Axel Legay
Saddek Bensalem (Eds.)

LNCS 8174

Runtime Verification

4th International Conference, RV 2013
Rennes, France, September 2013
Proceedings

Springer

Lecture Notes in Computer Science 8174

Axel Legay Saddek Bensalem (Eds.)

Runtime Verification

4th International Conference, RV 2013
Rennes, France, September 24-27, 2013
Proceedings

Springer

Volume Editors

Axel Legay
Inria Rennes
Campus de Beaulieu
263 Avenue Général Leclerc
35042 Rennes, France
E-mail: axel.legay@inria.fr

Saddek Bensalem
Université Joseph Fourier
VERIMAG Centre Équation
Avenue de Vignate 2
38618 Gières, France
E-mail: saddek.bensalem@imag.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40786-4 e-ISBN 978-3-642-40787-1
DOI 10.1007/978-3-642-40787-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013946888

CR Subject Classification (1998): D.2, F.2, D.2.4, D.1, C.2, F.3, D.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 2013 International Conference on Runtime Verification (RV 2013), held in Rennes, France, during September 24–27. RV 2013 was is the fourth of a series dedicated to the advancement of monitoring and analysis techniques for software and hardware system executions. The previous three editions of the RV conference took place in Malta (2010), San Fransisco (2011), and Istanbul (2012). The history of RV goes back to 2001 when it started as a workshop. It continued as an anual workshop series until 2009 and became a conference in 2010.

RV 2013 attracted 58 submissions (short/regular/tool papers) in response to the call for papers. Each submission was assigned to at least three members of the Program Committee; in many cases additional reviews were solicited from outside experts. The Program Committee discussed the submissions electronically, judging them on their perceived importance, originality, clarity, and appropriateness to the expected audience. The Program Committee selected 24 papers for presentation, leading to an acceptance rate of 41 %. In addition, two tutorials were selected for presentation during the main conference.

Complementing the contributed papers, the program of RV 2013 included invited lectures by Viktor Kuncak, Martin Leucker, and Klaus Ostermann. RV 2013 was preceded by a tutorial day and, for the first time as a stand-alone event, by a satellite workshop (SMC 2013, the first workshop on statistical model checking).

The chairs would like to thank the authors for submitting their papers to RV 2013. We are grateful to the reviewers who contributed to nearly 250 informed and detailed reports and discussions during the electronic Program Committee meeting. We also sincerely thank the Steering Committee for their advice. Finally, we would like to thank our local helpers, Louis-Marie Traonouez and Uli Fahrenberg, who took care of the website and of the proceedings, Edith Blin, who helped us with the social events, and Benoît Boyer and Cyrille Jégourel, who helped with the organization of the workshop. RV 2013 received sponsorships from INRIA Université de Rennes 1, Région Bretagne, Fondation Métivier, Rennes Métropole, and SISCOM Bretagne. The submission and evaluation of papers, as well as preparation if this proceedings volume, were handled by the EasyChair conference management system.

September 2013

Saddek Bensalem
Axel Legay

Organization

Program Committee

Cyrille Valentin Artho	AIST
Howard Barringer	The University of Manchester
Andreas Bauer	NICTA / Australian National University
Saddek Bensalem	VERIMAG, France
Eric Bodden	Fraunhofer / TU Darmstadt, Germany
Borzoo Bonakdarpour	University of Waterloo, Canada
Marco Faella	Università di Napoli Federico II, Italy
Ylies Falcone	Université Joseph Fourier, France
Gilles Geeraerts	Université Libre de Bruxelles, Belgium
Patrice Godefroid	Microsoft Research
Susanne Graf	Universite Joseph Fourier / CNRS / VERIMAG, France
Sylvain Hallé	Université du Québec à Chicoutimi, Canada
Klaus Havelund	Jet Propulsion Laboratory, California Institute of Technology, USA
Suresh Jagannathan	Purdue University, USA
Claude Jard	Université de Nantes, France
Thierry Jéron	INRIA Rennes - Bretagne Atlantique, France
Sarfraz Khurshid	The University of Texas at Austin, USA
Steve Kremer	INRIA Nancy - Grand Est, France
Insup Lee	University of Pennsylvania, USA
Axel Legay	IRISA/INRIA, Rennes, France
Oded Maler	CNRS-VERIMAG, France
Tiziana Margaria	Universität Potsdam, Germany
Darko Marinov	University of Illinois at Urbana-Champaign, USA
Atif Memon	University of Maryland, USA
Gordon Pace	University of Malta
Shaz Qadeer	Microsoft
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Arne Skou	Aalborg University, Denmark
Scott Smolka	Stony Brook University, USA
Oleg Sokolsky	University of Pennsylvania, USA
Serdar Tasiran	Koc University, Turkey
Martin Vechev	ETH Zürich, Switzerland
Xiangyu Zhang	Purdue University, USA
Lenore Zuck	University of Illinois in Chicago, USA

Additional Reviewers

Askarov, Aslan
Bacci, Giorgio
Bacci, Giovanni
Boudjadar, Abdeldjalil
Cho, Sungmin
Colombo, Christian
D'Amorim, Marcelo
Delaval, Gwenaël
Ehlers, Rüdiger
Ferrere, Thomas
Fischmeister, Sebastian
Francalanza, Adrian
Huang, Jeff
Ivanov, Radoslav
Kalajdzic, Kenan
Kim, Jin Hyun
King, Andrew
Kuester, Jan-Christoph
Kuncak, Victor
Lanik, Jan
Laurent, Mounier
Lee, Choonghwan
Luo, Qingzhou
Matar, Hassan Salehe
Medhat, Ramy
Morvan, Christophe
Mutlu, Erdal
Mutluergil, Suha Orhun
Naujokat, Stefan
Nickovic, Dejan
Olivo, Oswaldo
Ozkan, Burcu Kulahcioglu
Park, Junkil
Perrin, Matthieu
Reger, Giles
Rubin, Sasha
Ruething, Oliver
Rusu, Vlad
Rydeheard, David
Rüthing, Oliver
Sassolas, Mathieu
Schallhart, Christian
Swamy, Nikhil
Wang, Shaohui
Weimer, James
Weiss, Gera

Abstracts of Invited Talks

Executing Specifications Using Synthesis and Constraint Solving

Viktor Kuncak, Etienne Kneuss, and Philippe Suter

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
`firstname.lastname@epfl.ch`, `psuter@us.ibm.com`

Abstract. Specifications are key to improving software reliability as well as documenting precisely the intended behavior of software. Writing specifications is still perceived as expensive. Of course, writing implementations is at least as expensive, but is hardly questioned because there is currently no real alternative. Our goal is to give specifications a more balanced role compared to implementations, enabling the developers to compile, execute, optimize, and verify against each other mixed code fragments containing both specifications and implementations. To make specification constructs executable we combine deductive synthesis with run-time constraint solving, in both cases leveraging modern SMT solvers. Our tool decomposes specifications into simpler fragments using a cost-driven deductive synthesis framework. It compiles as many fragments as possible into conventional functional code; it executes the remaining fragments by invoking our constraint solver that extends an SMT solver to handle recursive functions. Using this approach we were able to execute constraints that describe the desired properties of integers, sets, maps and algebraic data types.

Runtime Verification with Data

Martin Leucker

Universtity of Lübeck
Institute for Software Engineering
and Programming Languages
Ratzeburger Allee 160
23562 Lübeck
leucker@isp.uni-luebeck.de

Abstract. In the talk accompanying this abstract several approaches for verifying properties involving data at runtime are reviewed. Starting with a typical, mainly academic account to runtime verification based on linear temporal logic (LTL) we present existing extensions such as LTL with parameterized propositions and first-order LTL. Moreover, we compare these approaches with further frameworks such as LOLA, RuleR and Eagle. The goal is to give a comprehensive picture of runtime verification in the light of data values.

Programming without Borders

Klaus Ostermann

Department of Computer Science and Mathematics
University of Marburg
Hans-Meerwein-Straße
35032 Marburg, Germany
kos@informatik.uni-marburg.de

Abstract. The standard programming model of most programming environments is characterized by several sharp borders: Compile-time versus run-time, compiler vs. program, object level vs. meta level, expressions vs types, and so forth. I argue that we should abandon or at least weaken these distinctions. To this end, I will present a library for collections that diffuses the compile-time/run-time distinction, an extensible programming language that unifies compiler extensions and object level programs, and a type system that coalesces the usual stratification into universes like terms, types, kinds. In each case, significant expressive power is gained by making borders permeable.

Table of Contents

Invited Paper

Regular Papers

Short Papers

Tool Papers

Tutorials

Executing Specifications Using Synthesis and Constraint Solving*

Viktor Kuncak[1], Etienne Kneuss[1], and Philippe Suter[1,2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{firstname.lastname}@epfl.ch, psuter@us.ibm.com

Abstract. Specifications are key to improving software reliability as well as documenting precisely the intended behavior of software. Writing specifications is still perceived as expensive. Of course, writing implementations is at least as expensive, but is hardly questioned because there is currently no real alternative. Our goal is to give specifications a more balanced role compared to implementations, enabling the developers to compile, execute, optimize, and verify against each other mixed code fragments containing both specifications and implementations. To make specification constructs executable we combine deductive synthesis with run-time constraint solving, in both cases leveraging modern SMT solvers. Our tool decomposes specifications into simpler fragments using a cost-driven deductive synthesis framework. It compiles as many fragments as possible into conventional functional code; it executes the remaining fragments by invoking our constraint solver that extends an SMT solver to handle recursive functions. Using this approach we were able to execute constraints that describe the desired properties of integers, sets, maps and algebraic data types.

1 Introduction

Specifications are currently second class citizens in software development. An implementation is obligatory; specification is optional. Our goal is to assign to specifications a more balanced role compared to implementations. For this to happen, we aim to allow developers to execute specifications, even if such execution is slower or less predictable than execution of imperative and functional code. We wish to permit developers to write mixed code fragments containing both specifications and implementations. They should be able to compile, execute, optimize, and verify such fragments against each other.

By *execution* of specifications we mean not only testing whether a constraint is true for known values of variables (as when checking e.g. assertions), but also computing a missing value so that the given constraint is satisfied. Such constraint solving functionality can also be thought as one way of automating

* This work is supported in part by the European Research Council (ERC) Project *Implicit Programming*.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 1–20, 2013.

the remedial action in case of assertion violation [57]. However, we believe that such constructs should not be treated as a sort of exception mechanism, but as one of the main ways of describing the desired common behavior.

This paper presents our experience in developing techniques to make such constraint solving executable. Our current approach combines deductive synthesis with run-time constraint solving, in both cases leveraging modern SMT solvers. We have built a tool as part of the Leon verification system [9] that incorporates both techniques and allows us to experiment with their trade-offs. A version of the Leon platform is publicly available in source code form for further experiments at `http://lara.epfl.ch/w/leon`. The tool decomposes specifications into simpler fragments using a cost-driven deductive synthesis framework [32,37,40–42]. It compiles as many fragments as possible into conventional functional code; it executes the remaining fragments by invoking a constraint solver at runtime. The solver extends a conventional SMT solver with the ability to handle recursive functions, in a manner similar to our previous systems [38,39]. Using this approach we were able to execute constraints that describe the desired properties of integers, sets, maps and algebraic data types.

In general, the deductive synthesis framework allows us to recursively split challenging problems into tractable subproblems and compile some of the subproblems into conventional code. If a subproblem remains too challenging for synthesis, we keep its declarative specification and execute it using run-time constraint solving. It turns out that in certain interesting cases, the resulting partial program is well-defined for simple, frequent paths and only relies on run-time constraint solving for complex cases.

In the rest of this paper we outline our approach, including the functional language setup, the run-time constraint solving approach, and synthesis techniques. We then illustrate our initial experience with combining run-time constraint solving and synthesis, hinting at some future directions. We finish with a (necessary biased) survey of related work.

2 Examples

We illustrate the benefits of enabling declarative specifications through a series of examples. We show how these examples can be effectively handled by our system. We start by defining a List data-structure with an abstraction function content from list to a set, and an invariant predicate isSorted.

```
abstract class List
case class Cons(head: Int, tail: List) extends List
case object Nil extends List

def content(l: List): Set[Int] = l match {
  case Cons(h, t) ⇒ Set(h) ++ content(t)
  case Nil ⇒ Set()
}

def isSorted(l: List): Boolean = l match {
  case Cons(h1, t1 @ Cons(h2, t2)) ⇒ (h1 ≤ h2) && isSorted(t1)
  case _ ⇒ true
}
```

Thanks to the abstraction function and the invariant, we can concisely specify an insert operation for sorted lists using a constraint:

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  choose{ (x: List) ⇒ isSorted(x) && (content(x) == content(l) ++ Set(v)) }
}
```

Our deductive synthesis procedure is able to translate this constraint into the following complete implementation in under 9 seconds:

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  l match {
    case Cons(head, tail) ⇒
      if (v == head) {
        l
      } elseif (v < head) {
        Cons(v, l)
      } else {
        insert(t, v)
      }
    case Nil ⇒
      Cons(v, Nil)
  }
}
```

However, as the complexity of the constraints increases, the deductive procedure may run short of available time to translate a constraint into complete efficient implementations. As an example, we can currently observe this limitation of our system for a red-black tree benchmark. The following method describes the insertion into a red-black tree.[1]

```
def insert(t: Tree, v: Int) = {
  require(isRedBlack(t))
  choose{ (x: Tree) ⇒ isRedBlack(x) && (content(x) == content(t) ++ Set(v)) }
}
```

Instead of using synthesis (for which this example may present a challenge), we can rely on the run-time constraint solving to execute the constraint. In such scenario, the run-time waits until the argument t and the value v are known, and finds a new tree value x such that the constraint holds. Thanks to our constraint solver, which has a support recursive functions and also leverages the Z3 SMT solver, this approach works well for small red-black trees. It is therefore extremely useful for prototyping and testing and we have previously

[1] We omit here the definition of the tree invariant for brevity, which is rather complex [15, 52], but still rather natural to describe using recursive functions.

explored it as a stand-alone technique for constraint programming in Scala [39]. However, the complexity of reasoning symbolically about complex trees makes this approach inadequate for large concrete inputs.

Fortunately, thanks to the nature of our deductive synthesis framework, we can combine synthesis and run-time constraint solving. We illustrate this using an example of a *red-black tree with a cache*. Such a tree contains a red-black tree, but also redundantly stores one of its elements.

```
case class CTree(cache: Int, data: Tree)
```

The specification of the invariant inv formalizes the desired property: the cache value must be contained in the tree unless the tree is empty.

```
def inv(ct: CTree) = {
  isRedBlack(ct.data) &&
  (ct.cache ∈ content(ct.data)) || (ct.data == Empty)
}
```

The contains operation tests membership in the tree.

```
def contains(ct: CTree, v: Int): Boolean = {
  require(inv(ct))
  choose{ (x: Boolean) ⇒ x == (v ∈ content(ct)) }
}
```

While not being able to fully translate it, the deductive synthesis procedure decomposes the problem and partially synthesizes the constraint. One of its possible results is the following *partial* implementation that combines actual code and a sub-constraint:

```
def contains(ct: CTree, v: Int): Boolean = ct.data match {
  case n: Node ⇒
    if (ct.cache == v) {
      true
    } else {
      choose { (x: Boolean) ⇒ x == (v ∈ content(n)) }
    }
  case Empty ⇒
    false
}
```

We notice that this partial implementation makes use of the cache in accordance with the invariant. The code accurately reflects the fact that the cache may not be trusted if the tree is empty. The remaining constraint is in fact a simpler problem that only relates to standard red-black trees. Our system can then compile the resulting code, where the fast path is compiled as the usual Scala code, and the choose construct is compiled using the run-time solving approach. In the sequel we give details both for our run-time solving approach and the compile-time deductive synthesis transformation framework. We then discuss our very first experience with combining these two approaches.

3 Language

We next present a simple functional language that we use to explore the ability to do verification as well as to execute and compile constraints.

3.1 Implementation Language

As the implementation language we consider a Turing-complete Scala fragment. For the purpose of this paper we assume that programs consist of a set of side-effect-free deterministic mutually recursive functions that manipulate countable data types including integers, n-tuples, algebraic data types, finite sets, and finite maps. We focus on functional code. Our implementation does support localized imperative features; for more details see [9]. We assume that recursive functions are terminating when their specified preconditions are met; our tool applies several techniques to establish termination of recursive functions.

3.2 Function Contracts

Following Scala's contract notation [51], we specify functions in the implementation language using preconditions (**require**) and postconditions (**ensuring**). The declaration

```
def f(x:A) : B = {
  body
} ensuring((res:B) ⇒ post(res))
```

indicates that the result computed by f should satisfy the specification post. Here res⇒post(res) is a lambda expression in which res is a bound variable; the **ensuring** operator binds res to the result of evaluating body. The expression post is itself a general expression in the implementation language, and can invoke recursive functions itself.

3.3 Key Concept: Constraints

Constraints are lambda expressions returning a Boolean value, precisely of the kind used after **ensuring** clauses. To express that a constraint should be solved for a given value, we introduce the construct choose. The expression

```
choose((res:B) ⇒ C(res))
```

should evaluate to a value of type B that satisfies the constraint C. For example, an implicit way to indicate that we expect that the value y is even and to compute y/2 is the following:

```
choose((res:Int) ⇒ res + res == y)
```

The above expression evaluates to y/2 whenever y is even. Note that C typically contains, in addition to the variable res, variables denoting other values in scope (in the above, example, the variable y). We call such variables *parameters* of the constraint.

4 Solving Constraints at Run-Time

We next describe the baseline approach that we use to execute constraints at run-time. This approach is general, as it works for essentially all computable functions on countable domains. On the other hand, it can be inefficient. The subsequent section will describe our synthesis techniques, which can replace such general-purpose constraint solving in a number of cases of interest.

4.1 Model-Generating SMT Solver

The main work horse of our run-time approach is an SMT solver, concretely, Z3 [16]. What is crucial for our application is that Z3 supports model generation: it not only detects unsatisfiable formulas, but in case a formula has a model, can compute and return one model. Other important aspects of Z3 are that it has good performance, supports algebraic data types and arrays [17], supports incremental solving, and has a good API, which we used to build a Scala layer to conveniently access its functionality [38].

4.2 Fair Unfolding of Recursive Functions

Although SMT solvers are very expressive, they do not directly support recursive functions. We therefore developed our own procedure for handling recursive function definitions. Given a deterministic recursive functions f viewed as a relation, assume that f is defined using the fixed point of a higher-order functional H, which implies the formula D:

$$D \quad \equiv \quad \forall x.\ f(x) = H(x, f)$$

The constraints we solve have the form $C \wedge D_k$, where both C and D_k are quantifier-free formulas that we map precisely into the language of an SMT solver.

We use an algorithm for fair unfolding of recursive definitions [69] to reduce the formula $C \wedge D$ to a series of over-approximations and under-approximations. From an execution point of view, such approximations describe all the executions up to certain depth. From a logical perspective, unfolding is a particular form of universal quantifier instantiation which generates a ground consequence D_k of the definition D. If $C \wedge D_k$ is unsatisfiable, so is $C \wedge D$. If $C \wedge D_k$ is satisfiable for the model $x = a$ then we can simply check whether the executable expression evaluates to **true**. In our implementation we have an additional option: we can use the SMT solver itself, to check whether a model of $C \wedge D_k$ depends on the values of partly interpreted functions denoted by f. To this extent, we instrument the logical representation of unfolding up to a given depth using propositional variables that can prevent the execution from depending on uninterpreted values of functions. We call the value of these variables the *control literals* B.

Figure 1 shows the pseudo-code of the resulting algorithm for solving constraints involving recursive functions. It is defined in terms of two subroutines,

```
def solve($C$, $D$) {
  ($C$, $D_0$, $B_0$) = unrollStep($C$, $D$, $\emptyset$, 0)
  k = 0
  while(true) {
    decide($C \land D_k \land \bigwedge_{b \in B_k} b$) match {
      case "SAT" $\Rightarrow$ return "SAT"
      case "UNSAT" $\Rightarrow$ decide($C \land D_k$) match {
        case "UNSAT" $\Rightarrow$ return "UNSAT"
        case "SAT" $\Rightarrow$ ($C$, $D_{k+1}$, $B_{k+1}$) = unrollStep($C$, $D$, $B_k$, k) }}
    k += 1
  }
}
```

Fig. 1. Pseudo-code of the solving algorithm. The decision procedure for the base theory is invoked through the calls to decide.

decide, which invokes the underlying SMT solver, and unrollStep. The fair nature of the unrolling step guarantees that all uninterpreted function values present in the formula are eventually unfolded, if needs be.

The formula without the control literals can be seen as an *over-approximation* of the formula with the semantics of the program, in that it accepts all the same models plus some models in which the interpretation of some invocations is incorrect. The formula with the control literals is an *under-approximation*, in the sense that it accepts only the models that do not rely on the guarded invocations. This explains why the UNSAT answer can be trusted in the first case and the SAT case in the latter.

This algorithm is the basis of the original Leon as a constraint solver and verifier for functional programs [9, 68, 69].

4.3 Executing Choose Construct at Run-Time

During the compilation of programs with choose constructs, we collect a symbolic representation of the constraints used. The actual call to choose is then substituted with a call back to the Leon system indicating both which constraint it refers to, but also propagating the run-time inputs.

During execution, these inputs are converted from concrete JVM objects back to their Leon representations, and finally substituted within the constraint. By construction, the resulting formula's free variables are all output variables.

Given a model for this formula, we translate the Leon representation of output values back to concrete objects which are then returned.

4.4 Evaluation

We evaluated the performance of the run-time constraint solving algorithm on two data structures with invariants: sorted lists and red-black trees. For each data structure, we implemented a declarative version of the *add* and *remove* operations. Thanks to the abstraction and predicate functions, the specifications of both operations are very concise and self-explanatory. We illustrate this by providing the corresponding code for red-black trees:

```
def add(t: Tree, e: Int): Tree = choose {
  (res: Tree) ⇒ content(res) == content(t) ++ Set(e) && isRedBlackTree(res)
}

def remove(t: Tree, e: Int): Tree = choose {
  (res: Tree) ⇒ content(res) == content(t) -- Set(e) && isRedBlackTree(res)
}
```

Solving is relatively efficient for small data structures: it finds models in under 400ms for lists and trees up to size 4. However, the necessary solving time increases exponentially with the size and goes as high as 35 seconds for synthesizing insertion into a red-black tree of size 10.

5 Synthesizing Functional Code from Constraints

In this section, we give an overview of our framework for deductive synthesis. The goal of the approach is to derive correct programs by successive steps. Each step is validated independently, and the framework ensures that composing steps results in global correctness.

5.1 Synthesis Problems and Solutions

A synthesis problem, or constraint, is fundamentally a relation between inputs and outputs. We represent this, together with contextual information, as a quadruple

$$\llbracket \bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x} \rrbracket$$

where:

- $\bar{a}$ denotes the set of *input variables*,
- $\bar{x}$ denotes the set of *output variables*,
- ϕ is the *synthesis predicate*, and
- Π is the *path condition* to the synthesis problem.

The free variables of ϕ must be a subset of $\bar{a} \cup \bar{x}$. The path condition denotes a property that holds for input at the program point where synthesis is to be performed, and the free variables of Π should therefore be a subset of $\bar{a}$.

As an example, consider the following call to choose:

```
def f(a : Int) : Int = {
  if(a ≥ 0) {
    choose((x : Int) ⇒ x ≥ 0 && a + x ≤ 5)
  } else ...
}
```

The representation of the corresponding synthesis problem is:

$$[\![a \ \langle a \geq 0 \rhd x \geq 0 \wedge a + x \leq 5\rangle \ x]\!] \tag{1}$$

We represent a solution to a synthesis problem as a pair $\langle P|\bar{T}\rangle$ where:

- P is the *precondition*, and
- $\bar{T}$ is the *program term.*

The free variables of both P and $\bar{T}$ must range over $\bar{a}$. The intuition is that, whenever the path condition and the precondition are satisfied, evaluating $\phi[\bar{x} \mapsto \bar{T}]$ should evaluate to $\top$ (true), i.e. $\bar{T}$ are realizers for a solution to $\bar{x}$ in ϕ given the inputs $\bar{a}$. Furthermore, for a solution to be as general as possible, the precondition must be as weak as possible.

Formally, for such a pair to be a solution to a synthesis problem, denoted as

$$[\![\bar{a} \ \langle \Pi \rhd \phi\rangle \ \bar{x}]\!] \vdash \langle P|\bar{T}\rangle$$

the following two properties must hold:

- *Relation refinement:* $\Pi \wedge P \models \phi[\bar{x} \mapsto \bar{T}]$
  This property states that whenever the path- and precondition hold, the program $\bar{T}$ can be used to generate values for the output variables $\bar{x}$ such that the predicate ϕ is satisfied.
- *Domain preservation:* $\Pi \wedge (\exists \bar{x} : \ \phi) \models P$
  This property states that the precondition P cannot exclude inputs for which an output exists.

As an example, a valid solution to the synthesis problem (1) is given by: $\langle a \leq 5|0\rangle$. The precondition $a \leq 5$ characterizes exactly the input values for which a solution exists, and for all such values, the constant 0 is a valid solution term for x. The solution is in general not unique; alternative solutions for this particular problem include $\langle a \leq 5|5 - a\rangle$, or $\langle a \leq 5|\mathsf{if}(a < 5) \ a + 1 \ \mathsf{else} \ 0\rangle$.

5.2 Inference Rules

The correctness conditions described above characterize the validity of solutions to synthesis problems. We now show how to derive such solutions. We present our techniques as a set of *inference rules*. As a simple first example, consider the following rule:

$$\frac{[\![\bar{a} \ \langle \Pi \rhd \phi[x_0 \mapsto t]\rangle \ \bar{x}]\!] \vdash \langle P|\bar{T}\rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a} \ \langle \Pi \rhd x_0 = t \wedge \phi\rangle \ x_0 , \bar{x}]\!] \vdash \langle P|\mathsf{val} \ \bar{x} := \bar{T}; \ (t , \bar{x})\rangle}$$

As is usual with inference rules, on top are the premisses and below is the goal. This particular rule captures the intuition that, whenever a term of the form $x_0 = t$ appears as a top-level conjunct in a synthesis problem, the problem can be simplified by assigning to the output variable x_0 the term t. The rule specifies both how the subproblem relates to the original one, and how its solution and precondition are used in the final program.

Another example is the following rule for decomposing disjunctions:

$$\frac{[\![\bar{a}\ \langle \Pi \rhd \phi_1\rangle\ \bar{x}]\!] \vdash \langle P_1|\bar{T}_1\rangle \qquad [\![\bar{a}\ \langle \Pi \land \neg P_1 \rhd \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_2|\bar{T}_2\rangle}{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \lor \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_1 \lor P_2|\mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\}\rangle}$$

Here, the rule states that a disjunction can be handled by considering each disjunct in isolation, and combining the solutions as an if-then-else expression, where the branching condition is the precondition for the first problem. Note that in the second subproblem, we have added the literal $\neg P_1$ to the path condition. This reflects the knowledge than, in the final program, the subprogram for the second disjunct only executes if the first one cannot compute a solution.

In general, a synthesis problem is solved whenever a derivation can be found for which all output variables are assigned to a program term.

For certain well-defined classes of synthesis problems, we can design sets of inference rules which, together with a systematic application strategy, are guaranteed to result in successful derivation. We have shown in previous work such complete strategies for integer linear arithmetic, rational arithmetic, or term algebras [32, 40, 67]. We call these *synthesis procedures*, analogously to decision procedures.

As a final example, we now show how our framework can express solutions that take the form of recursive functions traversing data structures. The next rule captures one particular yet very common form of such a traversal for Lists.

$$\frac{\begin{array}{c}(\Pi_1 \land P) \implies \Pi_2 \qquad \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t] \\ [\![\bar{a}\ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}]\rangle\ \bar{x}]\!] \vdash \langle \top|\bar{T}_1\rangle \\ [\![\bar{r}\,, h\,, t\,, \bar{a}\ \langle \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \land \phi[a_0 \mapsto t, \bar{x} \mapsto \bar{r}] \rhd \phi[a_0 \mapsto \mathsf{Cons}(h,t)]\rangle\ \bar{x}]\!] \vdash \langle \top|\bar{T}_2\rangle\end{array}}{[\![a_0\,, \bar{a}\ \langle \Pi_1 \rhd \phi\rangle\ \bar{x}]\!] \vdash \langle P|\mathsf{rec}(a_0,\bar{a})\rangle}$$

The goal of the rule is to derive a solution consisting of a single invocation of a (fresh) recursive function rec, of the following form:

```
def rec($a_0$, $\bar{a}$) = {
  require($\Pi_2$)
  $a_0$ match {
    case Nil ⇒ $\bar{T}_1$
    case Cons($h$, $t$) ⇒
      val $\bar{r}$ = rec($t$, $\bar{a}$)
      $\bar{T}_2$
  }
} ensuring($\bar{r} \Rightarrow \phi[\bar{x} \mapsto \bar{r}]$)
```

The rule decomposes the problem into two cases, corresponding to the alternatives in the data type, and assumes that the solution takes the form of a *fold* function, fixing the recursive call.

6 Combining Synthesis and Runtime Constraint Solving

The deductive synthesis framework allows us to split a challenging problem into tractable sub problems. In the case where the subproblems remain too challenging, we keep their corresponding declarative specifications and compile them into the run-time invocation of the constraint. This result in a partially implemented program. In certain cases, the partial program is well-defined for simple, frequent paths, and only falls back to run-time solving for complex cases.

As an illustrative example, we give here the partial derivation of the function contains on CTrees from Section 2 using rules such as the ones described in Section 5.

We start with the synthesis problem:

$$[\![\mathsf{c},\mathsf{d},\mathsf{v}\ \ \langle \mathsf{inv}(\mathsf{CTree}(\mathsf{c},\mathsf{d})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c},\mathsf{d}))\rangle\ \ \mathsf{x}]\!]$$

A first step is to perform case analysis on d, the tree. This generates two subproblems, for the cases Empty and Node respectively. For Empty, we have:

$$[\![\mathsf{c},\mathsf{v}\ \ \langle \mathsf{inv}(\mathsf{CTree}(\mathsf{c},\mathsf{Empty})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c},\mathsf{Empty}))\rangle\ \ \mathsf{x}]\!]$$

At this point, inv(CTree(c, Empty)) simplifies to $\top$ and content(CTree(c, Empty)) simplifies to $\emptyset$. The problem thus becomes:

$$[\![\mathsf{c},\mathsf{v}\ \ \langle \top \rhd \mathsf{x} \iff \mathsf{v} \in \emptyset\rangle\ \ \mathsf{x}]\!]$$

which is solved by $\langle \top | \mathsf{false}\rangle$. For the Node branch, we have:

$$[\![\mathsf{c},\mathsf{n},\mathsf{v}\ \ \langle \mathsf{n} \neq \mathsf{Empty} \wedge \mathsf{inv}(\mathsf{CTree}(\mathsf{c},\mathsf{n})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c},\mathsf{n}))\rangle\ \ \mathsf{x}]\!]$$

This is almost the original problem, with the additional contextual information that the tree is not empty. Given that we have two integer variables in scope, c and v (the cache and the value for which we are checking inclusion), a potential tactic is to perform case analysis on their equality. This yields two subproblems. For the equal case, we have one fewer variable:

$$[\![\mathsf{n},\mathsf{v}\ \ \langle \mathsf{n} \neq \mathsf{Empty} \wedge \mathsf{inv}(\mathsf{CTree}(\mathsf{v},\mathsf{n})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{v},\mathsf{n}))\rangle\ \ \mathsf{x}]\!]$$

At this point, because inv(CTree(v,n)) implies that v∈CTree(v,n), the problem is solved with $\langle \top | \mathsf{true}\rangle$.

For the final subproblem, where d is a Node and the cache does not hold the value v, our system is currently not efficient enough to derive a solution. Therefore, it falls back to emitting a run-time invocation of choose. Combining the solutions for all subproblems, we obtain the partially synthesized function contains as shown in Section 2.

6.1 Discussion

It is our hope that the combination of two technologies, run-time constraint solving and synthesis, can make execution of specifications practical. It will be then interesting to understand to what extend such complete specifications change the software development process. Writing data structures using constraints shows the productivity advantages of using constraints, because data structure invariants are reusable across all operations, whereas the remaining specification of each individual operation becomes extremely concise. The development process thus approaches the description of a data structure design from a textbook [15], which starts from basic invariants and uses them as guiding principle when presenting the implementation of data structure operations. We are thus hopeful that, among other results, we can soon enable automated generation of efficient unbounded data structures from high-level descriptions, analogously to recent breakthrough on cache coherence protocol generation [71].

An exciting future direction is to use run-time property verification techniques to efficiently combine code generated through speculative synthesis and constraint solving. In many synthesis approaches, due to a heavy use of example-driven techniques and the limitations of static verification, it is also quite possible that the automatically generated implementation is incorrect for some of the inputs. In such cases, techniques of run-time verification can be used to detect, with little overhead, the violation of specifications for given inputs. As a result, synthesis could be used to generate fast paths and likely code fragments, while ensuring the overall adherence to specification at run time.

In general, we are excited about future interplay between dynamic and static approaches to make specifications executable, which is related to partial evaluation and to techniques for compilation of declarative programming languages, as well as static optimization of run-time checks.

7 Related Work

We provide an overview of related work on executing constraints using general-purpose solvers at run-time, synthesizing constraints into conventional functional and imperative code, and combining these two extreme approaches by staging the computation between run-time and compile time.

