto q, v) &:= o \mapsto \text{timeAdv}(q, v)\\
\text{timeAdv}(Q \cup \{\mathrm{K}^f(\textsf{\textbf{rtDur}}(t,v');s)\}, v) &:= \text{timeAdv}(Q, v) \cup \{\mathrm{K}^f(\textsf{\textbf{rtDur}}(t, v'-v);s)\} \text{ if } (v'-v) \geq 0\\
\text{timeAdv}(Q \cup \{\mathrm{K}^f(\textsf{\textbf{rtDur}}(t,v');s)\}, v) &:= \text{timeAdv}(Q, v) \cup \{\mathrm{K}^f(\textsf{\textbf{rtDur}}(t, 0);s)\} \text{ if } (v'-v) < 0\\
\text{timeAdv}(Q \cup \{\mathrm{K}^f(s)\}) &:= \text{timeAdv}(Q, v) \cup \{\mathrm{K}^f(s)\}\\
\text{timeAdv}(\emptyset) &:= \emptyset\\[1ex]
\text{resRefill}(\widehat{\tau} \curvearrowright \mathit{newResEv}(r,v), \widehat{\sigma}) &:= \text{resRefill}(\widehat{\tau}, \widehat{\sigma}) \curvearrowright \mathit{avResEv}(r,v) \curvearrowright \widehat{\sigma}\\
\text{resRefill}(\widehat{\tau} \curvearrowright \widehat{\sigma}', \widehat{\sigma}) &:= \text{resRefill}(\widehat{\tau}, \widehat{\sigma})\\
\text{resRefill}(\widehat{\tau} \curvearrowright \widehat{ev}, \widehat{\sigma}) &:= \text{resRefill}(\widehat{\tau}, \widehat{\sigma}) \text{ if } \widehat{ev} \neq \mathit{newResEv}(\cdots)\\
\text{resRefill}(\varepsilon, \widehat{\sigma}) &= \widehat{\sigma}\\[1ex]
\text{time}(\widehat{\tau}) &:= t \text{ if lastTimeAdv}(\widehat{\tau}) == \mathit{timeAdvEv}(t)\\
\text{lastTimeAdv}(\widehat{\tau} \curvearrowright \widehat{\sigma}) &:= \text{lastTimeAdv}(\widehat{\tau})\\
\text{lastTimeAdv}(\widehat{\tau} \curvearrowright \widehat{ev}) &:= \text{lastTimeAdv}(\widehat{\tau}) \text{ if } \widehat{ev} \neq \mathit{timeAdvEv}(\cdots)\\
\text{lastTimeAdv}(\widehat{\tau} \curvearrowright \mathit{timeAdvEv}(t)) &:= \mathit{timeAdvEv}(t)\\[1ex]
\text{avRes}(\widehat{\tau}, r) &:= v \text{ if lastAvRes}(r, \widehat{\tau}) == \mathit{avResEv}(r,v)\\
\text{lastAvRes}(r, \widehat{\tau} \curvearrowright \widehat{\sigma}) &:= \text{lastAvRes}(r, \widehat{\tau})\\
\text{lastAvRes}(r, \widehat{\tau} \curvearrowright \widehat{ev}) &:= \text{lastAvRes}(r, \widehat{\tau}) \text{ if } \widehat{ev} \neq \mathit{timeAdvEv}(r, _)\\
\text{lastAvRes}(r, \widehat{\tau} \curvearrowright \mathit{avResEv}(r,v)) &:= \mathit{avResEv}(r,v)
\end{aligned}
$$

Fig. 9. Auxiliary functions used for time advance and resource refill.

stands for maximum time elapse and calculates the time to the next observable point, e.g., next time interval where the smallest timer in a duration statement is 0 or next time interval to refill resource to proceed with a cost statement. The function for time advance modifies the continuations in Σ by decreasing timers in run-time duration statements according to a maximum time elapse, passed as a parameter. The function resource refill returns a well-formed concrete trace where all the cost centres in a configuration have been refilled with their total amounts per time interval, as recorded when the cost centre was created, by appending an available resource event per cost centre. The functions time and available resources calculates the current global time and available resources of a cost centre in a given well-formed and concrete trace, by finding the last event with time advance or the last event with available resources for that cost centre, respectively. All these auxiliary functions will be used in the composition rules to generate concrete traces.

5.1 Global Trace Composition of Mini Real-Time ABS

This section explains the *composition* rules that combine local evaluation and process creation into global traces. Random scheduling is expressed declaratively using well-formedness constraints over traces. Given the non-deterministic nature of the semantics, many different traces can be obtained, depending on the different scheduling of processes. Starting in a concrete initial configuration $Conf_0 := \langle[\,]\rangle \curvearrowright timeAdvEv(0) \curvearrowright [\,], [main \mapsto \mathrm{K}^{f_0}(sc)]$, the global composition rules are applied recursively. If the current execution of the program produces a final trace, the execution will continue until all the objects maps to empty sets of continuations, generating the finite concrete trace $\widehat{\tau}$.

The rest of this section, explains the composition rules summarised in Fig. 10. Rule Non-empty-continuation repeatedly evaluates one continuation at a time and generates a new continuation with the remaining statements to be executed, then stitch the resulting concrete trace together with a given a concrete trace $\widehat{\tau}$, and update Σ with the new generated continuation. This rule can only be applied if the evaluated symbolic trace can be concretised and path condition pc evaluates to true according to a guessed substitution $\widehat{\rho}$, well-formedness conditions are satisfied and the evaluated trace does no contain a new resource event, a cost event or a duration event. Rule Mtd-binding generates a new process with an exiting future id from $\widehat{\sigma}$. Note that this new process will start with an invocation reaction event and will be artificially suspended (see Fig. 5). This rule can only be applied if there exist a previews invocation event which has introduced the future id and well-formedness conditions are satisfied, meaning that no other process has been created with the same future id. Since the newly created process is artificially suspended, method binding can happen at any moment, provided that the method call has been already issued. Rule Empty-continuation removes empty continuations in Σ. Rule New-Resource matches exactly the local evaluation of a new cost centre with a total amount of resources (capacity) per time interval. The rule refills the cost centre by adding an available resource trace event. This rule can only be applied if the evaluated symbolic trace can be concretised and path condition pc evaluates to true according to a guessed substitution $\widehat{\rho}$ and well-formedness conditions are satisfied, meaning that no other cost centre with the same id has been previously created. Rules Cost1 and Cost2 match exactly the local evaluation of a cost statement. In the case of Cost1, there is enough available resources, calculated by the function avRes in Fig. 9, therefore the cost statement is completely consumed and the available resources in that cost centre is updated accordingly by adding a additional available resources event. In the case of Cost2 there is not enough

$$\textsc{Non-empty-continuation}\quad \frac{\begin{array}{c}\Sigma(o) = Q \cup \{\mathrm{K}^f(s)\} \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \quad \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \widehat{\rho}(\tau)) \\ \mathit{newResEv}(\cdots) \notin \widehat{\rho}(\tau) \quad \mathit{costEv}(\cdots) \notin \widehat{\rho}(\tau) \quad \mathit{durEv}(\cdots) \notin \widehat{\rho}(\tau)\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \widehat{\rho}(\tau), \Sigma[o \mapsto Q \cup \{\mathrm{K}^f(s')\}]}$$

$$\textsc{Mtd-binding}\quad \frac{\begin{array}{c}o \in \mathrm{dom}(\Sigma) \quad m(\overline{x})\{s\} = \mathrm{lookup}(\mathcal{G}(o), m) \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \quad f \in \mathrm{dom}(\widehat{\sigma}) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s) \in [\![m(\overline{x})\{s\}]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \\ \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \widehat{\rho}(\tau))\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \widehat{\rho}(\tau), \Sigma[o \mapsto q \cup \{\mathrm{K}^f(s)\}]}$$

$$\textsc{Empty-continuation}\quad \frac{\Sigma(o) = Q \cup \{\mathrm{K}^f(\odot)\}}{\widehat{\tau}, \Sigma \to \widehat{\tau}\ \Sigma[o \mapsto Q]}$$

$$\textsc{New-Resource}\quad \frac{\begin{array}{c}\Sigma(o) = Q \cup \{\mathrm{K}^f(s)\} \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \quad \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \widehat{\rho}(\tau)) \\ \widehat{\rho}(\tau) == \mathit{newResEv}_{\widehat{\sigma}}(r, v) \curvearrowright \widehat{\sigma}[x \mapsto r]\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \widehat{\rho}(\tau ** \mathit{avResEv}_{\mathrm{last}(\tau)}(r, v)), \Sigma[o \mapsto Q \cup \{\mathrm{K}^f(s')\}]}$$

$$\textsc{Cost1}\quad \frac{\begin{array}{c}\Sigma(o) = Q \cup \{\mathrm{K}^f(s)\} \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \quad a = \mathrm{avRes}(\widehat{\tau}, r) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \\ \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \widehat{\rho}(\tau)) \\ a > 0 \quad 0 < v \leq a \quad a' = a - v \quad \widehat{\rho}(\tau) == \mathit{costEv}_{\widehat{\sigma}}(r, v)\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \mathit{costEv}_{\widehat{\sigma}}(r, v) ** \mathit{avResEv}_{\widehat{\sigma}}(r, a'), \Sigma[o \mapsto Q \cup \{\mathrm{K}^f(s')\}]}$$

$$\textsc{Cost2}\quad \frac{\begin{array}{c}\Sigma(o) = Q \cup \{\mathrm{K}^f(s)\} \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \quad a = \mathrm{avRes}(\widehat{\tau}, r) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \\ \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \mathit{costEv}_{\widehat{\sigma}}(r, a)) \\ a > 0 \quad v > a \quad v' = v - a \quad \widehat{\rho}(\tau) == \mathit{costEv}_{\widehat{\sigma}}(r, v)\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \mathit{costEv}_{\widehat{\sigma}}(r, a) ** \mathit{avResEv}_{\widehat{\sigma}}(r, 0), \Sigma[o \mapsto Q \cup \{\mathrm{K}^f(\textbf{cost}(r, v'); s')\}]}$$