7.1 Run-Time Constraint Solving

This proceedings volume contains a notable approach and tool Boogaloo [56], which enables execution of a rich intermediate language Boogie [43]. The original purpose of Boogie is static verification [6]. The usual methods to verify Boogie programs generate conservative *sufficient* verification, which become unprovable if the invariants are not inductive. Tools such as Boogaloo complement verification-condition generation approaches and help developer distinguish errors in programs or specifications that would manifest at run-time from those errors that come from inductive statements not being strong enough. A run-time interpreter for the annotations of the Jahob verification system [79] can also execute certain limited form of specifications, but does not use symbolic execution techniques and treats quantifiers more conservatively than the approach of Boogaloo. Our current Leon system works with a quantifier-free language; the developers write specifications using recursion instead of quantifiers. Our system allows developers to omit postconditions of defined functions and does not reporting spurious counterexamples. Therefore, it provides the users the advantages of both sound static verification and true counterexamples in a unified algorithm. As remarked, however, unfolding recursive functions can be viewed as a particular quantifier instantiation strategy.

Constraint solving is key for executing programs annotated with contracts because it enables the generation of concrete states that satisfy a given precondition. In our tool we use our approach of satisfiability modulo recursive (pure) functions. In a prior work we have focused on constraint programming using such system [39], embedding constraint programming with recursive functions and SMT solvers into the full Scala language and enabling ranked enumeration of solutions. The Boogaloo approach [56] uses symbolic execution where quantifiers are treated through a process that generalizes deterministic function unrolling to more general declarative constraints. Unfolding is also used in bounded model checking [7] and k-induction approaches [35]. Symbolic execution can also be performed at the level of bytecodes, as in the UDITA system that builds on Java Pathfinder and contains specialized techniques for generating non-isomorphic graph structures [26].

Functional logic programming [3] amalgamates the functional programming and logic programming paradigms into a single language. Functional logic languages, such as Curry [47] benefit from efficient demand-driven term reduction strategies proper to functional languages, as well as non-deterministic operations of logic languages, by using a technique called *narrowing*, a combination of term reduction and variable instantiation. Instantiation of unbound logic variables occur in constructive guessing steps, only to sustain computation when a reduction needs their values. The performance of non-deterministic computations depends on the evaluation strategy, which are formalized using definitional trees [2]. Applications using functional logic languages include programming of graphical and web user interfaces [28,29] as well as providing high-level APIs for accessing and manipulating databases [11]. The Oz language and the associated Mozart Programming System is another admirable combination of multiple paradigms [72], with applications in functional, concurrent, and logic programming. In particular, Oz supports a form of logical variables, and logic programming is enabled through unification. One limitation is that one cannot perform arithmetic operations with logical variables (which we have demonstrated in several of our examples), because unification only applies to constructor terms.

Monadic constraint programming [58] integrates constraint programming into purely functional languages by using monads to define solvers. The authors define monadic search trees, corresponding to a base search, that can be transformed by the use of *search transformers* in a composable fashion to increase performance. The Dminor language [8] introduces the idea of using an SMT solver to check subtyping relations between refinement types; in Dminor, all types are defined as logical predicates, and subtyping thus consists of proving an implication between two such predicates. The authors show that an impressive number of common types (including for instance algebraic data types) can be encoded using this formalism. In this context, generating values satisfying a predicate is framed as the *type inhabitation* problem, and the authors introduce the expression elementof$T$ to that end. It is evaluated by invoking Z3 at run-time and is thus conceptually comparable to our find construct but without support for recursive function unfolding. We have previously found that recursive function unfolding works better as a mechanism for satisfiability checking than using quantified axiomatization of recursive functions [69]. In general, we believe that our examples are substantially more complex than the experiences with elementof in the context of Dminor.

The ScalaZ3 library [38], used in Leon, integrates invocations to Z3 into a programming language. Because it is implemented purely as a library, we were then not able to integrate user-defined recursive functions and data types into constraints, so the main application is to provide an embedded domain-specific language to access the constraint language of Z3 (but not to extend it). A similar approach has been taken by others to invoke the Yices SMT solver [20] from Haskell.[2]

7.2 Synthesis of Functions

Our approach blends deductive synthesis [45,46,61], which incorporates transformation of specifications, inductive reasoning, recursion schemes and termination checking, with modern SMT techniques and constraint solving for executable constraints. As one of our subroutines we include complete functional synthesis for integer linear arithmetic [42] and extend it with a first implementation of complete functional synthesis for algebraic data types [32,67]. This gives us building blocks for synthesis of recursion-free code. To synthesize recursive code we build on and further advance the counterexample-guided approach to synthesis [64].

Deductive synthesis frameworks. Early work on synthesis [45, 46] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle.

Programming by refinement has been popularized as a manual activity [5,76]. Interactive tools have been developed to support such techniques in HOL [13]. A recent example of deductive synthesis and refinement is the Specware system from Kesterel [61]. We were not able to use the system first-hand due to its availability policy, but it appears to favor expressive power and control, whereas we favor automation.

A combination of automated and interactive development is analogous to the use of automation in interactive theorem provers, such as Isabelle [50]. However, whereas in verification it is typically the case that the program is available, the emphasis here is on constructing the program itself, starting from specifications.

Work on synthesis from specifications [65] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. The work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures, but appears to require more detailed information about the structure of the expected solution than our approach.

[2] `http://hackage.haskell.org/package/yices-easy`

Synthesis with input/output examples. One of the first works that addressed synthesis with examples and put inductive synthesis on a firm theoretical foundation is the one by Summers [66]. Subsequent work presents extensions of the classical approach to induction of functional Lisp-programs [**?**, 30]. These extensions include synthesizing a set of equations (instead of just one), multiple recursive calls and systematic introduction of parameters. Our current system lifts several restrictions of previous approaches by supproting reasoning about arbitrary datatypes, supporting multiple parameters in concrete and symbolic I/O examples, and allowing nested recursive calls and user-defined declarations.

Inductive programming and programming by demonstration. Inductive (logic) programming that explores automatic synthesis of (usually recursive) programs from incomplete specifications, most often being input/output examples [24,49], influenced our work. Recent work in the area of programming by demonstration has shown that synthesis from examples can be effective in a variety of domains, such as spreadsheets [60]. Advances in the field of SAT and SMT solvers inspired counter-example guided iterative synthesis [27, 64], which can derive input and output examples from specifications. Our tool uses and advances these techniques through two new counterexample-guided synthesis approaches.

Synthesis based on finitization techniques. Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [62–64]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. The tool we presented shows one way to move the ideas of sketching towards infinite domains. In this generalization we leverage reasoning about equations as much as SAT techniques.

Reactive synthesis. Synthesis of reactive systems generates programs that run forever and interact with the environment. However, known complete algorithms for reactive synthesis work with finite-state systems [55] or timed systems [4]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [73]. These techniques usually take specifications in a fragment of temporal logic [54] and have resulted in tools that can synthesize useful hardware components [33, 34]. Recently such synthesis techniques have been extended to repair that preserves good behaviors [23], which is related to our notion of partial programs that have remaining choose statements.

Automated inference of program fixes and contracts. These areas share the common goal of inferring code and rely on specialized software synthesis techniques [53, 74, 75]. Inferred software fixes and contracts are usually snippets of code that are synthesized according to the information gathered about the analyzed program. The core of these techniques lies in the characterization of runtime behavior that is used to guide the generation of fixes and contracts. Such characterization is done by analyzing program state across the execution of tests; state can be defined using user-defined query operations [74,75], and additional expressions extracted from the code [53]. Generation of program fixes and contracts is done using heuristically guided injection of (sequences of) routine calls into predefined code templates.

Our synthesis approach works with purely functional programs and does not depend on characterization of program behavior. It is more general in the sense that it focuses on synthesizing whole correct functions from scratch and does not depend on already existing code. Moreover, rather than using execution of tests to define starting points for synthesis and SMT solvers just to guide the search, our approach utilizes SMT solvers to guarantee correctness of generated programs and uses execution of tests to speedup the search. Coupling of flexible program generators and the Leon verifier provides more expressive power of the synthesis than filling of predefined code schemas.

7.3 Combining Run-Time and Compile-Time Approaches

We have argued that constraint solving generalizes run-time checking, and allows the underlying techniques to be applied in more scenarios than providing additional redundancy. The case of optimizing run-time checks also points out that there is a large potential for speedups in executing specifications: in the limit, a statically proved assertion can be eliminated, so its execution cost goes from traversing a significant portion of program state to zero. As is in general the case for compilation, such static pre-computation can be viewed as partial evaluation, and has been successfully applied for temporal finite-state properties [10].

Compilation and transformation of logic programs. Compilation of logic programs is important starting point for improving the baseline of compiled code. A potential inefficiency in the current approach (though still only a polynomial factor) is that the constraint solver and the underlying programming language use a different representation of values, so values need to be converted at the boundary of constraints and standard functional code. Techniques such as those employed in Warren's Abstract Machine (WAM) is relevant in this context [1]. Deeper optimizations and potentially exponential speedups can be obtained using tabling [14], program transformation [59] and partial evaluation [12, 25].

Specifications as a fallback to imperative code. The idea to use specifications as a fall-back mechanism for imperative code was adopted in [57]. Dynamic contract checking is applied and, upon violations, specifications can be *executed*. The technique ignores the erroneous state and computes output values for methods given concrete input values and the method contract. The implementation uses a relational logic similar to Alloy [31] for specifications, and deploys the Kodkod model finder [70]. A related tight integration between Java and the Kodkod engine is presented in [48]. We expect that automated synthesis will allow the developers to use specifications alone in such scenarios, with a candidate implementation generated automatically.

Data structure repair. It is worth mentioning that this proceedings also contains new results [78] in the exciting area of *data structure repair*. This general approach is related to solving constraints at run-time. The assumption in data structure repair is that, even if a given data structure does not satisfy the desired property, it may provide a strong hint at the desired data structure. Therefore, it is reasonable to use the current data structure as the starting point for finding the value that satisfies the desired constraint, hoping that the correct data structure is close to the current one. Although such approach is slightly more natural in the context of imperative than functional code, it is relevant whenever the data manipulated is large enough. The first specification-driven approach for data structure repair is by Demsky and Rinard [18, 19] where the goal is to recover from corrupted data structures by transforming states that are erroneous with respect to integrity constraints into valid ones, performing local heuristic search. Subsequent work uses less custom constraint solvers instead [21, 22]. We believe that SMT solvers could also play a role in this domain. Researchers [77] have also used method contracts instead of data structure integrity constraints to be able to support rich behavioral specifications, which makes it also more relevant for our scenarios. While the primary goal in most works is run-time recovery of data structures, recent work [44] extends the technique for debugging purposes, by abstracting concrete repair actions to program statements performing the same actions. As in general for run-time constraint solving, we expect that data structure repair can be productively applied to implementations generated using "speculative synthesis" that generates a not necessarily correct implementation.

References

1. Ait-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
2. Antoy, S.: Definitional trees. In: ALP, pp. 143–157 (1992)
3. Antoy, S., Hanus, M.: Functional logic programming. CACM 53(4), 74–85 (2010)
4. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Hybrid Systems II, pp. 1–20 (1995)
5. Back, R.-J., von Wright, J.: In: Refinement Calculus, Springer (1998)
6. 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)
7. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58 (2003)
8. Bierman, G.M., Gordon, A.D., Hritcu, C., Langworthy, D.E.: Semantic subtyping with an SMT solver. In: ICFP, pp. 105–116 (2010)
9. Blanc, R.W., Kneuss, E., Kuncak, V., Suter, P.: An overview of the Leon verification system: Verification by translation to recursive functions. In: Scala Workshop (2013)
10. Bodden, E., Lam, P., Hendren, L.J.: Partially evaluating finite-state runtime monitors ahead of time. ACM Trans. Program. Lang. Syst. 34(2), 7 (2012)
11. Braßel, B., Hanus, M., Müller, M.: High-level database programming in Curry. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 316–332. Springer, Heidelberg (2008)
12. Bruynooghe, M., Schreye, D.D., Krekels, B.: Compiling control. The Journal of Logic Programming 6(12), 135–162 (1989)
13. Butler, M., Grundy, J., Langbacka, T., Ruksenas, R., von Wright, J.: The refinement calculator: Proof support for program refinement. In: Formal Methods Pacific (1997)

14. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM 43(1), 20–74 (1996)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
16. 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)
17. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer-Aided Design (November 2009)
18. Demsky, B., Rinard, M.C.: Automatic detection and repair of errors in data structures. In: OOPSLA, pp. 78–95 (2003)
19. Demsky, B., Rinard, M.C.: Data structure repair using goal-directed reasoning. In: ICSE, pp. 176–185 (2005)
20. Dutertre, B., de Moura, L.: The Yices SMT solver (2006), `http://yices.csl.sri.com/tool-paper.pdf`
21. Elkarablieh, B., Khurshid, S.: Juzi: a tool for repairing complex data structures. In: ICSE, pp. 855–858 (2008)
22. Elkarablieh, B., Khurshid, S., Vu, D., McKinley, K.S.: Starc: static analysis for efficient repair of complex data. In: OOPSLA, pp. 387–404 (2007)
23. von Essen, C., Jobstmann, B.: Program repair without regret. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 896–911. Springer, Heidelberg (2013)
24. Flener, P., Partridge, D.: Inductive programming. Autom. Softw. Eng. 8(2), 131–137 (2001)
25. Gallagher, J., Peralta, J.: Regular tree languages as an abstract domain in program specialisation. Higher-Order and Symbolic Computation 14(2-3), 143–172 (2001)
26. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: International Conference on Software Engineering (ICSE) (2010)
27. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI, pp. 62–73 (2011)
28. Hanus, M.: Type-oriented construction of web user interfaces. In: PPDP, pp. 27–38 (2006)
29. Hanus, M., Kluß, C.: Declarative programming of user interfaces. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 16–30. Springer, Heidelberg (2008)
30. Hofmann, M.: IgorII - an analytical inductive functional programming system (tool demo). In: PEPM, pp. 29–32 (2010)
31. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)
32. Jacobs, S., Kuncak, V., Suter, P.: Reductions for synthesis procedures. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 88–107. Springer, Heidelberg (2013)
33. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD (2006)
34. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
35. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. In: 10th Int. Workshop Parallel and Distributed Methods in verifiCation, PDMC 2011 (2011)
36. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. JMLR 7, 429–454 (2006)
37. Kneuss, E., Kuncak, V., Kuraj, I., Suter, P.: On integrating deductive synthesis and verification systems. Technical Report EPFL-REPORT-186043, EPFL (2013)

38. Köksal, A.S., Kuncak, V., Suter, P.: Scala to the power of Z3: Integrating SMT and programming. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 400–406. Springer, Heidelberg (2011)
39. Köksal, A., Kuncak, V., Suter, P.: Constraints as control. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL (2012)
40. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: ACM SIGPLAN Conf. Programming Language Design and Implementation, PLDI (2010)
41. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. Software Tools for Technology Transfer (STTT), TBD (TBD) (2012)
42. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. Communications of the ACM (2012)
43. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178, working draft (June 24, 2008)
44. Malik, M.Z., Siddiqui, J.H., Khurshid, S.: Constraint-based program debugging using data structure repair. ICST, 190–199 (2011)
45. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. 2(1), 90–121 (1980)
46. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM 14(3), 151–165 (1971)
47. Michael Hanus, E.: Curry: An integrated functional logic language vers. 0.8.2 (2006), http://www.curry-language.org
48. Milicevic, A., Rayside, D., Yessenov, K., Jackson, D.: Unifying execution of imperative and declarative code. In: ICSE, pp. 511–520 (2011)
49. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. J. Log. Program. 19(20), 629–679 (1994)
50. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
51. Odersky, M.: Contracts for Scala. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 51–57. Springer, Heidelberg (2010)
52. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
53. Pei, Y., Wei, Y., Furia, C.A., Nordio, M., Meyer, B.: Evidence-based automated program fixing. CoRR, abs/1102.1059 (2011)
54. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
55. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 179–190. ACM, New York (1989)
56. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before. In: Int. Conf. Runtime Verification (2013)
57. Samimi, H., Aung, E.D., Millstein, T.: Falling back on executable specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
58. Schrijvers, T., Stuckey, P.J., Wadler, P.: Monadic constraint programming. J. Funct. Program. 19(6), 663–697 (2009)
59. Senni, V., Fioravanti, F.: Generation of test data structures using constraint logic programming. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 115–131. Springer, Heidelberg (2012)

60. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 634–651. Springer, Heidelberg (2012)
61. Smith, D.R.: Generating programs plus proofs by refinement. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 182–188. Springer, Heidelberg (2008)
62. Solar-Lezama, A., Arnold, G., Tancau, L., Bodík, R., Saraswat, V.A., Seshia, S.A.: Sketching stencils. In: PLDI, pp. 167–178 (2007)
63. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI (2008)
64. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
65. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. In: POPL (2010)
66. Summers, P.D.: A methodology for LISP program construction from examples. JACM 24(1), 161–175 (1977)
67. Suter, P.: Programming with Specifications. PhD thesis, EPFL (December 2012)
68. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: ACM POPL (2010)
69. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011)
70. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
71. Udupa, A., Raghavan, A., Deshmukh, J., Mador-Haim, S., Martin, M., Alur, R.: Transit: Specifying protocols with concolic snippets. In: ACM Conference on Programming Language Design and Implementation (2013)
72. Van Roy, P.: Logic programming in Oz with Mozart. In: ICLP (1999)
73. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 139–154. Springer, Heidelberg (2009)
74. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: ICSE (2011)
75. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 61–72 (2010)
76. Wirth, N.: Program development by stepwise refinement. Commun. ACM 26(1), 70–74 (1983) (reprint)
77. Nokhbeh Zaeem, R., Khurshid, S.: Contract-based data structure repair using alloy. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 577–598. Springer, Heidelberg (2010)
78. Zaeem, R.N., Malik, M.Z., Khurshid, S.: Repair abstractions for more efficient data structure repair. In: Int. Conf. Runtime Verification (2013)
79. Zee, K., Kuncak, V., Taylor, M., Rinard, M.: Runtime checking for program verification. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 202–213. Springer, Heidelberg (2007)

Informative Types and Effects for Hybrid Migration Control

Ana Almeida Matos and Jan Cederquist

Instituto de Telecomunicações (SQIG) and Instituto Superior Técnico,
Lisbon, Portugal
{ana.matos,jan.cederquist}@ist.utl.pt

Abstract Flow policy confinement is a property of programs whose declassifications respect the allowed flow policy of the context in which they execute. In a distributed setting where computation domains enforce different allowed flow policies, code migration between domains implies dynamic changes to the relevant allowed policy. Furthermore, when programs consist of more than one thread running concurrently, the same program might need to comply to more than one allowed flow policy simultaneously. In this scenario, confinement can be enforced as a migration control mechanism. In the present work we compare three type-based enforcement mechanisms for confinement, regarding precision and efficiency of the analysis. In particular, we propose an efficient hybrid mechanism based on statically annotating programs with the declassification effect of migrating code. This is done by means of an informative type and effect pre-processing of the program, and is used for supporting runtime decisions.

1 Introduction

Research in language based security has placed a lot of attention on the study of information flow properties and enforcement mechanisms [1]. Information flow security regards the control of how dependencies between information of different security levels can lead to information leakage during program execution. Information flow properties range in strictness from pure absence of information leaks, classically known as non-interference [2], to more flexible properties that allow for *declassification* to take place in a controlled manner [3].

Separating the problems of enabling and of controlling flexible information flow policies paves the way to a modular composition of security properties that can be studied independently. Here we consider a distributed setting with runtime remote thread creation, and the problem of ensuring that declassifications that are performed by mobile code comply to the flow policy that is allowed at the computation domain where they are performed. We refer to this property as *flow policy confinement*, and treat it as a migration control problem [4,5].

An illustrative scenario could be that of a set of personal mobile appliances, such as smartphones. Due to their inter-connectivity (web, Bluetooth), they form networks of highly responsive computing devices with relatively limited

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 21–39, 2013.

resources, and that handle sensitive information (personal location, contacts, passwords). This combination demands for scalable and efficient mechanisms for ensuring privacy in a distributed setting with code mobility. From an abstract perspective, each device forms a *computation domain* with specific capabilities and restrictions, and in particular information flow policies for protecting data and other computing *threads* that are running concurrently in the same domain. We refer to these policies as the *allowed flow policy* of the domain. Flow policy confinement ensures that domains do not execute code that might perform declassifications that break their own allowed policies.

Let us consider, for example, an application for supporting two users (Alice and Bob) in choosing the best meeting point and path for reaching each other by means of public transportation. In order to produce advice that takes into account the current context (recent user locations, traffic conditions, weather) threads containing code for building updated travel maps are downloaded by Alice and Bob during runtime (their travel). The recommended path and meeting point can be improved by deducing the users' personal preferences from data that it collects from the mobile devices (e.g. content of stored images, file types). Users might, however, have privacy restrictions regarding that data, in the form of allowed flow policies that the downloaded threads must comply to. The following naive program creates a thread for gathering data that helps select the meeting point. Since the meeting point will necessarily be revealed to Bob, this part of the program should only allowed to run if it respects which private information Alice accepts to leak to Bob.

```
newthread {                      // Creates thread at Alice's device
   ref zoo=0; ref bookstore=0;   // to choose between zoo or bookstore
   allowed                       // If allowed by Alice's policy
      (L_IMGS < L_BOB /\         // to leak image contents
       L_FILES < L_BOB)          // and file types to Bob
      flow (L_IMGS < L_BOB /\    //   Declares a declassification
            L_FILES < L_BOB)     //   with same policy
         processImgs(zoo);       //     weighs images with animals
         searchFiles(bookstore); //     weighs e-book files
         if (zoo > bookstore)    //     inspects sensitive data...
            meetAt(ZOO);         //       and influences meeting point
            meetAt(BOOKSTORE);
   meetAt(random);               // If not allowed, uses other criteria
} at D_ALICE
```

As the above code is deployed, device `D_ALICE` must decide whether it is safe to execute the thread or not. Clearly, the decision must be taken quickly so as to not disrupt the purpose of the application. Ultimately, it is based on an analysis of the code, giving special attention to the points where declassifications occur.

This paper addresses the technical problem of how to build suitable enforcement mechanisms that enable domains to check incoming code against their own allowed flow policies. Previous work [6] introduces, as a proof-of-concept, a runtime migration control mechanism for enforcing confinement that lacks precision and efficiency. In this paper we look closely at the problem of the overhead that is implied by using types and effects for checking programs during runtime.

We study three type and effect-based mechanisms for enforcing confinement that place different weight over static and run time: First, we present a purely static type and effect system. Second, we increase its precision by letting most of the control be done dynamically, at the level of the operational semantics. To this end, the migration instruction is conditioned by a type check by means of a standard type and effect checking system. Third, we provide a mechanism for removing the runtime weight of typing migrating programs. It consists in statically annotating programs with information about the declassifying behavior of migrating threads, in the form of a *declassification effect*, and using it to support efficient runtime checks.

This work is formulated over an expressive distributed higher-order imperative lambda-calculus with remote thread creation. This language feature implies that programs might need to comply to more than one dynamically changing allowed flow policy simultaneously. The main contributions are:

1. A purely static type and effect system for enforcing flow policy confinement.
2. A type and effect system for checking migrating threads at runtime, that is more precise than the one in point 1.
3. A static-time informative pre-processing type and effect system for annotating programs with a declassification effect, for a more efficient and precise mechanism than the one in point 2.

We start by presenting the security setting (Section 2) and language (Section 3). The formal security property of Flow Policy Confinement (Section4) follows. Then, we study three type and effect-based enforcement mechanisms (Section 5) and draw conclusions regarding their efficiency and precision. Finally we discuss related work (Section 6) and conclude (Section 7). An extended version of this article (available from the authors) presents the detailed proofs.

2 Security Setting

The study of confidentiality traditionally relies on a lattice of security levels [7], corresponding to security clearances, that is associated to information containers in the programming language. The idea is that information pertaining to references labeled with l_2 can be legally transferred to references labeled with l_1 only if l_1 is at least as confidential as l_2. In this paper we do not deal explicitly with security levels, but instead with *flow policies* that define how information should be allowed to flow between security levels. Formally, flow policies can be seen as downward closure operators over a basic lattice of security levels [8].

Flow policies $A, F \in \boldsymbol{\mathit{Flo}}$ can be ordered according to their permissiveness by means of a *permissiveness relation* $\preccurlyeq$, where $F_1 \preccurlyeq F_2$ means that F_1 is at least as permissive as F_2. We assume that flow policies form a lattice that supports a pseudo-subtraction operation $\langle \boldsymbol{\mathit{Flo}}, \preccurlyeq, \curlywedge, \curlyvee, \mho, \Omega, \smile \rangle$, where: the meet operation $\curlywedge$ gives, for any two flow policies F_1, F_2, the strictest policy that allows for both F_1 and F_2; the join operation $\curlyvee$ gives, for any two flow policies F_1, F_2, the most permissive policy that only allows what both F_1 and F_2 allow; the most restrictive flow policy $\mho$ does not allow any information flows; and the most permissive flow policy Ω that allows all information flows. Finally, the pseudo-subtraction operation $\smile$ between two flow policies F_1 and F_2 (used only in Subsection 5.3) represents the most permissive policy that allows everything that is allowed by the first (F_1), while excluding all that is allowed by the second (F_2); it is defined as the relative pseudo-complement of F_2 with respect to F_1, i.e. the greatest F such that $F \curlywedge F_2 \preccurlyeq F_1$.

Considering a concrete example of a lattice of flow polices that meets the abstract requirements defined above can provide helpful intuitions. Flow policies that operate over the security lattice where security levels are sets of principals $p, q \in \boldsymbol{Pri}$ provide such a case. In this setting, security levels are ordered by means of the flow relation $\supseteq$. Flow policies then consist of binary relations on $\boldsymbol{Pri}$, which can be understood as representing additional directions in which information is allowed to flow between principals: a pair $(p, q) \in F$, most often written $p \prec q$, is to be understood as "information may flow from p to q". New more permissive security lattices are obtained by collapsing security levels into possibly lower ones, by closing them with respect to the valid flow policy. Writing $F_1 \preccurlyeq F_2$ means that F_1 allows flows between at least as many pairs of principals as F_2. The relation is here defined as $F_1 \preccurlyeq F_2$ iff $F_2 \subseteq F_1^*$ (where F^* denotes the reflexive and transitive closure of F): The meet operation is then defined as $\curlywedge = \cup$, the join operation is defined as $F_1 \curlyvee F_2 = F_1^* \cap F_2^*$, the top flow policy is given by $\mho = \emptyset$, the bottom flow policy is given by $\Omega = \boldsymbol{Pri} \times \boldsymbol{Pri}$, and the pseudo-subtraction operation is given by $\smile = -$.

3 Language

The language extends an imperative higher order lambda calculus that includes reference and concurrent thread creation, a declassification construct, and a policy-context testing construct, with basic distribution and code mobility features. Computation domains hold a local allowed flow policy, which imposes a limit on the permissiveness of the declassifications that are performed within the domain. A remote thread creation construct serves as a code migration primitive.

$$\begin{array}{lrcll}
\textit{Variables} & x & \in & \boldsymbol{Var} & \qquad \textit{Flow Policies} \quad A, F \in \boldsymbol{Flo} \\
\textit{Reference Names} & a & \in & \boldsymbol{Ref} & \qquad \textit{Domain Names} \quad d \in \boldsymbol{Dom} \\
\textit{Values} & V & \in & \boldsymbol{Val} & ::= () \mid x \mid a \mid (\lambda x.M) \mid tt \mid ff \\
\textit{Pseudo-values} & X & \in & \boldsymbol{Pse} & ::= V \mid (\varrho x.X) \\
\textit{Expressions} & M, N & \in & \boldsymbol{Exp} & ::= X \mid (M\ N) \mid (M; N) \mid (\text{if } M \text{ then } N_t \text{ else } N_f) \mid \\
 & & & & \quad (\text{ref}_\theta\ M) \mid (!\ N) \mid (M := N) \mid \mathbf{(flow}\ \boldsymbol{F}\ \mathbf{in}\ \boldsymbol{M}\mathbf{)} \mid \\
 & & & & \quad \mathbf{(allowed}\ \boldsymbol{F}\ \mathbf{then}\ \boldsymbol{N_t}\ \mathbf{else}\ \boldsymbol{N_f}\mathbf{)} \mid \mathbf{(thread}\ \boldsymbol{M}\ \mathbf{at}\ \boldsymbol{d}\mathbf{)}
\end{array}$$

Fig. 1. Syntax of Expressions

3.1 Syntax

The syntax of expressions defined in Figure 1 is based on a λ-calculus extended with the imperative constructs of ML, conditional branching and boolean values, where the $(\varrho x.X)$ construct provides for recursive values. Names of references (a), domains (d), and threads (m, n), are drawn from disjoint countable sets $\boldsymbol{Ref}$, $\boldsymbol{Dom} \neq \emptyset$ and $\boldsymbol{Nam}$, respectively. References are information containers to which values of the language pertaining to a given type in $\boldsymbol{Typ}$ can be assigned.

Declassification is introduced in the language by means of *flow policy declarations* [9]. They have the form (flow F in M), and are used to locally weaken the information flow policy that is valid for the particular execution context, by enabling information flows that comply to the flow policy F to take place within the scope of the delimited block of code M. Expression M is executed in the context of the current flow policy extended with F; after termination the current flow policy is restored, that is, the scope of F is M. For context-policy awareness, programs can inspect the allowed flow policy of the current domain by means of the *allowed-condition*, which is written (allowed F then N_t else N_f). The construct tests whether the flow policy F is allowed by the current domain and executes branches N_t or N_f accordingly, in practice offering alternative behaviors to be taken in case the domains they end up are too restrictive. For migration and concurrency, the thread creator (thread M at d) spawns the thread M in domain d, to be executed concurrently with other threads at that domain.

Networks are flat juxtapositions of domains, each containing a store and a pool of threads, which are subjected to the allowed flow policy of the domain. Threads run concurrently in *pools* $P : \boldsymbol{Nam} \rightarrow \boldsymbol{Exp}$, which are mappings from thread names to expressions (denoted as sets of threads). *Stores* $S : \boldsymbol{Ref} \rightarrow \boldsymbol{Val}$ map reference names to values. *Position-trackers* $T : \boldsymbol{Nam} \rightarrow \boldsymbol{Dom}$, map thread names to domain names, and are used to keep track of the locations of threads in the network. The pool P containing all the threads in the network, the mapping T that keeps track of their positions, and the store S containing all the references in the network, form *configurations* $\langle P, T, S\rangle$. The flow policies that are allowed by each domain are kept by the *allowed-policy mapping* $W : \boldsymbol{Dom} \rightarrow \boldsymbol{Flo}$ from domain names to flow policies, which is considered fixed in this model.

3.2 Operational Semantics

The small step operational semantics of the language is defined in Figure 2. The '$W \vdash^{\Sigma}$' turnstile makes explicit the allowed flow policy of each domain in the network, and the *reference labeling* Σ that determines the type of values that is assigned to each reference name. Other security-related information, such as security levels, could be added for the purpose of an information flow analysis.

The call-by-value evaluation order is specified by representing expressions using *evaluation contexts*.

$$\begin{array}{r} \textit{Evaluation Contexts}\ \mathrm{E} ::= [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid (\mathrm{ref}_{\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \\ (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid (\mathrm{if}\ \mathrm{E}\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f) \mid \mathbf{(flow}\ \boldsymbol{F}\ \mathbf{in}\ \mathbf{E)} \end{array} \quad (1)$$

$$\begin{aligned}
W \vdash^{\Sigma} \langle \{\mathrm{E}[((\lambda x.M)\ V)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[\{x \mapsto V\}M]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathrm{if}\ tt\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[N_t]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \mathrm{E}[(\mathrm{if}\ \mathit{ff}\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[N_f]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(V; N)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[N]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(\varrho x.X)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[(\{x \mapsto (\varrho x.X)\}\ X)]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathrm{flow}\ F\ \mathrm{in}\ V)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[V]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(!\ a)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[S(a)]^m\}, T, S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(a := V)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[()]^m\}, T, [a := V]S\rangle \\
W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathrm{ref}_\theta\ V)]^m\}, T, S\rangle &\xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[a]^m\}, T, [a := V]S\rangle,\ \textit{a fresh in S} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textit{and } \Sigma(a) = \theta
\end{aligned}$$

$$\frac{\boldsymbol{W(T(m)) \preccurlyeq F}}{\boldsymbol{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathbf{allowed}\ F\ \mathbf{then}\ N_t\ \mathbf{else}\ N_f)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[N_t]^m\}, T, S\rangle}}$$

$$\frac{\boldsymbol{W(T(m)) \not\preccurlyeq F}}{\boldsymbol{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathbf{allowed}\ F\ \mathbf{then}\ N_t\ \mathbf{else}\ N_f)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[N_f]^m\}, T, S\rangle}}$$

$$\boldsymbol{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\mathbf{thread}\ N\ \mathbf{at}\ d)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E}\rceil]{} \langle \{\mathrm{E}[()]^m, N^n\}, [n := d]T, S\rangle,}$$
$$\boldsymbol{n\ \textit{fresh in}\ \mathbf{T}}$$

$$\frac{W \vdash^{\Sigma} \langle P, T, S\rangle \xrightarrow[F]{} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle\ \textit{is well formed}}{W \vdash^{\Sigma} \langle P \cup Q, T, S\rangle \xrightarrow[F]{} \langle P' \cup Q, T', S'\rangle}$$

Fig. 2. Operational Semantics

We write $\mathrm{E}[M]$ to denote an expression where the sub-expression M is placed in the evaluation context E, obtained by replacing the occurrence of $[]$ in E by M. The flow policy that is permitted by the evaluation context E is denoted by $\lceil \mathrm{E}\rceil$. It consists a lower bound (see Section 2) to all the flow policies that are declared by the context:

$$\begin{aligned}
\lceil [] \rceil = \mho, \qquad\qquad &\lceil (\mathrm{flow}\ F\ \mathrm{in}\ \mathrm{E}) \rceil = F \curlywedge \lceil \mathrm{E} \rceil, \\
\lceil \mathrm{E}'[\mathrm{E}] \rceil = \lceil \mathrm{E} \rceil,\ &\textit{when } \mathrm{E}' \textit{ does not contain flow declarations}
\end{aligned} \tag{2}$$

The following basic notations and conventions are useful for defining transitions. For a mapping Z, we define $\mathrm{dom}(Z)$ as the domain of a given mapping Z. We say a name is fresh in Z if it does not occur in $\mathrm{dom}(Z)$. We denote by $\mathrm{rn}(P)$ and $\mathrm{dn}(P)$ the set of reference and domain names, respectively, that occur in the expressions of P. We let $\mathrm{fv}(M)$ be the set of variables occurring free in M. We restrict our attention to well formed configurations $\langle P, T, S\rangle$ satisfying the conditions that $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$, that $\mathrm{dn}(P) \subseteq \mathrm{dom}(W)$, that $\mathrm{dom}(P) \subseteq \mathrm{dom}(T)$, and that, for any $a \in \mathrm{dom}(S)$, $\mathrm{rn}(S(a)) \subseteq \mathrm{dom}(S)$ and $\mathrm{dn}(S(a)) \subseteq \mathrm{dom}(W)$.

We denote by $\{x \mapsto W\}M$ the capture-avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

The transition rules of our semantics are decorated with the flow policy declared by the evaluation context where the step is performed. The lifespan of the flow declaration terminates when the expression M that is being evaluated terminates (that is, M becomes a value). In particular, the evaluation of (flow F in M) simply consists in the evaluation of M, annotated with a flow policy that is at least as permissive as F. The flow policy that decorates the transition steps is used only by the rules for (allowed F then N_t else N_f), where the choice of the branch depends on whether F is allowed to be declared or not. The thread creation construct functions as a migration construct when the new domain of the created thread is different from that of the parent thread. The last rule establishes that the execution of a pool of threads is compositional (up to the expected restriction on the choice of new names). Notice that W, representing the allowed flow policies associated to each domain, is never changed.

For simplicity, we assume memory to be shared by all programs and every computation domain, in a transparent form. This does not remove the distributed nature of the model, as programs' behavior depends on where they are [6].

4 Security Property

In a distributed setting with concurrent mobile code, programs might need to comply simultaneously to different allowed flow policies that change dynamically. The property of flow policy confinement deals with this difficulty by placing individual restrictions on each step that might be performed by a part of the program, taking into account the possible location where it might take place.

Compatibility. Since we are considering a higher-order language, values stored in memory can be used by programs to build expressions that are then executed. In order to avoid deeming all such programs insecure, memories are assumed to be compatible to the given security setting and typing environment, requiring typability of their contents with respect to the relevant type system and parameters. Informally, a memory S is said to be (W, Σ, Γ)-compatible if for every reference $a \in \text{dom}(S)$ its value $S(a)$ is typable. This predicate will be defined for each security analysis, and can be shown to be preserved by the semantics.

Flow Policy Confinement. The property is defined co-inductively for *located threads*, consisting of pairs $\langle d, M^m \rangle$ that carry information about the location d of a thread M^m. The location of each thread determines which allowed flow policy it should obey at that point, and is used to place a restriction on the flow policies that decorate the transitions: at any step, they should comply to the allowed flow policy of the domain where the thread who performed it is located.

Definition 1 ((W, Σ, Γ)-Confined Located Threads). *Consider an allowed-policy mapping W, a reference labeling Σ, and a typing environment Γ. A set $\mathcal{C}$*

of located threads is a set of (W, Σ, Γ)*-*confined located threads *if the following holds for all* $\langle d, M^m\rangle \in \mathcal{C}$*, for all* T *such that* $T(m) = d$*, and for all* (W, Σ, Γ)*-compatible memories* S*:*

- $W \vdash^{\Sigma} \langle\{M^m\}, T, S\rangle \xrightarrow[F]{} \langle\{M'^m\}, T', S'\rangle$ *implies* $W(T(m)) \preccurlyeq F$ *and also* $\langle T'(m), M'^m\rangle \in \mathcal{C}$*. Furthermore,* S' *is still* (W, Σ, Γ)*-compatible.*
- $W \vdash^{\Sigma} \langle\{M^m\}, T, S\rangle \xrightarrow[F]{} \langle\{M'^m, N^n\}, T', S'\rangle$ *implies* $W(T(m)) \preccurlyeq F$ *and also* $\langle T'(m), M'^m\rangle, \langle T'(n), N^n\rangle \in \mathcal{C}$*. Furthermore,* S' *is still* (W, Σ, Γ)*-compatible.*

Note that for any W, Σ, and Γ there exists a set of (W, Σ, Γ)-confined located threads, like for instance $\boldsymbol{Dom} \times (\boldsymbol{Val} \times \boldsymbol{Nam})$. Furthermore, the union of a family of sets of (W, Σ, Γ)-confined located threads is a set of (W, Σ, Γ)-confined located threads. The largest set of (W, Σ, Γ)-confined threads is denoted by $\mathcal{C}_W^{\Sigma,\Gamma}$.

We say that a thread M^m is (W, Σ, Γ)-confined when located at d, if $\langle d, M^m\rangle \in \mathcal{C}_W^{\Sigma,\Gamma}$. A well formed *thread configuration* $\langle P, T\rangle$, satisfying the applicable rules of a well formed configuration, is said to be (W, Σ, Γ)-confined if all located threads in $\{\langle T(m), M^m\rangle \mid M^m \in P\}$ are (W, Σ, Γ)-confined.

Notice that this property speaks strictly about what *flow declarations* a thread can do *while* it is at a specific domain. In particular, it does not restrict threads from migrating to more permissive domains in order to perform a declassification. More importantly, the property does not deal with information flows. So for instance it offers no assurance that information leaks that are encoded at each point of the program do obey the declared flow policies for that point. Such an analysis can be done independently, cf. *non-disclosure* in [9].

5 Enforcement Mechanisms

In this section we start by studying a type system for statically ensuring that global computations always comply to the locally valid allowed flow policy. This type system is inherently restrictive, as the domains where each part of the code will actually compute cannot in general be known statically (Subsection 5.1). We then present a more precise type system to be used at runtime by the semantics of the language for checking migrating threads against the allowed flow policy of the destination domain (Subsection 5.2). Finally, we propose a yet more precise type and effect system that computes information about the declassification behaviors of programs. This information will be used more efficiently at runtime by the semantics of the language in order to control migration of programs.

5.1 Purely Static Type Checking

We have seen that in a setting where code can migrate between domains with different allowed security policies, the computation domain might change *during* computation, along with the allowed flow policy that the program must comply to. This can happen in particular within the branch of an allowed condition:

$$(\text{allowed } F \text{ then } (\text{thread } (\text{flow } F \text{ in } M_1) \text{ at } d) \text{ else } M_2) \qquad (3)$$

$$[\textsc{Nil}]\ W;\Gamma \vdash^{\Sigma}_{A} () : \mathsf{unit} \qquad [\textsc{Bt}]\ W;\Gamma \vdash^{\Sigma}_{A} tt : \mathsf{bool} \qquad [\textsc{Bf}]\ W;\Gamma \vdash^{\Sigma}_{A} f\!f : \mathsf{bool}$$

$$[\textsc{Loc}]\ W;\Gamma \vdash^{\Sigma}_{A} a : \Sigma(a) \text{ ref} \qquad [\textsc{Var}]\ \Gamma, x : \tau \vdash^{\Sigma}_{A} x : \tau$$

$$[\textsc{Abs}]\ \frac{W;\Gamma, x : \tau \vdash^{\Sigma}_{A} M : \sigma}{W;\Gamma \vdash^{\Sigma}_{A'} (\lambda x.M) : \tau \xrightarrow{A} \sigma} \qquad [\textsc{Rec}]\ \frac{W;\Gamma, x : \tau \vdash^{\Sigma}_{A} X : \tau}{W;\Gamma \vdash^{\Sigma}_{A} (\varrho x.X) : \tau}$$

$$[\textsc{Ref}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta}{W;\Gamma \vdash^{\Sigma}_{A} (\text{ref}_{\theta}\ M) : \theta \text{ ref}} \qquad [\textsc{Der}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta \text{ ref}}{W;\Gamma \vdash^{\Sigma}_{A} (!\ M) : \theta}$$

$$[\textsc{Ass}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \theta \text{ ref} \quad W;\Gamma \vdash^{\Sigma}_{A} N : \theta}{W;\Gamma \vdash^{\Sigma}_{A} (M := N) : \mathsf{unit}} \qquad [\textsc{Seq}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \tau \quad W;\Gamma \vdash^{\Sigma}_{A} N : \sigma}{W;\Gamma \vdash^{\Sigma}_{A} (M; N) : \sigma}$$

$$[\textsc{Cond}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \mathsf{bool} \quad \begin{array}{c} W;\Gamma \vdash^{\Sigma}_{A} N_t : \tau \\ W;\Gamma \vdash^{\Sigma}_{A} N_f : \tau \end{array}}{W;\Gamma \vdash^{\Sigma}_{A} (\text{if } M \text{ then } N_t \text{ else } N_f) : \tau}$$

$$[\textsc{App}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} M : \tau \xrightarrow{A} \sigma \quad W;\Gamma \vdash^{\Sigma}_{A} N : \tau}{W;\Gamma \vdash^{\Sigma}_{A} (M\ N) : \sigma} \qquad [\textbf{Flow}]\ \frac{W;\Gamma \vdash^{\Sigma}_{A} N : \tau \quad A \preccurlyeq F}{W;\Gamma \vdash^{\Sigma}_{A} (\text{flow } F \text{ in } N) : \tau}$$

$$[\textbf{Allow}]\ \frac{\begin{array}{c} W;\Gamma \vdash^{\Sigma}_{A \curlywedge F} N_t : \tau \\ W;\Gamma \vdash^{\Sigma}_{A} N_f : \tau \end{array}}{W;\Gamma \vdash^{\Sigma}_{A} (\text{allowed } F \text{ then } N_t \text{ else } N_f) : \tau}$$

$$[\textbf{Mig}]\ \frac{W;\Gamma \vdash^{\Sigma}_{W(d)} M : \mathsf{unit}}{W;\Gamma \vdash^{\Sigma}_{A} (\text{thread } M \text{ at } d) : \mathsf{unit}}$$

Fig. 3. Type and effect system for checking Confinement

In this program, the flow declaration of the policy F is executed only if F has been tested as being allowed by the domain where the program was started. It might then seem that the flow declaration is "protected" by an appropriate allowed construct. However, by the time the flow declaration is performed, the thread is already located at another domain, where that flow policy might not be allowed. It is clear that a static enforcement of a confinement property requires tracking the possible locations where threads might be executing at each point.

Figure 3 presents a new type and effect system [10] for statically enforcing confinement over a migrating program. The type system guarantees that when operations are executed by a thread within the scope of a flow declaration, the declared flow complies to the allowed flow policy of the current domain. The typing judgments have the form

$$W;\Gamma \vdash^{\Sigma}_{A} M : \tau \tag{4}$$

meaning that expression M is typable with type τ in typing context $\Gamma : \boldsymbol{Var} \rightarrow \boldsymbol{Typ}$, which assigns types to variables. In addition to the reference mapping Σ, the turnstile has as parameter the flow policy A that is *allowed by the context*, which includes all flow policies that have been positively tested by the program as being allowed at the computation domain where the expression M is running. Finally, W represents the mapping of domain names to allowed flow policies.

Types have the standard syntax (t is a type variable)

$$\tau, \sigma, \theta \ \in \ \textbf{\textit{Typ}} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \text{ ref} \mid \tau \xrightarrow{A} \sigma \tag{5}$$

where the reference type records the type θ of values that the reference contains, and the functional type records the latent allowed policy A that is used to type the application of the function to an argument.

Our type system applies restrictions to programs in order to ensure that flow declarations can only declare flow policies that are allowed by the context (rule FLOW). These restrictions are relaxed when typing the first branch of allowed conditions, by extending the flow policy allowed by the context with the policy that guards the condition (rule ALLOW). In rule MIG, the flow policy allowed by the context is adjusted to that of the destination computation domain $W(d)$ that is specified by the (thread M at d) construct.

Note that if an expression is typable with respect to an allowed flow policy A, then it is also so for any more permissive allowed policy A'. In particular, due to the ABS rule, the process of typing an expression is not deterministic. For instance, the expression $(\lambda x.())$ can be given any type of the form $\tau \xrightarrow{F} \mathsf{unit}$.

We refer to the enforcement mechanism that consists of statically type checking all threads in a network according to the type and effect system of Figure 3, with respect to the allowed flow policies of each thread's initial domain, using the semantics represented in Figure 2, as *Enforcement mechanism I.*

Soundness. Enforcement mechanism I guarantees security of networks with respect to confinement, as is formalized by the following result. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \text{dom}(S)$ to store a value satisfying $W; \Gamma \vdash^{\Sigma}_{\mho} S(a) : \Sigma(a)$.

Theorem 1 (Soundness of Enforcement Mechanism I). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exists τ such that $W; \Gamma \vdash^{\Sigma}_{W(T(m))} M : \tau$. Then $\langle P, T\rangle$ is (W, Σ, Γ)-confined.*

Proof. By showing that the set $\{\langle d, M^m\rangle \mid m \in \textbf{\textit{Nam}} \text{ and } \exists\tau.\, W; \Gamma \vdash^{\Sigma}_{W(d)} M : \tau\}$ is a set of (W, Σ, Γ)-confined located threads, using induction on the inference of $W; \Gamma \vdash^{\Sigma}_{W(d)} M : \tau$.

Precision. Given the purely static nature of this migration control analysis, some secure programs are bound to be rejected. There are different ways to increase the precision of a type system, which are all intrinsically limited to what can conservatively be predicted before runtime. For example, for the program

$$(\text{if } (!\,a) \text{ then } (\text{thread } (\text{flow}\, F \text{ in}\, M) \text{ at}\, d_1) \text{ else } (\text{thread } (\text{flow}\, F \text{ in}\, M) \text{ at}\, d_2)) \tag{6}$$

it is in general not possible to predict which branch will be executed (or, in practice, to which domain the thread will migrate), for it depends on the contents of the memory. It will then be rejected if $W(d_2) \not\preccurlyeq F$ or $W(d_1) \not\preccurlyeq F$.

5.2 Runtime Type Checking

In this subsection we study a hybrid mechanism for enforcing confinement, that makes use of a relaxation of the type system of Figure 3 during runtime. Migration is now controlled by means of a runtime check for typability of migrating threads with respect to the allowed flow policy of the destination domain. The condition represents the standard theoretical requirement of checking incoming code before allowing it to execute in a given machine.

The relaxation is achieved by replacing rule Mig by the following one:

$$[\textsc{Mig}]\ \frac{\Gamma \vdash^{\Sigma}_{\Omega} M : \mathsf{unit}}{\Gamma \vdash^{\Sigma}_{A} (\text{thread } M \text{ at } d) : \mathsf{unit}} \tag{7}$$

The new type system no longer imposes *future* migrating threads to conform to the policy of their destination domain, but only to the most permissive allowed flow policy Ω. The rationale is that it only worries about confinement of the non-migrating parts of the program. This is sufficient, as all threads that are to be spawned by the program will be re-checked at migration time.

The following modification to the migration rule of the semantics of Figure 2 introduces the runtime check that controls migration (n fresh in T). The idea is that a thread can only migrate to a domain if it respects its allowed flow policy:

$$\frac{\Gamma \vdash^{\Sigma}_{W(d)} N : \mathsf{unit}}{W \vdash^{\Sigma} \langle\{\mathrm{E}[(\text{thread } N \text{ at } d)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{n} \langle\{\mathrm{E}[()]^m, N^n\}, [n := d]T, S\rangle} \tag{8}$$

The new remote thread creation rule (our migration primitive), now depends on typability of the migrating thread. The typing environment Γ (which is constant) is now an implicit parameter of the operational semantics. If only closed threads are considered, then also migrating threads are closed. The allowed flow policy of the destination site now determines whether or not a migration instruction may be consummated, or otherwise block execution.

Notice that, thanks to postponing the migration control to runtime, the type system no longer needs to be parameterized with information about the allowed flow policies of all domains in the network, which in practice could be impossible. The only relevant one are those of the destination domain of migrating threads.

We refer to the enforcement mechanism that consists of statically type checking all threads in a network according to the type and effect system of Figure 3 modified using the new Mig rule represented in Rule (8), with respect to the allowed flow policies of each thread's initial domain, using the semantics of Figure 2 modified according to Rule (7), as *Enforcement mechanism II.*

Soundness. Enforcement mechanism II guarantees security of networks with respect to confinement, as is formalized by the following result. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \mathrm{dom}(S)$ to store a value satisfying $\Gamma \vdash^{\Sigma}_{\mho} S(a) : \Sigma(a)$.

Theorem 2 (Soundness of Enforcement Mechanism II). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exists τ such that $\Gamma \vdash^{\Sigma}_{W(T(m))} M : \tau$. Then $\langle P, T\rangle$ is (W, Σ, Γ)-confined.*

Proof. By showing that the set $\{\langle d, M^m\rangle \mid m \in \boldsymbol{Nam} \textit{ and } \exists\tau \,.\, \Gamma \vdash^{\Sigma}_{W(d)} M : \tau\}$ is a set of (W, Σ, Γ)-confined located threads, using induction on the inference of $\Gamma \vdash^{\Sigma}_{W(d)} M : \tau$.

Safety, precision and efficiency. The proposed mechanism does not offer a safety result, guaranteeing that programs never "get stuck". Indeed, the side condition of the thread creation rule introduces the possibility for the execution of a thread to block, since no alternative is given. This can happen in Example 3 (in page 28), if the flow policy F is not permitted by the allowed policy of the domain of the branch that is actually executed, then the migration will not occur, and execution will not proceed. In order to have safety, we could design the thread creation instruction as including an alternative branch for execution in case the side condition fails. Nevertheless, Example 3 might have better been written

$$(\text{thread } (\text{allowed } F \text{ then } (\text{flow } F \text{ in } M_1) \text{ else } M_2) \text{ at } d) \tag{9}$$

in effect using the allowed condition for encoding such alternative behaviors.

Returning to Example 6 (in page 6), thanks to the relaxed Mig rule, this program is now *always* accepted statically by the type system. Depending on the result of the test, the migration might also be allowed to occur if a safe branch is chosen. This means that enforcement mechanism II accepts more secure programs. Because of the possibility of blockage mentioned above, an information flow analysis might reject some of the programs accepted here, in case for instance the reference a is assigned a "high" security level, and a "low" write is performed after the test. This issue is however orthogonal to our aims here.

A drawback with this enforcement mechanism lies in the computation weight of the runtime type checks. This is particularly acute for an expressive language such as the one we are considering. Indeed, recognizing typability of ML expressions has exponential (worst case) complexity [11].

5.3 Static Informative Typing for Runtime Effect Checking

We have seen that bringing the type-based migration control of programs to runtime allows to increase the precision of the confinement analysis. This is, however, at the cost of performance. It is possible to separate the program analysis as to what are the declassification operations that are performed by migrating threads, from the safety problem of determining whether those declassification operations should be allowed at a given domain. To achieve this, we now present an *informative* type system [8] that statically calculates a summary of all the declassification operations that might be performed by a program, in the form of a *declassification effect.* Furthermore, this type system produces a version of the

$$[\textsc{Nil}_\textsc{I}]\ \Gamma \vdash^{\Sigma} () \hookrightarrow () : \mho, \mathsf{unit} \quad [\textsc{Bt}_\textsc{I}]\ \Gamma \vdash^{\Sigma} tt \hookrightarrow tt : \mho, \mathsf{bool} \quad [\textsc{Bf}_\textsc{I}]\ \Gamma \vdash^{\Sigma} \mathit{ff} \hookrightarrow \mathit{ff} : \mho, \mathsf{bool}$$

$$[\textsc{Loc}_\textsc{I}]\ \Gamma \vdash^{\Sigma} a \hookrightarrow a : \mho, \Sigma(a) \text{ ref} \qquad [\textsc{Var}_\textsc{I}]\ \Gamma, x : \tau \vdash^{\Sigma} x \hookrightarrow x : \mho, \tau$$

$$[\textsc{Abs}_\textsc{I}]\ \frac{\Gamma, x : \tau \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \sigma}{\Gamma \vdash^{\Sigma} (\lambda x.M) \hookrightarrow (\lambda x.\hat{M}) : \mho, \tau \xrightarrow{s} \sigma} \qquad [\textsc{Rec}_\textsc{I}]\ \frac{\Gamma, x : \tau \vdash^{\Sigma} X \hookrightarrow \hat{X} : s, \tau}{\Gamma \vdash^{\Sigma} (\varrho x.X) \hookrightarrow (\varrho x.\hat{X}) : s, \tau}$$

$$[\textsc{Ref}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta' \quad \theta \preccurlyeq \theta'}{\Gamma \vdash^{\Sigma} (\text{ref}_\theta\ M) \hookrightarrow (\text{ref}_\theta\ \hat{M}) : s, \theta \text{ ref}} \qquad [\textsc{Der}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta \text{ ref}}{\Gamma \vdash^{\Sigma} (!\ M) \hookrightarrow (!\ \hat{M}) : s, \theta}$$

$$[\textsc{Ass}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \theta \text{ ref} \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s', \theta' \quad \theta \preccurlyeq \theta'}{\Gamma \vdash^{\Sigma} (M := N) \hookrightarrow (\hat{M} := \hat{N}) : s \curlywedge s', \mathsf{unit}}$$

$$[\textsc{Seq}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s', \sigma}{\Gamma \vdash^{\Sigma} (M; N) \hookrightarrow : s \curlywedge s', \sigma}$$

$$[\textsc{Cond}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \mathsf{bool} \quad \begin{array}{c}\Gamma \vdash^{\Sigma} N_t \hookrightarrow \hat{N}_t : s_t, \tau_t \\ \Gamma \vdash^{\Sigma} N_f \hookrightarrow \hat{N}_f : s_f, \tau_f\end{array} \quad \tau_t \approx \tau_f}{\Gamma \vdash^{\Sigma} (\text{if } M \text{ then } N_t \text{ else } N_f) \hookrightarrow (\text{if } \hat{M} \text{ then } \hat{N}_t \text{ else } \hat{N}_f) : s \curlywedge s_t \curlywedge s_f, \tau_t \curlywedge \tau_f}$$

$$[\textsc{App}_\textsc{I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \xrightarrow{s'} \sigma \quad \Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s'', \tau'' \quad \tau \preccurlyeq \tau''}{\Gamma \vdash^{\Sigma} (M\ N) \hookrightarrow (\hat{M}\ \hat{N}) : s \curlywedge s' \curlywedge s'', \sigma}$$

$$[\mathbf{Flow_I}]\ \frac{\Gamma \vdash^{\Sigma} N \hookrightarrow \hat{N} : s, \tau}{\Gamma \vdash^{\Sigma} (\text{flow } F \text{ in } N) \hookrightarrow (\text{flow } F \text{ in } \hat{N}) : s \curlywedge F, \tau}$$

$$[\mathbf{Allow_I}]\ \frac{\begin{array}{c}\Gamma \vdash^{\Sigma} N_t \hookrightarrow \hat{N}_t : s_t, \tau_t \\ \Gamma \vdash^{\Sigma} N_f \hookrightarrow \hat{N}_f : s_f, \tau_f\end{array} \quad \tau_t \approx \tau_f}{\Gamma \vdash^{\Sigma} (\text{allowed } F \text{ then } N_t \text{ else } N_f) \hookrightarrow (\text{allowed } F \text{ then } \hat{N}_t \text{ else } \hat{N}_f) : s_t \smile F \curlywedge s_f, \tau_t \curlywedge \tau_f}$$

$$[\mathbf{Mig_I}]\ \frac{\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \mathsf{unit}}{\Gamma \vdash^{\Sigma} (\text{thread } M \text{ at } d) \hookrightarrow (\text{thread}^s\ \hat{M} \text{ at } d) : \mho, \mathsf{unit}}$$

Fig. 4. Informative Type and Effect System for obtaining the Declassification Effect

program that is annotated with the relevant information for deciding, at runtime, whether its migrating threads can be considered safe by the destination domain. The aim is to bring the overhead of the runtime check to static time.

The typing judgments of the type system in Figure 4 have the form:

$$\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \tag{10}$$

Comparing with the typing judgments of Subsection 5.2, while the flow policy allowed by the context parameter is omitted from the turnstile '$\vdash$', the security effect s represents a flow policy which corresponds to the *declassification effect*: a lower bound to the flow policies that are declared in the typed expression. The second expression $\hat{M}$ is the result of annotating M. The syntax of annotated expressions differs only in the thread creation construct, that has an additional flow policy F as parameter, written (thread^F M at d). The syntax of types is the same as the ones used in Subsections 5.1 and 5.2.

It is possible to relax the type system by matching types that have the same structure, even if they differ in flow policies pertaining to them. We achieve this by overloading $\preccurlyeq$ to relate types where certain latent effects in the first are at least as permissive as the corresponding ones in the second. The more general relation $\approx$ matches types where certain latent effects differ: Finally, we define an operation $\curlywedge$ between two types τ and τ' such that $\tau \approx \tau'$:

$$\begin{aligned} \tau \preccurlyeq \tau' \text{ iff } \tau = \tau',\ \text{or } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma' \text{ with } F \preccurlyeq F' \text{ and } \sigma \preccurlyeq \sigma' \\ \tau \approx \tau' \text{ iff } \tau = \tau',\ \text{or } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma' \text{ with } \sigma \approx \sigma' \\ \tau \curlywedge ! \tau' = \tau,\ \text{if } \tau = \tau',\ \text{or } \theta \xrightarrow{F \curlywedge F'} \sigma \curlywedge \sigma',\ \text{if } \tau = \theta \xrightarrow{F} \sigma \text{ and } \tau' = \theta \xrightarrow{F'} \sigma' \end{aligned} \tag{11}$$

The $\preccurlyeq$ relation is used in rules Ref$_\text{I}$, Ass$_\text{I}$ andApp$_\text{I}$, in practice enabling to associate to references and variables (by reference creation, assignment and application) expressions with types that contain stricter policies than required by the declared types. The relation $\approx$ is used in rules Cond$_\text{I}$ andAllow$_\text{I}$ in order to accept that two branches of the same test construct can differ regarding some of their policies. Then, the type of the test construct is constructed from both using $\curlywedge$, thus reflecting the flow policies in both branches.

The declassification effect is constructed by aggregating (using the meet operation) all relevant flow policies that are declared within the program. The effect is updated in rule Flow$_\text{I}$, each time a flow declaration is performed, and "grows" as the declassification effects of sub-expressions are met in order to form that of the parent command. However, when a part of the program is "protected" by an allowed condition, some of the information in the declassification effect can be discarded. This happens in rule Allow$_\text{I}$, where the declassification effect of the first branch is not used entirely: the part that will be tested during execution by the allowed-condition is omitted. In rule Mig$_\text{I}$, the declassification effect of migrating threads is also not recorded in the effect of the parent program, as they will be executed (and tested) elsewhere. That information is however used to annotate the migration instruction.

One can show that the type system is deterministic, in the sense that it assigns to a non-annotated expression a single annotated version of it, a single declassification effect, and a single type.

Modified operational semantics, revisited. By executing annotated programs, the type check that conditions the migration instruction can be replaced by a simple declassification effect inspection. The new migration rule is similar to the one in Subsection 5.2, but now makes use of the declassification effect (n fresh in T):

$$\frac{W(d) \preccurlyeq s}{W \vdash^{\Sigma} \langle \{\mathrm{E}[(\text{thread}^s\ N \text{ at } d)]^m\}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{n} \langle \{\mathrm{E}[()]^m, N^n\}, [n := d]T, S\rangle} \tag{12}$$

In the remaining rules of the operational semantics the annotations are ignored.

We refer to the mechanism that consists of statically annotating all threads in a network according to the type and effect system of Figure 4, assuming that each thread's declassification effect is allowed by its initial domain, using the semantics of Figure 2 modified according to Rule (12), as *Enforcement mechanism III.*

Soundness. We will now see that the declassification effect can be used for enforcing confinement. The (W, Σ, Γ)-compatibility predicate that is used to define confinement here requires all references $a \in \mathrm{dom}(S)$ to store a value that results from annotating some other value V according to $\Gamma \vdash^{\Sigma}_{\mho} V \hookrightarrow S(a) : \Sigma(a)$.

Theorem 3 (Soundness of Enforcement Mechanism III). *Consider a fixed allowed-policy mapping W, a given reference labeling Σ and typing environment Γ, and a thread configuration $\langle P, T\rangle$ such that for all $M^m \in P$ there exist $\hat{M}$, s and τ such that $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau$ and $W(T(m)) \preccurlyeq s$. Then $\langle \hat{P}, T\rangle$, formed by annotating the threads in $\langle P, T\rangle$, is (W, Σ, Γ)-confined.*

Proof. By showing that the following is a set of (Σ, Γ)-confined located threads

$$\{\langle d, \hat{M}^m\rangle \mid m \in \textbf{\textit{Nam}} \textit{ and } \exists M, s, \tau \,.\, \Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau \text{ and } W(d) \preccurlyeq s\} \quad (13)$$

using induction on the inference of $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau$.

Precision and efficiency. The relaxed type system of Subsection 5.2 for checking confinement, and its informative counterpart of Figure 4, are strongly related. The following result states that typability according to latter type system is at least as precise as the former. It is proven by induction on the inference of $\Gamma \vdash^{\Sigma}_{A} M : \tau$.

Proposition 1. *Consider a given a typing environment Γ and reference labeling Σ. If there exist A, τ such that $\Gamma \vdash^{\Sigma}_{A} M : \tau$, then there exist $\hat{M}$, τ' and s such that $\Gamma \vdash^{\Sigma} M \hookrightarrow \hat{M} : s, \tau'$ and $A \preccurlyeq s$ with $\tau \preccurlyeq \tau'$.*

The converse direction is not true, i.e. enforcement mechanism III accepts strictly more programs than enforcement mechanism II. This can be seen by considering the secure program where, $\theta_1 = \tau \xrightarrow{F_1} \sigma$ and $\theta_2 = \tau \xrightarrow{F_2} \sigma$:

$$(\text{if } (!\ a) \text{ then } (!\ (\text{ref}_{\theta_1}\ M_1)) \text{ else } (!\ (\text{ref}_{\theta_2}\ M_2))) \quad (14)$$

This program is not accepted by the type system of Section 5.2 because it cannot give the same type to both branches of the conditional (the type of the dereference of a reference of type θ is precisely θ). However, since the two types satisfy $\theta_1 \approx \theta_2$, the informative type system can accept it and give it the type $\theta_1 \curlywedge \theta_2$.

A more fundamental difference between the two enforcement mechanisms lays in the timing of the computation overhead that is required by each mechanism. While mechanism II requires heavy runtime type checks to occur each time a thread migrates, in III the typability analysis is anticipated to static time, leaving only a comparison between two flow policies to be performed at migration time. The complexity of this comparison depends on the concrete representation of flow policies. In the worst case, that of flow policies as general downward closure operators (see Section 2), it is linear on the number of security levels that are considered. When flow policies are flow relations, then it consists on a subset relation check, which is polynomial on the size of the flow policies.

6 Related Work

Controlling declassification. Most previous mechanisms for controlling declassification [12] target flexible versions of an information flow property. Departing from this approach, the work by Boudol and Kolundzija [13] on combining access control and declassification is the first to treat declassification control separately from the underlying information flow problem. In [13], standard access control primitives are used to control the access level of programs that perform declassifications in the setting of a local language, ensuring that a program can only declassify information that it has the right to read.

Controlling code mobility. A wide variety of distributed network models have been designed with the purpose of studying mechanisms for controlling code mobility. These range from type systems for statically controlling migration as an access control mechanism [5,14], to runtime mechanisms that are based on the concept of programmable domain. In the latter, computing power is explicitly associated to the *membranes* of computation domains, and can be used for controlling boundary transposition. This control can be performed by processes that interact with external and internal programs [15,16,4], or by more specific automatic verification mechanisms [17]. In the present work we abstract away from the particular machinery that implements the migration control checks, and express declaratively, via the language semantics, the condition that must be satisfied for the boundary transposition to be allowed.

Checking the validity of the declassification effect as a certificate is not simpler than checking the program against a concrete allowed policy (as presented in Subsection 5.2), meaning that it does not consist of a case of Proof Carrying Code. The concept of trust can be used to lift the checking requirements of code whose history of visited domains provides enough reassurance [17,5]. These ideas could be applied to the present work, assisting the decision of trusting the declassification effect, otherwise leading to a full type check of the code.

Hybrid mechanisms. The use of hybrid mechanisms for enforcing information flow policies is currently an active research area (see [18] for a review of related work). The closest to ours is perhaps the study of securing information release for a simple language with dynamic code evaluation in the form of a string eval command, which includes an on-the-fly information flow static analysis [19].

Focusing on declassification control, the idea of using a notion of declassification effect for building a runtime migration control mechanism was put forward in [6] for a similar language with local thread creator and a basic goto migration instruction. In spite of the restrictions that are pointed out in Subsection 5.1 for a static analysis, the type system presented as part of Enforcement Mechanism I is more refined than the proof-of-concept presented earlier. Indeed, in the previous work, migration was not taken into account when analyzing the declassifications occurring within the migrating code. So while there the following program would be rejected if F was not allowed by $W(d_1)$

$$(\text{thread } (\text{thread } (\text{flow } F \text{ in } M) \text{ at } d_2) \text{ at } d_1) \tag{15}$$

the type system of Figure 3 only rejects it if F is not allowed by $W(d_2)$. Enforcement Mechanism II adopts part of the idea in [6] of performing a runtime type analysis to migrating programs, but uses a more permissive "checking" type system. Enforcement Mechanism III explores a mechanism that allows to take advantage of the efficiency of flow policy comparisons. It uses a type and effect system for calculating declassification effects that is substantially more precise than previous ones, thanks to the matching relations and operations that it uses.

The concept of informative type and effect system was introduced in [8], where a different notion of declassification effect was defined and applied to the problem of dealing with dynamic updates to a local allowed flow policy.