$$\textsc{Duration}\quad \frac{\begin{array}{c}\Sigma(o) = Q \cup \{\mathrm{K}^f(s)\} \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad pc \triangleright \tau \cdot \mathrm{K}^f(s') \in [\![s]\!]_{\widehat{\sigma}}^{o,f,\mathrm{now}} \quad \mathrm{concrete}(\widehat{\rho}, \tau) \quad [\![pc]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau} ** \widehat{\rho}(\tau)) \\ \widehat{\rho}(\tau) == \mathit{durEv}_{\widehat{\sigma}}(o, t, v)\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau} ** \widehat{\rho}(\tau), \Sigma[o \mapsto Q \cup \{\mathrm{K}^f(\textbf{rtDur}(t, v); s')\}]}$$

$$\textsc{Time-Advance}\quad \frac{\begin{array}{c}\mathrm{blocked}(\widehat{\tau}, \Sigma) \quad \neg\mathrm{pending}(\widehat{\tau}) \quad \mathrm{now} = \mathrm{time}(\widehat{\tau}) \\ \widehat{\sigma} = \mathrm{last}(\widehat{\tau}) \quad \widehat{\tau}' = \widehat{\tau} ** \mathit{timeAdvEv}_{\widehat{\sigma}}(\mathrm{now} + v) ** \mathrm{resRefill}(\widehat{\tau}, \widehat{\sigma}) \\ v = \mathrm{mte}(\Sigma) \quad \Sigma' = \mathrm{timeAdv}(\Sigma, v)\end{array}}{\widehat{\tau}, \Sigma \to \widehat{\tau}', \Sigma'}$$

Fig. 10. Trace composition rules for mini Real-Time ABS.

$$\textsc{Example-Duration}\quad \frac{\begin{array}{c}\Sigma_1(\mathit{main}) = \emptyset \cup \{\mathrm{K}^{f_{main}}(s_8)\} \quad \mathrm{now} = 0 \\ \widehat{\sigma}_1 = \mathrm{last}(\widehat{\tau}_1) \quad \mathtt{tt} \triangleright \tau \cdot \mathrm{K}^{main}(s_9) \in [\![s_8]\!]_{\widehat{\sigma}_1}^{main, f_{main}, \mathrm{now}} \\ \mathrm{concrete}(\widehat{\rho}_1, \tau) \quad [\![\mathtt{tt}]\!]_{\widehat{\rho}} \quad \mathrm{wf}(\widehat{\tau}_1 ** \widehat{\rho}_1(\tau)) \\ \widehat{\rho}(\tau) == \mathit{durEv}_{\widehat{\sigma}_1}(\mathit{main}, 0, 1)\end{array}}{\widehat{\tau}_1, \Sigma_1 \to \widehat{\tau}_2, \Sigma_2}$$

$$\textsc{Example-Time-Advance}\quad \frac{\begin{array}{c}\mathrm{blocked}(\widehat{\tau}_2, \Sigma_2) \quad \neg\mathrm{pending}(\widehat{\tau}_2) \quad \mathrm{now} = 0 \\ \widehat{\sigma}_1 = \mathrm{last}(\widehat{\tau}_2) \quad \widehat{\tau}_3 = \widehat{\tau}_2 ** \mathit{timeAdvEv}_{\widehat{\sigma}_1}(0+1) ** \mathrm{resRefill}(\widehat{\tau}_2, \widehat{\sigma}_1) \\ 1 = \mathrm{mte}(\Sigma_2) \quad \Sigma_3 = \mathrm{timeAdv}(\Sigma_2, 1)\end{array}}{\widehat{\tau}_2, \Sigma_2 \to \widehat{\tau}_3, \Sigma_3}$$

$$\textsc{Example-Non-Empty-Continuation}\quad \frac{\begin{array}{c}\Sigma_3(\mathit{main}) = \emptyset \cup \{\mathrm{K}^{f_{main}}(\mathbf{rtDur}(0,0); s_9)\} \quad \mathrm{now} = 1 \\ \widehat{\sigma}_1 = \mathrm{last}(\widehat{\tau}_3) \quad pc_1 \triangleright \tau_1 \cdot \mathrm{K}^{f_{main}}(s_{12}) \in [\![\mathbf{rtDur}(0,0); s_9]\!]_{\widehat{\sigma}_1}^{main, f_{main}, \mathrm{now}} \\ \mathrm{concrete}(\widehat{\rho}_2, \tau_1) \quad [\![pc_1]\!]_{\widehat{\rho}_2} \quad \mathrm{wf}(\widehat{\tau}_3 ** \widehat{\rho}_2(\tau_1)) \\ \mathit{newResEv}(\cdots) \notin \widehat{\rho}_2(\tau_1) \quad \mathit{costEv}(\cdots) \notin \widehat{\rho}_2(\tau_1) \quad \mathit{durEv}(\cdots) \notin \widehat{\rho}_2(\tau_1)\end{array}}{\widehat{\tau}_3, \Sigma_3 \to \widehat{\tau}_3 ** \widehat{\rho}_2(\tau_1), \Sigma_3[\mathit{main} \mapsto \emptyset \cup \{\mathrm{K}^{f_{main}}(s_{12})\}]}$$

Fig. 11. Sample of the application of the trace composition rules in the example.

available resources to consume the cost statement, therefore a new cost statement is added to the continuation with the resources that were not consumed and an available resource event with zero resources is added, which will block the object until new resources are refilled when time advances. Rule Duration matches exactly the local evaluation of a duration statement and replaces it with a run-time duration statement which will block execution in that process until time advances and reduces the local timer in the run-time duration statement to zero, at this point the Rule Non-empty-continuation can be applied. Duration rule can only be applied if the evaluated symbolic trace can be concretised and path condition *pc* evaluates to true according to a guessed substitution $\widehat{\rho}$ and well-formedness conditions are satisfied,meaning that the added duration event should match the current time of $\widehat{\tau}$. Rule Time-Advance can only be applied if all the continuations in Σ are blocked and there is not pending method calls. This rule will advance the time by adding a time advance trace event, it will also advance the time in all continuations in Σ by decreasing the local timers in run-time duration statements, according to a calculated maximum time elapse function mte, and it will refill all the cost centres that have been created.

5.2 Global Semantics of the Running Example

This section shows some of the applications of the composition rules to the example in Fig. 2, using the local evaluation rules of the example in Fig. 6. Samples of such applications are shown in Fig. 11. Let us consider a global table of classes $\mathcal{G} ::= [c \mapsto \langle [\mathsf{r}], \{\mathsf{request(c,start)}\{\mathbf{cost(this}.\mathsf{r,c});\ \mathbf{return}\ (\mathbf{now}-\mathsf{start});\}\}\rangle]$, a concrete trace $\widehat{\tau}_1 := \langle [\ ] \rangle \curvearrowright \mathit{timeAdvEv}(0) \curvearrowright [\ ] \curvearrowright \widehat{\sigma}_1$, where $\widehat{\sigma}_1 :=$

$[cpu \mapsto nil, net \mapsto nil, w \mapsto nil, fut \mapsto f_{main}, resp \mapsto 0]$, recording execution of the example until variable declaration in Line 7. Let us assume a $\widehat{\rho}_1 := [\ ]$. The application of the Rule Duration, instantiated as Example-Duration in Fig. 11, will generate $\widehat{\tau}_2 := \langle[\ ]\rangle \curvearrowright timeAdvEv(0) \curvearrowright [\ ] \curvearrowright \widehat{\sigma}_1 \curvearrowright durEv(main, 0, 1) \curvearrowright \widehat{\sigma}_1$ and $\Sigma_2 := [main \mapsto \{\mathrm{K}^{f_{main}}(\mathbf{rtDur}(0, 1); s_9)\}]$ by replacing the duration statement of the example in Line 8 with a run-time duration statement, and by adding a duration event to the concrete trace. Since the only continuation in this configuration is blocked, then the only rule that can be applied is Time-Advance, which is instantiated as Example-Time-Advance in Fig. 11. It will generate $\widehat{\tau}_3 := \widehat{\tau}_2 \curvearrowright timeAdvEv(1) \curvearrowright \widehat{\sigma}_1$ and $\Sigma_3 := [main \mapsto \{\mathrm{K}^{f_{main}}(\mathbf{rtDur}(0, 0); s_9)\}]$. The next rule that can be applied after time advance is Rule Non-empty-continuation, instantiated as Example-Non-empty-continuation in Fig. 11. This rule matches the execution of Line 8 to Line 11 in the example. Let is assume $\widehat{\rho}_2 := [o_0 \mapsto o_w]$, the generated symbolic trace will have $pc_1 := 0 = 0$, $\widehat{\rho}_2(\tau_1) := \widehat{\sigma}_1 \curvearrowright durREv(main, 0) \curvearrowright \widehat{\sigma}_1 \curvearrowright newResEv(r_1, 50) \curvearrowright \widehat{\sigma}_1 \curvearrowright \widehat{\sigma}_2 \curvearrowright newResEv(r_2, 10) \curvearrowright \widehat{\sigma}_2 \curvearrowright \widehat{\sigma}_3 \curvearrowright newObEv(o_w) \curvearrowright \widehat{\sigma}_4 \curvearrowright \widehat{\sigma}_5$, where $\widehat{\sigma}_2 := \widehat{\sigma}_1[cpu \mapsto r_1]$, $\widehat{\sigma}_3 := \widehat{\sigma}_2[net \mapsto r_2]$, $\widehat{\sigma}_4 := \widehat{\sigma}_3[o_0 \mapsto o_w]$ and $\widehat{\sigma}_5 := \widehat{\sigma}_4[w \mapsto o_w, o_w.r \mapsto r_1]$. Since the example produces final traces, the rest of the example will recursively apply the composition rules until all the continuations in Σ_n are empty.