7 Conclusion

We have considered an instance of the problem of enforcing compliance of declassifications to a dynamically changing allowed flow policy. In our setting, changes in the allowed flow policy result from the migration of programs during execution. We approach the problem from a migration control perspective. To this end, we chose a network model that abstracts away the details of the migration control architecture. This allows us to prove soundness of a concrete network level security property, guaranteeing that programs can roam over the network, never performing declassifications that violate the network confinement property.

While our results are formulated for a particular security property – flow policy confinement – we expect that similar ideas can be used for other properties. One could add expressiveness to the property by taking into account the history of domains that a thread has visited when defining secure code migrations. For instance, one might want to forbid threads from moving to domains with more favorable allowed flow policies. This would be easily achieved by introducing a condition on the allowed flow policies of origin and destination domains.

By performing comparisons between three related enforcement mechanisms, we have argued that the concept of declassification effect offers a good balance between precision and efficiency. We believe that similar mechanisms can be applied in other contexts. For future work, we plan to study others instances of enabling dynamic changing allowed flow policies.

Acknowledgments. This work benefits from the authors' visits to the Indes Team at INRIA Sophia Antipolis and the Language Based Security Group at Chalmers University of Technology, partially supported by the Portuguese Foundation for Science and Technology. Thanks are due to José Santos for insightful feedback.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
2. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symp. on Security and Privacy, pp. 11–20. IEEE Computer Society (1982)
3. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop, pp. 255–269. IEEE Computer Society (2005)
4. Boudol, G.: A generic membrane model (Note). In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 208–222. Springer, Heidelberg (2005)
5. Martins, F., Vasconcelos, V.T.: History-based access control for distributed processes. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 98–115. Springer, Heidelberg (2005)
6. Almeida Matos, A.: Flow policy awareness for distributed mobile code. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 53–68. Springer, Heidelberg (2009)
7. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
8. Matosand, A.A., Santos, J.F.: Typing illegal information flows as program effects. In: Proceedings of the 7th Workshop on Programming Languages and Analysis for Security, PLAS 2012, pp. 1:1–1:12. ACM (2012)
9. Matos, A.A., Boudol, G.: On declassification and the non-disclosure policy. Journal of Computer Security 17(5), 549–597 (2009)
10. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: 15th ACM Symp. on Principles of Programming Languages, pp. 47–57. ACM Press (1988)
11. Mairson, H.G.: Deciding ml typability is complete for deterministic exponential time. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, pp. 382–401. ACM (1990)
12. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. 17, 517–548 (2009)
13. Boudol, G., Kolundzija, M.: Access Control and Declassification. In: Gorodetsky, V., Kotenko, I., Skormin, V.A. (eds.) MMM-ACNS 2007. CCIS, vol. 1, pp. 85–98. Springer, Heidelberg (2007)
14. Hennessy, M., Merro, M., Rathke, J.: Towards a behavioural theory of access and mobility control in distributed systems. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 282–298. Springer, Heidelberg (2003)
15. Levi, F., Sangiorgi, D.: Controlling interference in ambients. In: POPL 2000: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 352–364. ACM, New York (2000)

16. Schmitt, A., Stefani, J.-B.: The m-calculus: a higher-order distributed process calculus. SIGPLAN Not 38(1), 50–61 (2003)
17. Gorla, D., Hennessy, M., Sassone, V.: Security policies as membranes in systems for global computing. In: Foundations of Global Ubiquitous Computing, FGUC 2004. ENTCS, pp. 23–42. Elsevier (2005)
18. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium, CSF 2011, pp. 146–160. IEEE Computer Society (2011)
19. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, CSF 2009, pp. 43–59. IEEE Computer Society (2009)

Monitoring of Temporal First-Order Properties with Aggregations

David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu

Institute of Information Security, ETH Zurich, Switzerland

Abstract. Compliance policies often stipulate conditions on aggregated data. Current policy monitoring approaches are limited in the kind of aggregations that they can handle. We rectify this as follows. First, we extend metric first-order temporal logic with aggregation operators. This extension is inspired by the aggregation operators common in database query languages like SQL. Second, we provide a monitoring algorithm for this enriched policy specification language. Finally, we experimentally evaluate our monitor's performance.

1 Introduction

Motivation. Compliance policies represent normative regulations, which specify permissive and obligatory actions for agents. Both public and private companies are increasingly required to monitor whether agents using their IT systems, i.e., users and their processes, comply with such policies. For example, US hospitals must follow the US Health Insurance Portability and Accountability Act (HIPAA) and financial services must conform to the Sarbanes-Oxley Act (SOX). First-order temporal logics are not only well-suited for formalizing such regulations, they also admit efficient monitoring. When used online, these monitors observe the agents' actions as they are made and promptly report violations. Alternatively, the actions are logged and the monitor checks them later, such as during an audit. See, for example, [6, 18].

Current logic-based monitoring approaches are limited in their support for expressing and monitoring aggregation conditions. Such conditions are often needed for compliance policies, such as the following simple example from fraud prevention: A user must not withdraw more than \$10,000 within a 30 day period from his credit card account. To formalize this policy, we need an operator to express the aggregation of the withdrawal amounts over the specified time window, grouped by the users. In this paper, we address the problem of expressing and monitoring first-order temporal properties built from such aggregation operators.

Solution. First, we extend metric first-order temporal logic (MFOTL) with aggregation operators and with functions. This follows Hella et al.'s [19] extension of first-order logic with aggregations. We also ensure that the semantics of aggregations and grouping operations in our language mimics that of SQL.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 40–58, 2013.

As illustration, a formalization in our language of the above fraud-detection policy is

$$\Box\, \forall u.\, \forall s.\, [\mathsf{SUM}_a\, a.\, \blacklozenge_{[0,31)}\, \mathit{withdraw}(u,a)](s;u) \rightarrow s \preceq 10000\,. \qquad \text{(P0)}$$

The SUM operator, at the current time point, groups all withdrawals, in the past 30 days, for a user u and then sums up their amounts a. The aggregation formula defines a binary relation where the first coordinate is the SUM's result s and the second coordinate is the user u for whom the result is calculated. If the user's sum is greater than 10000, then the policy is violated at the current time point. Finally, the formula states that the aggregation condition must hold for each user and every time point.

A corresponding SQL query for determining the violations with respect to the above policy at a specific time is

$$\mathsf{SELECT\ SUM}(a)\ \mathsf{AS}\ s, u\ \mathsf{FROM}\ W\ \mathsf{GROUP\ BY}\ u\ \mathsf{HAVING\ SUM}(a) > 10000\,,$$

where W is the dynamically created view consisting of the withdrawals of all users within the 30 day time window relative to the given time. Note that the subscript a of the formula's aggregation operator in (P0) corresponds to the a in the SQL query and the third appearance of a in (P0) is implicit in the query, as it is fixed by the view's definition. The second a in (P0) is redundant and emphasizes that the variable a is quantified, i.e., it does not correspond to a coordinate in the resulting relation.

Not all formulas in our language are monitorable. Unrestricted use of logic operators may require infinite relations to be built and manipulated. The second part of our solution, therefore, is a monitorable fragment of our language. It can express all our examples, which represent typical policy patterns, and it allows the liberal use of aggregations and functions. We extend our monitoring algorithm for MFOTL [7] to this fragment. In more detail, the algorithm processes log files sequentially and handles aggregation formulas by translating them into extended relational algebra. Functions are handled similarly to Prolog, where variables are instantiated before functions are evaluated.

We have implemented and evaluated our monitoring solution. For the evaluation, we use fraud-detection policy examples and synthetically generated log files. We first compare the performance of our prototype implementation with the performance of the relational database management system PostgreSQL [22]. Our language is better suited for expressing the policy examples and our prototype's performance is superior to PostgreSQL's performance. This is not surprising since the temporal reasoning must be explicitly encoded in SQL queries and PostgreSQL does not process logged data in the time sequential manner. We also compare our prototype implementation with the stream-processing tool STREAM [2]. Its performance is better than our tool's performance because, in contrast to our tool, STREAM is limited to a restricted temporal pattern for which it is optimized. Although we have not explored performance optimizations for our tool, it is, nevertheless, already efficient enough for practical use.

Contributions. Although aggregations have appeared previously in monitoring, to our knowledge, our language is the first to add expressive SQL-like aggregation operators to a first-order temporal setting. This enables us to express complex compliance policies with aggregations. Our prototype implementation of the presented monitoring algorithm is therefore the first tool to handle such policies, and it does so with acceptable performance.

Related Work. Our MFOTL extension is inspired by the aggregation operators in database query languages like SQL and by Hella et al.'s extension of first-order logic with aggregation operators [19]. Hella et al.'s work is theoretically motivated: they investigate the expressiveness of such an extension in a non-temporal setting. A minor difference between their aggregation operators and ours is that their operators yield terms rather than formulas as in our extension.

Monitoring algorithms for different variants of first-order temporal logics have been proposed by Hallé and Villemaire [18], Bauer at al. [9], and Basin et al. [7]. Except for the counting quantifier [9], none of them support aggregations. Bianculli et al. [10] present a policy language based on a first-order temporal logic with a restricted set of aggregation operators that can only be applied to atomic formulas. For monitoring, they require a fixed finite domain and provide a translation to a propositional temporal logic. Such a translation is not possible in our setting since variables range over an infinite domain. In the context of database triggers and integrity constraints, Sistla and Wolfson [23] describe an integration of aggregation operators into their monitoring algorithm for a first-order temporal logic. Their aggregation operators are different from those presented here in that they involve two formulas that select the time points to be considered for aggregation and they use a database query to select the values to be aggregated from the selected time points.

Other monitoring approaches that support different kinds of aggregations are LarvaStat [13], LOLA [15], EAGLE [4], and an approach based on algebraic alternating automata [16]. These approaches allow one to aggregate over the events in system traces, where events are either propositions or parametrized propositions. They do not support grouping, which is needed to obtain statistics per group of events, e.g., the events generated by the same agent. Moreover, quantification over data elements and correlating data elements is more restrictive in these approaches than in a first-order setting.

Most data stream management systems like STREAM [2] and Gigascope [14] handle SQL-like aggregation operators. For example, in STREAM's query language CQL [3] one selects events in a specified time range, relative to the current position in the stream, into a table on which one performs aggregations. The temporal expressiveness of such languages is weaker than our language, in particular, linear-time temporal operators are not supported.

Organization. In Section 2, we extend MFOTL with aggregation operators. In Section 3, we present our monitoring algorithm, which we evaluate in Section 4. In Section 5, we draw conclusions. Additional details are given in the appendix.

2 MFOTL with Aggregation Operators

2.1 Preliminaries

We use standard notation for sets and set operations. We also use set notation with sequences. For instance, for a set A and a sequence $\bar{s} = (s_1, \ldots, s_n)$, we write $A \cup \bar{s}$ for the union $A \cup \{s_i \mid 1 \leq i \leq n\}$ and we denote the length of $\bar{s}$ by $|\bar{s}|$. Let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[b, b') := \{a \in \mathbb{N} \mid b \leq a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$.

A *multi-set* M with domain D is a function $M : D \to \mathbb{N} \cup \{\infty\}$. This definition extends the standard one to multi-sets where elements can have an infinite multiplicity. A multi-set is *finite* if $M(a) \in \mathbb{N}$ for any $a \in D$ and the set $\{a \in D \mid M(a) > 0\}$ is finite. We use the brackets $\{\!|$ and $|\!\}$ to specify multi-sets. For instance, $\{\!| 2 \cdot \lfloor n/2 \rfloor \mid n \in \mathbb{N} |\!\}$ denotes the multi-set $M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ with $M(n) = 2$ if n is even and $M(n) = 0$ otherwise.

An *aggregation operator* is a function from multi-sets to $\mathbb{Q} \cup \{\bot\}$ such that finite multi-sets are mapped to elements of $\mathbb{Q}$ and infinite multi-sets are mapped to $\bot$. Common examples are $\mathsf{CNT}(M) := \sum_{a \in D} M(a)$, $\mathsf{SUM}(M) := \sum_{a \in D} M(a) \cdot a$, $\mathsf{MIN}(M) := \min\{a \in D \mid M(a) > 0\}$, $\mathsf{MAX}(M) := \max\{a \in D \mid M(a) > 0\}$, and $\mathsf{AVG}(M) := \mathsf{SUM}(M)/\mathsf{CNT}(M)$ if $\mathsf{CNT}(M) \neq 0$ and $\mathsf{AVG}(M) := 0$ otherwise, where $M : D \to \mathbb{N} \cup \{\infty\}$ is a finite multi-set. We assume that the given aggregation operators are only applied over the multisets with the domain $\mathbb{Q}$.

2.2 Syntax

A *signature* $\mathcal{S}$ is a tuple $(\mathsf{F}, \mathsf{R}, \iota)$, where F is a finite set of function symbols, R is a finite set of predicate symbols disjoint from F, and the function $\iota : \mathsf{F} \cup \mathsf{R} \to \mathbb{N}$ assigns to each symbol $s \in \mathsf{F} \cup \mathsf{R}$ an arity $\iota(s)$. In the following, let $\mathcal{S} = (\mathsf{F}, \mathsf{R}, \iota)$ be a signature and V a countably infinite set of variables, where $\mathsf{V} \cap (\mathsf{F} \cup \mathsf{R}) = \emptyset$.

Function symbols of arity 0 are called *constants*. Let $\mathsf{C} \subseteq \mathsf{F}$ be the set of constants of $\mathcal{S}$. *Terms* over $\mathcal{S}$ are defined inductively: Constants and variables are terms, and $f(t_1, \ldots, t_n)$ is a term if $t_1, \ldots, t_n$ are terms and f is a function symbol of arity $n > 0$. We denote by $fv(t)$ the set of the variables that occur in the term t. We denote by T the set of all terms over $\mathcal{S}$, and by $\mathsf{T}_\emptyset$ the set of ground terms. A *substitution* θ is a function from variables to terms. We use the same symbol θ to denote its homomorphic extension to terms.

Given a finite set Ω of aggregation operators, the MFOTL$_\Omega$ *formulas* over the signature $\mathcal{S}$ are given by the grammar

$$\varphi ::= r(t_1, \ldots, t_{\iota(r)}) \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\exists x.\, \varphi) \mid (\bullet_I\, \varphi) \mid (\varphi\; \mathsf{S}_I\; \psi) \mid [\omega_t\, \bar{z}.\, \varphi](y; \bar{g})\,,$$

where r, t and the t_is, I, and ω range over the elements in R, T, $\mathbb{I}$, and Ω, respectively, x and y range over elements in V, and $\bar{z}$ and $\bar{g}$ range over sequences of elements in V. Note that we overload notation: ω denotes both an aggregation operator and its corresponding symbol. This grammar extends MFOTL's [20] in two ways. First, it introduces aggregation operators. Second, terms may also be built from function symbols and not just from variables and constants. For ease of exposition, we do not consider future-time temporal operators.

We call $[\omega_t\, \bar{z}.\, \psi](y; \bar{g})$ an *aggregation formula*. It is inspired by the homonymous relational algebra operator. Intuitively, by viewing variables as (relation) attributes, $\bar{g}$ are the attributes on which grouping is performed, t is the term on which the aggregation operator ω is applied, and y is the attribute that stores the result. The variables in $\bar{z}$ are ψ's attributes that do not appear in the described relation. We define the semantics in Section 2.3, where we also provide examples.

The set of *free variables* of a formula φ, denoted $fv(\varphi)$, is defined as expected for the standard logic connectives. For an aggregation formula, it is defined as $fv\big([\omega_t\, \bar{z}.\, \varphi](y; \bar{g})\big) := \{y\} \cup \bar{g}$. A variable is *bound* if it is not free. We denote by $\overline{fv}(\varphi)$ the sequence of free variables of a formula φ that is obtained by ordering the free variables of φ by their occurrence when reading the formula from left to right. A formula is *well-formed* if for each of its subformulas $[\omega_t\, \bar{z}.\, \psi](y; \bar{g})$, it holds that (a) $y \notin \bar{g}$, (b) $fv(t) \subseteq fv(\psi)$, (c) the elements of $\bar{z}$ and $\bar{g}$ are pairwise distinct, and (d) $\bar{z} = fv(\psi) \setminus \bar{g}$. Note that, given condition (d), the use of one of the sequences $\bar{z}$ and $\bar{g}$ is redundant. However, we use this syntax to make explicit the free and bound variables in aggregation formulas. Throughout the paper, we consider only well-formed formulas.

To omit parenthesis, we assume that Boolean connectives bind stronger than temporal connectives, and unary connectives bind stronger than binary ones, except for the quantifiers, which bind weaker than Boolean ones. As syntactic sugar, we use standard Boolean connectives such as $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, the universal quantifier $\forall x.\, \varphi := \neg\exists x.\, \neg\varphi$, and the temporal operators $\blacklozenge_I\, \varphi := (p \vee \neg p)\, \mathsf{S}_I\, \varphi$, $\blacksquare_I\, \varphi := \neg \blacklozenge_I \neg\varphi$, where $I \in \mathbb{I}$ and p is some predicate symbol of arity 0, assuming without loss of generality that R contains such a symbol. Non-metric variants of the temporal operators are easily defined, e.g., $\blacklozenge\, \varphi := \blacklozenge_{[0,\infty)}\, \varphi$.

2.3 Semantics

We distinguish between predicate symbols whose corresponding relations are *rigid* over time and those that are *flexible*, i.e., their interpretations can change over time. We denote by R_r and R_f the sets of rigid and flexible predicate symbols, where $\mathsf{R} = \mathsf{R}_r \cup \mathsf{R}_f$ with $\mathsf{R}_r \cap \mathsf{R}_f = \emptyset$. We assume R_r contains the binary predicate symbols $\approx$ and $\prec$, which have their expected interpretation, namely, equality and ordering.

A *structure* $\mathcal{D}$ over the signature $\mathcal{S}$ consists of a domain $\mathbb{D} \neq \emptyset$ and interpretations $f^{\mathcal{D}} \in \mathbb{D}^{\iota(f)} \to \mathbb{D}$ and $r^{\mathcal{D}} \subseteq \mathbb{D}^{\iota(r)}$, for each $f \in \mathsf{F}$ and $r \in \mathsf{R}$. A *temporal structure* over the signature $\mathcal{S}$ is a pair $(\bar{\mathcal{D}}, \bar{\tau})$, where $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ is a sequence of structures over $\mathcal{S}$ and $\bar{\tau} = (\tau_0, \tau_1, \dots)$ is a sequence of non-negative integers, with the following properties.

1. The sequence $\bar{\tau}$ is monotonically increasing, that is, $\tau_i \leq \tau_{i+1}$, for all $i \geq 0$. Moreover, $\bar{\tau}$ makes progress, that is, for every $\tau \in \mathbb{N}$, there is some index $i \geq 0$ such that $\tau_i > \tau$.
2. All structures $\mathcal{D}_i$, with $i \geq 0$, have the same domain, denoted $\mathbb{D}$.

3. Function symbols and rigid predicate symbols have rigid interpretations, that is, $f^{\mathcal{D}_i} = f^{\mathcal{D}_{i+1}}$ and $p^{\mathcal{D}_i} = p^{\mathcal{D}_{i+1}}$, for all $f \in \mathsf{F}$, $p \in \mathsf{R}_r$, and $i \geq 0$. We also write $f^{\mathcal{D}}$ and $p^{\mathcal{D}}$ for $f^{\mathcal{D}_i}$ and $p^{\mathcal{D}_i}$, respectively.

We call the elements in the sequence $\bar{\tau}$ *timestamps* and the indices of the elements in the sequences $\bar{\mathcal{D}}$ and $\bar{\tau}$ *time points*.

A *valuation* is a mapping $v : \mathsf{V} \to \mathbb{D}$. For a valuation v, the variable sequence $\bar{x} = (x_1, \ldots, x_n)$, and $\bar{d} = (d_1, \ldots, d_n) \in \mathbb{D}^n$, we write $v[\bar{x} \mapsto \bar{d}]$ for the valuation that maps x_i to d_i, for $1 \leq i \leq n$, and the other variables' valuation is unaltered. We abuse notation by also applying a valuation v to terms. That is, given a structure $\mathcal{D}$, we extend v homomorphically to terms.

For the remainder of the paper, we fix a countable domain $\mathbb{D}$ with $\mathbb{Q} \cup \{\bot\} \subseteq \mathbb{D}$. We only consider a single-sorted logic. One could alternatively have sorts for the different types of elements like data elements and the aggregations. Furthermore, note that function symbols are always interpreted by total functions. Partial functions like division over scalar domains can be extended to total functions, e.g., by mapping elements outside the function's domain to $\bot$. Since the treatment of partial functions is not essential to our work, we treat $\bot$ as any other element of $\mathbb{D}$. Alternative treatments are, e.g., based on multi-valued logics [21].

Definition 1. *Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure over the signature $\mathcal{S}$, with $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \ldots)$ and $\bar{\tau} = (\tau_0, \tau_1, \ldots)$, φ a formula over $\mathcal{S}$, v a valuation, and $i \in \mathbb{N}$. We define the relation $(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \varphi$ inductively as follows:*

$$\begin{array}{lll}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models p(t_1, \ldots, t_{\iota(r)}) & \textit{iff} & \big(v(t_1), \ldots, v(t_{\iota(r)})\big) \in p^{\mathcal{D}_i} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \neg\psi & \textit{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \vee \psi' & \textit{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \textit{ or } (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi' \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \exists x.\, \psi & \textit{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v[x \mapsto d], i) \models \psi, \textit{ for some } d \in \mathbb{D} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \bullet_I\, \psi & \textit{iff} & i > 0,\ \tau_i - \tau_{i-1} \in I,\ \textit{and } (\bar{\mathcal{D}}, \bar{\tau}, v, i-1) \models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi\, \mathsf{S}_I\, \psi' & \textit{iff} & \textit{for some } j \leq i,\ \tau_i - \tau_j \in I,\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi', \\
 & & \textit{and } (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \psi,\ \textit{for all } k \textit{ with } j < k \leq i \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models [\omega_t\, \bar{z}.\, \psi](y; \bar{g}) & \textit{iff} & v(y) = \omega(M),
\end{array}$$

where $M : \mathbb{D} \to \mathbb{N} \cup \{\infty\}$ is the multi-set

$$\{\!\!\{ v[\bar{z} \mapsto \bar{d}](t) \;\big|\; (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{z} \mapsto \bar{d}], i) \models \psi,\ \textit{for some } \bar{d} \in \mathbb{D}^{|\bar{z}|} \}\!\!\}.$$

Note that the semantics for the aggregation formula is independent of the order of the variables in the sequence $\bar{z}$.

For a temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$, a time point $i \in \mathbb{N}$, a formula φ, a valuation v, and a sequence $\bar{z}$ of variables with $\bar{z} \subseteq \mathit{fv}(\varphi)$, we define the set

$$[\![\varphi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}_{\bar{z}, v} := \{\bar{d} \in \mathbb{D}^{|\bar{z}|} \mid (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{z} \mapsto \bar{d}], i) \models \varphi\}\,.$$

We drop the superscript when it is clear from the context. We drop the subscript when $\bar{z} = \bar{\mathit{fv}}(\varphi)$. Note that in this case the valuation v is irrelevant and $[\![\varphi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$ denotes the set of satisfying elements of φ at time point i in $(\bar{\mathcal{D}}, \bar{\tau})$.

With this notation, we illustrate the semantics for aggregation formulas in the case where we aggregate over a variable. We use the same notation as in

x	y	g
1	b	a
2	b	a
1	c	a
4	c	b

x	y	g
$\boxed{1}$	b	a
$\boxed{2}$	b	
$\boxed{1}$	c	
$\boxed{4}$	c	b

Fig. 1. Relation $p^{\mathfrak{D}_0}$ from Example 2. The two boxes represent the multi-set M for the two valuations v_1 and v_2, respectively.

Definition 1. In particular, consider a formula $\varphi = [\omega_x\, \bar{z}.\, \psi](y; \bar{g})$, with $x \in \mathsf{V}$, and a valuation v. Note that v (and thus also $v[\bar{z} \mapsto \bar{d}]$) fixes the values of the variables in $\bar{g}$ because these are free in φ. The multi-set M is as follows. If $x \notin \bar{g}$, then $M(a) = |\{\bar{d} \in [\![\varphi]\!]_{\bar{z},v} \mid d_j = a\}|$, for any $a \in \mathbb{D}$, where j is the index of x in $\bar{z}$. If $x \in \bar{g}$, then $M(v(x)) = |[\![\varphi]\!]_{\bar{z},v}|$ and $M(a) = 0$, for any $a \in \mathbb{D} \setminus \{v(x)\}$.

Example 2. Let $(\bar{\mathfrak{D}}, \bar{\tau})$ be a temporal structure over a signature with a ternary predicate symbol p with $p^{\mathfrak{D}_0} = \{(1, b, a), (2, b, a), (1, c, a), (4, c, b)\}$. Moreover, let φ be the formula $[\mathsf{SUM}_x\, x, y.\, p(x, y, g)](s; g)$ and $\bar{z} = (x, y)$. At time point 0, for a valuation v_1 with $v_1(g) = a$, we have $[\![p(x, y, g)]\!]_{\bar{z},v_1} = \{(1, b), (2, b), (1, c)\}$ and $M = \{\!|1, 2, 1|\!\}$. For a valuation v_2 with $v_2(g) = b$, we have $[\![p(x, y, g)]\!]_{\bar{z},v_2} = \{(4, c)\}$ and $M = \{\!|4|\!\}$. Finally, for a valuation v_3 with $v_3(g) \notin \{a, b\}$, we have that $[\![p(x, y, g)]\!]_{\bar{z},v_3}$ and M are empty. So the formula φ is only satisfied under a valuation v with $v(s) = 4$ and either $v(g) = a$ or $v(g) = b$. Indeed, we have $[\![\varphi]\!] = \{(4, a), (4, b)\}$. The tables in Figure 1 illustrate this example. We obtain $[\![[\mathsf{SUM}_x\, y, g.\, p(x, y, g)](s; x)]\!] = \{(2, 1), (2, 2), (4, 4)\}$, if we group on the variable x instead of g and $[\![[\mathsf{SUM}_x\, x, y, g.\, p(x, y, g)](s)]\!] = \{(8)\}$, if we do not group.

Example 3. Consider the formula $\varphi = [\mathsf{SUM}_a\, a.\, \psi](s; u)$, where ψ is the formula $\blacklozenge_{[0,31)}\, \mathit{withdraw}(u, a)$. Let $(\bar{\mathfrak{D}}, \bar{\tau})$ be a temporal structure with the relations $\mathit{withdraw}^{\mathfrak{D}_0} = \{(\text{Alice}, 9), (\text{Alice}, 3)\}$ and $\mathit{withdraw}^{\mathfrak{D}_1} = \{(\text{Alice}, 3)\}$, and the timestamps $\tau_0 = 5$ and $\tau_1 = 8$. We have that $[\![\psi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},0)} = [\![\psi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},1)} = \{(\text{Alice}, 9), (\text{Alice}, 3)\}$ and therefore $[\![\varphi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},0)} = [\![\varphi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},1)} = \{(12, \text{Alice})\}$. Our semantics ignores the fact that the tuple $(\text{Alice}, 3)$ occurs at both time points 0 and 1. Note that the withdraw events do not have unique identifiers in this example.

To account for multiple occurrences of an event, we can attach to each event additional information to make it unique. For example, assume we have a predicate symbol *ts* at hand that records the timestamp at each time point, i.e., $\mathit{ts}^{\mathfrak{D}_i} = \{\tau_i\}$, for $i \in \mathbb{N}$. For the formula $\varphi' = [\mathsf{SUM}_a\, a.\, \psi'](s; u)$ with $\psi' = \blacklozenge_{[0,31)}\, \mathit{withdraw}(u, a) \wedge \mathit{ts}(t)$, we have that $[\![\varphi']\!]^{(\bar{\mathfrak{D}},\bar{\tau},0)} = \{(12, \text{Alice})\}$ and $[\![\varphi']\!]^{(\bar{\mathfrak{D}},\bar{\tau},1)} = \{(15, \text{Alice})\}$ because $[\![\psi']\!]^{(\bar{\mathfrak{D}},\bar{\tau},0)} = \{(\text{Alice}, 9, 5), (\text{Alice}, 3, 5)\}$ while $[\![\psi']\!]^{(\bar{\mathfrak{D}},\bar{\tau},1)} = \{(\text{Alice}, 9, 5), (\text{Alice}, 3, 5), (\text{Alice}, 3, 8)\}$. To further distinguish between withdraw events at time points with equal timestamps, we would need additional information about the occurrence of an event, e.g.,

$$\frac{p \in \mathsf{R}_f \quad x_1,\dots,x_{\iota(p)} \in \mathsf{V} \text{ are pairwise distinct}}{p(x_1,\dots,x_{\iota(p)}) \in \mathcal{F}}\ \mathsf{FLX}$$

$$\frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)}{\varphi \wedge p(t_1,\dots,t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}_{\wedge} \qquad \frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)}{\varphi \wedge \neg p(t_1,\dots,t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}_{\wedge\neg}$$

$$\frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1, i\neq j}^{\iota(p)} fv(t_i) \subseteq fv(\varphi) \quad t_j \in \mathsf{V} \quad j \in H_p}{\varphi \wedge p(t_1,\dots,t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}'_{\wedge}$$

$$\frac{\varphi, \psi \in \mathcal{F}}{\varphi \wedge \psi \in \mathcal{F}}\ \mathsf{GEN}_{\wedge} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\psi) \subseteq fv(\varphi)}{\varphi \wedge \neg\psi \in \mathcal{F}}\ \mathsf{GEN}_{\wedge\neg} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\psi) = fv(\varphi)}{\varphi \vee \psi \in \mathcal{F}}\ \mathsf{GEN}_{\vee}$$

$$\frac{\varphi \in \mathcal{F}}{\exists x.\, \varphi \in \mathcal{F}}\ \mathsf{GEN}_{\exists} \qquad \frac{\varphi \in \mathcal{F}}{\bullet_I\, \varphi \in \mathcal{F}}\ \mathsf{GEN}_{\bullet} \qquad \frac{\varphi \in \mathcal{F}}{[\omega_t\, \bar{z}.\, \varphi](y; \bar{g}) \in \mathcal{F}}\ \mathsf{GEN}_{\omega}$$

$$\frac{\varphi, \psi \in \mathcal{F} \quad fv(\varphi) \subseteq fv(\psi)}{\varphi\, \mathsf{S}_I\, \psi \in \mathcal{F}}\ \mathsf{GEN}_{\mathsf{S}} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\varphi) \subseteq fv(\psi)}{\neg\varphi\, \mathsf{S}_I\, \psi \in \mathcal{F}}\ \mathsf{GEN}_{\neg\mathsf{S}}$$

Fig. 2. The derivation rules defining the fragment $\mathcal{F}$ of monitorable formulas

information obtained from a predicate symbol *tpts* that is interpreted as $tpts^{\mathcal{D}_i} = \{(i, \tau_i)\}$, for $i \in \mathbb{N}$.

The multiplicity issue illustrated by Example 3 also appears in databases. SQL is based on a multi-set semantics and one uses the DISTINCT keyword to switch to a set-based semantics. However, it is problematic to define a multi-set semantics for first-order logic, i.e., one that attaches a multiplicity to a tuple $\bar{d} \in \mathbb{D}^{|fv(\varphi)|}$ for how often it satisfies the formula φ instead of a Boolean value. For instance, there are several ways to define a multi-set semantics for disjunction: the multiplicity of $\bar{d}$ for $\psi \vee \psi'$ can be either the maximum or the sum of the multiplicities of $\bar{d}$ for ψ and ψ'. Depending on the choice, standard logical laws become invalid, namely, distributivity of existential quantification or conjunction over disjunction. Defining a multi-set semantics for negation is even more problematic.

3 Monitoring Algorithm

We assume that policies are of the form $\Box\, \forall \bar{x}.\, \varphi$, where φ is an MFOTL$_{\Omega}$ formula and $\bar{x}$ is the sequence of free variables of φ. The policy requires that $\forall \bar{x}.\, \varphi$ holds at every time point in temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$. In the following, we assume that $(\bar{\mathcal{D}}, \bar{\tau})$ is a *temporal database*, i.e., (1) the domain $\mathbb{D}$ is countably infinite, (2) the relation $p^{\mathcal{D}_i}$ is finite, for each $p \in \mathsf{R}_f$ and $i \in \mathbb{N}$, (3) $p^{\mathcal{D}}$ is a recursive relation, for each $p \in \mathsf{R}_r$, and (4) $f^{\mathcal{D}}$ is computable, for each $f \in \mathsf{F}$. We also assume that the aggregation operators in Ω are computable functions on finite multi-sets.

The inputs of our monitoring algorithm are a formula ψ, which is logically equivalent to $\neg\varphi$, and a temporal database $(\bar{\mathcal{D}}, \bar{\tau})$, which is processed iteratively. The algorithm outputs, again iteratively, the relation $[\![\psi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$, for each $i \geq 0$.

As ψ and $\neg\varphi$ are equivalent, the tuples in $[\![\psi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},i)}$ are the policy violations at time point i. Note that we drop the outermost quantifier as we are interested not only in whether the policy is violated. An instantiation of the free variables $\bar{x}$ that satisfies ψ provides additional information about the violations.

3.1 Monitorable Fragment

Not all formulas are effectively monitorable. Consider, for example, the policy $\Box\,\forall x.\,\forall y.\,p(x) \rightarrow q(x,y)$ with the formula $\psi = p(x) \wedge \neg q(x,y)$ that we use for monitoring. There are infinitely many violations for time points i with $p^{\mathfrak{D}_i} \neq \emptyset$, namely, any tuple $(a,b) \in \mathbb{D}^2 \setminus q^{\mathfrak{D}_i}$ with $a \in p^{\mathfrak{D}_i}$. In such a case, $[\![\psi]\!]^{(\bar{\mathfrak{D}},\bar{\tau},i)}$ is infinite and its elements cannot be enumerated in finite time. We define a fragment of MFOTL$_\Omega$ that guarantees finiteness. Furthermore, the set of violations at each time point can be effectively computed bottom-up over the formula structure. In the following, we treat the Boolean connective $\wedge$ as a primitive.