6 Related Work

Active objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time, with cooperative scheduling of method activation inside each object. This concurrency model is an alternative to multi-thread concurrency in object-orientation (e.g., [8]). Concurrent objects support compositional verification of concurrent software [2,9]. Din *et al.* introduced a proof system based on four communication events [11,14,15]. This four-event proof system, are the starting point for the well-formedness conditions of events used in this paper. In contrast, the work presented here uses events which are not only restricted to communication, but also to object and cost centre creation and to duration statements, cost statements and time advance.

The time and resource model presented in this paper is based on a *maximal progress* semantics and follows the semantics ideas introduced in [6,18]. However, this paper captures the interleaved execution by means of local continuations, such that the global trace is obtained by gradually unfolding traces that correspond to local symbolic executions with continuations. Early work of the author modelled tightly coupled cost centres to objects [18]. The resource model presented in this paper is loosely coupled, which gives the freedom to the modeller in expressing and combining different consumptions of various cost centres.

Cost centres with resources that are orthogonal to time and use take and release [4,5], (similar to semaphores) can be easily integrated in this semantics, e.g., memory resources, locations, etc. In this case, the local rule for take will only proceed if the cost center has enough resources, which can be checked in the path condition, while the release rule will always proceed and add the released resources to the cost center. In this context, events with well-formlessness conditions at the global level can guarantee that the total amount of resources in a cost center is consistent.

Cost statements in this paper captures how the modeller specify dynamic resource consumption that can be recharged, e.g., computation, energy, etc. Note that the expressions in cost statements could be automatically derived using static analysis techniques, if the code in the model is fully implemented. One possibility will be to use COSTABS [3], a cost analysis tool for ABS which supports *concurrent* object-oriented programs, based on the notion of cost centres. Previous work with Albert *et al.*, has explored the use of cost analysis tools for memory analysis of ABS models [4].

7 Conclusion and Future Work

This paper presents a compositional trace semantics for a time and resource sensitive active object language, inspired by Real-Time ABS. This semantics uses the trace semantics framework of [12,13] and contributes with a modular extension of time and resources, following the ideas of the operational semantics presented in [18]. As future work, it would be interesting to develop the full LAGC semantics of Real-Time ABS and formally compare it with the existing semantics presented in [18], which is the basis for the simulation tool of Real-Time ABS [22]. It would also be interesting to develop a program logic and a calculus for Real-Time ABS to be able to express time and resource related properties, e.g., the response of a request with input e will be available in less than $3 * e$ time units. We can take as starting point results from e.g., [21]. Other interesting extension would be to develop a loosely coupled resource model semantics for Real-Time ABS with independent cost centres, as the one presented in this paper.

Acknowledgement. I would like to thank Einar Broch Johnsen and Rudolf Schlatte for the interesting discussions while writing this paper.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming (2012). 2010 (in press)
3. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: a cost and termination analyzer for ABS. In: Kiselyov, O., Thompson, S. (eds.) Proceeding Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 151–154. ACM (2012)

4. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating concurrent behaviors with worst-case cost bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_27
5. Rizwan Ali, M., Ka I Pun, V.: Towards a resource-aware formal modelling language for workflow planning. In: Bellatreche, L., Chernishev, G., Corral, A., Ouchani, S., Vain, J. (eds.) MEDI 2021. CCIS, vol. 1481, pp. 251–258. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-87657-9_19
6. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations Syst. Softw. Eng. **9**(1), 29–43 (2013). https://doi.org/10.1007/s11334-012-0184-5
7. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing 1974. Elsevier, North-Holland, Amsterdam (1974)
8. Caromel, D., Henrio., L.: A Theory of Distributed Object. Springer (2005). https://doi.org/10.1007/3-540-27245-3_9
9. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
10. de Boer, F.S.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1-76:39 (2017)
11. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: component reasoning for concurrent objects. J. Logic Algebraic Program. **81**(3), 227–256 (2012)
12. Din, C.C., Hähnle, R., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. In: Schmidt, R.A., Nalon, C. (eds.) TABLEAUX 2017. LNCS (LNAI), vol. 10501, pp. 22–43. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66902-1_2
13. Din, C.C., Hähnle, R., Henrio, L., Johnsen, E.B., Pun, V.K.I., Tapia Tarifa, S.L.: LAGC semantics of concurrent programming languages, February 2022. https://arxiv.org/abs/2202.12195
14. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Log. Algebraic Meth. Program. **83**(5–6), 360–383 (2014)
15. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2014). https://doi.org/10.1007/s00165-014-0322-y
16. Hähnle, R., Huisman, M.: Deductive verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives. LNCS, vol. 10000, pp. 345–373. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-319-91908-9_18
17. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University Press, Cambridge (2004)
18. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebraic Meth. Program. **84**(1), 67–91 (2015)
19. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
20. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)

21. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed kripke structures and real-time rewrite theories. Sci. Comput. Programm. **99**, 128–192 (2015)
22. Schlatte, R., Johnsen, E.B., Kamburjan, E., Tapia Tarifa, S.L.: Modeling and analyzing resource-sensitive actors: a tutorial introduction. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 3–19. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_1

Transparent Treatment of for-Loops in Proofs

Nathan Wasser(✉)

SharpMind, 60313 Frankfurt, Germany
nate@sharpmind.de
https://www.sharpmind.de/

Abstract. Indexed loop scopes have been shown to be a helpful tool in creating sound loop invariant rules in dynamic logic for programming languages with abrupt completion, such as Java. These rules do not require program transformation of the loop body, as other approaches to dealing with abrupt completion do. However, indexed loop scopes were designed specifically to provide a loop invariant rule for `while` loops and work rather opaquely. Here we propose replacing indexed loop scopes with a more transparent solution, which also lets us extend this idea from `while` loops to `for` loops. We further present sound loop unrolling rules for `while`, `do` and `for` loops, which require neither program transformation of the loop body, nor the use of nested modalities. This approach allows `for` loops to be treated as first-class citizens in proofs – rather than the usual approach of transforming `for` loops into `while` loops – which makes semi-automated proofs more transparent and easier to follow for the user, whose interactions may be required in order to close the proofs.

Keywords: Theorem proving · Dynamic logic · Loop invariants

1 Introduction

Sound program transformation in real world programming languages such as Java [8] is not easy, with potential pitfalls hiding in constructs such as Java's `try-finally` statement. Thus, when reasoning about programs it is useful to avoid complex program transformations whenever possible.

Indexed loop scopes were introduced in [20] to allow a sound loop invariant rule (which does not require program transformation of the loop body) in dynamic logic [10] for `while` loops containing statements which complete abruptly [8, Chapter 14.1]. In [18] it was shown that an implementation of this new loop invariant rule in KeY[1] [1] also decreases proof size when compared to the existing rule.

However, indexed loop scopes were tailored specifically to treat the case of applying a loop invariant to a `while` loop. While we made attempts to re-use

[1] https://www.key-project.org/.

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

indexed loop scopes for loop unrolling [20] and application to `for` loops [21], these were suboptimal.

In this paper we refine the concept of the loop scope, splitting it into two distinct parts: (1.) the `attempt- continuation` statement providing a non-active prefix [1] for loop bodies; and (2.) the logic to determine whether the loop invariant or the original formula should be proven, which was rather opaquely contained in symbolic execution rules for loop scopes. Splitting these orthogonal concerns allows using an `attempt-continuation` statement in simple loop unrolling rules for `while`, `do` and `for` loops, which avoid program transformation of the loop body and do not require the use of nested modalities, as the approach in [20] did for `while` loop unrolling. It also allows for a more transparent loop invariant rule for `while` loops and we can introduce a transparent loop invariant rule for `for` loops, which also both avoid program transformation of the loop body.

With this, we can treat `for` loops fully as first-class citizens in proofs, without the need to transform them into `while` loops, which involves non-trivial program transformation.

Section 2 provides background on dynamic logic and JavaDL in particular, as well as on indexed loop scopes and the loop invariant rule using them. In Sect. 3 we introduce the `attempt-continuation` statement and new specialized loop unrolling rules for each loop type. We propose new specialized loop invariant rules for `while` and `for` loops in Sect. 4, while Sect. 5 contains an evaluation of previous work and the changes proposed in this paper. In Sect. 6 we compare this approach with related work. Finally, we conclude and offer ideas for future work in Sect. 7.

2 Background

One approach to *deductive software verification* [6] which has been quite useful is *dynamic logic* [10]. The idea behind dynamic logic is to contain the program under test within the logic itself by use of dynamic logic *modalities*. Classically, for all formulae ϕ and all programs $\mathtt{p}$ the formula $[\mathtt{p}]\phi$ holds iff ϕ holds in all terminating states reachable by executing $\mathtt{p}$. The dual is defined as: $\langle \mathtt{p} \rangle \phi \equiv \neg([\mathtt{p}](\neg\phi))$. One advantage of dynamic logic is that many proofs can be abstracted away from the *semantics* of the underlying programming language and instead rely only on axioms expressed using the programming language's *syntax* within a dynamic logic modality. Initially proposed using *Kleene's regular expression operators* [14] as programming language, it has been extended to various other programming languages, in particular to Java [8] in *Java dynamic logic* (JavaDL) [2]. While Kleene's regular expression operators contain complexities such as non-determinism, which makes reasoning about them far from simple, there is no concept of *abrupt completion*[2]: either an operation completes normally or blocks. Additionally, program elements in Java can "catch" these

[2] In Java, statements can complete abruptly due to `breaks`, `continues` and `returns`, while both statements and expressions can complete abruptly due to thrown exceptions [8, Chapter 14.1].

abrupt completions and execute different code due to them, then either complete normally or complete abruptly for the same or a different reason. Thus, it is not as simple a matter to give meaning to $[\texttt{while (e) st}]\phi$ for the `while` loop of a Java program, while the axiom for `while` in a simple WHILE language can be expressed through *loop unrolling*:

$$[\texttt{WHILE (e) st}]\phi \equiv [\texttt{IF e \{ st WHILE (e) st \}}]\phi$$

One solution, proposed for example in [16], would be to introduce new modalities for each type of completion.

Definition 1 (Set of all labels, sets of completion types). *$\mathcal{L}$ is an infinite set of labels. The set of completion types $\mathcal{T}$ and its subsets $\mathcal{N}$ (*normal*), $\mathcal{A}$ (*abrupt*), $\mathcal{B}_l$ (*breaking*) and $\mathcal{C}_l$ (*continuing completion types*) are given as:*

$$\mathcal{N} = \{normal\},\ \mathcal{A} = \{break, continue\} \cup \bigcup_{l \in \mathcal{L}} \{break_l, continue_l\},\ \mathcal{T} = \mathcal{N} \cup \mathcal{A},$$
$$\forall l \in \mathcal{L}.\ \mathcal{B}_l = \{break, break_l\}, \qquad \forall l \in \mathcal{L}.\ \mathcal{C}_l = \{normal, continue, continue_l\}$$

We write $[\texttt{p}]_S\ \phi$ as short form for $\bigwedge_{t \in S}([\texttt{p}]_t\ \phi)$. The axioms given in this paper hold for all $l \in \mathcal{L}$. Figure 1 contains relatively straightforward axioms for some simple Java statements, as well as the `try-finally` statement.

$$[\texttt{;}]_{\mathcal{N}}\ \phi \equiv \phi \quad (1) \qquad\qquad [\texttt{;}]_{\mathcal{A}}\ \phi \quad (6)$$
$$[\texttt{break;}]_{break}\ \phi \equiv \phi \quad (2) \qquad\qquad [\texttt{break;}]_{\mathcal{T}\setminus\{break\}}\ \phi \quad (7)$$
$$[\texttt{continue;}]_{continue}\ \phi \equiv \phi \quad (3) \qquad\qquad [\texttt{continue;}]_{\mathcal{T}\setminus\{continue\}}\ \phi \quad (8)$$
$$[\texttt{break}\ l\texttt{;}]_{break_l}\ \phi \equiv \phi \quad (4) \qquad\qquad [\texttt{break}\ l\texttt{;}]_{\mathcal{T}\setminus\{break_l\}}\ \phi \quad (9)$$
$$[\texttt{continue}\ l\texttt{;}]_{continue_l}\ \phi \equiv \phi \quad (5) \qquad\qquad [\texttt{continue}\ l\texttt{;}]_{\mathcal{T}\setminus\{continue_l\}}\ \phi \quad (10)$$

$$[\texttt{st1 st2}]_{\mathcal{N}}\ \phi \equiv [\texttt{st1}]_{\mathcal{N}}[\texttt{st2}]_{\mathcal{N}}\ \phi \quad (11)$$
$$\forall a \in \mathcal{A}.\ [\texttt{st1 st2}]_a\ \phi \equiv [\texttt{st1}]_a\ \phi \wedge [\texttt{st1}]_{\mathcal{N}}[\texttt{st2}]_a\ \phi \quad (12)$$
$$\forall t \in \mathcal{T}.\ [\texttt{if (e) st1 else st2}]_t\ \phi \equiv [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow [\texttt{st1}]_t\ \phi) \wedge (\neg\texttt{b} \rightarrow [\texttt{st2}]_t\ \phi)) \quad (13)$$

$$[\texttt{try \{ p \} finally \{ q \}}]_{\mathcal{N}}\ \phi \equiv [\texttt{p}]_{\mathcal{N}}[\texttt{q}]_{\mathcal{N}}\ \phi \quad (14)$$
$$\forall a \in \mathcal{A}.\ [\texttt{try \{ p \} finally \{ q \}}]_a\ \phi \equiv [\texttt{p}]_a[\texttt{q}]_{\mathcal{N}}\ \phi \wedge [\texttt{p}]_{\mathcal{T}}[\texttt{q}]_a\ \phi \quad (15)$$

Fig. 1. Axioms for *skip*, `breaks`, `continues`, *sequence*, `if` and `try-finally`

We write "`if (e) st`" as short form for "`if (e) st else ;`".

Figure 2 contains axioms for the `while` statement. The axiom (16) expresses that the loop can: (1) continue normally, or by a matching `continue` statement; and (2) be exited normally or by a matching `break` statement. Axiom (17) expresses that the loop can complete abruptly by a labeled `break` or `continue` that does not match the loop label. Axiom (18) expresses that a `while` loop can never complete abruptly due to a matching `break` or `continue` statement.

$$[l\texttt{: while (e) st}]_{\mathcal{N}}\ \phi \ \equiv\ [\texttt{b = e;}]_{\mathcal{N}}((\neg\texttt{b} \rightarrow \phi)\ \wedge\ (\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{B}_l}\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[l\texttt{: while (e) st}]_{\mathcal{N}}\ \phi))) \tag{16}$$

$$\forall t \in \bigcup_{k\in\mathcal{L}\setminus\{l\}} \{break_k, continue_k\}.$$

$$[l\texttt{: while (e) st}]_t\ \phi \ \equiv\ [\texttt{b = e;}]_{\mathcal{N}}(\texttt{b} \rightarrow ([\texttt{st}]_t\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[l\texttt{: while (e) st}]_t\ \phi)) \tag{17}$$

$$[l\texttt{: while (e) st}]_{\{break, continue, break_l, continue_l\}}\ \phi \tag{18}$$

Fig. 2. Axioms for `while`

While this approach of adding many new modalities provides a sound theoretical grounding, a calculus directly using these axioms as rules is problematic in practice (in particular when using *symbolic execution* [13]), as it becomes quite complex very quickly. It should be pointed out that the axioms for the modalities covering exception throwing and returning from a method are more involved than the somewhat simpler modalities dealing with `break`s and `continue`s. Additionally, modalities need to be analyzed multiple times, as can be seen by applying (16) to $[\texttt{l: while (e) \{ st1 st2 \}}]_{normal}\ \phi$ and then simplifying with (11) and (12), leading to three separate occurences of $[\texttt{b = e;}]_{normal}[\texttt{st1}]_{normal}(\cdot)$. Using symbolic execution, this involves multiple symbolic executions of the exact same program fragment in the same state with the same context, which is a waste of resources.

For these and other reasons, the authors of JavaDL chose to instead keep track of the context *within the program part* of the modality, rather than creating additional modality types. To this end they defined *legal program fragments* [1], which may occur in the program part of a modality:

Definition 2. *Let Prg be a Java program. A* legal program fragment `p` *is a sequence of Java statements, where there are local variables* $\texttt{a}_1, \ldots, \texttt{a}_n$ *of Java types* $\texttt{T}_1, \ldots, \texttt{T}_n$ *such that extending Prg with an additional class* `C` *yields again a legal program according to the rules of the Java language specification [8], except that* `p` *may refer to fields, methods and classes that are not visible in* `C`*, and* `p` *may contain* extended Java statements *in addition to normal Java statements; where the class* `C` *is declared:*

```
public class C {
  public static void m(T₁ a₁, ..., Tₙ aₙ) { p }
}
```

In [1] the only *extended Java statement* allowed was the *method-frame*, a way to track the context of within which method call (of which object or class) a program fragment was to be executed. This allows for method calls within a program fragment to be replaced with method-frames containing their expanded method bodies.

Definition 3. *The set of all* JavaDL formulae *is defined as the smallest set containing all:*

– *first-order formulae,*
– $[\mathtt{p}]\phi$*, where* `p` *is a legal program fragment and* ϕ *is a JavaDL formula, and*
– $\{\mathcal{U}\}\phi$*, where* ϕ *is a JavaDL formula and* $\mathcal{U}$ *is an* update*.*

Definition 4. *An* update $\mathcal{U}$ *expresses state changes. An* elementary update $\mathtt{x} := t$ *represents the states where the variable* `x` *is set to the value of the term* t*, while a* parallel update $\mathcal{U}_1 \parallel \mathcal{U}_2$ *expresses both updates simultaneously (with a last-wins to resolve conflicts). Updates can be applied to terms ($\{\mathcal{U}\}t$), formulae ($\{\mathcal{U}\}\phi$) and other updates ($\{\mathcal{U}_1\}\mathcal{U}_2$), creating new terms, formulae and updates representing the changed state.*

A legal program fragment has the form "π `st` ω", where the *non-active prefix* π initially consisted only of an arbitrary sequence of opening braces "`{` ", labels, beginnings "`method-frame(...) {`" of method invocation statements, and beginnings "`try {`" of `try`-(`catch`)-`finally` statements; `st` is the *active statement*; and ω is the rest of the program, in particular including closing braces corresponding to the opening braces in π. Certain active statements can interact with the non-active prefix.

JavaDL uses a *sequent calculus* in which rules consist of one conclusion and any number of premisses, and are applied bottom-up. In addition to first-order logic rules, there are *symbolic execution rules*, which operate on the active statement inside a legal program fragment.

Example 1. We consider: $\{\mathtt{x} := 1\}$[`l : { y = x; break l; y = 0; }`]$(y \neq 0)$

Here "`l : {` " is the non-active prefix, while "`y = x;`" is the active statement. JavaDL contains a symbolic execution rule to execute a simple assignment, which leads to the formula $\{\mathtt{x} := 1\}\{y := x\}$[`l : { break l; y = 0; }`]$(y \neq 0)$. Now the active statement "`break l;`" interacts with the non-active prefix, removing the labeled block completely and leaving the formula $\{\mathtt{x} := 1\}\{y := x\}[](y \neq 0)$ which is equivalent to $\{\mathtt{x} := 1\}\{y := x\}(y \neq 0)$. Applying the updates gives first $\{\mathtt{x} := 1\}(x \neq 0)$ and then $1 \neq 0$, which obviously holds. The update $\mathtt{x} := 1$ could have alternatively been applied to the update $y := x$, yielding the parallel update $\mathtt{x} := 1 \parallel y := \{\mathtt{x} := 1\}x$, which simplifies to $\mathtt{x} := 1 \parallel y := 1$. Applying this update to $(y \neq 0)$ also leads to the formula $(1 \neq 0)$.