Definition 4. *The set $\mathcal{F}$ of* monitorable formulas *with respect to $(H_p)_{p\in\mathsf{R}_r}$ is defined by the rules given in Figure 2, where $H_p \subseteq \{1,\dots,\iota(p)\}$, for each $p \in \mathsf{R}_r$.*

Let ℓ be a label of a rule from Figure 2. We say that a formula $\varphi \in \mathcal{F}$ is of *kind* ℓ if there is a derivation tree for φ having as root a rule labeled by ℓ.

Before describing some of the rules, we first explain the meaning of the set H_p, for $p \in \mathsf{R}_r$ with arity k. The set H_p contains the indexes j for which we can determine the values of the variable x_j that satisfy $p(x_1,\dots,x_k)$, given that the values of the variables x_i with $i \neq j$ are fixed. Formally, given a temporal database $(\bar{\mathfrak{D}},\bar{\tau})$ and a rigid predicate symbol p of arity $k > 0$, we say that an index j, with $1 \leq j \leq k$, is *effective* for p if for any $\bar{a} \in \mathbb{D}^{k-1}$, the set $\{d \in \mathbb{D} \mid (a_1,\dots,a_{j-1},d,a_j,\dots,a_{k-1}) \in p^{\mathfrak{D}}\}$ is finite. For instance, for the rigid predicate $\approx$, the set of effective indexes is $H_\approx = \{1,2\}$. Similarly, for the rigid predicate $\prec_\mathbb{N}$, defined as $a \prec_\mathbb{N} b$ iff $a,b \in \mathbb{N}$ and $a < b$, we have $H_{\prec_\mathbb{N}} := \{1\}$.

We describe the intuition behind the first four rules in Figure 2. The meaning of the other rules should then be obvious. The first rule (FLX) requires that in an atomic formula $p(\bar{t})$ with $p \in \mathsf{R}_f$, the terms t_i are pairwise distinct variables. This formula is monitorable since we assume that p's interpretation is always a finite relation. For the rules ($\mathsf{RIG}_\wedge$) and ($\mathsf{RIG}_{\wedge\neg}$), consider formulas of the form $\varphi \wedge p(\bar{t})$ and $\varphi \wedge \neg p(\bar{t})$ with $p \in \mathsf{R}_r$ and $\bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)$. In both cases, the second conjunct restricts on the tuples satisfying φ. A simple example is the formula $p(x,y) \wedge x + 1 \approx y$. If φ is monitorable, such a formula is also monitorable as its evaluation can be performed by filtering out the tuples in $[\![\varphi]\!]$ that do not satisfy the second conjunct. The rule ($\mathsf{RIG}'_\wedge$) treats the case where one of the terms t_i is a variable that does not appear in φ. We require here that the index j is effective, so that the values of this variable are determined by the values of the other variables, which themselves are given by the tuples in $[\![\varphi]\!]$. An example is the formula $p(x,y) \wedge z \approx x + y$. The required conditions on t_j are necessary. If j is not effective, then we cannot guarantee finiteness. Consider, e.g., the formula $q(x) \wedge x \not\approx y$. If t_j is neither a variable nor a constant, then we must solve equations to determine the value of the variable that does not occur in φ. Consider, e.g., the formula $q(x) \wedge x \approx y \cdot y$.

The rule (FLX) may seem very restrictive. However, one can often rewrite a formula of the form $p(t_1, \dots, t_n)$ with $p \in \mathsf{R}_f$ into an equivalent formula in $\mathcal{F}$. For instance, $p(x+1, x)$ can be rewritten to $\exists y.\, p(y, x) \wedge x+1 \approx y$. Alternatively, one can add additional rules that handle such cases directly.

We now show that φ's membership in $\mathcal{F}$ guarantees the finiteness of $[\![\varphi]\!]$.

Lemma 5. *Let $(\bar{\mathfrak{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$ a time point, φ a formula, and H_p the set of effective indexes for p, for each $p \in \mathsf{R}_r$. If φ is a monitorable formula with respect to $(H_p)_{p \in \mathsf{R}_r}$, then $[\![\varphi]\!]^{(\bar{\mathfrak{D}}, \bar{\tau}, i)}$ is finite.*

There are formulas like $(x \approx y) \,\mathsf{S}\, p(x, y)$ that describe finite relations but are not in $\mathcal{F}$. However, the policies considered in this paper all fall into the monitorable fragment. They follow the common pattern $\Box \forall \bar{x}, \bar{y}.\, \varphi(\bar{x}, \bar{y}) \wedge c(\bar{x}, \bar{y}) \to \psi(\bar{y}) \wedge c'(\bar{y})$, where c and c' represent restrictions, i.e., formulas of the form $r(\bar{t})$ and $\neg r(\bar{t})$ with $r \in \mathsf{R}_r$. The formula to be monitored, i.e., $\varphi(\bar{x}, \bar{y}) \wedge c(\bar{x}, \bar{y}) \wedge \neg(\psi(\bar{y}) \wedge c'(\bar{y}))$ is in $\mathcal{F}$ if φ and ψ are in $\mathcal{F}$, and c, c' satisfy the conditions of the (RIG) rules.

Finiteness can also be guaranteed by semantic notions like domain independence or syntactic notions like range restriction, see, e.g., [1] and also [7, 12] for a generalization of these notions to a temporal setting. If we restrict ourselves to MFOTL without future operators, the range restricted fragment in [7] is more general than the fragment $\mathcal{F}$. This is because, in contrast to the rules in Figure 2, range restrictions are not local conditions, that is, conditions that only relate formulas with their direct subformulas. However, the evaluation procedures in [1,7,12] also work in a bottom-up recursive manner. So one still must rewrite the formulas to evaluate them bottom-up. No rewriting is needed for formulas in $\mathcal{F}$. Furthermore, the fragment ensures that aggregation operators are always applied to *finite* multi-sets. Thus, for any $\varphi \in \mathcal{F}$, the element $\bot \in \mathbb{D}$ never appears in a tuple of $[\![\varphi]\!]$, provided that $p^{\mathfrak{D}_i} \subseteq D^{\iota(p)}$ and $f^{\mathfrak{D}}(\bar{a}) \in D$, for every $p \in \mathsf{R}$, $f \in \mathsf{F}$, $i \in \mathbb{N}$, and $\bar{a} \in D^{\iota(f)}$, where $D = \mathbb{D} \setminus \{\bot\}$.

3.2 Extended Relational Algebra Operators

Our monitoring algorithm is based on a translation of MFOTL$_\Omega$ formulas in $\mathcal{F}$ to extended relational algebra expressions. The translation uses equalities, which we present in Section 3.3, that extend the standard ones [1] expressing the relationship between first-order logic (without function symbols) and relational algebra to function symbols, temporal operators, and group-by operators. In this section, we introduce the extended relational algebra operators.

We start by defining constraints. We assume a given infinite set of variables $Z = \{z_1, z_2, \dots\} \subseteq \mathsf{V}$, ordered by their indices. A *constraint* is a formula $r(t_1, \dots, t_n)$ or its negation, where r is a rigid predicate symbol of arity n and the t_is are constraint terms, i.e., terms with variables in Z. We assume that for each domain element $d \in \mathbb{D}$, there is a corresponding constant, denoted also by d. A tuple $(a_1, \dots, a_k)$ satisfies the constraint $r(t_1, \dots, t_n)$ iff $\bigcup_{i=1}^{n} fv(t_i) \subseteq \{z_1, \dots, z_k\}$

and $(v(t_1),\dots,v(t_n)) \in r^{\mathcal{D}}$, where v is a valuation with $v(z_i) = a_i$, for all $i \in \{1,\dots,k\}$. Satisfaction for a constraint $\neg r(t_1,\dots,t_n)$ is defined similarly.

In the following, let C be a set of constraints, $A \subseteq \mathbb{D}^m$, and $B \subseteq \mathbb{D}^n$. The *selection* of A with respect to C is the m-ary relation

$$\sigma_C(A) := \{\bar{a} \in A \mid \bar{a} \text{ satisfies all constraints in } C\}\,.$$

The integer i is a *column* in A if $1 \le i \le m$. Let $\bar{s} = (s_1, s_2, \dots, s_k)$ be a sequence of $k \ge 0$ columns in A. The *projection* of A on $\bar{s}$ is the k-ary relation

$$\pi_{\bar{s}}(A) := \big\{(a_{s_1}, a_{s_2}, \dots, a_{s_k}) \in \mathbb{D}^k \,\big|\, (a_1, a_2, \dots, a_m) \in A\big\}\,.$$

Let $\bar{s}$ be a sequence of columns in $A \times B$. The *join* and the *antijoin* of A and B with respect to $\bar{s}$ and C is defined as

$$A \bowtie_{\bar{s},C} B := (\pi_{\bar{s}} \circ \sigma_C)(A \times B) \qquad \text{and} \qquad A \rhd_{\bar{s},C} B := A \setminus (A \bowtie_{\bar{s},C} B)\,.$$

Let ω be an operator in Ω, G a set of $k \ge 0$ columns in A, and t a constraint term. The ω-*aggregate* of A on t with grouping by G is the $(k+1)$-ary relation

$$\omega_t^G(A) := \big\{(b, \bar{a}) \,\big|\, \bar{a} = (a_{g_1}, a_{g_2}, \dots, a_{g_k}) \in \pi_{\bar{g}}(A) \text{ and } b = \omega(M_{\bar{a}})\big\}\,.$$

Here $\bar{g} = (g_1, g_2, \dots, g_k)$ is the maximal subsequence of $(1, 2, \dots, m)$ such that $g_i \in G$, for $1 \le i \le k$, and $M_{\bar{a}} : \mathbb{D}^{m-k} \to \mathbb{N}$ is the finite multi-set

$$M_{\bar{a}} := \{\!\!\{ (\pi_{\bar{h}} \circ \sigma_{\{d \approx t\} \cup D})(A) \,\big|\, d \in \mathbb{D} \}\!\!\}\,,$$

where $\bar{h}$ is the maximal subsequence of $(1, 2, \dots, m)$ with no element in G and $D := \{a_i \approx z_{g_i} \mid 1 \le i \le k\}$.

3.3 Translation to Extended Relational Algebra

Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$, and $\varphi \in \mathcal{F}$. We express $\llbracket\varphi\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)}$ in terms of the generalized relational algebra operators defined in Section 3.2.

Kind (FLX). This case is straightforward: for a predicate symbol $p \in \mathsf{R}_f$ of arity n and pairwise distinct variables $x_1, \dots, x_n \in \mathsf{V}$,

$$\llbracket p(x_1, \dots, x_n)\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)} = p^{\mathcal{D}_i}\,.$$

Kind ($\mathsf{RIG}_\wedge$). Let $\psi \wedge p(t_1, \dots, t_n)$ be a formula of kind ($\mathsf{RIG}_\wedge$). Then

$$\llbracket\psi \wedge p(t_1, \dots, t_n)\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)} = \sigma_{\{p(\theta(t_1),\dots,\theta(t_n))\}}\big(\llbracket\psi\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)}\big)\,,$$

where the substitution $\theta : fv(\psi) \to \{z_1, \dots, z_{|fv(\psi)|}\}$ is given by $\theta(x) = z_j$ with j the index of x in $\bar{fv}(\psi)$. For instance, if $\varphi \in \mathcal{F}$ is the formula $\psi(x, y) \wedge (x - y)\ mod\ 2 \approx 0$ then $\llbracket\varphi\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)} = \sigma_{\{(z_1 - z_2)\ mod\ 2 \approx 0\}}\llbracket\psi\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)}$.

Kind ($\mathsf{GEN_S}$). Let $\psi \,\mathsf{S}_I\, \psi'$ be a formula of kind ($\mathsf{GEN_S}$) with $\bar{fv}(\psi) = (y_1, \dots, y_n)$ and $\bar{fv}(\psi') = (y'_1, \dots, y'_\ell)$. Then

$$\llbracket\psi \,\mathsf{S}_I\, \psi'\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup_{j \in \{i' \mid i' \le i,\ \tau_i - \tau_{i'} \in I\}} \Big(\llbracket\psi'\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \big(\bigcap_{k \in \{j+1,\dots,i\}} \llbracket\psi\rrbracket^{(\bar{\mathcal{D}},\bar{\tau},k)}\big)\Big)\,,$$

where (a) $\bar{s} = (1, \dots, n, n + i_1, \dots, n + i_\ell)$ with i_j such that $(i_1, \dots, i_\ell)$ is the maximal subsequence of $(1, \dots, \ell)$ with $y'_{i_j} \notin fv(\psi)$ and (b) $C = \{z_j \approx z_{n+h} \mid y_j = y'_h,\ 1 \le j \le n, \text{ and } 1 \le h \le \ell\}$. For instance, for $\bar{fv}(\psi) = (x, y, z)$ and $\bar{fv}(\psi') = (z, z', x)$, we have $\bar{s} = (1, 2, 3, 5)$ and $C = \{z_1 \approx z_6, z_3 \approx z_4\}$.

Kind (GEN_ω). Let $[\omega_t\, \bar{z}'.\, \psi](y; \bar{g})$ be a formula of kind (GEN_ω). It holds that

$$[\![[\omega_t\, \bar{z}'.\, \psi](y; \bar{g})]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \omega^{G}_{\theta(t)}\big([\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\big)\,,$$

where $\overline{\mathit{fv}}(\psi) = (y_1, \dots, y_n)$, for some $n \geq 0$, $G = \{i \mid y_i \in \bar{g}\}$, and $\theta : \mathit{fv}(\psi) \to \{z_1, \dots, z_n\}$ is given by $\theta(x) = z_j$ with j being the index of x in $\overline{\mathit{fv}}(\psi)$. For instance, for $[\mathsf{SUM}_{x+y}\, x, y.\, p(x, y, z)](s; z)$, we have $G = \{3\}$ and $\theta(t) = z_1 + z_2$.

Other kinds. The case for $(\mathsf{RIG}_{\wedge\neg})$ is similar to the one for $(\mathsf{RIG}_{\wedge})$. The cases for $(\mathsf{GEN}_{\wedge})$, $(\mathsf{GEN}_{\wedge\neg})$, and $(\mathsf{GEN}_{\neg\mathsf{S}})$ are similar to the one for $(\mathsf{GEN}_{\mathsf{S}})$. The cases for $(\mathsf{GEN}_{\wedge\neg})$ and $(\mathsf{GEN}_{\neg\mathsf{S}})$ use the antijoin instead of the join. The cases for $(\mathsf{GEN}_{\vee})$, $(\mathsf{GEN}_{\exists})$, $(\mathsf{GEN}_{\bullet})$ are obvious. Additional details are in the appendix of the full version of the paper available at the authors' web pages.

3.4 Algorithmic Realization

Our monitoring algorithm for MFOTL_Ω is inspired by those in [7,8,11]. We only sketch it here. Further details are given in the appendix.

For a formula $\psi \in \mathcal{F}$, the algorithm iteratively processes the temporal database $(\bar{\mathcal{D}}, \bar{\tau})$. At each time point i, it calls the procedure eval to compute $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$. The input of eval at time point i is the formula ψ, the time point i with its timestamp τ_i, and the interpretations of the flexible predicate symbols, i.e., $r^{\mathcal{D}_i}$, for each $r \in \mathsf{R}_f$. Note that $\bar{\mathcal{D}}$'s domain and the interpretations of the rigid predicate symbols and the function symbols, including the constants, do not change over time. We assume that they are fixed in advance.

The computation of $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$ is by recursion over ψ's formula structure and is based on the equalities in Section 3.3. Note that extended relational algebra operators have standard, efficient implementations [17], which can be used to evaluate the expressions on the right-hand side of the equalities from Section 3.3.

To accelerate the computation of $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$, the monitoring algorithm maintains state for each temporal subformula, storing previously computed intermediate results. The monitor's state is initialized by the procedure init and updated in each iteration by the procedure eval. For subformulas of the form $\bullet_I\, \psi'$, we store at time point $i > 0$, the tuples that satisfy ψ' at time-point $i-1$, i.e., the relation $[\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i-1)}$. For formulas of the form $\psi_1\, \mathsf{S}_{[a,b)}\, \psi_2$, we store at time point i, the list of relations $[\![\psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \big(\bigcap_{j<k\leq i} [\![\psi_1]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)}\big)$ with $j \leq i$ such that $\tau_i - \tau_j < b$, where $\bar{s}$ and C are defined as in Section 3.3. By storing these relations, the subformulas ψ', ψ_1, and ψ_2 need not be evaluated again at time points $j < i$ during the evaluation of ψ at time point i. Further optimizations are possible. For instance, one can store and reuse some of the intermediate relations used for computing the relation $[\![\psi_1\, \mathsf{S}_{[a,b)}\, \psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$ from the relations stored in the previously mentioned list. Also, when $a = 0$ and $b = \infty$, it is sufficient to store the resulting relation from the previous time point, as $[\![\psi_1\, \mathsf{S}\, \psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![\psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} \cup \big([\![\psi_1\, \mathsf{S}\, \psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i-1)} \bowtie [\![\psi_1]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\big)$.

Theorem 6. *Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$, and $\psi \in \mathcal{F}$. The procedure* $\mathsf{eval}(\psi,\, i,\, \tau_i,\, \Gamma_i)$ *returns the relation $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$, whenever* $\mathsf{init}(\psi)$, $\mathsf{eval}(\psi,\, 0,$

$$\Box\, \forall u.\, \forall s.\, [\mathsf{SUM}_a\, a, i.\, \blacklozenge_{[0,31)}\, \psi(u,a,i)](s;u) \to s \preceq 10000 \tag{P1}$$

$$\begin{aligned}\Box\, \forall u.\, \forall s.\, [\mathsf{SUM}_a\, a, i.\, \blacklozenge_{[0,31)}\, \psi(u,a,i)](s;u) \wedge (\neg \mathit{limit\_off}(u)\ \mathsf{S}\ \mathit{limit\_on}(u)) \to\\ s \preceq 10000\end{aligned} \tag{P2}$$

$$\begin{aligned}\Box\, \forall u.\, \forall s.\, \forall m.\, [\mathsf{AVG}_a\, a, i.\, \blacklozenge_{[0,91)}\, \psi(u,a,i)](s;u) \wedge\\ [\mathsf{MAX}_a\, a.\, \blacklozenge_{[0,8)}\, \mathit{withdraw}(u,a)](m;u) \to m \preceq 2 \cdot s\end{aligned} \tag{P3}$$

$$\Box\, \forall s.\, [\mathsf{AVG}_u\, u, c.\, [\mathsf{CNT}_i\, a, i.\, \blacklozenge_{[0,31)}\, \psi(u,a,i)](c;u)](s) \to s \preceq 150 \tag{P4}$$

$$\begin{aligned}\Box\, \forall u.\, \forall c.\, [\mathsf{CNT}_j\ v, p, j.\, [\mathsf{AVG}_a\, a, i.\, \blacklozenge_{[0,31)}\, \psi(u,a,i)](v;u) \wedge\\ \blacklozenge_{[0,31)}\, \psi(u,p,j) \wedge 2 \cdot v \prec p](c;u) \to c \preceq 5\end{aligned} \tag{P5}$$

Fig. 3. Policy formalizations, where $\psi(u,a,i)$ abbreviates $\mathit{withdraw}(u,a) \wedge \mathit{ts}(i)$.

τ_0, Γ_0), ..., $\mathsf{eval}(\psi,\ i-1,\ \tau_{i-1},\ \Gamma_{i-1})$ were called previously in this order, where $\Gamma_j = (p^{\mathcal{D}_j})_{p \in \mathsf{R}_f}$ is the family of interpretations of flexible predicates at j, for every time point $j \in \mathbb{N}$.

4 Experimental Evaluation

We compare our prototype implementation, which extends our monitoring tool MonPoly [5] for MFOTL, with the relational database PostgreSQL [22] and the stream-processing tool STREAM [2]. For our evaluation, we consider the following five policies. Figure 3 contains their MFOTL$_\Omega$ formalizations.

(P1) The sum of withdrawals of each user over the last 30 days does not exceed the limit of $10,000.

(P2) Similar to (P1), except that the withdrawals must not exceed $10,000 only when the flag for checking the limit is set.

(P3) The maximal withdrawal of each user over the last seven days must be at most be twice as large as the average of the user's withdrawals over the last 90 days.

(P4) The average of the number of withdrawals of all users over the last 30 days should be less than a given threshold of 150.

(P5) For each user, the number of peaks over the last 30 days does not exceed a threshold of 5, where a peak is a value at least twice the average over some time window.

Note that in the formalization of the policy (P2), the event $\mathit{limit\_on}(u)$ sets the limit flag for the user u, while $\mathit{limit\_off}(u)$ unsets it.

We use synthetically generated logs[1] with different time spans (in days). The logs contain withdraw events from 500 users, except for (P5), for which we consider only 100 users. Each user makes on average five withdrawals per day. Table 1 shows the running times of the three tools on a standard desktop computer with

[1] Our prototype, the formulas, and the input data are available as an archive at `https://projects.developer.nokia.com/MonPoly/files/rv13-experiments.tgz`.

Table 1. Running times (STREAM / MonPoly extension / PostgreSQL) in seconds

time span policy	400	800	1200	1600	2000
(P1)	8 / 9 / 76	9 / 19 / 279	11 / 29 / 610	12 / 39 / 1065	14 / 48 / 1650
(P2)	21 / 10 / 247	23 / 20 / 1646	24 / 30 / 5233	26 / 40 / 11989	28 / 50 / 23260
(P3)	† / 22 / 168	† / 44 / 604	† / 66 / 1230	† / 88 / 2251	† / 110 / 3458
(P4)	12 / 9 / 75	15 / 19 / 280	15 / 29 / 612	17 / 38 / 1068	19 / 48 / 1650
(P5)	24 / 76 / 83	33 / 157 / 337	41 / 234 / 745	49 / 313 / 1351	59 / 395 / 2099

8 GB of RAM and an Intel Core i5 CPU with 2.67 GHz. The SQL queries for PostgreSQL and the CQL queries for STREAM were manually obtained from the corresponding MFOTL_Ω formulas. For the considered policies and logs, the semantic differences between the languages are not substantial. In particular, the tools output the same violations. PostgreSQL's running times only account for the query evaluation, performed once per log file, and not for populating the database. For MAX aggregations, STREAM aborts with a runtime error, and we mark this with the symbol †.

Note that the formulas in Figure 3 vary in their complexity: e.g., they contain different numbers of aggregations and temporal operators, with time windows of different sizes. STREAM and our tool scale linearly on these examples with respect to the time spans of the logs. This is not the case for PostgreSQL. Overall, our tool's performance is between STREAM's and PostgreSQL's on these examples.

We first focus on the performance of our tool. (P2) is only slightly slower to monitor than (P1) because the relations for the additional subformula are not large: they contain around 50 tuples, as the limit flag is toggled for each user, on average, every 10 days. (P3) takes longer to monitor for two reasons. First, it contains a significantly larger time window. Second, the join of two relations is computed, which is also the case for (P5). For (P3), the two input relations and the output relation each have size n, where n is the number of users. For (P5), the size of the input relations is approximately $31mn$, where m is the average number of withdrawals per day of a user, while the output relation is approximately of size 31^2m^2n. This explains why (P5) takes longer to monitor than (P3). Since aggregating over a relation does not increase its size, the nesting of aggregation operators has only a minor impact on the running times, compare (P1) and (P4).

PostgreSQL performs worst in these experiments. This is not surprising as PostgreSQL is not designed for this application domain. In particular, PostgreSQL has no support for temporal reasoning and we must treat time as just another data value. In more detail, we load log files into database tables that have two additional attributes to represent the time point and the timestamp of an event occurrence, and we adapt the standard embedding of temporal logic into first-order logic to represent MFOTL_Ω formulas as SQL queries. Treating time as data has the following disadvantages. First, it is not suited for online processing of events: query evaluation does not scale, because the query must be reevaluated on the entire database each time new events are added. Second, even for offline processing (as done in our experiments), the query evaluation procedure does not take advantage of the temporal ordering of events. This deficiency is most evident when evaluating the SQL query for (P2).

In contrast to PostgreSQL, STREAM is designed for online event processing. However, temporal reasoning in STREAM is limited. In particular, CQL's only temporal construct collects all event tuples within a specified time range relative to the current time. It roughly corresponds[2] to the $\blacklozenge_I$ operator in MFOTL$_\Omega$, where I is of the form $[0, t)$ with $t \in \mathbb{N} \cup \{\infty\}$. We cannot select only tuples from a time window that is strictly in the past. It is therefore not clear how to handle temporal properties of the form $\blacklozenge_I \varphi$ with $0 \notin I$. It is also not clear how to handle nested temporal operators as this also requires handling time windows that do not contain the current time point. Finally, it is also not obvious how to check that certain event patterns happen at every time point in a given time window. Consider, e.g., the policy stating that a user may not make large withdrawals if he is continuously in an over-withdrawn state during the last seven days. In MFOTL$_\Omega$, the policy is naturally expressed as

$$\Box\, \forall u.\, \big(\blacksquare_{[0,8)}(\neg \textit{out-debt}(u)\ \mathsf{S}\ \textit{in-debt}(u))\big) \to \neg \exists a.\, \textit{withdraw}(u, a) \wedge a \succ 1000\,.$$

Note that the subformula $\neg \textit{out-debt}(u)\ \mathsf{S}\ \textit{in-debt}(u)$ can be encoded in CQL by requiring for each user u that at the current time the total number of $\textit{out-debt}(u)$ events is smaller than the total number of $\textit{in-debt}(u)$ events. We have used such an encoding for (P2). We remark that the addition to (P1) of the since subformula in (P2) has a larger impact on STREAM's performance than on our tool.

While MFOTL$_\Omega$ has a richer tool set than CQL to express temporal patterns, STREAM's performance is consistently better than our tool's. Nevertheless, the differences are not as large as one might expect for a prototype implementation. Our prototype has not yet been systematically optimized. We expect substantial performance improvements by carefully adapting data structures and query evaluation techniques used in databases and stream processing.

5 Conclusion

Existing logic-based policy monitoring approaches offer little support for aggregations. To rectify this shortcoming we extended metric first-order temporal logic with expressive SQL-like aggregation operators and presented a monitoring algorithm for this language. Our experimental results for a prototype implementation of the algorithm are promising. The prototype's performance is in the reach of optimized stream-processing tools, despite its richer input language and its lack of systematic optimization. As future work, we will investigate performance

[2] CQL's time model differs from that of MFOTL$_\Omega$. In CQL, there is no notion of time point and query evaluation is performed for each timestamp $\tau \in \mathbb{N}$. Furthermore, CQL has a multi-set semantics.

optimizations for our monitor. In general, it remains to be seen how logic-based monitoring approaches can benefit from the techniques used in stream processing.

Acknowledgements. This work was partially supported by the Zurich Information Security and Privacy Center. It represents the views of the authors.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
2. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford stream data manager. IEEE Data Eng. Bull. 26(1), 19–26 (2003)
3. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–144 (2006)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
5. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: Monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 360–364. Springer, Heidelberg (2012)
6. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng., (to appear), `http://doi.ieeecomputersociety.org/10.1109/TSE.2013.18`
7. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: Proceedings of the 28th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008). Leibniz International Proceedings in Informatics (LIPIcs), vol. 2, pp. 49–60 (2008)
8. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer, Heidelberg (2012)
9. Bauer, A., Goré, R., Tiu, A.: A first-order policy language for history-based transaction monitoring. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 96–111. Springer, Heidelberg (2009)
10. Bianculli, D., Ghezzi, C., San Pietro, P.: The tale of SOLOIST: A specification language for service compositions interactions. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 55–72. Springer, Heidelberg (2013)
11. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
12. Chomicki, J., Toman, D., Böhlen, M.H.: Querying ATSQL databases with temporal logic. ACM Trans. Database Syst. 26(2), 145–178 (2001)
13. Colombo, C., Gauci, A., Pace, G.J.: LarvaStat: Monitoring of statistical properties. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 480–484. Springer, Heidelberg (2010)
14. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: A stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 647–651 (2003)

15. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), pp. 166–174 (2005)
16. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. Form. Method. Syst. Des. 27(3), 253–274 (2005)
17. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems: The complete book. Pearson Education (2009)
18. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. 5(2), 192–206 (2012)
19. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. J. ACM 48(4), 880–907 (2001)
20. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
21. Owe, O.: Partial logics reconsidered: A conservative approach. Form. Asp. Comput. 5(3), 208–223 (1993)
22. PostgreSQL Global Development Group. PostgreSQL, Version 9.1.4 (2012), `http://www.postgresql.org/`
23. Sistla, A.P., Wolfson, O.: Temporal conditions and integrity constraints in active database systems. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 269–280 (1995)

A Appendix

The pseudo-code of the procedures init and eval is given in Figure 4. Our pseudo-code is written in a functional-programming style with pattern matching. The symbol $\langle\rangle$ denotes the empty sequence, $+\!\!+$ sequence concatenation, and $h :: L$ the sequence with head h and tail L.

We describe the eval procedure in the following in more detail. The cases correspond to the rules defining the set of monitorable formulas. The pseudo-code for the cases corresponding to non-temporal connectives follows closely the equalities given in Section 3.3 and also given in the appendix of the full version of the paper. The predicates $\mathsf{kind\_rig}$ and $\mathsf{kind\_rig}$' check whether the input formula φ is indeed of the intended kind. The $\mathsf{get\_info\_}*$ procedures return the parameters used by the corresponding relational algebra operators. For instance, $\mathsf{get\_info\_rig}$ returns the singleton set consisting of the constraint corresponding to the restrictions $p(\bar{t})$ or $\neg p(\bar{t})$. Similarly, $\mathsf{get\_info\_rig}$' returns the effective index corresponding to the only variable that appears only in the right conjunct of φ. The procedure $\mathsf{reval}(p, k, \bar{a})$ returns the set $\{d \in \mathbb{D} \mid (a_1, \dots, a_{k-1}, d, a_k, \dots, a_{n-1}) \in p^{\mathcal{D}}\}$, for any $\bar{a} \in \mathbb{D}^{n-1}$, where n is the arity of the rigid predicate symbol p.

The case for the formulas of the form $\bullet_I\, \psi$ is straightforward. We recursively evaluate the subformula ψ, we update the state, and we return the relation resulting from the evaluation of ψ at the previous time point, provided that the temporal constraint is satisfied. Otherwise we return the empty relation.

The case for the formulas φ of the form $\psi \mathsf{S}_I\, \psi'$ or $\neg\psi \mathsf{S}_I\, \psi'$ is more involved. It is mainly handled by the sub-procedure $\mathsf{eval\_since}$, given in Figure 5. The notation $\lambda x.f(x)$ denotes a function f. For the clarity of the presentation, we

```
proc init($\varphi$)
  for each $\psi \in \mathsf{sf}(\varphi)$ with $\psi = \psi\ \mathsf{S}_I\ \psi'$ do
    $L_\psi \leftarrow \langle\rangle$
  for each $\psi \in \mathsf{sf}(\varphi)$ with $\psi = \bullet_I\ \psi'$ do
    $A_\psi \leftarrow \emptyset$
    $\tau_\psi \leftarrow 0$


proc eval($\varphi$, $i$, $\tau$, $\Gamma$)
  case $\varphi = p(x_1, \ldots, x_n)$
    return $\Gamma_p$

  case $\varphi = \psi \wedge p(t_1, \ldots, t_n)$ & kind_rig($\varphi$)
  case $\varphi = \psi \wedge \neg p(t_1, \ldots, t_n)$ & kind_rig($\varphi$)
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $C \leftarrow$ get_info_rig($\varphi$)
    return $\sigma_C(A)$

  case $\varphi = \psi \wedge p(t_1, \ldots, t_n)$ & kind_rig'($\varphi$)
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $k \leftarrow$ get_info_rig'($\varphi$)
    $R \leftarrow \emptyset$
    for each $\bar{a} \in A$
      $R \leftarrow R \cup$ reval($p, k, \bar{a}$)
    return $R$

  case $\varphi = \psi \wedge \neg\psi'$
  case $\varphi = \psi \wedge \psi'$
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $A' \leftarrow$ eval($\psi'$, $i$, $\tau$, $\Gamma$)
    $C, \bar{s} \leftarrow$ get_info_and($\varphi$)
    if $\varphi = \psi \wedge \psi'$ then
      return $A \bowtie_{C,\bar{s}} B$
    else
      return $A \rhd_{C,\bar{s}} B$
```

```
  case $\varphi = \psi \vee \psi'$
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $A' \leftarrow$ eval($\psi'$, $i$, $\tau$, $\Gamma$)
    return $A \cup A'$

  case $\varphi = \exists \bar{x}.\, \psi$
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $\bar{s} \leftarrow$ get_info_exists($\varphi$)
    return $\pi_{\bar{s}}(A)$

  case $\varphi = [\omega_t\ \bar{z}.\, \psi](y; \bar{g})$
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $H, t' \leftarrow$ get_info_agg($\varphi$)
    return $\omega^H_{t'}(A)$

  case $\varphi = \bullet_I\ \psi$
    $A' \leftarrow A_\varphi$
    $A_\varphi \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $\tau' \leftarrow \tau_\varphi$
    $\tau_\varphi \leftarrow \tau$
    if $i > 0$ and $(\tau - \tau') \in I$ then
      return $A'$
    else
      return $\emptyset$

  case $\varphi = \neg\psi\ \mathsf{S}_I\ \psi'$
  case $\varphi = \psi\ \mathsf{S}_I\ \psi'$
    $A \leftarrow$ eval($\psi$, $i$, $\tau$, $\Gamma$)
    $A' \leftarrow$ eval($\psi'$, $i$, $\tau$, $\Gamma$)
    return eval_since($\varphi$, $\tau$, $A$, $A'$)
```

Fig. 4. The init and eval procedures

```
proc eval_since($\varphi$, $\tau$, $A$, $A'$)
  b $\leftarrow$ interval_right_margin($\varphi$)
  drop_old($L_\varphi$, $b$, $\tau$)
  $C, \bar{s} \leftarrow$ get_info_and($\varphi$)
  case $\varphi = \neg\psi\ \mathsf{S}_I\ \psi'$ then
    $f \leftarrow \lambda B.B \rhd_{\bar{s},C} A$
  case $\varphi = \psi\ \mathsf{S}_I\ \psi'$ then
    $f \leftarrow \lambda B.B \bowtie_{\bar{s},C} A$
  $g \leftarrow \lambda(\kappa, B).(\kappa, f(B))$
  $L_\varphi \leftarrow$ map($g$, $L_\varphi$)
  $L_\varphi \leftarrow L_\varphi +\!\!+ \langle(\tau, A')\rangle$
  return fold_left(aux_since, $\emptyset$, $L_\varphi$)
```

```
proc drop_old($L$, $b$, $\tau$)
  case $L = \langle\rangle$
    return $\langle\rangle$
  case $L = (\kappa, B) :: L'$
    if $\tau - \kappa \geq b$ then
      return drop_old($L', b, \tau$)
    else return $L$


proc aux_since($R$, $(\kappa, B)$)
  if $(\tau - \kappa) \in I$ then return $R \cup B$
  else return $R$
```

Fig. 5. The eval_since procedure

assume that $\varphi = \psi\ \mathsf{S}_I\ \psi'$, the other case being similar. The evaluation of φ reflects the logical equivalence $\psi\ \mathsf{S}_I\ \psi' \equiv \bigvee_{d \in I} \psi\ \mathsf{S}_{[d,d]}\ \psi'$. Note that we abuse notation here, as the right-hand side is not necessarily a formula, because I may be infinite. The function interval_right_margin(φ) returns b, where $I = [a, b)$ for some $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$.