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := se\}[\pi\ \ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathtt{x}\ \mathtt{=}\ se\mathtt{;}\ \ \omega]\phi, \Delta}$$

$$\text{blockBreak}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ l_1, \ldots, \mathtt{l}, \ldots l_n\mathtt{:}\ \mathtt{\{\ break\ l;\ p\ \}}\ \ \omega]\phi, \Delta}$$

Initially there was no designated non-active prefix that allowed interaction with unlabeled `break`s as well as labeled and unlabeled `continue`s, which can occur in loop bodies. This makes a simple loop unrolling rule impossible, therefore loop bodies were transformed when unrolling the loop or when applying the loop invariant rule directly to a `while` loop. With a `for` loop, the entire loop was first transformed, creating a `while` loop, with a further program transformation of the loop body when dealing with said `while` loop. However, sound program transformation rules for a complex language such as Java lead to very opaque program fragments, which have next to no relation to the original program, as can be seen in Examples 2 and 3.

In [20] the concept of an *indexed loop scope* (a further *extended Java statement* $\circlearrowleft_{\mathtt{x}}\ \mathtt{st}\ {}_{\mathtt{x}}\circlearrowright$) was proposed, allowing a designated non-active prefix for loop bodies (although the semantics of the indexed loop scope were such that it is directly useful only for a loop invariant rule for `while` loops). Symbolic execution rules for `continue`s and unlabeled `break`s, as well as interaction between the various completion statements and the loop scope were defined. This allowed for the loop invariant rule below, which avoids program transformation of the loop body. Additionally, it was shown in [18] that an implementation of this rule in KeY was more efficient than the loop invariant rule relying on program transformation.

loopInvariantWhileWithLoopScopes

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta \\ \mathit{Inv} \Longrightarrow [\pi\ \ \circlearrowleft_{\mathtt{x}}\ \mathtt{if}\ \mathtt{(}nse\mathtt{)}\ \mathtt{\{\ p\ continue;\ \}}\ {}_{\mathtt{x}}\circlearrowright\ \ \omega]((\mathtt{x} \doteq \mathrm{FALSE} \rightarrow \mathit{Inv}) \\ \qquad\qquad\qquad\qquad\qquad \&\ (\mathtt{x} \doteq \mathrm{TRUE} \rightarrow \phi))\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathtt{while}\ \mathtt{(}nse\mathtt{)}\ \mathtt{p}\ \ \omega]\phi, \Delta}$$

The first premiss ensures that the invariant holds in the program state before the first iteration of the loop. The second premiss ensures both that normal

Example 2. Sound program transformation of the `for` loop in Listing 1.1 leads to the `while` loop in Listing 1.2.

```
for (; x > 1; x = x / 2) {
  if (x % 2 == 0) continue;
  if (x % 5 == 0) break;
}
```

Listing 1.1. Original `for` loop

```
b: {
     while (x > 1) {
c:     {
         if (x % 2 == 0) break c;
         if (x % 5 == 0) break b;
       }
       x = x / 2;
     }
   }
```

Listing 1.2. Transformed `while` loop

Example 3. Consider the program fragment in Listing 1.3. Sound program transformation of this loop's body (in order to apply the loop invariant rule) must track abrupt completion within the body, but also reset and restore this tracking when encountering the `finally` block to ensure that the semantics are not altered. This leads to the program fragment shown in Listing 1.4.

```
while (x != 0) {
  try {
    if (x > 0) return x;
    x = x + 100;
    break;
  } finally {
    if (x > 10) {
      x = -1;
      continue;
    }
  }
}
```

Listing 1.3. Original loop

After executing this transformed loop body, the proof then continues on multiple branches for: (1.) the "preserves invariant" case where `brk` and `rtn` are `false`, and `thrown` is `null`; (2.) the "exceptional use case" where `thrown` is not null; (3.) the "return use case" where `rtn` is `true`; and (4.) the "break use case" where `brk` is `true`.

```
Throwable thrown = null;
boolean brk = false;
boolean cnt = false;
boolean rtn = false;
int rtnVal = 0;
try {
  l: {
    try {
      if (x > 0) {
        rtnVal = x;
        rtn = true;
        break l;
      }
      x = x + 100;
      brk = true;
      break l;
    } finally {
      boolean saveBrk = brk;
      brk = false;
      boolean saveCnt = cnt;
      cnt = false;
      boolean saveRtn = rtn;
      rtn = false;
      if (x > 10) {
        x = -1;
        cnt = true;
        break l;
      }
      brk = saveBrk;
      cnt = saveCnt;
      rtn = saveRtn;
    }
  }
} catch (Throwable t) {
  thrown = t;
}
```

Listing 1.4. Transformed loop body

and abrupt continuation of the loop body preserves the invariant; and that after leaving the loop normally or abruptly and executing the remaining program the original formula ϕ holds. As this must hold for *any* iteration, the assumptions $\Gamma \cup \neg\Delta$ and the update $\mathcal{U}$ expressing the program state before the first iteration are removed, with only the invariant as an assumption for the second premiss.

However, loop scopes work in a fairly opaque way: as can be seen in the rule above, the loop scope index `x` is never explicitly set anywhere in the rule, but rather will implicitly be set by the symbolic execution rules operating on loop scopes (with `continue` setting it to false, and everything else setting it to true). In this paper we show how to create a more transparent solution.

3 New Loop Unrolling Rules for JavaDL

In order to introduce new loop unrolling rules specifically for `while`, `do` and `for` loop, which do not require program transformation of the loop bodies, we require a non-active prefix for loop bodies in JavaDL. To this end we introduce the `attempt-continuation` statement:

3.1 Introducing the `attempt-continuation` Statement

Definition 5. *An* `attempt-continuation` *statement is an* extended Java statement *of the form "*$\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }`*" where* `l` $\in \mathcal{L}$ *is a label, and* `p` *and* `q` *are (extended) Java statements. Non-active prefixes may additionally contain beginnings "*$\texttt{attempt}_\texttt{l}$ `{`*" of* `attempt-continuation` *statements.*

If `p` does not contain any labeled `break` or `continue` statements matching the label `l`, "$\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }`" is equivalent to its *unlabeled* counterpart "`attempt { p } continuation { q }`". Non-active prefixes may therefore contain unlabeled `attempt-continuation` beginnings "`attempt {`".

The semantic meaning of $\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }` is that `p` is executed first, then there is a choice:

1. If `p` completes normally or completes abruptly due to a matching `continue` statement (`continue l;` or `continue;`), `q` is executed and the statement $\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }` completes for the same reason as `q`.
2. If `p` completes abruptly due to a matching `break` (`break l;` or `break;`), `q` is *not* executed and $\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }` completes normally.
3. If `p` completes abruptly for any other reason (including due to a statement `continue l';` or `break l';` where `l` $\neq$ `l'`), `q` is *not* executed and $\texttt{attempt}_\texttt{l}$ `{ p } continuation { q }` completes abruptly for the same reason `p` completed abruptly.

Axioms for `attempt-continuation` statements are shown in Fig. 3.

Correct unrolling of a `while` loop is now possible with the help of `attempt-continuation` statements, as shown in Theorem 1.

Theorem 1 (Correctness of loop unrolling). $[\texttt{l: while (e) st}]_t\ \phi$ *is equivalent to* $[\texttt{if (e) attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \}}]_t\ \phi$ *for all completion types* $t \in \mathcal{T}$*.*

Proof. See appendix.

$$[\texttt{attempt}_\texttt{l}\ \texttt{\{ p \} continuation \{ q \}}]_\mathcal{N}\ \phi \ \equiv\ [\texttt{p}]_{\mathcal{C}_l}[\texttt{q}]_\mathcal{N}\ \phi\ \wedge\ [\texttt{p}]_{\mathcal{B}_l}\ \phi \tag{19}$$

$\forall t \in \{\mathit{break}, \mathit{continue}, \mathit{break}_l, \mathit{continue}_l\}$.

$$[\texttt{attempt}_\texttt{l}\ \texttt{\{ p \} continuation \{ q \}}]_t\ \phi \ \equiv\ [\texttt{p}]_{\mathcal{C}_l}[\texttt{q}]_t\ \phi \tag{20}$$

$\forall t \in \bigcup_{k \in \mathcal{L}\setminus\{l\}} \{\mathit{break}_k, \mathit{continue}_k\}$.

$$[\texttt{attempt}_\texttt{l}\ \texttt{\{ p \} continuation \{ q \}}]_t\ \phi \ \equiv\ [\texttt{p}]_{\mathcal{C}_l}[\texttt{q}]_t\ \phi\ \wedge\ [\texttt{p}]_t\ \phi \tag{21}$$

Fig. 3. Axioms for `attempt-continuation`

3.2 Symbolic Execution Rules for `attempt-continuation`

We introduce new symbolic execution rules for the `attempt-continuation` statement into JavaDL as follows:

For an empty `attempt` block:

$$\textsf{emptyAttempt}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{q}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

We combine two rules into one here, by writing "$\texttt{attempt}_{\texttt{l}^?}$" to express that there is a rule for the labeled `attempt-continuation` statement and a rule for the unlabeled `attempt-continuation` statement.

For an `attempt` block with a leading `continue` statement:

attemptContinueNoLabel

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{q}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ continue; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

attemptContinue

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{q}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_\texttt{l}\ \texttt{\{ continue l; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

attemptContinueNoMatch

$$\texttt{l} \neq \texttt{l}':\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{continue l';}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ continue l'; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

For an `attempt` block with a leading `break` statement:

attemptBreakNoLabel

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ break; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

attemptBreak

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_\texttt{l}\ \texttt{\{ break l; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

attemptBreakNoMatch

$$\texttt{l} \neq \texttt{l}':\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{break l';}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ break l'; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

For an `attempt` block with a leading `throw` statement:

attemptThrow
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw}\ se\texttt{;}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ throw}\ se\texttt{; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

For an `attempt` block with a leading `return` statement:

attemptEmptyReturn
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{return;}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ return; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

attemptReturn
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{return}\ se\texttt{;}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ return}\ se\texttt{; p \} continuation \{ q \}}\ \omega]\phi, \Delta}$$

Further symbolic execution rules in JavaDL for `continue` statements and unlabeled `break` statements when encountering other non-active prefixes are identical to those given in [20]. These merely propagate the abruptly completing statements upwards (executing the `finally` block first, in the case of a `try`-(`catch`)-`finally` statement). As an example, where *cs* is a possibly empty list of `catch`-blocks:

$$\text{tryContinueNoLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{r continue;}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{try \{ continue; p \}}\ cs\ \texttt{finally \{ r \}}\ \omega]\phi, \Delta}$$

3.3 JavaDL Loop Unwinding Rules Using `attempt-continuation`

We can also use `attempt-continuation` statements in a loop unwinding rule for `while` loops in JavaDL. This does not require nested modalities as used in [20]:

unwindWhileLoop
$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{if (}nse\texttt{) attempt}_{\texttt{l}^?}\ \texttt{\{ p \}}\\ \qquad\qquad\qquad \texttt{continuation \{ l}^?\texttt{: while (}nse\texttt{) p \}}\ \omega]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^?\texttt{: while (}nse\texttt{) p}\ \omega]\phi, \Delta}$$

We unroll and execute one iteration of the loop, winding up back at the beginning of the loop unless the loop body completes abruptly (not due to a matching `continue`). This closely resembles the loop unrolling equivalence in Theorem 1.

The loop unwinding rule for `do` loops is almost the same, except that the condition is not checked before the first iteration:

unwindDoLoop
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ p \} continuation \{ l}^?\texttt{: while (}nse\texttt{) p \}}\omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^?\texttt{: do p while (}nse\texttt{);}\ \omega]\phi, \Delta}$$

As can be seen, a single loop unwinding turns a `do` loop into a `while` loop.

We can also introduce a loop unwinding rule for the `for` loop. As will be seen later, we have a rule to pull out the initializer of the `for` loop, so the rule only considers `for` loops with empty initializers:

unwindForLoop

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{if (}g'\texttt{) attempt}_{\texttt{l}^?}\ \texttt{\{ p \}}\ \texttt{continuation \{ } upd'\ \texttt{l}^?\texttt{: for (; } g\texttt{; } upd\texttt{) p \}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^?\texttt{: for (; } g\texttt{; } upd\texttt{) p}\ \omega]\phi, \Delta}$$

Here upd' is a statement list equivalent to the expression list upd, and g' is an expression equivalent to the guard g (if g is empty, g' is `true`).

As in the rules for `while` and `do` loops, the loop body is executed in an `attempt` block. But before re-entering the loop in the continuation, we execute the `for` loop's update. This ensures that we execute the `for` loop's update whether the loop body completes normally or completes abruptly due to a matching `continue` statement.

4 New Loop Invariant Rules for JavaDL

In order for the loop invariant rule based on loop scopes to be sound, when a `continue` statement reached a loop scope the appropriate symbolic execution rule in JavaDL needed to opaquely do two things: (1.) set the loop scope index to false and (2.) remove the entire surrounding legal program fragment. Thanks to `attempt-continuation` statements we can explcitly set a variable in the continuation in order to transparently solve the first of these issues. However in order to solve the second issue transparently, we require the addition of a further extended Java statement, which explicitly halts the program.

4.1 Introducing the Halt Statement

Definition 6. *The* halt statement *(written $\downarrow$) is an* extended Java statement *that, when executed, immediately halts the entire legal program fragment in which it is contained, ensuring that no further statements are executed (not even statements in* `finally` *blocks).*

The dynamic logic with modalities for each type of completion can be extended with new modalities $[\texttt{p}]_\downarrow(\cdot)$ for all legal program fragments `p`. Axioms for $\downarrow$ and the new modalities are shown in Fig. 4. In particular, loop unrolling using `attempt-continuation` statements is also valid in the halt modalities:

Theorem 2 (Correctness of loop unrolling in the halt modalities). $[\texttt{if (e) attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \}}]_\downarrow\ \phi$ *is equivalent to* $[\texttt{l: while (e) st}]_\downarrow\ \phi$.

Proof. See appendix.

Halting in JavaDL

The single symbolic execution rule in JavaDL required for the halt statement is:

$$\textsf{halt}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \downarrow\ \omega]\phi, \Delta}$$

$$[\texttt{;}]_{\downarrow}\ \phi \quad (22) \qquad [\texttt{break;}]_{\downarrow}\ \phi \quad (23) \qquad [\texttt{continue;}]_{\downarrow}\ \phi \quad (25) \qquad [\downarrow]_{\downarrow}\ \phi \ \equiv\ \phi \quad (27)$$

$$[\texttt{break}\ l\texttt{;}]_{\downarrow}\ \phi \quad (24) \qquad [\texttt{continue}\ l\texttt{;}]_{\downarrow}\ \phi \quad (26) \qquad [\downarrow]_{\mathcal{T}}\ \phi \quad (28)$$

$$[\texttt{st1 st2}]_{\downarrow}\ \phi \ \equiv\ [\texttt{st1}]_{\downarrow}\ \phi \ \land\ [\texttt{st1}]_{\mathcal{N}}[\texttt{st2}]_{\downarrow}\ \phi \quad (29)$$

$$\begin{aligned}&[\texttt{if (e) st1 else st2}]_{\downarrow}\phi\\ \equiv\ &[\texttt{b = e;}]_{\downarrow}\phi \ \land\ [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow [\texttt{st1}]_{\downarrow}\phi) \ \land\ (\neg\texttt{b} \rightarrow [\texttt{st2}]_{\downarrow}\phi)) \quad (30)\\ &[l\texttt{: while (e) st}]_{\downarrow}\phi\\ \equiv\ &[\texttt{b = e;}]_{\downarrow}\phi \ \land\ [\texttt{b = e;}]_{\mathcal{N}}(\texttt{b} \rightarrow ([\texttt{st}]_{\downarrow}\phi \ \land\ [\texttt{st}]_{\mathcal{C}_l}[l\texttt{: while (e) st}]_{\downarrow}\ \phi)) \quad (31)\end{aligned}$$

$$[\texttt{try \{ p \} finally \{ q \}}]_{\downarrow}\phi \ \equiv\ [\texttt{p}]_{\downarrow}\phi \ \land\ [\texttt{p}]_{\mathcal{T}}[\texttt{q}]_{\downarrow}\phi \quad (32)$$

$$[\texttt{attempt}_{\texttt{l}}\ \texttt{\{ p \} continuation \{ q \}}]_{\downarrow}\phi \ \equiv\ [\texttt{p}]_{\downarrow}\phi \ \land\ [\texttt{p}]_{\mathcal{C}_l}[\texttt{q}]_{\downarrow}\phi \quad (33)$$

Fig. 4. Axioms for the halt statement and halt modality

Provided correct modalities for `throw` and `return`, as well as further axioms for missing Java statements (in particular `throw`, `try-catch`, `return`, *assignment* and dealing with method calls), Conjecture 1 claims equivalence between JavaDL and the dynamic logic with modalities for each type of completion.

Conjecture 1. The JavaDL formula $[\texttt{p}]\phi$ must hold iff ϕ holds in all normally completing or halting states reachable by executing `p`:

$$[\texttt{p}]\phi \ \equiv\ [\texttt{p}]_{normal}\ \phi \ \land\ [\texttt{p}]_{\downarrow}\phi$$

4.2 Loop Invariant Rule for `while` Loops Using `attempt-continuation`

Thanks to `attempt-continuation` and halt statements we introduce the following loop invariant rule for `while` loops, where `x` is a fresh boolean variable not occuring anywhere in the legal program fragment "π $\texttt{l}^?$`: while (`*nse*`) p` ω":

loopInvariantWhile

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta\\ \mathit{Inv} \Longrightarrow [\pi\ \texttt{x = true;}\\ \qquad\qquad \texttt{if (}\mathit{nse}\texttt{)}\\ \qquad\qquad\qquad \texttt{attempt}_{\texttt{l}^?}\ \texttt{\{ } p \texttt{ \} continuation \{ x = false; } \downarrow \texttt{ \}}\\ \qquad\quad \omega]((\texttt{x} \doteq \text{FALSE} \rightarrow \mathit{Inv})\ \&\ (\texttt{x} \doteq \text{TRUE} \rightarrow \phi))\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^?\texttt{: while (}\mathit{nse}\texttt{) p}\ \omega]\phi, \Delta}$$

As the `continuation` block is constructed only from a simple assignment and the halt statement, if `p` completes normally or completes abruptly due to a matching `continue`, it is guaranteed to set `x` to false and complete due to the halt statement, leaving the invariant to be proven in the state reached after execution of a single loop iteration.

In all other cases `x` retains its initial value true, leaving $\{\mathcal{U}'\}[\pi\ \texttt{abrupt}\ \omega]\phi$ to be proven, with $\mathcal{U}'$ expressing the state the program is in when the loop is left.

If *nse* evaluates to false or `p` completes abruptly due to a matching `break`, then `abrupt` is empty and it remains to prove $\{\mathcal{U}'\}[\pi\ \ \omega]\phi$. If `p` completes abruptly due to any other statement, `abrupt` is equal to that abruptly completing statement.