The state at time point i, that is, after the procedure $\mathsf{eval}(\varphi, i, \tau_i, \Gamma_i)$ has been executed, consists of the list L_φ of tuples (τ_j, R^i_j) ordered with j ascending, where j is such that $j \leq i$ and $\tau_i - \tau_j < b$ and where

$$R^i_j := [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \left(\bigcap\nolimits_{j<k\leq i} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)}\right),$$

with $\bar{s}$ and C defined as in Section 3.3. We have

$$[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup\nolimits_{j\leq i, \tau_i-\tau_j\in I} R^i_j.$$

The computation of this union is performed in the last line of the $\mathsf{eval\_since}$ procedure. Note that, in general, not all the relations R^i_j in the list L_φ are needed for the evaluation of φ at time point i. However, the relations R^i_j with j such that $\tau_i - \tau_j \notin I$, that is $\tau_i - \tau_j < a$, are stored for the evaluation of φ at future time points $i' > i$.

We now explain how the state is updated at time point i from the state at time point $i-1$. We first drop from the list L_φ the tuples that are not longer relevant. More precisely, we drop the tuples that have as first component a timestamp τ_j for which the distance to the current timestamp τ_i is too large with respect to the right margin of I. This is done by the procedure $\mathsf{drop\_old}$. Next, the state is updated according to the logical equivalence $\alpha \;\mathsf{S}\; \beta \;\equiv\; (\alpha \wedge \bullet\; (\alpha \;\mathsf{S}\; \beta)) \vee \beta$. This is done in two steps. First, we update each element of L_φ so that the tuples in the stored relations also satisfy ψ at the current time point i. This step corresponds to the conjunction in the above equivalence and it is performed by the map function. The update is based on the equality $R^i_j = R^{i-1}_j \bowtie_{\bar{s},C} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$. Note that the join distributes over the intersection. The second step, which corresponds to the disjunction in the above equivalence, consists of appending the tuple (τ_i, R^i_i) to L_φ. Note that $R^i_i = [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$.

Finally, we note that the proof of Theorem 6 follows the above presentation of the algorithm, and is done by induction using the lexicographic ordering on tuples $(i, |\varphi|)$, where $i \in \mathbb{N}$ and $|\varphi|$ denotes φ's size, defined as expected. Furthermore, the proof of Lemma 5 is straightforward. It follows by induction on the formula structure and from the equalities given in Section 3.3, as each relational algebra operator produces a finite relation when applied to finite relations.

From Propositional to First-Order Monitoring

Andreas Bauer[1,2], Jan-Christoph Küster[1,2], and Gil Vegliach[1]

[1] NICTA* Software Systems Research Group
[2] Australian National University

Abstract. The main purpose of this paper is to introduce a first-order temporal logic, $\mathrm{LTL}^{\mathrm{FO}}$, and a corresponding monitor construction based on a new type of automaton, called spawning automaton.

Specifically, we show that monitoring a specification in $\mathrm{LTL}^{\mathrm{FO}}$ boils down to an undecidable decision problem. The proof of this result revolves around specific ideas on what we consider a "proper" monitor. As these ideas are general, we outline them first in the setting of standard LTL, before lifting them to the setting of first-order logic and $\mathrm{LTL}^{\mathrm{FO}}$. Although due to the above result one cannot hope to obtain a complete monitor for $\mathrm{LTL}^{\mathrm{FO}}$, we prove the soundness of our automata-based construction and give experimental results from an implementation. These seem to substantiate our hypothesis that the automata-based construction leads to efficient runtime monitors whose size does not grow with increasing trace lengths (as is often observed in similar approaches). However, we also discuss formulae for which growth is unavoidable, irrespective of the chosen monitoring approach.

1 Introduction

In the area of runtime verification (cf. [17,16,12,8]), a *monitor* typically describes a device or program which is automatically generated from a formal specification capturing undesired (resp. desired) system behaviour. The monitor's task is to passively observe a running system in order to detect if the behavioural specification has been satisfied or violated by the observed system behaviour. While, arguably, the majority of runtime verification approaches are based on propositional logic (or expressively conservative parametric extensions thereof; cf. §6 for an overview), there exist works that have considered full first-order logic (cf. [16,5,4]). Monitoring first-order specifications has also gained prior attention in the database community, especially in the context of so called temporal triggers, which correspond to first-order temporal logic specifications that are evaluated wrt. a linear sequence of database updates (cf. [10,11,23]). Although the underlying logics are generally undecidable, the monitors in these works usually address decidable problems, such as "is the so far observed behaviour a violation of a given specification φ?" Additionally, in many approaches, φ must only ever be a safety or domain independent property for this problem to actually be decidable (cf. [10,4]), which can be ensured by syntactic restrictions on the input formula, for example.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 59–75, 2013.

As there exist many different ways in which a system can be monitored in this abstract sense, we are going to put forth very specific assumptions concerning the properties and inner-workings of what we consider a "proper" monitor. None of these assumptions is particularly novel or complicated, but they help describe and distinguish the task of a "proper" monitor from that of, say, a model checker, which can also be used to solve monitoring problems as we shall see.

The two basic assumptions are easy to explain: Firstly, we demand that a monitor is what we call *trace-length independent*, meaning that its efficiency does not decline with an increasing number of observations. Secondly, we demand that a monitor is *monotonic* wrt. reporting violations (resp. satisfication) of a specification, meaning that once the monitor returns "SAT" to the user, additional observations do not lead to it returning "UNSAT" (and vice versa). We are going to postulate further assumptions, but these are mere consequences of the two basic ones and are explained in §2.

At the heart of this paper, however, is a custom first-order temporal logic, in the following referred to as $\mathrm{LTL}^{\mathrm{FO}}$, which is undecidable. Yet we outline a sound, albeit incomplete, monitor construction for it based on a new type of automaton, called spawning automaton. $\mathrm{LTL}^{\mathrm{FO}}$ was originally developed for the specification of runtime verification properties of Android "Apps" and has already been used in that context (see [6] for details). Although [6] gave a monitoring algorithm for $\mathrm{LTL}^{\mathrm{FO}}$ based on formula rewriting, it turns out that the automata-based construction given in this paper leads to practically more efficient results.

As our definition of what constitutes a "proper" monitor is not tied to a particular logic we will develop it first for standard LTL (§2), the quasi-standard in the area of runtime verification. In §3, we give a more detailed account of $\mathrm{LTL}^{\mathrm{FO}}$ than was available in [6], before we lift the results of §2 to the first-order setting (§4). The automata-based monitor construction for $\mathrm{LTL}^{\mathrm{FO}}$ along with experimental results is described in §5, related work in §6. Detailed proofs can be found in [7].

2 Complexity of Monitoring in the Propositional Case

In what follows, we assume basic familiarity with LTL and topics like model checking (cf. [3] for an overview). Despite that, let us first state a formal LTL semantics, since we will consider its interpretation on infinite and finite traces. For that purpose, let AP denote a set of propositions, $\mathrm{LTL}(\mathrm{AP})$ the set of well-formed LTL formulae over that set, and for some set X set $X^\infty = X^\omega \cup X^*$ to be the union of the sets of all infinite and finite traces over X. When AP is clear from the context or does not matter, we use LTL instead of $\mathrm{LTL}(\mathrm{AP})$. Also, for a given trace $w = w_0 w_1 \ldots$, the trace w^i is defined as $w_i w_{i+1} \ldots$. As a convention we use $u, u', \ldots$ to denote finite traces, by σ the trace of length 1, and w for infinite ones or where the distinction is of no relevance.

Definition 1. *Let $\varphi \in \mathrm{LTL}(\mathrm{AP})$, $w \in (2^{\mathrm{AP}})^\infty$ be a non-empty trace, and $i \in \mathbb{N}_0$, then*

$$\begin{aligned}
w^i &\models p && \text{iff } p \in w_i, \text{ where } p \in \mathrm{AP},\\
w^i &\models \neg\varphi && \text{iff } w^i \models \varphi \text{ does not hold},\\
w^i &\models \varphi \wedge \psi && \text{iff } w^i \models \varphi \text{ and } w^i \models \psi,\\
w^i &\models \mathsf{X}\varphi && \text{iff } |w| > i \text{ and } w^{i+1} \models \varphi,\\
w^i &\models \varphi \mathsf{U} \psi && \text{iff there is a } k \text{ s.t. } i \le k < |w|, w^k \models \psi, \text{ and for all } i \le j < k, w^j \models \varphi.
\end{aligned}$$

And if $w^0 \models \varphi$ holds, we usually write $w \models \varphi$ instead. Although this semantics, which was also proposed in [21], gives rise to mixed languages, i.e., languages consisting of finite and infinite traces, we shall only ever be concerning ourselves with either finite-trace or infinite-trace languages, but not mixed ones. It is easy to see that over infinite traces this semantics matches the definition of standard LTL. Recall, LTL is a decidable logic; in fact, the satisfiability problem for LTL is known to be PSpace-complete [22].

As there are no commonly accepted rules for what qualifies as a monitor (not even in the runtime verification community), there exist a myriad of different approaches to checking that an observed behaviour satisfies (resp. violates) a formal specification, such as an LTL formula. Some of these (cf. [17,5]) consist in solving the word problem (see Definition 2). A monitor following this idea can either first record the entire system behaviour in form of a trace $u \in \Sigma^+$, where Σ is a finite alphabet of events, or process the events incrementally as they are emitted by the system under scrutiny. Both approaches are documented in the literature (cf. [17,15,16,5]), but only the second one is suitable to detect property violations (resp. satisfaction) right when they occur.

Definition 2. *The* word problem for LTL *is defined as follows.*
Input: *A formula* $\varphi \in \mathrm{LTL}(\mathrm{AP})$ *and some trace* $u \in (2^{\mathrm{AP}})^+$.
Question: *Does* $u \models \varphi$ *hold?*

In [21], a bilinear algorithm for this problem was presented (an even more efficient solution was recently given in [19]). Hence, the first sort of monitor, which is really more of a test oracle than a monitor, solves a classical decision problem. The second sort of monitor, however, solves an entirely different kind of problem, which cannot be stated in complexity-theoretical terms at all: its input is an LTL formula and a finite albeit unbounded trace which *grows* incrementally. This means that this monitor solves the word problem for each and every new event that is added to the trace at runtime. We can therefore say that the word problem acts as a lower bound on the complexity of the monitoring problem that such a monitor solves; or, in other words, the problem that the online monitor solves is at least as hard as the problem that the offline monitor solves.

There are approaches to build efficient (i.e., trace-length independent) monitors that repeatedly answer the word problem (cf. [17]). However, such approaches violate our second basic assumption, mentioned in the introduction, in that they are necessarily non-monotonic. To see this, consider $\varphi = a\mathrm{U}b$ and some trace $u = \{a\}\{a\}\ldots\{a\}$ of length n. Using our finite-trace interpretation, $u \not\models \varphi$. However, if we add $u_{n+1} = \{b\}$, we get $u \models \varphi$.[1] For the user, this essentially means that she cannot trust the verdict of the monitor as it may flip in the future, unless of course it is obvious from the start that, e.g., only safety properties are monitored and the monitor is built merely to detect violations, i.e., bad prefixes. However, if we take other monitorable languages into account as we do in this paper, i.e., those that have either good or bad prefixes (or both), we need to distinguish between satisfaction and violation of a property (and want the monitor to report either occurrence truthfully).

[1] Note that this effect is not particular to our choice of finite-trace interpretation. Had we used, e.g., what is known as the weak finite-trace semantics, discussed in [14], we would first have had $u \models \varphi$ and if $u_{n+1} = \emptyset$, subsequently $u \not\models \varphi$.

Definition 3. *For any $L \subseteq \Sigma^\omega$, $u \in \Sigma^*$ is called a* good prefix *(resp.* bad prefix*) iff $u\Sigma^\omega \subseteq L$ holds (resp. $u\Sigma^\omega \cap L = \emptyset$).*

We shall use $\text{good}(L) \subseteq \Sigma^*$ (resp. $\text{bad}(L)$) to denote the set of good (resp. bad) prefixes of L. For brevity, we also write $\text{good}(\varphi)$ instead of $\text{good}(\mathcal{L}(\varphi))$, and do the same for $\text{bad}(\mathcal{L}(\varphi))$.

A monitor that detects good (resp. bad) prefixes has been termed impartial in [12] as it not only states something about the past, but also about the future: once a good (resp. bad) prefix has been detected, no matter how the system would evolve in an indefinite future, the property would remain satisfied (resp. violated). In that sense, impartial monitors are monotonic by definition. Moreover in [8], a construction is given, showing how to obtain trace-length independent (even optimal) impartial monitors for LTL and a timed extension called TLTL. The obtained monitor basically returns $\top$ to the user if $u \in \text{good}(\varphi)$ holds, $\bot$ if $u \in \text{bad}(\varphi)$ holds, and ? otherwise. Not surprisingly though, the monitoring problem such a monitor solves is computationally more involved than the word problem. It solves what we call the prefix problem (of LTL), which can easily be shown PSpace-complete by way of LTL satisfiability.

Definition 4. *The* prefix problem for LTL *is defined as follows.*
Input: *A formula $\varphi \in \text{LTL}(\text{AP})$ and some trace $u \in (2^{\text{AP}})^*$.*
Question: *Does $u \in \text{good}(\varphi)$ (resp. $\text{bad}(\varphi)$) hold?*

Theorem 1. *The prefix problem for* LTL *is PSpace-complete.*

Proof. For brevity, we will only focus on bad prefixes. It is easy to see that $u \in \text{bad}(\varphi)$ iff $\mathcal{L}(u_0 \wedge \mathsf{X}u_1 \wedge \mathsf{XX}u_2 \wedge \ldots \wedge \varphi) = \emptyset$. Constructing this conjunction takes polynomial time and the corresponding emptiness check can be performed in PSpace [22]. For hardness, we proceed with a reduction of LTL satisfiability. Again, it is easy to see that $\mathcal{L}(\varphi) \neq \emptyset$ iff $\sigma \notin \text{bad}(\mathsf{X}\varphi)$ for any $\sigma \in 2^{\text{AP}}$. This reduction is linear, and as PSpace $=$ co-PSpace, the statement follows. □

We would like to point out the possibility of building an impartial though trace-length dependent LTL monitor using an "off the shelf" model checker, which accepts a propositional Kripke structure and an LTL formula as input. Note that here we make the assumption that Kripke structures produce infinite as opposed to finite traces.

Definition 5. *The* model checking problem for LTL *is defined as follows.*
Input: *A formula $\varphi \in \text{LTL}(\text{AP})$ and a Kripke structure $\mathcal{K}$ over 2^{AP}.*
Question: *Does $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ hold?*

As in LTL the model checking and the satisfiability problems are both PSpace-complete [22], we can use a model checking tool as monitor: given that it is straightforward to construct $\mathcal{K}$, s.t. $\mathcal{L}(\mathcal{K}) = u(2^{\text{AP}})^\omega$, in no more than polynomial time, we return $\top$ to the user if $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ holds, $\bot$ if $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\neg\varphi)$ holds, and ? if neither holds. One could therefore be tempted to think of monitoring merely in terms of a model checking problem, but we shall see that as soon as the logic in question has an undecidable satisfiability problem this reduction fails. Besides, it can be questioned whether monitoring as model checking leads to a desirable monitor with its obvious trace-length dependence and having to repeatedly solve a PSpace-complete problem for each new event.

3 LTL$^{\text{FO}}$—Formal Definitions and Notation

Let us now introduce our first-order specification language LTL$^{\text{FO}}$ and related concepts in more detail. The first concept we need is that of a *sorted first-order signature*, given as $\varGamma = (\mathbf{S}, \mathbf{F}, \mathbf{R})$, where $\mathbf{S}$ is a finite non-empty set of sorts, $\mathbf{F}$ a finite set of function symbols and $\mathbf{R} = \mathbf{U} \cup \mathbf{I}$ a finite set of a priori uninterpreted and interpreted predicate symbols, s.t. $\mathbf{U} \cap \mathbf{I} = \emptyset$ and $\mathbf{R} \cap \mathbf{F} = \emptyset$. The former set of predicate symbols are referred to as $\mathbf{U}$-operators and the latter as $\mathbf{I}$-operators. As is common, 0-ary functions symbols are also referred to as constant symbols. We assume that all operators in $\varGamma$ have a given arity that ranges over the sorts given by $\mathbf{S}$, respectively. We also assume an infinite supply of variables, $\mathbf{V}$, that also range over $\mathbf{S}$ and where $\mathbf{V} \cap (\mathbf{F} \cup \mathbf{R}) = \emptyset$. Let us refer to the first-order language determined by $\varGamma$ as $\mathcal{L}(\varGamma)$. While *terms* in $\mathcal{L}(\varGamma)$ are made up of variables and function symbols, *formulae* of $\mathcal{L}(\varGamma)$ are defined as follows:

$$\varphi ::= p(t_1, \ldots, t_n) \mid r(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \mathsf{U} \varphi \mid \forall(x_1, \ldots, x_n) : p.\ \varphi,$$

where $t_1, \ldots, t_n$ are terms, $p \in \mathbf{U}$, $r \in \mathbf{I}$, and $x_1, \ldots, x_n \in \mathbf{V}$. As variables are sorted, in the quantified formula $\forall(x_1, \ldots, x_n) : p.\ \varphi$, the $\mathbf{U}$-operator p with arity $\tau_1 \times \ldots \times \tau_n$, defines the sorts of variables $x_1, \ldots, x_n$ to be $\tau_1, \ldots, \tau_n$, with $\tau_i \in \mathbf{S}$, respectively. For terms $t_1, \ldots, t_n$, we say that $p(t_1, \ldots, t_n)$ is well-sorted if the sort of every t_i is τ_i. This notion is inductively applicable to terms. Moreover, we consider only well-sorted formulae and refer to the set of all well-sorted $\mathcal{L}(\varGamma)$ formulae over a signature $\varGamma$ in terms of LTL$^{\text{FO}}_{\varGamma}$. When a specific $\varGamma$ is either irrelevant or clear from the context, we will simply write LTL$^{\text{FO}}$ instead. When convenient and a certain index is of no importance in the given context, we also shorten notation of a vector $(x_1, \ldots, x_n)$ by a (bold) $\boldsymbol{x}$.

A *$\varGamma$-structure*, or just *first-order structure* is a pair $\mathfrak{A} = (|\mathfrak{A}|, I)$, where $|\mathfrak{A}| = |\mathfrak{A}|_1 \cup \ldots \cup |\mathfrak{A}|_n$, is a non-empty set called domain, s.t. every sub-domain $|\mathfrak{A}|_i$ is either a non-empty finite or countable set (e.g., set of all integers or strings) and I an interpretation. I assigns to each sort $\tau_i \in \mathbf{S}$ a specific sub-domain $\tau_i^I = |\mathfrak{A}|_i$, to each function symbol $f \in \mathbf{F}$ of arity $\tau_1 \times \ldots \times \tau_l \longrightarrow \tau_m$ a function $f^I : |\mathfrak{A}|_1 \times \ldots \times |\mathfrak{A}|_l \longrightarrow |\mathfrak{A}|_m$, and to every $\mathbf{I}$-operator r with arity $\tau_1 \times \ldots \times \tau_m$ a relation $r^I \subseteq |\mathfrak{A}|_1 \times \ldots \times |\mathfrak{A}|_m$. We restrict ourselves to computable relations and functions. In that regard, we can think of I as a mapping between $\mathbf{I}$-operators (resp. function symbols) and the corresponding algorithms which compute the desired return values, each conforming to the symbols' respective arities. Note that the interpretation of $\mathbf{U}$-operators is rather different from $\mathbf{I}$-operators, as it is closely tied to what we call a trace and therefore discussed after we introduce the necessary notions and notation.

We model observed system behaviour in terms of *actions*: Let $p \in \mathbf{U}$ with arity $\tau_1 \times \ldots \times \tau_m$ and $\boldsymbol{d} \in D_p = |\mathfrak{A}|_1 \times \ldots \times |\mathfrak{A}|_m$, then we call $(p, \boldsymbol{d})$ an *action*. We refer to finite sets of actions as *events*. A system's behaviour is therefore a finite *trace* of events, which we also denote as a sequence of sets of ground terms $\{sms(1234)\}\{login(\text{“user”})\}\ldots$ when we mean the sequence of tuples $\{(sms, 1234)\}\{(login, \text{“user”})\}\ldots$ Therefore the occurrence of some action $sms(1234)$ in the trace at position $i \in \mathbb{N}_0$, written $sms(1234) \in w_i$, indicates that, at time i, $sms(1234)$ holds (or, from a practical point of view, an SMS was sent to number 1234). We follow the convention that only symbols from $\mathbf{U}$ appear in a trace, which therefore gives these symbols their respective interpretations. The following formalises this notion.

A *first-order temporal structure* is a tuple $(\overline{\mathfrak{A}}, w)$, where $\overline{\mathfrak{A}} = (|\mathfrak{A}_0|, I_0)(|\mathfrak{A}_1|, I_1)\ldots$ is a (possibly infinite) sequence of first-order structures and $w = w_0 w_1 \ldots$ a corresponding trace. We demand that for all $\mathfrak{A}_i$ and $\mathfrak{A}_{i+1}$ from $\overline{\mathfrak{A}}$, it is the case that $|\mathfrak{A}_i| = |\mathfrak{A}_{i+1}|$ for all $f \in \mathbf{F}$, $f^{I_{i+1}} = f^{I_i}$, and for all $\tau \in \mathbf{S}$, $\tau^{I_i} = \tau^{I_{i+1}}$. For any two structures, $\mathfrak{A}$ and $\mathfrak{A}'$, which satisfy these conditions, we write $\mathfrak{A} \sim \mathfrak{A}'$. Moreover given some $\overline{\mathfrak{A}}$ and $\mathfrak{A}$, if for all $\mathfrak{A}_i$ from $\overline{\mathfrak{A}}$, we have that $\mathfrak{A}_i \sim \mathfrak{A}$, we also write $\overline{\mathfrak{A}} \sim \mathfrak{A}$. Finally, the interpretation of an $\mathbf{U}$-operator p with arity $\tau_1 \times \ldots \times \tau_m$ is then defined wrt. a position i in w as $p^{I_i} = \{\boldsymbol{d} \mid (p, \boldsymbol{d}) \in w_i\}$. Essentially this means that, unlike function symbols, $\mathbf{U}$- and $\mathbf{I}$-operators don't have to be rigid.

Note also that from this point forward, we consider only the case where the policy to be monitored is given as a closed formula, i.e., a sentence. This is closely related to our means of quantification: a quantifier in LTL$^{\mathrm{FO}}$ is restricted to those elements that appear in the trace, and not arbitrary elements from a (possibly infinite) domain. While certain policies cannot be expressed with this restriction (e.g., "for all phone numbers x that are not in the contact list, $r(x)$ is true"), this restriction bears the advantage that, when examining a given trace, functions and relations are only ever evaluated over known objects. The advantages of this type of quantification in monitoring first-order languages have also been pointed out in [16,5]. In other words, had we allowed free variables, a monitor might end up having to "try out" all the different domain elements in order to evaluate such policies, which runs counter to our design rationale of quantification.

In what follows, let us fix a particular Γ. The semantics of LTL$^{\mathrm{FO}}$ can now be defined wrt. a quadruple $(\overline{\mathfrak{A}}, w, v, i)$ as follows, where $i \in \mathbb{N}_0$, and v is an (initially empty) set of valuations assigning domain values to variables:

$$\begin{array}{rl}
(\overline{\mathfrak{A}}, w, v, i) \models p(t_1, \ldots, t_n) & \text{iff } (t_1^{I_i}, \ldots, t_n^{I_i}) \in p^{I_i}, \\
(\overline{\mathfrak{A}}, w, v, i) \models r(t_1, \ldots, t_n) & \text{iff } (t_1^{I_i}, \ldots, t_n^{I_i}) \in r^{I_i}, \\
(\overline{\mathfrak{A}}, w, v, i) \models \neg\varphi & \text{iff } (\overline{\mathfrak{A}}, w, v, i) \models \varphi \text{ is not true}, \\
(\overline{\mathfrak{A}}, w, v, i) \models \varphi \wedge \psi & \text{iff } (\overline{\mathfrak{A}}, w, v, i) \models \varphi \text{ and } (\overline{\mathfrak{A}}, w, v, i) \models \psi, \\
(\overline{\mathfrak{A}}, w, v, i) \models \mathsf{X}\varphi & \text{iff } |w| > i \text{ and } (\overline{\mathfrak{A}}, w, v, i+1) \models \varphi, \\
(\overline{\mathfrak{A}}, w, v, i) \models \varphi \mathsf{U} \psi & \text{iff for some } k \geq i, (\overline{\mathfrak{A}}, w, v, k) \models \psi, \\
 & \text{and } (\overline{\mathfrak{A}}, w, v, j) \models \varphi \text{ for all } i \leq j < k, \\
(\overline{\mathfrak{A}}, w, v, i) \models \forall (x_1, \ldots, x_n) : p.\ \varphi & \text{iff for all } (p, d_1, \ldots, d_n) \in w_i, \\
 & (\overline{\mathfrak{A}}, w, v \cup \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}, i) \models \varphi,
\end{array}$$

where terms are evaluated inductively and x^I treated as $v(x)$. If $(\overline{\mathfrak{A}}, w, v, 0) \models \varphi$, we write $(\overline{\mathfrak{A}}, w, v) \models \varphi$, and if v is irrelevant or clear from the context, $(\overline{\mathfrak{A}}, w) \models \varphi$.

Later we will also make use of the (possibly infinite) set of all events wrt. $\mathfrak{A}$, given as $(\mathfrak{A})\text{-Ev} = \bigcup_{p \in \mathbf{U}} \{(p, \boldsymbol{d}) \mid \boldsymbol{d} \in D_p\}$, and take the liberty to omit the trailing $(\mathfrak{A})$ whenever a particular $\mathfrak{A}$ is either irrelevant or clear from the context. We can then describe the *generated language* of φ, $\mathcal{L}(\varphi)$ (or simply the language of φ, i.e., the set of all logical models of φ) compactly as $\mathcal{L}(\varphi) = \{(\overline{\mathfrak{A}}, w) \mid w_i \in 2^{\mathrm{Ev}} \text{ and } (\overline{\mathfrak{A}}, w) \models \varphi\}$, although, as before, we shall only ever concern ourselves with either infinite- or finite-trace languages, but not mixed ones. Finally, we will use common syntactic "sugar", including $\exists (x_1, \ldots, x_n) : p.\ \varphi = \neg(\forall (x_1, \ldots, x_n) : p.\ \neg\varphi)$, etc.

Example 1. For brevity, cf. [6] for some $\mathrm{LTL}^{\mathrm{FO}}$ example policies formalised in $\mathrm{LTL}^{\mathrm{FO}}$. However, to give at least an intuition, let's pick up the idea of monitoring Android "Apps" again, and specify that "Apps" must not send SMS messages to numbers not in a user's contact database. Assuming there exists an **U**-operator sms, which is true / appears in the trace, whenever an "App" sends an SMS message to phone number x, we could formalise said policy in terms of $\mathsf{G}\forall x : sms.\ contact(x)$. Note how in this formula the meaning of x is given implicitly by the arity of sms and must match the definition of $contact$ in each world. Also note how sms is interpreted indirectly via its occurrence in the trace, whereas $contact$ never appears in the trace, even if true. $contact$ can be thought of as interpreted via a program that queries a user's contact database, whose contents may change over time.

4 Complexity of Monitoring in the First-Order Case

$\mathrm{LTL}^{\mathrm{FO}}$ as defined above is undecidable as can be shown by way of the following lemma. It basically helps us reduce finite satisfiability of standard first-order logic to $\mathrm{LTL}^{\mathrm{FO}}$.

Lemma 1. *Let φ be a sentence in first-order logic, then we can construct a corresponding $\psi \in \mathrm{LTL}^{\mathrm{FO}}$ s.t. φ has a finite model iff ψ is satisfiable.*

Theorem 2. $\mathrm{LTL}^{\mathrm{FO}}$ *is undecidable.*

Proof (Sketch). Follows from Lemma 1 and Trakhtenbrot's Theorem (cf. [20, §9]).

Let's now define what is meant by Kripke structures in our new setting. They either give rise to infinite-trace languages (i.e., have a left-total transition relation), or represent finite traces (i.e, are essentially linear structures). For brevity, we shall restrict to the definition of the former. Note that we will also skip detailed redefinitions of the decision problems discussed in §2, since the concepts transfer in a straightforward manner.

Definition 6. *Given some $\mathfrak{A}$, a* ($\mathfrak{A}$)-Kripke structure*, or just first-order Kripke structure, is a state-transition system $\mathcal{K} = (S, s_0, \lambda, \to)$, where S is a finite set of states, $s_0 \in S$ a distinguished initial state, $\lambda : S \longrightarrow \widehat{\mathfrak{A}} \times \mathrm{Ev}$, where $\widehat{\mathfrak{A}} = \{\mathfrak{A}' \mid \mathfrak{A}' \sim \mathfrak{A}\}$, a labelling function, and $\to \subseteq S \times S$ a (left-total) transition relation.*

Definition 7. *For a $(\mathfrak{A})$-Kripke structure $\mathcal{K}$ with states $s_0, \ldots, s_n$, the* generated language *is given as $\mathcal{L}(\mathcal{K}) = \{(\overline{\mathfrak{A}}, w) \mid (\mathfrak{A}_0, w_0) = \lambda(s_0)$ and for all $i \in \mathbb{N}$ there exist some $j, k \in [0, n]$ s.t. $(\mathfrak{A}_i, w_i) = \lambda(s_j), (\mathfrak{A}_{i-1}, w_{i-1}) = \lambda(s_k)$ and $(s_k, s_j) \in \to\}$.*

The inputs to the $\mathrm{LTL}^{\mathrm{FO}}$ word problem are therefore an $\mathrm{LTL}^{\mathrm{FO}}$ formula and a linear first-order Kripke structure, representing a finite input trace. Unlike in standard LTL,

Theorem 3. *The word problem for* $\mathrm{LTL}^{\mathrm{FO}}$ *is PSpace-complete.*

The inputs to the $\mathrm{LTL}^{\mathrm{FO}}$ model checking problem, in turn, are a left-total first-order Kripke structure, which gives rise to an infinite-trace language, and an $\mathrm{LTL}^{\mathrm{FO}}$ formula.

Theorem 4. *The model checking problem for* $\mathrm{LTL}^{\mathrm{FO}}$ *is in ExpSpace.*

The reason for this result is that we can devise a reduction of that problem to LTL model checking in exponential space. While the PSpace-lower bound is easy, e.g., via reduction of the $\mathrm{LTL}^{\mathrm{FO}}$ word problem, we currently do not know how tight these bounds are and, therefore, leave this as an open problem. Note also that the results of both Theorem 3 and Theorem 4 are obtained even without taking into account the complexities of the interpretations of function symbols and **I**-operators; that is, for these results to hold, we assume that interpretations do not exceed polynomial, resp. exponential space.

We have seen in §2 that the prefix problem lies at the heart of an impartial monitor. While in LTL it was possible to build an impartial monitor using a model checker (albeit a very inefficient one), the following shows that this is no longer possible.

Lemma 2. *Let* $\mathfrak{A}$ *be a first-order structure and* $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$*, then* $\mathcal{L}(\varphi)_{\mathfrak{A}} = \{(\overline{\mathfrak{A}}, w) \mid \overline{\mathfrak{A}} \sim \mathfrak{A}, w \in (2^{\mathrm{Ev}})^{\omega},$ *and* $(\overline{\mathfrak{A}}, w) \models \varphi\}$*. Testing if* $\mathcal{L}(\varphi)_{\mathfrak{A}} \neq \emptyset$ *is generally undecidable.*

Theorem 5. *The prefix problem for* $\mathrm{LTL}^{\mathrm{FO}}$ *is undecidable.*

Proof (Sketch). As in Theorem 1: $(\mathfrak{A}, \sigma) \in \mathrm{bad}(\mathsf{X}\varphi)$ iff $\mathcal{L}(\varphi)_{\mathfrak{A}} = \emptyset$ for any $\sigma \in \mathrm{Ev}$.

5 Monitoring $\mathrm{LTL}^{\mathrm{FO}}$

A corollary of Theorem 5 is that there cannot exist a complete monitor for $\mathrm{LTL}^{\mathrm{FO}}$-definable infinite trace languages. Yet one of the main contributions of our work is to show that one can build a sound and efficient $\mathrm{LTL}^{\mathrm{FO}}$ monitor using a new kind of automaton. Before we go into the details of the actual monitoring algorithm, let us first consider the automaton model, which we refer to as *spawning automaton* (SA). SAs are called that, because when they process their input, they potentially "spawn" a positive Boolean combination of "children SAs" (i.e., subautomata) in each such step. Let $\mathcal{B}^+(X)$ denote the set of all positive Boolean formulae over the set X. We say that some set $Y \subseteq X$ satisfies a formula $\beta \in \mathcal{B}^+(X)$, written $Y \models \beta$, if the truth assignment that assigns true to all elements in Y and false to all $X - Y$ satisfies β.

Definition 8. *A* spawning automaton*, or simply SA, is given by* $\mathcal{A} = (\Sigma, l, Q, Q_0, \delta_{\rightarrow}, \delta_{\downarrow}, \mathcal{F})$*, where* Σ *is a countable set called alphabet,* $l \in \mathbb{N}_0$ *the level of* $\mathcal{A}$*,* Q *a finite set of states,* $Q_0 \subseteq Q$ *a set of distinguished initial states,* $\delta_{\rightarrow}$ *a transition relation,* $\delta_{\downarrow}$ *what is called a spawning function, and* $\mathcal{F} = \{F_1, \ldots, F_n \mid F_i \subseteq Q\}$ *an acceptance condition (to be defined later on). We have* $\delta_{\rightarrow} : Q \times \Sigma \longrightarrow 2^Q$ *and* $\delta_{\downarrow} : Q \times \Sigma \longrightarrow \mathcal{B}^+(\mathcal{A}^{<l})$*, where* $\mathcal{A}^{<l} = \{\mathcal{A}' \mid \mathcal{A}'$ *is an SA with level less than* $l\}$*.*

Definition 9. *A* run *of* $\mathcal{A}$ *over input* $w \in \Sigma^{\omega}$ *is a mapping* $\rho : \mathbb{N}_0 \longrightarrow Q$*, s.t.* $\rho(0) \in Q_0$ *and* $\rho(i+1) \in \delta_{\rightarrow}(\rho(i), w_i)$ *for all* $i \in \mathbb{N}_0$*.* ρ *is* locally accepting *if* $\mathrm{Inf}(\rho) \cap F_i \neq \emptyset$ *for all* $F_i \in \mathcal{F}$*, where* $\mathrm{Inf}(\rho)$ *denotes the set of states visited infinitely often. It is called* accepting *if* $l = 0$ *and it is locally accepting. If* $l > 0$*,* ρ *is called accepting if it is locally accepting and for all* $i \in \mathbb{N}_0$ *there is a set* $Y \subseteq \mathcal{A}^{<l}$*, s.t.* $Y \models \delta_{\downarrow}(\rho(i), w_i)$ *and all automata* $\mathcal{A}' \in Y$ *have an accepting run,* ρ'*, over* w^i*. The* accepted language *of* $\mathcal{A}$*,* $\mathcal{L}(\mathcal{A})$*, consists of all* $w \in \Sigma^{\omega}$*, for which it has at least one accepting run.*

5.1 Spawning Automata Construction

Given some $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, let us now examine how to build the corresponding SA, $\mathcal{A}_\varphi = (\Sigma, l, Q, Q_0, \delta_\rightarrow, \delta_\downarrow, \mathcal{F})$ s.t. $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds. To this end, we set $\Sigma = \{(\mathfrak{A}, \sigma) \mid \sigma \in (\mathfrak{A})\text{-Ev}\}$. If φ is not a sentence, we write $\mathcal{A}_{\varphi,v}$ to denote the spawning automaton for φ in which free variables are mapped according to a finite set of valuations v.[2] To define the set of states for an SA, we make use of a restricted subformula function, $\mathrm{sf}|_\forall(\varphi)$, which is defined like a generic subformula function, except if φ is of the form $\forall\, \boldsymbol{x} : p.\ \psi$, we have $\mathrm{sf}|_\forall(\varphi) = \{\varphi\}$. This essentially means that an SA for a formula φ on the topmost level looks like the generalised Büchi automaton (GBA, cf. [3]) for φ, where quantified subformulae have been interpreted as atomic propositions.

For example, if $\varphi = \psi \wedge \forall\, \boldsymbol{x} : p.\ \psi'$, where ψ is a quantifier-free formula, then $\mathcal{A}_\varphi$, at the topmost level n, is like the GBA for the LTL formula $\psi \wedge a$, where a is an atomic proposition; or in other words, $\mathcal{A}_\varphi$ handles the subformula $\forall\, \boldsymbol{x} : p.\ \psi'$ separately in terms of a subautomaton of level $n-1$ (see also definition of $\delta_\downarrow$ below).

Finally, we define the closure of φ wrt. $\mathrm{sf}|_\forall(\varphi)$ as $\mathrm{cl}(\varphi) = \{\neg\psi \mid \psi \in \mathrm{sf}|_\forall(\varphi)\} \cup \mathrm{sf}|_\forall(\varphi)$, i.e., the smallest set containing $\mathrm{sf}|_\forall(\varphi)$, which is closed under negation. The *set of states* of $\mathcal{A}_\varphi$, Q, consists of all complete subsets of $\mathrm{cl}(\varphi)$; that is, a set $q \subseteq \mathrm{cl}(\varphi)$ is complete iff

- for any $\psi \in \mathrm{cl}(\varphi)$ either $\psi \in q$ or $\neg\psi \in q$, but not both; and
- for any $\psi \wedge \psi' \in \mathrm{cl}(\varphi)$, we have that $\psi \wedge \psi' \in q$ iff $\psi \in q$ and $\psi' \in q$; and
- for any $\psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)$, we have that if $\psi \mathsf{U} \psi' \in q$ then $\psi' \in q$ or $\psi \in q$, and if $\psi \mathsf{U} \psi' \notin q$, then $\psi' \notin q$.

Let $q \in Q$ and $\mathfrak{A} = (|\mathfrak{A}|, I)$. The *transition function* $\delta_\rightarrow(q, (\mathfrak{A}, \sigma))$ is defined iff

- for all $p(\boldsymbol{t}) \in q$, we have $\boldsymbol{t}^I \in p^I$ and for all $\neg p(\boldsymbol{t}) \in q$, we have $\boldsymbol{t}^I \notin p^I$,
- for all $r(\boldsymbol{t}) \in q$, we have $\boldsymbol{t}^I \in r^I$ and for all $\neg r(\boldsymbol{t}) \in q$, we have $\boldsymbol{t}^I \notin r^I$.

In which case, for any $q' \in Q$, we have that $q' \in \delta_\rightarrow(q, (\mathfrak{A}, \sigma))$ iff

- for all $\mathsf{X}\psi \in \mathrm{cl}(\varphi)$, we have $\mathsf{X}\psi \in q$ iff $\psi \in q'$, and
- for all $\psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)$, we have $\psi \mathsf{U} \psi' \in q$ iff $\psi' \in q$ or $\psi \in q$ and $\psi \mathsf{U} \psi' \in q'$.

This is similar to the well known syntax directed construction of GBAs (cf. [3]), except that we also need to cater for quantified subformulae. For this purpose, an inductive *spawning function* is defined as follows. If $l > 0$, then $\delta_\downarrow(q, (\mathfrak{A}, \sigma))$ yields

$$\left(\bigwedge_{\forall \boldsymbol{x}:p.\psi \in q} \left(\bigwedge_{(p,\boldsymbol{d}) \in \sigma} \mathcal{A}_{\psi,v'} \right) \right) \wedge \left(\bigwedge_{\neg\forall \boldsymbol{x}:p.\psi \in q} \left(\bigvee_{(p,\boldsymbol{d}) \in \sigma} \mathcal{A}_{\neg\psi,v''} \right) \right),$$

where $v' = v \cup \{\boldsymbol{x} \mapsto \boldsymbol{d}\}$ and $v'' = v \cup \{\boldsymbol{x} \mapsto \boldsymbol{d}\}$ are sets of valuations, otherwise $\delta_\downarrow(q, (\mathfrak{A}, \sigma))$ yields $\top$. Moreover, we set $Q_0 = \{q \in Q \mid \varphi \in q\}$, $\mathcal{F} = \{F_{\psi \mathsf{U} \psi'} \mid$

[2] Considering free variables, even though our runtime policies can only ever be sentences, is necessary, because an SA for a policy φ is inductively defined in terms of SAs for its subformulae (i.e., $\mathcal{A}_\varphi$'s subautomata), some of which may contain free variables.

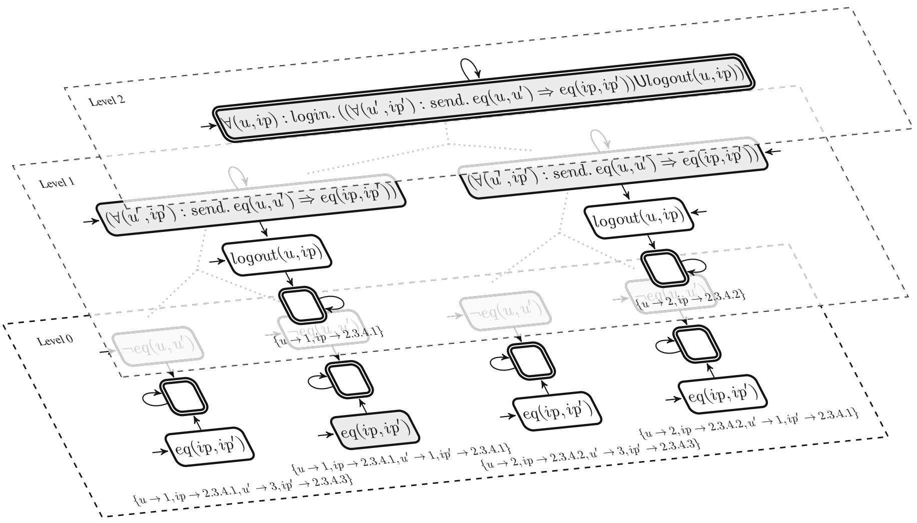

Level 2
∀(u, ip) : login. ((∀(u′, ip′) : send. eq(u, u′) ⇒ eq(ip, ip′))Ulogout(u, ip))
Level 1
(∀(u′, ip′) : send. eq(u, u′) ⇒ eq(ip, ip′))
logout(u, ip)
(∀(u′, ip′) : send. eq(u, u′) ⇒ eq(ip, ip′))
logout(u, ip)
Level 0
¬eq(u, u′)
¬eq(u, u′)
{u → 1, ip → 2.3.4.1}
¬eq(u, u′)
¬eq(u, u′)
{u → 2, ip → 2.3.4.2}
eq(ip, ip′)
eq(ip, ip′)
eq(ip, ip′)
eq(ip, ip′)
{u → 2, ip → 2.3.4.2, u′ → 1, ip′ → 2.3.4.1}
{u → 1, ip → 2.3.4.1, u′ → 1, ip′ → 2.3.4.1}
{u → 2, ip → 2.3.4.2, u′ → 3, ip′ → 2.3.4.3}
{u → 1, ip → 2.3.4.1, u′ → 3, ip′ → 2.3.4.3}

Fig. 1. Spawning on $\{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$

$\psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)\}$ with $F_{\psi \mathsf{U} \psi'} = \{q \in Q \mid \psi' \in q \vee \neg(\psi \mathsf{U} \psi') \in q\}$, and $l = \mathrm{depth}(\varphi)$, where $\mathrm{depth}(\varphi)$ is called the *quantifier depth* of φ. For some $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, $\mathrm{depth}(\varphi) = 0$ iff φ is a quantifier free formula. The remaining cases are inductively defined as follows: $\mathrm{depth}(\forall \boldsymbol{x} : p.\ \psi) = 1 + \mathrm{depth}(\psi)$, $\mathrm{depth}(\psi \wedge \psi') = \mathrm{depth}(\psi \mathsf{U} \psi') = \max(\mathrm{depth}(\psi), \mathrm{depth}(\psi'))$ and $\mathrm{depth}(\neg\varphi) = \mathrm{depth}(\mathsf{X}\varphi) = \mathrm{depth}(\varphi)$.

Lemma 3. *Let $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ (not necessarily a sentence) and v be a valuation. For each accepting run ρ in $\mathcal{A}_{\varphi,v}$ over input $(\overline{\mathfrak{A}}, w)$, $\psi \in \mathrm{cl}(\varphi)$, and $i \geq 0$, we have that $\psi \in \rho(i)$ iff $(\overline{\mathfrak{A}}, w, v, i) \models \psi$.*

Theorem 6. *The constructed SA is correct in the sense that for any sentence $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, we have that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$.*

Example 2. Consider the graphical representation of an SA for $\varphi = \mathsf{G}(\forall(u, ip) : login.\ ((\forall(u', ip') : send.\ eq(u, u') \Rightarrow eq(ip, ip'))\mathsf{U}logout(u, ip)))$ in Fig. 1. In a nutshell, φ states that once user u has logged in to the system from IP-address ip, she must not send anything from an IP-address other than ip until logged out. While φ is not meant to represent a realistic security policy as is, it does help highlight the features of an SA: We first note that level l of $\mathcal{A}_\varphi$ is given by $\mathrm{depth}(\varphi) = 2$. As φ is of the form $\mathsf{G}\forall(u, ip) : login.\ \psi$, $\mathcal{A}_\varphi$'s individual state space is de facto that of an ordinary GBA for an LTL formula of the form $\mathsf{G}p$. Let's now assume that $\sigma = \{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$ is an event, which we want $\mathcal{A}_\varphi$ to process. Due to φ's outmost quantifier, the two $login$-actions will lead to the spawning of a conjunction of two subautomata of respective levels $l - 1$ (downward dotted lines). The individual state space of these subautomata is de facto that of an ordinary GBA for an LTL formula of the form $a\mathsf{U}b$ as one can see in Fig. 1, level 1. These SAs also keep track of a quantified formula, hence the two $send$-actions will also spawn a conjunction of subautomata, basically, to check if $eq(u, u') \Rightarrow eq(ip, ip')$ holds. The respective valuations are given below each SA, whereas the respective current states are marked in grey.

5.2 Monitor Construction

Before we look at the actual monitor construction, let us first introduce some additional concepts and notation: For a finite run ρ in $\mathcal{A}_\varphi$ over $(\overline{\mathfrak{A}}, u)$, we call $\delta_\downarrow(\rho(j), (\mathfrak{A}_j, u_j)) = obl_j$ an *obligation*, where $0 \leq j < |u|$, in that obl_j represents the language to be satisfied after j inputs. That is, obl_j refers to the language represented by the positive Boolean combination of spawned SAs. We say obl_j is *met* by the input, if $(\overline{\mathfrak{A}}^j, u^j) \in \text{good}(obl_j)$ and *violated* if $(\overline{\mathfrak{A}}^j, u^j) \in \text{bad}(obl_j)$. Furthermore, ρ is called *potentially locally accepting*, if it can be extended to a run ρ' over $(\overline{\mathfrak{A}}, u)$ together with some infinite suffix, such that ρ' is locally accepting.

The monitor for a formula $\varphi \in \text{LTL}^{\text{FO}}$ can now be described in terms of two mutually recursive algorithms: The main entry point is Algorithm M. It reads an event and issues two calls to a separate Algorithm T: one for φ (under a possibly empty valuation v) and one for $\neg\varphi$ (under a possibly empty valuation v). The purpose of Algorithm T is to detect bad prefixes wrt. the language of its argument formula, call it ψ. It does so by keeping track of those finite runs in $\mathcal{A}_{\psi,v}$ that are potentially locally accepting and where its obligations haven't been detected as violated by the input. If at any time not at least one such run exists, then a bad prefix has been encountered. Algorithm T, in turn, uses Algorithm M to evaluate if obligations of its runs are met or violated by the input observed so far (i.e., it inductively creates submonitors): after the ith input, it instantiates Algorithm M with argument ψ' (under corresponding valuation v') for each $\mathcal{A}_{\psi',v'}$ that occurs in obl_i and forwards to it all observed events from time i on.

Algorithm M (*Monitor*). The algorithm takes a $\varphi \in \text{LTL}^{\text{FO}}$ (under a possibly empty valuation v). Its abstract behaviour is as follows: Let us assume an initially empty first-order temporal structure $(\overline{\mathfrak{A}}, u)$. Algorithm M reads an event $(\mathfrak{A}, \sigma)$, prints "$\top$" if $(\overline{\mathfrak{A}}\mathfrak{A}, u\sigma) \in \text{good}(\varphi)$ (resp. "$\bot$" for $\text{bad}(\varphi)$), and returns. Otherwise it prints "?", whereas we now assume that $(\overline{\mathfrak{A}}, u) = (\overline{\mathfrak{A}}\mathfrak{A}, u\sigma)$ holds.[3]

M1. [Create instances of Algorithm T.] Create two instances of Algorithm T: one with φ and one with $\neg\varphi$, and call them $T_{\varphi,v}$ and $T_{\neg\varphi,v}$, respectively.

M2. [Forward next event.] Wait for next event $(\mathfrak{A}, \sigma)$ and forward it to $T_{\varphi,v}$ and $T_{\neg\varphi,v}$.

M3. [Communicate verdict.] If $T_{\varphi,v}$ sends "no runs", print $\bot$ and return. If $T_{\neg\varphi,v}$ sends "no runs", print $\top$ and return. Otherwise, print "?" and go to M2. ▮

Algorithm T (*Track runs*). The algorithm takes a $\varphi \in \text{LTL}^{\text{FO}}$ (under a corresponding valuation v), for which it creates an SA, $\mathcal{A}_{\varphi,v}$. It then reads an event $(\mathfrak{A}, \sigma)$ and returns, if $\mathcal{A}_{\varphi,v}$, after processing $(\mathfrak{A}, \sigma)$, does not have any potentially locally accepting runs, for which obligations haven't been detected as violated. Otherwise, it saves the new state of $\mathcal{A}_{\varphi,v}$, waits for new input, and then checks again, and so forth.

[3] Obviously, the monitor does not really keep $(\overline{\mathfrak{A}}, u)$ around, or it would be necessarily trace-length dependent. $(\overline{\mathfrak{A}}, u)$ is merely used here to explain the inner workings of the monitor.

T1. [Create SA.] Create an SA, $\mathcal{A}_{\varphi,v}$, in the usual manner.

T2. [Wait for new event.] Let $(\mathfrak{A}, \sigma)$ be the event that was read.

T3. [Update potentially locally accepting runs.] Let B and B' be (initially empty) buffers. If $B = \emptyset$, for each $q \in Q_0$ and for each $q' \in \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma))$: add $(q', [\delta_{\downarrow}(q, (\mathfrak{A}, \sigma))])$ to B. Otherwise, set $B' = B$, and subsequently $B = \emptyset$. Next, for all $(q, [\mathit{obl}_1, \ldots, \mathit{obl}_n]) \in B'$ and for all $q' \in \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma))$: add $(q', [\mathit{obl}_{\mathit{new}}, \mathit{obl}_1, \ldots, \mathit{obl}_n])$ to B, where $\mathit{obl}_{\mathit{new}} = \delta_{\downarrow}(q, (\mathfrak{A}, \sigma))$.

T4. [Create submonitors.] For each $(q, [\mathit{obl}_{\mathit{new}}, \mathit{obl}_1 \ldots, \mathit{obl}_n]) \in B$: call Algorithm M with argument ψ (under corresponding v') for each $\mathcal{A}_{\psi,v'}$ that occurs in $\mathit{obl}_{\mathit{new}}$.

T5. [Iterate over candidate runs.] Assume $B = \{b_0, \ldots, b_m\}$. Create a counter $j = 0$ and set $(q, [\mathit{obl}_0, \ldots, \mathit{obl}_n]) = b_j$ to be the jth element of B.

T6. [Send, receive, replace.] For all $0 \leq i \leq n$: send $(\mathfrak{A}, \sigma)$ to all submonitors corresponding to SAs occurring in obl_i, and wait for the respective verdicts. For every returned $\top$ (resp. $\bot$) replace the corresponding SA in obl_i with $\top$ (resp. $\bot$).

T7. [Corresponding run has violated obligations?] For all $0 \leq i \leq n$: if $\mathit{obl}_i = \bot$, remove b_j from B and go to T9.

T8. [Obligations met?] For all $0 \leq i \leq n$: if $\mathit{obl}_i = \top$, remove obl_i.

T9. [Next run in buffer.] If $j \leq m$, set j to $j + 1$ and go to step T6.

T10. [Communicate verdict.] If $B = \emptyset$, send "no runs" to the calling Algorithm M and return, otherwise send "some run(s)" and go back to T2. ∎

For a given $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ and $(\overline{\mathfrak{A}}, u)$, let us use $M_{\varphi}(\overline{\mathfrak{A}}, u)$ to denote the successive application of Algorithm M for formula φ, first on u_0, then u_1, and so forth. We then get

Theorem 7. $M_{\varphi}(\overline{\mathfrak{A}}, u) = \top \Rightarrow (\overline{\mathfrak{A}}, u) \in \mathrm{good}(\varphi)$ *(resp. for* $\bot$ *and* $\mathrm{bad}(\varphi)$*).*

5.3 Experimental Results

To demonstrate feasibility and to get an intuition on runtime performance we have implemented the above.[4] The only liberty we took in deviating from our description is the following: since the SAs for $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ on the different levels basically consist of ordinary GBAs for the respective subformulae of φ, we have used an "off the shelf" GBA generator, lbt[5]. Moreover, our algorithm bears the advantage that it is possible to precompute the SAs that are required at runtime (i.e., we replaced step T1 in Algorithm T with a look-up in a precomputed table of SAs and merely use a new valuation each time). We also compared our implementation with the somewhat naive (but, arguably, easier to implement) approach of monitoring, described in [6]. There, we used formula rewriting, sometimes referred to as progression (cf. [2]): a function, P, continuously "rewrites" a formula $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ using an observed event, σ, s.t., $\sigma w \models \varphi \Leftrightarrow w \models P(\varphi, \sigma)$ holds.

As a benchmark for our tests, we have used several formulae derived from the well-known specification patterns [13], and added quantification to crucial positions.

[4] Available as open source Scala project on
`https://github.com/jckuester/ltlfo2mon`

[5] `http://www.tcs.hut.fi/Software/maria/tools/lbt/`

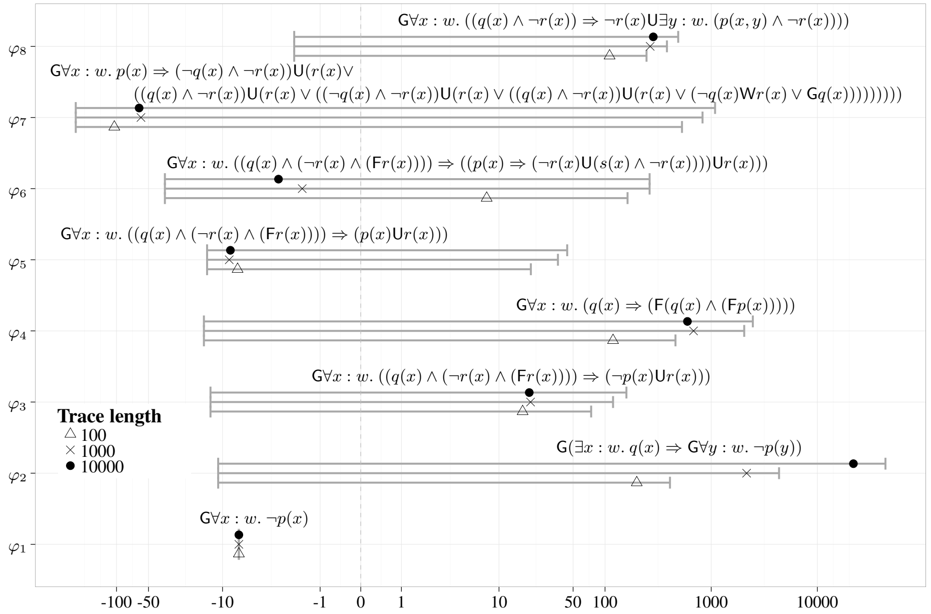


Fig. 2. Difference in space consumption at runtime: SA-based monitor vs. progression

Some of the results are visualised in Fig. 2. For each LTL$^{\text{FO}}$ formula corresponding to a pattern, we randomly generated 20 traces of lengths 100, 1000, and 10000, respectively, and passed them to both algorithms. We then measured the average space consumption of each algorithm at different trace lengths. For progression this is measured simply in terms of the length of the formula at a given time, whereas for the SA-based monitor $M_{\varphi,v}$ it is determined recursively as follows: Recall, $M_{\varphi,v}$ first creates two instances of Algorithm T, $T_{\varphi,v}$ and $T_{\neg\varphi,v}$, each of which creates a buffer, call it B_φ, resp. $B_{\neg\varphi}$. Let $B = B_\varphi \cup B_{\neg\varphi}$, and $(q_i, [obl_{i,0}, \ldots, obl_{i,n}])$ be the i-th element of B, then $|M_{\varphi,v}| = \sum_{i=0}^{|B|-1} |(q_i, [obl_{i,0}, \ldots, obl_{i,n}])| = \sum_{i=0}^{|B|-1}(1 + |\widetilde{obl_{i,0}}| + \ldots + |\widetilde{obl_{i,n}}|)$, where $|\widetilde{obl_{i,j}}| = |obl_{i,j}| + \sum_{\mathcal{A}_{\psi,v} \in obl_{i,j}} |M_{\psi,v}|$, i.e., the sum of the top-level monitor's constituents as well as that of all of its submonitors. Finally, we also need to add the total size of the precomputed GBA look-up table.

The end markers on the left of each horizontal bar show how much bigger in the *worst case* an SA-based monitor is for a given formula compared to the corresponding progression-based monitor (and vice versa for the right markers). The small shapes in the middle denote the *average* size difference of the two monitors over the whole length of a trace. This difference is most striking for φ_2 on longer traces (e.g., $\Delta \geq 10000$ for traces of length 10000), where the average almost coincides with the worst case. As such, this example brings to surface one of the potential pitfalls of progression, namely that a lot of redundant information can accumulate over time: If $\exists x : w.\ q(x)$ ever becomes true, then P, which operates purely on a syntactic level, will produce a new conjunct $\mathsf{G}\forall y : w.\ \neg p(y)$ for each new event, even though semantically it is not necessary. Hence, the longer the trace, the greater the average difference in size (similar in case of φ_3 and φ_4).

At first glance, it may seem a curious coincidence that the left markers of each bar align perfectly, because this indicates that for all three traces that belong to a given formula, the SA-based monitor is in the worst case by exactly the same constant k bigger than the progression-based one, irrespective of the trace. However, it makes sense if we consider when this worst case occurs; that is, whenever the SA-based monitor (and consequently also the progression-based one) does not have to memorise any data at all, in which case the size of the SA-based monitor's look-up table weighs the most; that is, the size of the look-up table is almost equal to k. Usually this happens when monitoring commences, hence, there is a perfect alignment on all traces. On the other hand, the worst case for progression arises whenever the amount of data to be memorised by the monitor has reached its maximum. As this depends not only on the formula, but also on the content of the randomly generated traces (and in some of the examples also on their lengths, as seen in the previous example), we generally don't observe alignment on the right.

For those examples that, on average, favour progression, note that the difference in size is less dramatic—a fact, which may be slightly obfuscated by the pseudo-logarithmic scale of the x-axis. Again, the differences in these examples (φ_1, φ_5, φ_6, and φ_7) can be mostly explained by the fact that an SA-based monitor generally wastes more space for "book keeping". Naturally, all examples are also exposed to a degree of randomness due to the generated traces, which would also deserve closer investigation. Hence, these tests are indicative as well as promising, but certainly not conclusive yet.

6 Related Work

This is by no means the first work to discuss monitoring of first-order specifications. Motivated by checking temporal triggers and temporal constraints, the monitoring problem for different types of first-order logic has been widely studied, e.g., in the database community. In that context, Chomicki [10] presents a method to check for violations of temporal constraints, specified using (metric) past temporal operators. The logic in [10] differs from $\mathrm{LTL}^{\mathrm{FO}}$, in that it allows natural first-order quantification over a single countable and constant domain, whereas quantified variables in $\mathrm{LTL}^{\mathrm{FO}}$ range over elements that occur at the current position of the trace (see also [16,5]). Presumably, to achieve the same effect, [10] demands that policies are what is called "domain independent", so that all statements refer to known objects. As such, domain independence is a property of the policy and shown to be undecidable. In contrast, one could say that $\mathrm{LTL}^{\mathrm{FO}}$ has a similar notion of domain independence already built-in, because of its quantifier. Like $\mathrm{LTL}^{\mathrm{FO}}$, the logic in [10] is also undecidable; no function symbols are allowed and relations are required to be finite. However, despite the fact that the prefix problem is not phrased as a decision problem, its basic idea is already denoted by Chomicki as the potential constraint satisfaction problem. In particular, he shows that the set of prefixes of models for a given formula is not recursively enumerable. On the other hand, the monitor in [10] does not tackle this problem and instead solves what we have introduced as the word problem, which, unlike the prefix problem, is decidable.

Basin et al. [4] extend Chomicki's monitor towards bounded future operators using the same logic. Furthermore, they allow infinite relations as long as these are representable by automatic structures, i.e., automata models. In this way, they show that the restriction on formulae to be domain independent is no longer necessary. $\mathrm{LTL}^{\mathrm{FO}}$, in comparison, is more general, in that it allows computable relations and functions. On the other hand, $\mathrm{LTL}^{\mathrm{FO}}$ lacks syntax to directly specify metric constraints.

The already cited work of Hallé and Villemaire [16] describes a monitoring algorithm for a logic with quantification identical to ours, but without function symbols or arbitrary computable relations. The resulting monitors are generated "on the fly" by using syntax-directed tableaux. In our approach, however, it is possible to pre-compute the individual BAs for the respective subformulae of a policy/levels of the SA, and thereby bound the complexity of that part of our monitor at runtime by a constant factor.

Sistla and Wolfson [23] also discuss a monitor for database triggers whose conditions are specified in a logic, which uses an assignment quantifier that binds a single value or a relation instance to a global, rigid variable. Their monitor is represented by a graph structure, which is extended by one level for each updated database state, and as such proportional in size to the number of updates.

Finally, there are works dealing with so called parametric monitoring which, although not based on first-order logic, offer support for monitoring traces carrying data (cf. [1,24,9]). The approach followed by [9] is to "slice" a trace according to the parameters occurring in it and then to forward the n (effectively propositional) subtraces to n monitor instances of the same specification; for example, one per logged-in user or per opened file. In [1], a similar technique is applied for matching regular expressions with the program trace, when restricted to the symbols declared in an expression. All approaches allow the user to add variables to a specification, but only [24] offers quantifiers. However, to restrict their scope, they must directly precede a positive so called parameterised proposition, which is ensured by syntactic rules that prohibit arbitrary nesting or use of negation that could otherwise help to get around this constraint. None of the approaches support arbitrary nesting of quantifiers and temporal operators, use of negation, or function symbols to name just some important restrictions. However, on the plus side, one is able to use optimised monitoring techniques, developed in the propositional domain, and apply them—with these restrictions in mind—to traces carrying data. For Java, a widely used such implementation is JavaMOP [18].

7 Conclusions

To the best of our knowledge, our algorithm is the first to devise impartial monitors, i.e., address the prefix problem instead of a (variant of the) word problem, for policies given in an undecidable first-order temporal logic. Moreover, unlike other approaches, such as [23,16] and even [6], we are even able to precompute most of the state space required at runtime as the different levels of our SAs correspond to standard GBAs that can be generated before monitoring commences. As required, our monitor is monotonic and in principle trace-length independent. The latter, however, deserves closer examination. Consider formula φ_8 in Fig. 2: once the left hand side of the implication holds, it basically forces the monitor to memorise all occurrences of x for all events and keep

Table 1. Overview of complexity results

	Satisfiability	**Word problem**	**Model checking**	**Prefix problem**
LTL	PSpace-complete	$<$ Bilinear-time	PSpace-complete	PSpace-complete
$\mathrm{LTL}^{\mathrm{FO}}$	Undecidable	PSpace-complete	ExpSpace-membership, PSpace-hard	Undecidable

them around until the right hand side of the U-operator holds. If the right hand side of the U-operator never holds (or not for a very long time), the space consumption of the monitor is *bound* to grow. Hence, unlike in standard LTL, trace-length dependence is not merely a property of the monitor, but also of the specification.

We conjecture that trace-length dependence is generally undecidable. However, if the formula is *not* trace-length dependent, then our monitor is trace-length independent, as desired. Given a $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ of which we know that it is trace-length independent in principle, our monitor's size at runtime at any given time is bounded by $O(|\sigma|^{\mathrm{depth}(\varphi)} \cdot 2^{|\varphi|})$, where σ is the current input to the monitor: Throughout the $\mathrm{depth}(\varphi)$ levels of the monitor, there are a total of $O(|\sigma|^{\mathrm{depth}(\varphi)})$ submonitors, which are of size $O(2^{|\varphi|})$, respectively. In contrast, the size of a progression-based monitor, even for obviously trace-length independent formulae, such as φ_2 in Fig. 2 is, in the worst case, proportional to the trace length.

In Table 1 we have summarised the main results of §2–§4, highlighting again the differences of LTL compared to $\mathrm{LTL}^{\mathrm{FO}}$. Note that as far as trace-length dependence goes, for LTL it is always possible to devise a trace-length independent monitor, irrespective of the specification at hand (cf. [8]).