4.3 Loop Invariant Rule for `for` Loops Using `attempt-continuation`

In order to prove that the loop invariant of a `for` loop initially holds, we must first reach the "initial" entry point of the loop. This is the point after full execution of the loop initializer. We therefore introduce the following rule to pull out the loop initializer of a `for` loop, where *init′* is a statement list equivalent to the loop initializer *init*:

$$\mathsf{pullOutLoopInitializer}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{(}\mathit{init'}\ \texttt{l}^{?}\texttt{: for (;}\ \mathit{guard}\texttt{;}\ \mathit{upd}\texttt{) p \}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^{?}\texttt{: for (}\mathit{init}\texttt{;}\ \mathit{guard}\texttt{;}\ \mathit{upd}\texttt{) p}\ \omega]\phi, \Delta}$$

The following loop invariant rule can then be applied to `for` loops without loop initializers, where `x` is a fresh boolean variable not occurring anywhere in the legal program fragment "π $\texttt{l}^{?}$`: for (;` *guard*`;` *upd*`) p` ω", *upd′* is a statement list equivalent to the expression list *upd*, and *guard′* is an expression equivalent to the guard *guard* (`true`, if *guard* is empty):

$$\mathsf{loopInvariantFor}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta \\ \mathit{Inv} \Longrightarrow [\pi\ \texttt{x = true;} \\ \qquad\quad \texttt{if (}\mathit{guard'}\texttt{)} \\ \qquad\qquad \texttt{attempt}_{\texttt{l}^{?}}\ \texttt{\{}\ p\ \texttt{\} continuation \{}\ \mathit{upd'}\ \texttt{x = false;}\ \downarrow\ \texttt{\}} \\ \qquad\ \omega]((\texttt{x} \doteq \mathrm{FALSE} \rightarrow \mathit{Inv})\ \&\ (\texttt{x} \doteq \mathrm{TRUE} \rightarrow \phi))\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^{?}\texttt{: for (;}\ \mathit{guard}\texttt{;}\ \mathit{upd}\texttt{) p}\ \omega]\phi, \Delta}$$

As the continuation is constructed only from the modified `for` loop update *upd′*, a simple assignment and the halt statement, it cannot contain `break`s, `continue`s or `return`s. It also cannot contain an explicit `throw`, but implicitly exceptions can be thrown in *upd′*. Thus if `p` completes normally or completes abruptly due to a matching `continue`, causing symbolic execution of the continuation, this will either set `x` to false and complete due to the halt statement, leaving the invariant to be proven in the state reached after execution of a single loop iteration; or it will complete abruptly due to a statement `throw se;` (keeping `x` set to its initial value of true), leaving $\{\mathcal{U}'\}[\pi\ \texttt{throw se;}\ \omega]\phi$ to be proven, with $\mathcal{U}'$ expressing the state the program is in when the loop is left abruptly due to the exception. All other cases are identical to those for the `while` loop invariant above.

Theorem 3. *The symbolic execution loop invariant rules* loopInvariantWhile *and* loopInvariantFor *are sound.*

Proof (Sketch). See the extended technical report [22].

As can be seen, introducing `attempt-continuation` and halt statements has allowed us to have a loop invariant rule specifically for `for` loops, which does not require program transformation of the loop body and only minimal program transformation of the loop update. This allows `for` loops to be treated as first-class citizens in proofs and lets user interactions occur on legal program fragments which are still reasonably close to the original program, rather than on those which have been transformed in such a way that it is unclear how they relate to the original program. This increases the transparency of the proof.

4.4 Why No Loop Invariant Rule for `do` Loops?

One could imagine that a similar case could be made to treat `do` loops as first-class citizens in proofs, by supplying a loop invariant rule specifically for `do` loops. However, this is not really the case. As with the other loop types, the loop invariant for a `do` loop needs to hold only just before the condition is checked. However, unlike the other loop types, this is not the case for `do` loops until after the first loop iteration. This makes a loop invariant rule for `do` loops actually *less* transparent, than the reasonably simple steps of (1.) converting the `do` loop into a `while` loop and (2.) applying the loop invariant rule for `while` loops on the resulting `while` loop. This transformation of a `do` loop into a `while` loop can happen in one of two ways: (i) by applying the unwindDoLoop rule to the `do` loop and symbolically executing the unrolled body until the `attempt`-block is exited and the `while` loop in the `continuation`-block becomes the active statement, or (ii) by applying the program transformation rule from [9] to the `do` loop, producing a `while` loop directly without needing to symbolically execute the first loop iteration:

transformDoToWhile

$$\frac{\Gamma \Longrightarrow \{\mathcal{U} \parallel \texttt{fst} := \mathrm{TRUE}\}[\texttt{l}^{?}\texttt{: while (fst || } \mathit{nse}\texttt{) \{ fst = false; p \} } \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{l}^{?}\texttt{: do p while (}\mathit{nse}\texttt{); } \omega]\phi, \Delta}$$

Here `fst` is a fresh boolean variable. The loop invariant then applied to the resulting `while` loop can use the value of `fst` if the invariant of the original `do` loop is only established after at least one iteration of the loop has been executed.

5 Evaluation

Based on our previous work on providing a loop invariant rule specifically for `for` loops using loop scopes [21], Benedikt Dreher implemented this loop invariant in KeY and evaluated it in [5]. He found that the efficiency of the new rule was similar to the pure program transformation rule and the rule using loop scopes on `while` loops produced by program transformation of the `for` loop. The new rule required only about 80% as many nodes and execution steps as the pure program transformation rule, while creating slightly more branches (creating an average of 27.86 to 27.5 branches in the examples). It was about 10% less efficient than the rule using loop scopes on `while` loops produced by program transformation of the `for` loop. However, the new rule provided more transparency, as it was easier to see in the proof tree which statement in the original `for` loop was being processed, as well as seeing directly what the result of applying the loop invariant rule to a `for` loop would produce.

The rules proposed in this paper should be slightly more efficient, as they do not require the unnecessary steps of resetting the loop scope index before symbolically executing the `for` loop's update and then setting the loop scope index afterwards, as the implemented rule from [21] does. Additionally, the transparency of the rules proposed in this paper should be even greater, as the opacity of the loop scope has been completely replaced with the transparency of `attempt-continuation` and halt statements.

6 Related Work

We have already compared our approach to other JavaDL approaches using program transformation of the loop body or indexed loop scopes, showing that our approach here is much more transparent. We have also compared this approach to using a dynamic logic with typed modalities for each completion type, which has drawbacks in particular when using symbolic execution. We unfortunately could not find any work formally explaining the handling of irregular control flow in loops for VeriFast [11], a symbolic execution system for C and Java; the most formal paper we could find [12] describes only a reduced language without `break`s and `continue`s. The symbolic execution calculus for KIV [19] is also a dynamic logic variant. However, they sequentially decompose (*flatten*) statements, such that a non-active prefix is not needed. This is accomplished by including both heavy program transformation and tracking of *mode information*, which has similarities to using a dynamic logic with typed modalities for each completion type. Additionally, their approach cannot deal directly with `continue`s, as they claim that these are problematic for loop unwinding; we have shown that this is not the case with our approach, providing loop unwinding rules for not only `while`, but also `do` and `for` loops. OpenJML [4] and other approaches using *verification condition generation* work by translating the program into an intermediate language. Abrupt completion is usually modelled by branches to basic blocks. This might make these approaches efficient, but the treatment of *all* loop types becomes completely opaque. While intermediate languages are less complex (which can be helpful), the translation into them can require compromises concerning soundness [7] and is a non-trivial and error-prone task [15] in any case.

Finally, concurrently to this work we came up with *Completion Scopes*[3] [17], which are non-active prefixes that are powerful enough to express all other non-active prefixes. There are both advantages and disadvantages to specialized solutions versus generalized solutions, but we feel that there should be enough room in this research area for both to coexist peacefully.

[3] To clarify: Nathan Wasser came up with the general idea of completion scopes, while Dominic Steinhöfel fleshed this idea out with some limited input from Nathan.

7 Conclusion and Future Work

We have introduced `attempt-continuation` and halt statements as extended Java statements that allow more localized reasoning for loops and a way to express immediately halting the Java program. Axioms for these statements and the appropriately typed modalities have been given in a dynamic logic with modalities for various completion types. These statements are of particular interest in JavaDL, where we have supplied symbolic execution rules for them.

We have shown that using `attempt-continuation` statements rather than indexed loop scopes lets us gain great potential:

1. We are able to express a loop invariant rule specifically for `for` loops which does not require program transformation of the loop body and allows a transparent treatment of `for` loops as first-class citizens in proofs.
2. We are able to express loop unrolling rules for `while`, `do` and `for` loops which require neither program transformation of the loop body, nor the use of nested modalities.
3. The rule for a `continue` reaching the `attempt`-block (the non-active prefix responsible for loop bodies) is more transparent than the corresponding rule for loop scopes, simply executing the continuation (whatever it may be), rather than opaquely setting the loop scope index to false.

As future work we would like to implement these ideas into KeY, performing an evaluation of the loop invariant rules for `while` and `for` loops with this approach on the examples tested in [18] and [5], so as to compare them with the loop scope approach. We would also like to evaluate the new loop unrolling rules and are looking to find an appropriate benchmark for that.

Additionally, we would be interested in adding a *halts* clause to JML [3] method contracts, in order to express what must hold if a method executes the halt statement. While no Java method can syntactically contain the halt statement, the Java virtual machine does provide the effect of halting, with the methods `Runtime.exit()` and `System.exit()` [8, Chapter 12.8]. Providing a way to express halting in a method contract is therefore somewhat of interest.

Finally, a comparison between implementations of this work and completion scopes would be very insightful.

Acknowledgements. We thank Benedikt Dreher for his implementation and evaluation of our previous attempt at using indexed loop scopes to create a loop invariant rule specifically for `for` loops. We thank Richard Bubel for all his helpful comments and the wonderful discussions these lead to. We owe a great debt of gratitude to Dominic Steinhöfel for his help on the technical report which lead to this paper, as well as for all the fruitful (and sometimes wine-ful) discussions on loop scopes and their offspring.

Appendix

Proofs for the Theorems

Theorem 1 (Correctness of loop unrolling). $[\texttt{l: while (e) st}]_t\ \phi$ *is equivalent to* $[\texttt{if (e) attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \}}]_t\ \phi$ *for all completion types* $t \in \mathcal{T}$.

Proof. $[\texttt{if (e) attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \}}]_t\ \phi$ *expands to*: $[\texttt{if (e) attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \}else ;}]_t\ \phi$.

As $\mathcal{A} = \{\mathit{break}, \mathit{continue}, \mathit{break}_{\texttt{l}}, \mathit{continue}_{\texttt{l}}\} \cup \bigcup_{k\in\mathcal{L}\setminus\{\texttt{l}\}}\{\mathit{break}_k, \mathit{continue}_k\}$ *and* $\mathcal{T} = \mathcal{N} \cup \mathcal{A}$, *by case distinction:*

If $t \in \mathcal{N}$**:**

$$
\begin{aligned}
& [\texttt{if (e) attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \} else ;}]_{\mathcal{N}}\ \phi \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}} \\
& \quad ((\texttt{b} \rightarrow [\texttt{attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \}}]_{\mathcal{N}}\ \phi) \\
& \quad \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_{\mathcal{N}}\ \phi)) && \text{by (13)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_{\mathcal{N}}\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{B}_l}\ \phi))\ \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_{\mathcal{N}}\ \phi)) \\
& && \text{by (19)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_{\mathcal{N}}\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{B}_l}\ \phi))\ \wedge\ (\neg\texttt{b} \rightarrow \phi)) \\
& && \text{by (1)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}((\neg\texttt{b} \rightarrow \phi)\ \wedge\ (\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{B}_l}\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_{\mathcal{N}}\ \phi))) \\
& && \text{by } \textit{commutativity} \text{ of } \wedge \\
\equiv\ & [\texttt{l: while (e) st}]_{\mathcal{N}}\ \phi && \text{by (16)} \quad \square
\end{aligned}
$$

If $t \in \{\mathit{break}, \mathit{continue}, \mathit{break}_{\texttt{l}}, \mathit{continue}_{\texttt{l}}\}$**:**

$$
\begin{aligned}
& [\texttt{if (e) attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \} else ;}]_t\ \phi \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}} \\
& \quad ((\texttt{b} \rightarrow [\texttt{attempt}_{\texttt{l}}\texttt{ \{ st \} continuation \{ l: while (e) st \}}]_t\ \phi) \\
& \quad \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_t\ \phi)) && \text{by (13)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi)\ \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_t\ \phi)) && \text{by (20)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}((\texttt{b} \rightarrow [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi)\ \wedge\ (\neg\texttt{b} \rightarrow \mathit{true})) && \text{by (6)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}(\texttt{b} \rightarrow [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi) && \text{by definition of} \rightarrow \text{and} \wedge \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}(\texttt{b} \rightarrow [\texttt{st}]_{\mathcal{C}_l}\ \mathit{true}) && \text{by (18)} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}(\texttt{b} \rightarrow \mathit{true}) && \text{by } \textit{necessitation} \\
\equiv\ & [\texttt{b = e;}]_{\mathcal{N}}\ \mathit{true} && \text{by definition of} \rightarrow \\
\equiv\ & \mathit{true} && \text{by } \textit{necessitation} \\
\equiv\ & [\texttt{l: while (e) st}]_t\ \phi && \text{by (18)} \quad \square
\end{aligned}
$$

Otherwise, $t \in \bigcup_{k\in\mathcal{L}\setminus\{\mathtt{l}\}}\{\mathit{break}_k, \mathit{continue}_k\}$**:**

$$\begin{aligned}
& [\texttt{if (e) attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \} else ;}]_t\ \phi \\
\equiv\ & [\texttt{b = e;}]_\mathcal{N} \\
& \quad ((\texttt{b} \rightarrow [\texttt{attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \}}]_t\ \phi) \\
& \quad\ \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_t\ \phi)) && \text{by (13)} \\
\equiv\ & [\texttt{b = e;}]_\mathcal{N}((\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi\ \wedge\ [\texttt{st}]_t\ \phi))\ \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_t\ \phi)) \\
& && \text{by (21)} \\
\equiv\ & [\texttt{b = e;}]_\mathcal{N}((\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi\ \wedge\ [\texttt{st}]_t\ \phi))\ \wedge\ (\neg\texttt{b} \rightarrow \mathit{true})) \\
& && \text{by (6)} \\
\equiv\ & [\texttt{b = e;}]_\mathcal{N}(\texttt{b} \rightarrow ([\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi\ \wedge\ [\texttt{st}]_t\ \phi)) \\
& && \text{by definition of} \rightarrow \text{and} \wedge \\
\equiv\ & [\texttt{b = e;}]_\mathcal{N}(\texttt{b} \rightarrow ([\texttt{st}]_t\ \phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_t\ \phi)) \\
& && \text{by } \textit{commutativity} \text{ of } \wedge \\
\equiv\ & [\texttt{l: while (e) st}]_t\ \phi && \text{by (17)} \quad \square
\end{aligned}$$

Theorem 2 (Correctness of loop unrolling in the halt modalities). $[\texttt{if (e) attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \}}]_\downarrow\ \phi$ *is equivalent to* $[\texttt{l: while (e) st}]_\downarrow\ \phi$.

Proof.

$$\begin{aligned}
& [\texttt{if (e) attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \} else ;}]_\downarrow\ \phi \\
\equiv\ & [\texttt{b = e;}]_\downarrow\phi\ \ \wedge \\
& [\texttt{b = e;}]_\mathcal{N}((\texttt{b} \rightarrow [\texttt{attempt}_\texttt{l}\ \texttt{\{ st \} continuation \{ l: while (e) st \}}]_\downarrow\phi) \\
& \qquad\qquad \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_\downarrow\phi)) && \text{by (30)} \\
\equiv\ & [\texttt{b = e;}]_\downarrow\phi\ \wedge\ [\texttt{b = e;}]_\mathcal{N}((\texttt{b} \rightarrow ([\texttt{st}]_\downarrow\phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_\downarrow\phi)) \\
& \qquad\qquad\qquad\qquad \wedge\ (\neg\texttt{b} \rightarrow [\texttt{;}]_\downarrow\phi)) && \text{by (33)} \\
\equiv\ & [\texttt{b = e;}]_\downarrow\phi\ \wedge\ [\texttt{b = e;}]_\mathcal{N}((\texttt{b} \rightarrow ([\texttt{st}]_\downarrow\phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_\downarrow\phi)) \\
& \qquad\qquad\qquad\qquad \wedge\ (\neg\texttt{b} \rightarrow \mathit{true})) && \text{by (22)} \\
\equiv\ & [\texttt{b = e;}]_\downarrow\phi\ \wedge\ [\texttt{b = e;}]_\mathcal{N}(\texttt{b} \rightarrow ([\texttt{st}]_\downarrow\phi\ \wedge\ [\texttt{st}]_{\mathcal{C}_l}[\texttt{l: while (e) st}]_\downarrow\phi)) \\
& && \text{by definition of} \rightarrow \text{and} \wedge \\
\equiv\ & [\texttt{l: while (e) st}]_\downarrow\ \phi && \text{by (31)} \quad \square
\end{aligned}$$

Theorem 3. *The symbolic execution loop invariant rules* loopInvariantWhile *and* loopInvariantFor *are sound.*

Proof (Sketch). See the extended technical report [22].

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer (2016)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach, LNCS, vol. 4334. Springer (2007)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_16
4. Cok, D.R.: OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Proceedings 1st Workshop on Formal Integrated Development Environment, pp. 79–92 (2014)
5. Dreher, B.: Transparent treatment of loops in JavaDL. B.Sc. thesis, Darmstadt University of Technology, Germany (2019)
6. Filliâtre, J.C.: Deductive software verification. Int. J. Softw. Tools Technol. Transfer **13**(5), 397 (2011)
7. Flanagan, C., Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on POPL, pp. 193–205. ACM (2001)
8. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)
9. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reason. **62**(1), 93–126 (2019)
10. Harel, D., Tiuryn, J., Kozen, D.: Dynamic Logic. MIT Press (2000)
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
12. Jacobs, B., Vogels, F., Piessens, F.: Featherweight VeriFast. Log. Methods Comput. Sci. **11**(3), 1–57 (2015)
13. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
14. Kleene, S.C.: Representation of events in nerve nets and finite automata (1951)
15. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. J. Log. Algebr. Program. **58**, 89–106 (2004)
16. Schlager, S.: Symbolic execution as a framework for deductive verification of object-oriented programs. Ph.D. thesis, Karlsruhe Institute of Technology (2007)
17. Steinhöfel, D.: Abstract execution: automatically proving infinitely many programs. Ph.D. thesis, Darmstadt University of Technology, Germany (2020). http://tuprints.ulb.tu-darmstadt.de/8540/
18. Steinhöfel, D., Wasser, N.: A new invariant rule for the analysis of loops with non-standard control flows. In: 13th International Conference on Integrated Formal Methods IFM, pp. 279–294 (2017)
19. Stenzel, K.: Verification of Java card programs. Ph.D. thesis, Universität Augsburg (2005)

20. Wasser, N.: Automatic generation of specifications using verification tools. Ph.D. thesis, Darmstadt University of Technology, Germany (2016)
21. Wasser, N., Steinhöfel, D.: Using loop scopes with `for`-loops. Tech. rep., Darmstadt University of Technology, Germany (2019). https://arxiv.org/abs/1901.06839
22. Wasser, N., Steinhöfel, D.: Treating for-loops as first-class citizens in proofs. Tech. rep., Darmstadt University of Technology, Germany (2020). https://arxiv.org/abs/2002.00776

Author Index