Acknowledgements. Our thanks go to Patrik Haslum, Michael Norrish and Peter Baumgartner for helpful comments on earlier drafts of this paper.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proc. 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 345–364. ACM (2005)
2. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence 22, 5–27 (1998)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
4. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
5. Bauer, A., Gore, R., Tiu, A.: A first-order policy language for history-based transaction monitoring. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 96–111. Springer, Heidelberg (2009)
6. Bauer, A., Küster, J.-C., Vegliach, G.: Runtime verification meets Android security. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 174–180. Springer, Heidelberg (2012)

7. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. Computing Research Repository (CoRR) abs/1303.3645. ACM (March 2013)
8. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology 20(4), 14 (2011)
9. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009)
10. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
11. Chomicki, J., Niwinski, D.: On the feasibility of checking temporal integrity constraints. J. Comput. Syst. Sci. 51(3), 523–535 (1995)
12. Dong, W., Leucker, M., Schallhart, C.: Impartial anticipation in runtime-verification. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 386–396. Springer, Heidelberg (2008)
13. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: Proc. 21st Intl. Conf. on Softw. Eng. (ICSE), pp. 411–420. IEEE (1999)
14. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
15. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers: When an efficient LTL algorithm on sequences is needed to model-check traces. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 557–572. Springer, Heidelberg (2006)
16. Halle, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: Proc. 12th Enterprise Distr. Object Comp. Conf. (EDOC), pp. 63–72. IEEE (2008)
17. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Software Tools for Technology Transfer 6(2), 158–173 (2004)
18. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: Proc. 34th Intl. Conf. on Softw. Eng. (ICSE), pp. 1427–1430. IEEE (2012)
19. Kuhtz, L., Finkbeiner, B.: Efficient parallel path checking for linear-time temporal logic with past and bounds. Logical Methods in Computer Science 8(4) (2012)
20. Libkin, L.: Elements of Finite Model Theory. Springer (2004)
21. Markey, N., Schnoebelen, P.: Model checking a path. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 251–265. Springer, Heidelberg (2003)
22. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM 32(3), 733–749 (1985)
23. Sistla, A.P., Wolfson, O.: Temporal triggers in active databases. IEEE Trans. Knowl. Data Eng. 7(3), 471–486 (1995)
24. Stolz, V.: Temporal assertions with parametrized propositions. J. Log. Comp. 20(3), 743–757 (2010)

Right-Universality of Visibly Pushdown Automata

Véronique Bruyère[1], Marc Ducobu[1], and Olivier Gauwin[2]

[1] University of Mons, UMONS, Belgium
[2] University of Bordeaux, LaBRI, France

Abstract. Visibly pushdown automata (VPAs) express properties on structures with a nesting relation such as program traces with nested method calls. In the context of runtime verification, we are interested in the following problem: given u, the beginning of a program trace, and $\mathcal{A}$, a VPA expressing a property to be checked on this trace, can we ensure that any extension uv of u will be accepted by $\mathcal{A}$? We call this property right-universality w.r.t. u. We propose an online algorithm detecting at the earliest position of the trace, whether this trace is accepted by $\mathcal{A}$. The decision problem associated with right-universality is ExpTime-complete. Our algorithm uses antichains and other optimizations, in order to avoid the exponential blow-up in most cases. This is confirmed by promising experiments conducted on a prototype implementation.

1 Introduction

Program traces describe the control flows of program executions, including subroutine calls and returns. Some properties of program traces are specific to the nesting structure of calls and returns, as for instance reentrant locks [BYBC10]. These properties are not captured by regular languages, but can be expressed by visibly pushdown languages [AM09]. These latter languages rely on a partitioning of the input alphabet into internal letters, call letters, and return letters, which naturally fits to program traces. A language over such a partitioned alphabet is *visibly pushdown* if there exists a *visibly pushdown automaton* (VPA) recognizing it. VPAs are pushdown automata, where operations on the stack are driven by the letter type: call letters can only push, return letters can only pop, while internal letters do not have access to the stack.

Runtime verification amounts to check a property during a program execution. In this paper we are interested in finding the earliest point during the execution where the property can be asserted, for properties defined by VPAs. More formally, given a word u over a partitioned alphabet and a VPA $\mathcal{A}$, we say that $\mathcal{A}$ is *right-universal w.r.t.* u if, for every possible continuation v of u, the word uv is accepted by $\mathcal{A}$. Hence, if u is a prefix of the trace w and $\mathcal{A}$ is known to be right-universal w.r.t. u, then it can be asserted that w verifies the property described by $\mathcal{A}$, without reading w entirely. Our aim is to check right-universality *incrementally* after each incoming event of the trace w, as described

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 76–93, 2013.

Algorithm 1 Checking right-universality incrementally

```
function INCREMENTAL-RIGHT-UNIVERSALITY($\mathcal{A}$)
    $u \leftarrow \epsilon$
    while trace $w$ is not completely read do
        $a \leftarrow$ next letter of $w$
        $u \leftarrow ua$
        if $\mathcal{A}$ is right-universal w.r.t. $u$ then
            return True
        end if
    end while
    return False
end function
```

in Algorithm 1. By incremental, we mean that some information is propagated from one event (reading a letter in w) to the next one (reading the next letter in w), avoiding repeated identical computations. Note that our algorithm will not store u, but enough information for asserting right-universality of $\mathcal{A}$ w.r.t. u.

For *safety properties*, right-universality allows to know at the earliest time point, whether the execution is sure, and thus whether controls can be stopped. This is typically the case for properties looking for a pattern (potentially complicated) that must occur during the execution. For *properties to be avoided*, this permits to stop the program before it enters an unsafe configuration, hence avoiding potential attacks [BJLW08].

These questions started to be addressed in the context of XML through *earliest query answering* of XPath expressions [BYFJ05, GNT09]. Indeed, XML documents also entail a nesting structure, similarly to program traces, and properties (or queries) over these documents can be expressed using VPAs. An algorithm asserting at the earliest time point, whether an XML document is accepted by a deterministic VPA has been proposed in [GNT09]. This algorithm runs in polynomial time at each incoming event. For non-deterministic VPAs, however, the decision problem associated with right-universality (i.e. given $\mathcal{A}$ and u, is $\mathcal{A}$ right-universal w.r.t. u?) is known to be ExpTime-complete [GNT09].

Algorithms that incrementally check right-universality of *non-deterministic* VPAs are challenging in several aspects. First, VPAs are usually an intermediate object, resulting from the translation of a property expressed in some logics, like XPath for XML documents. Such translations rely on non-determinism, as for instance when referring to any call inside a given procedure call (which corresponds to the descendant axis in XPath). Any VPA can be determinized, but the procedure yields VPAs of exponential size [AM09]. Second, right-universality w.r.t. u can be considered as a variant of universality, parameterized by u. Recent techniques have been proposed to check universality of non-deterministic VPAs efficiently [TO12, FKL13, BDG13]. Among them, *antichains* have been successfully applied to decision problems related to non-deterministic automata: universality and inclusion for finite word automata [DDHR06], and for non-deterministic bottom-up tree automata [BHH$^+$08]. Whether these techniques also apply to incrementally check right-universality is an interesting issue. Third, the algorithm proposed in the deterministic case [GNT09] does not directly generalize to the non-deterministic case.

Our contributions are the following. We propose an efficient incremental algorithm checking right-universality of a non-deterministic VPA $\mathcal{A}$. This algorithm relies on the progressive computation of *safe sets of configurations*, the configuration of a VPA being a state together with a stack content. It avoids to enterily determinize the given VPA. This algorithm uses antichains in order to get a compact representation of safe sets of configurations. We also propose efficient operators and data structures for saving useless computation. We report on some experiments, performed on randomly generated VPAs, and also on VPAs resulting from a translation from XPath expressions. These results exhibit the benefits of our optimizations.

Verification of traces with a nesting structure has already been addressed, through different aspects. Let us mention some of them. In [AEM04], the logic CaRet over words with a nesting structure is introduced, and a model checking algorithm is proposed. An extension is presented in [RCB08], with a corresponding monitor synthesis algorithm. VPAs are also sometimes used as an intermediate model to express properties on program traces [CA07, FJJ$^+$12].

The paper is structured as follows. In Section 2 we introduce VPAs and right-universality. Safe sets of configurations are defined in Section 3. Our antichain-based algorithm is described in Section 4, and further optimizations in Section 5. Experiments are reported in Section 6.

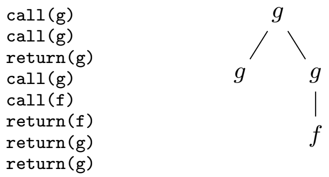
2 Visibly Pushdown Automata and Right-Universality

Visibly pushdown automata (VPAs) are pushdown automata working on a partitioned alphabet where only call symbols can push, return symbols can pop, and internal symbols can fire transitions without considering the stack [AM04, AM09].

In this paper, we consider trees, instead of words with a nesting structure (i.e. their linearization). Such a mapping is illustrated in Figure 1. Each call and its matching return are mapped to a node, and the calls and returns directly nested under this call correspond to the children of this node. These trees are unranked, as the number of children of a node is not determined by its label. We use VPAs as *unranked tree acceptors*, operating on their linearization [GNR08]. In particular these VPAs do not use internal symbols.

2.1 Unranked Trees

We here recall the standard definition of unranked trees, as provided for instance in [CDG$^+$07]. Let Σ be a finite *alphabet*, and Σ^* (resp. Σ^+) be the set of all words (resp. non empty words) over Σ. The empty word is denoted by ϵ. Given two words $v, w \in \Sigma^*$ over Σ, v is a *prefix* (resp. *proper prefix*) of w if there exists a word $v' \in \Sigma^*$ (resp. $v' \in \Sigma^+$) such that $vv' = w$.



```
call(g)
call(g)
return(g)
call(g)
call(f)
return(f)
return(g)
return(g)
```

Fig. 1. Representation of a program trace as a tree t

An *unranked tree* t over Σ is a tree such that its nodes are labeled by a letter of Σ and have an arbitrary number of children (the children are ordered from left to right). We call *a-node* a node with label $a \in \Sigma$. The set of all unranked trees over Σ is denoted by T_Σ. A *hedge* h over Σ is a finite sequence (empty or not) of unranked trees over Σ. The empty hedge is denoted by ϵ, and the set of all hedges over Σ is denoted by H_Σ.

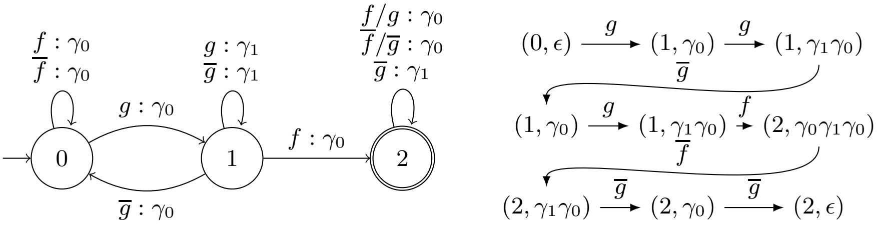
Trees can be described by well-balanced words which correspond to a depth-first traversal of the tree. An opening tag is used to notice the arrival on a node and a closing tag to notice the departure from a node. For each $a \in \Sigma$, let a itself represent the opening tag and $\overline{a}$ the related closing tag. The *linearization* $[t]$ of $t \in T_\Sigma$ is the *well-balanced* word over $\Sigma \cup \overline{\Sigma}$, with $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$, inductively defined by: $[t] = a\ [t_1] \cdots [t_n]\ \overline{a}$, with a is the label of the root and $t_1, \ldots, t_n$ are its n subtrees (from left to right). The linearization is extended to hedges as follows. Let $h = t_1 \cdots t_n$ be the sequence of trees t_i, $1 \leq i \leq n$. Then $[h] = [t_1] \cdots [t_n]$. We denote by $[T_\Sigma]$ (resp. $[H_\Sigma]$) the set of linearizations of all trees in T_Σ (resp. hedges in H_Σ). Consider for instance the tree t in Figure 1: its linearization is the word $[t] = gg\overline{g}gf\overline{f}\overline{g}\overline{g}$. Let $\mathit{Pref}(T_\Sigma)$ denote the set of all prefixes of $[T_\Sigma]$: $\mathit{Pref}(T_\Sigma) = \{u \in (\Sigma \cup \overline{\Sigma})^* \mid \exists v \in (\Sigma \cup \overline{\Sigma})^*,\ uv \in [T_\Sigma]\}$.

2.2 Visibly Pushdown Automata

Definition 1. *A* visibly pushdown automaton $\mathcal{A}$ *over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ where Q is a finite set of states containing initial states $Q_i \subseteq Q$ and final states $Q_f \subseteq Q$, a finite set Γ of stack symbols, and a finite set Δ of rules. Each rule in Δ is of the form $q \xrightarrow{a:\gamma} q'$ with $a \in \Sigma \cup \overline{\Sigma}$, $q, q' \in Q$, and $\gamma \in \Gamma$.*

The left-hand side of a rule $q \xrightarrow{a:\gamma} p \in \Delta$ is (q, a) if $a \in \Sigma$, and (q, a, γ) if $a \in \overline{\Sigma}$. A VPA is *deterministic* if it has at most one initial state, and it does not have two distinct rules with the same left-hand side. We provide an example of deterministic VPA in Figure 2a.

A *configuration* of a VPA $\mathcal{A}$ is a pair (q, σ) where $q \in Q$ is a state and $\sigma \in \Gamma^*$ a stack content. A configuration is *initial* (resp. *final*) if $q \in Q_i$ (resp. $q \in Q_f$) and $\sigma = \epsilon$. For $a \in \Sigma \cup \overline{\Sigma}$, we write $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if there is a transition $q \xrightarrow{a:\gamma} q'$ in Δ verifying $\sigma' = \gamma \cdot \sigma$ if $a \in \Sigma$, and $\sigma = \gamma \cdot \sigma'$ if $a \in \overline{\Sigma}$. We extend this notation to words $u = a_1 \cdots a_n$, by writing $(q_0, \sigma_0) \xrightarrow{u} (q_n, \sigma_n)$ whenever there exist configurations (q_i, σ_i) such that $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$ for all $1 \leq i \leq n$.


$f : \gamma_0$
$\overline{f} : \gamma_0$
$g : \gamma_1$
$\overline{g} : \gamma_1$
$f/g : \gamma_0$
$\overline{f}/\overline{g} : \gamma_0$
$\overline{g} : \gamma_1$
$g : \gamma_0$
$f : \gamma_0$
$\overline{g} : \gamma_0$
0
1
2
$(0,\epsilon) \xrightarrow{g} (1,\gamma_0) \xrightarrow{g} (1,\gamma_1\gamma_0)$
$\overline{g}$
$(1,\gamma_0) \xrightarrow{g} (1,\gamma_1\gamma_0) \xrightarrow{f} (2,\gamma_0\gamma_1\gamma_0)$
$\overline{f}$
$(2,\gamma_1\gamma_0) \xrightarrow{\overline{g}} (2,\gamma_0) \xrightarrow{\overline{g}} (2,\epsilon)$

(a) VPA $\mathcal{A}$ over alphabet $\Sigma = \{f, g\}$.

(b) Run of $\mathcal{A}$ on $[t]$.

Fig. 2. An automaton and one of its runs

A *run* of a VPA $\mathcal{A}$ on a linearization $[t] = a_1 \cdots a_n$ of $t \in T_\Sigma$ is a sequence $(q_0, \sigma_0) \cdots (q_n, \sigma_n)$ of configurations such that (q_0, σ_0) is initial, and for every $1 \leq i \leq n$, $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$. Such a run is *accepting* if (q_n, σ_n) is final. A tree t is *accepted* by $\mathcal{A}$ if there is an accepting run on its linearization $[t]$. The set of accepted trees is called the *language* of $\mathcal{A}$ and is written $L(\mathcal{A})$.[1] For instance, given a tree t, the VPA of Figure 2a checks whether t has a g-node with an f-child. An accepting run on $[t]$ for the tree t of Figure 1, is depicted in Figure 2b.

2.3 Universality and Right-Universality

We conclude the section of preliminaries with the notions of universality[2] and right-universality, that we will study in the remainder of the paper.

Definition 2. *A VPA $\mathcal{A}$ over Σ is said* universal *if $\mathcal{A}$ accepts all trees $t \in T_\Sigma$, i.e. $L(\mathcal{A}) = T_\Sigma$. Let $u \in Pref(T_\Sigma) \setminus \{\epsilon\}$ be a non empty prefix of $[t_0]$ for some tree $t_0 \in T_\Sigma$. The VPA $\mathcal{A}$ is said* right-universal w.r.t. u *if for all trees $t \in T_\Sigma$, if u is a prefix of $[t]$, then t is accepted by $\mathcal{A}$.*

In other words, right-universality w.r.t. u allows to assert that any tree linearization beginning with u is accepted by the automaton. In this definition, notice that when $u = [t_0]$, then $\mathcal{A}$ is right-universal w.r.t. u iff t_0 is accepted by $\mathcal{A}$. Moreover, as a universal VPA is right-universal w.r.t. all non empty words $u \in Pref(T_\Sigma)$, we assume in the sequel that VPAs are not universal.

In this article, our objective is to propose an *incremental* algorithm for right-universality of a VPA $\mathcal{A}$, in the sense described in the introduction (see Algorithm 1). More precisely, the linearization $[t_0]$ of a given tree t_0 is read letter by letter[3], and while $\mathcal{A}$ is not right-universal w.r.t. the current read prefix u of $[t_0]$, the next letter of $[t_0]$ is read. When processing a new letter, we try to reuse

[1] VPAs considered in this article are tree acceptors, with acceptance on empty stack.

[2] This notion of universality is different from the one proposed for usual VPAs, where a VPA is universal if it accepts all words of $(\Sigma \cup \overline{\Sigma})^*$ (and not only linearizations of trees).

[3] $[t_0]$ is the trace w of Algorithm 1.

prior computations as much as possible. In the sequel t_0 always refers to this particular tree.

We recall that the right-universality problem is ExpTime-complete for VPAs, but in PTime for deterministic VPAs [GNT09]. A dual problem to right-universality w.r.t. u is to ask whether every word starting with u is *rejected* by a given VPA. This problem is in PTime, even when the VPA is not deterministic. Indeed, this amounts to check whether no configuration reached after reading u can reach a final configuration.

2.4 Towards an Algorithm

In this section we consider several approaches for addressing right-universality of VPAs, and justify our choice of considering safe sets of configurations. Readers not familiar with the related litterature can skip this section at first reading. As explained in the introduction, we discard the naive approach consisting in determinizing the VPA and then applying the algorithm in [GNT09], in order to avoid state-space explosion.

The algorithm given in [GNT09] for deterministic VPAs is a progressive computation of *safe states*, with the property that the VPA is right-universal w.r.t. u iff the state reached after reading u is safe. Note that safe states cannot be determined statically: a state can be safe at some position of the word, but not at another one. A first idea for the non-deterministic case is a subset construction, using sets of safe states instead of safe states, but this direct adaptation is not correct. Hence, *safe sets of configurations* will be the basis of our algorithm, instead of safe states. This point raises new questions, as the configuration space is infinite, unlike the state space. In fact we will see that at each event, safe sets of configurations have the same stack height as the depth of the word u leading to this event, and are thus of finite (but potentially huge) cardinality at each time point.

Recently, several authors have proposed efficient algorithms to check universality of VPAs [TO12, FKL13, BDG13]. The method that we give in [BDG13] computes in an incremental way the set $\mathcal{R}$ of accessibility relations for all hedges (i.e. $\{\mathsf{rel}_h \mid h \in H_\Sigma\}$ in the present paper, also called summaries in [AM04]). It uses *antichains* to get smaller objects to manipulate (in particular to limit $\mathcal{R}$ to a strict smaller subset) and to avoid an explicit determinization step. It can be easily adapted for checking right-universality of VPAs [BDG12, Section 3.4]. However we face the problem of computing the whole set $\mathcal{R}$ (instead of a strict subset): experiments indicate that this step is too much time consuming [BDG13]. For this reason we do not follow this approach, but use antichains over safe sets of configurations rather than accessibility relations. This approach is developed in the next section.

3 Safe Sets of Configurations

In this section, $[t_0]$ is a fixed tree whose linearization is read letter by letter, and u denotes the current read prefix. We introduce the new notion of safe set of configurations related to u. We show how safe sets of configurations can be defined from such sets associated to smaller prefixes u' of $[t_0]$. This property will be useful to derive an incremental algorithm for checking right-universality of a VPA.

3.1 A Notion of Safety for Right-Universality

Definition 3. *Let $\mathcal{A}$ be a VPA and $\mathscr{C} \subseteq Q \times \Gamma^*$ be a non empty set of configurations. Let $u \in \mathit{Pref}(T_\Sigma) \setminus \{\epsilon\}$ be a non empty prefix of t_0.*

- $\mathscr{C}$ is safe for u *if for every v such that $uv \in [T_\Sigma]$, there exist $(q, \sigma) \in \mathscr{C}$ and $p \in Q_f$ such that $(q, \sigma) \xrightarrow{v} (p, \epsilon)$ in $\mathcal{A}$.*
- $\mathscr{C}$ is leaf-safe for u *if for every $v = \overline{a}v'$ with $\overline{a} \in \overline{\Sigma}$ such that $uv \in [T_\Sigma]$, there exist $(q, \sigma) \in \mathscr{C}$ and $p \in Q_f$ such that $(q, \sigma) \xrightarrow{v} (p, \epsilon)$ in $\mathcal{A}$.*[4]

Let $\mathit{Safe}(u) = \{\mathscr{C} \mid \mathscr{C}$ is safe for $u\}$ and $\mathit{LSafe}(u) = \{\mathscr{C} \mid \mathscr{C}$ is leaf-safe for $u\}$.

Intuitively, as stated in Theorem 1 below, if $\mathscr{C}$ is the set of configurations reached in $\mathcal{A}$ after reading u, then $\mathcal{A}$ is right-universal w.r.t. u iff $\mathscr{C}$ is safe for u. Indeed, for every possible v, one can find in $\mathscr{C}$ at least one configuration leading to an accepting configuration after reading v. Before stating this theorem, let us give the next definition. Let $\mathit{Reach}(u)$ denote the set of configurations (q, σ) such that $(q_0, \sigma_0) \xrightarrow{u} (q, \sigma)$ for some initial configuration (q_0, σ_0) of $\mathcal{A}$.

Theorem 1. *$\mathcal{A}$ is right-universal w.r.t. u iff $\mathit{Reach}(u) \in \mathit{Safe}(u)$.*

Let us illustrate this on the VPA $\mathcal{A}$ depicted in Figure 2a. Recall that this VPA checks that the tree has a g-node with an f-child. After opening a g-root, the set of safe configurations is $\mathit{Safe}(g) = \{\{(q_2, \gamma_1)\}, \{(q_2, \gamma_0)\}\}$, which does not contain the set of reached configurations, as $\mathit{Reach}(g) = \{(q_1, \gamma_0)\}$. Indeed, $\mathcal{A}$ is not right-universal w.r.t. g, as no f-node has been encountered yet under a g-node. For the word gf, as expected, the situation differs: $\mathit{Safe}(gf) = \{\{(q_2, \gamma_1\gamma_0)\}, \{(q_2, \gamma_0\gamma_0)\}\}$, and $\mathit{Reach}(gf) = \{(q_2, \gamma_0\gamma_0)\}$ is a safe set of configurations. By Theorem 1, $\mathcal{A}$ is right-universal w.r.t. gf.

Hence, an algorithm computing both $\mathit{Reach}(u)$ and $\mathit{Safe}(u)$ can decide right-universality w.r.t. u. The set $\mathit{Reach}(u)$ is easy to compute, just by firing transitions of the VPA. In Sections 3.2-3.4, we detail how the set $\mathit{Safe}(u)$ can be defined from the set $\mathit{Safe}(u')$ with u' a proper prefix of u. In this way, while reading the linearization $[t_0]$ of a given tree t_0, the set $\mathit{Safe}(u)$ with $u \neq \epsilon$ prefix of $[t_0]$, can be incrementally defined. In Section 4, we turn this approach into an algorithm.

[4] The name "leaf-safe" comes from the fact that we only consider suffixes starting with a $\overline{\Sigma}$ symbol, and thus continuations of the tree that do not add children to the current node.

3.2 Starting Point

We begin with $\mathit{Safe}(a)$ with $a \in \Sigma$, for which we recall the definition.

$$\mathit{Safe}(a) = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \forall h \in H_\Sigma, \exists q_f \in Q_f, \exists (q,\sigma) \in \mathscr{C} : (q,\sigma) \xrightarrow{h\overline{a}} (q_f,\epsilon)\} \quad (1)$$

3.3 Reading a Letter $\overline{a} \in \overline{\Sigma}$

Suppose that we are reading an $\overline{a} \in \overline{\Sigma}$ and we have to compute $\mathit{Safe}(u\overline{a})$. Two cases occur : either $u\overline{a} \neq [t_0]$, or $u\overline{a} = [t_0]$. Let us study these two cases and show how to manage them from an algorithmic point of view.

If $u\overline{a} \neq [t_0]$, we can retrieve safe sets configurations from prior computed such sets: $\mathit{Safe}(u\overline{a}) = \mathit{Safe}(u')$ where $u' \neq \epsilon$ is the unique prefix of u such that $u = u'a[h]$, with $h \in H_\Sigma$. Indeed as shown by Lemma 1 below, we have $\mathit{Safe}(u'a[h]\overline{a}) = \mathit{Safe}(u')$. Hence, from an algorithmic point of view, we just have to use a stack $\mathcal{S}$ to store these safe sets of configurations. When opening a, we put $\mathit{Safe}(u')$ on the stack, and when closing $\overline{a}$, we pop it. As h is a hedge, the stack before reading $\overline{a}$ is exactly the stack after reading a.

Lemma 1. *If $h \in H_\Sigma$, then $\mathit{Safe}(u[h]) = \mathit{Safe}(u)$ and $\mathit{LSafe}(u[h]) = \mathit{LSafe}(u)$.*

If $u\overline{a} = [t_0]$, the previous argument is no longer correct since $u' = \epsilon$ and $\mathit{Safe}(u')$ is not defined for the empty word. Nevertheless, by definition $\mathit{Safe}(u\overline{a}) = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \exists (q,\epsilon) \in \mathscr{C}$ with $q \in Q_f\}$. Therefore we can again use stack $\mathcal{S}$ as described before if it is initialized with the set

$$\mathit{Init} = \{\mathscr{C} \subseteq Q \times \Gamma^* \mid \exists (q,\epsilon) \in \mathscr{C} \text{ with } q \in Q_f\}. \quad (2)$$

3.4 Reading a Letter $a \in \Sigma$

Let us now consider the much more involved case of sets $\mathit{Safe}(ua)$ with $a \in \Sigma$. When reading an $a \in \Sigma$, two successive steps are performed, with leaf-safe sets of configurations as intermediate object:

$$\mathit{Safe}(u) \quad \xrightarrow{\text{Step 1}} \quad \mathit{LSafe}(ua) \quad \xrightarrow{\text{Step 2}} \quad \mathit{Safe}(ua)$$

For this purpose, we introduce the notion of *predecessor*. Let $\mathscr{C}, \mathscr{C}'$ be two sets of configurations, let $\overline{a} \in \overline{\Sigma}$ and $h \in H_\Sigma$.

- *$\mathscr{C}$ is an $\overline{a}$-predecessor of $\mathscr{C}'$* if $\forall (q',\sigma') \in \mathscr{C}'$, $\exists (q,\sigma) \in \mathscr{C}$, $(q,\sigma) \xrightarrow{\overline{a}} (q',\sigma')$.
- *$\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$* if $\forall (q',\sigma') \in \mathscr{C}'$, $\exists (q,\sigma) \in \mathscr{C}$, $(q,\sigma) \xrightarrow{[h]} (q',\sigma')$.

Let $\mathit{Pred}_{\overline{a}}(\mathscr{C}') = \{\mathscr{C} \mid \mathscr{C}$ is an $\overline{a}$-predecessor of $\mathscr{C}'\}$ and $\mathit{Pred}_h(\mathscr{C}') = \{\mathscr{C} \mid \mathscr{C}$ is a h-predecessor of $\mathscr{C}'\}$.

The next proposition can be used to perform Step 1 and Step 2. It states that safe sets of configurations are only among predecessors of prior safe sets of configurations.

Proposition 1. *Let $ua \in \mathit{Pref}(T_\Sigma)$.*

$$\mathscr{C} \in \mathit{LSafe}(ua) \iff \exists \mathscr{C}' \in \mathit{Safe}(u),\ \mathscr{C} \in \mathit{Pred}_{\overline{a}}(\mathscr{C}') \tag{3}$$

$$\mathscr{C} \in \mathit{Safe}(ua) \iff \forall h \in H_\Sigma,\ \exists \mathscr{C}' \in \mathit{LSafe}(ua),\ \mathscr{C} \in \mathit{Pred}_{\overline{h}}(\mathscr{C}') \tag{4}$$

However, to get an algorithm, we face the problem that the number of hedges to consider in Equivalence (4) is infinite. We use relations to overcome this. Also the size of $\mathit{Safe}(u)$ may be huge and not all configurations of $\mathit{Safe}(u)$ are crucial for checking right-universality w.r.t. u. We use antichains to get a compact representation of $\mathit{Safe}(u)$. These two concepts are explained in the following section.

4 An Antichain-Based Algorithm for Right-Universality

In this section, we give an antichain-based algorithm for incrementally checking right-universality of a VPA, that uses the approach given in Sections 3.2-3.4.

Let us summarize this approach. Let $[t_0]$ be the linearization of a given tree t_0. We initialize a stack $\mathcal{S}$ with set *Init* defined in (2) and we start by computing $\mathit{Safe}(u)$ with u being the first letter of $[t_0]$ (see (1)). Suppose that $\mathit{Safe}(u)$ has been computed from the current read prefix u of $[t_0]$. Then, if the next letter read in $[t_0]$ is $a \in \Sigma$, we compute $\mathit{Safe}(ua)$ from $\mathit{Safe}(u)$ by Steps 1 and 2, and we put $\mathit{Safe}(u)$ on the stack $\mathcal{S}$. If the next letter read in $[t_0]$ is $\overline{a} \in \overline{\Sigma}$, then we pop the stack $\mathcal{S}$. The element that has been popped is the set $\mathit{Safe}(u') = \mathit{Safe}(u\overline{a})$ where u' is the unique prefix of u such that $u = u'a[h]$ (except if $u\overline{a} = t_0$ in which case the popped set is equal to $\mathit{Init} = \mathit{Safe}(u\overline{a})$), see Section 3.3. At each computation of $\mathit{Safe}(u)$, we check whether $\mathit{Reach}(u) \in \mathit{Safe}(u)$ (see Theorem 1).

4.1 Finite Number of Hedges

We begin by showing that only a finite number of hedges has to be considered in Equivalence (4). The reason is that a hedge h does not change the stack of a configuration during a run of a VPA, that is $(q, \sigma) \xrightarrow{[h]} (q', \sigma)$. So h can be considered as a function mapping each state q to the set of states obtained when traversing h from q. Formally, for every $h \in H_\Sigma$, rel_h is the function from Q to 2^Q such that $q' \in \mathsf{rel}_h(q)$ iff $(q, \sigma) \xrightarrow{[h]} (q', \sigma)$ for some $\sigma \in \Gamma^*$. The number of such functions is finite, and bounded by $|Q| \cdot 2^{|Q|}$. These functions naturally define an equivalence relation of finite index over H_Σ:

$$h \sim h' \iff \mathsf{rel}_h = \mathsf{rel}_{h'}.$$

Let us note H for a subset containing one hedge per $\sim$-class. We have $|H| \leq |Q| \cdot 2^{|Q|}$. The next lemma indicates that the computation of h-predecessors can be limited to $h \in H$.

Lemma 2. *For every $h \in H_\Sigma$, $\mathscr{C}$ is a h-predecessor of $\mathscr{C}'$ iff there exists $h' \in H$ with $h \sim h'$, such that $\mathscr{C}$ is a h'-predecessor of $\mathscr{C}'$.*

The set H can be computed by a saturation method based on the VPA rules.

4.2 Antichains

Let us now introduce antichains. Consider the set $2^{Q\times\Gamma^*}$ of all sets of configurations, with the $\subseteq$ operator. An *antichain* is a set of pairwise incomparable sets of configurations with respect to $\subseteq$. Given a set α of sets of configurations, we denote by $\lfloor\alpha\rfloor$ the $\subseteq$*-minimal* elements of α. The set α is $\subseteq$*-upward closed* if for all $\mathscr{C}\in\alpha$ and $\mathscr{C}\subseteq\mathscr{C}'$, we have $\mathscr{C}'\in\alpha$. From their definition, it is immediate that $\mathit{Safe}(u)$, $\mathit{LSafe}(u)$, $\mathit{Pred}_{\overline{a}}(\mathscr{C})$ and $\mathit{Pred}_h(\mathscr{C})$ are $\subseteq$-upward closed sets.

The idea is to compute the antichain $\lfloor\mathit{Safe}(u)\rfloor$ (resp. $\lfloor\mathit{LSafe}(u)\rfloor$) instead of the whole $\subseteq$-upward closed set $\mathit{Safe}(u)$ (resp. $\mathit{LSafe}(u)$). The next corollary of Theorem 1 indicates that such a limited computation is enough to check whether a VPA $\mathcal{A}$ is right-universal w.r.t. u.

Corollary 1. *$\mathcal{A}$ is right-universal w.r.t. u iff there exists $\mathscr{C}\in\lfloor\mathit{Safe}(u)\rfloor$ such that $\mathscr{C}\subseteq\mathit{Reach}(u)$.*

Moreover, it can be shown that the antichains $\lfloor\mathit{Safe}(u)\rfloor$ and $\lfloor\mathit{LSafe}(u)\rfloor$ are finite and only contain finite sets $\mathscr{C}$ of configurations such that $\mathscr{C}\subseteq Q\times\Gamma^k$ for some k. We now try to use these antichains at the starting point, and in Steps 1 and 2 of our approach.

4.3 Starting Point with Antichains

Let us explain how to compute $\mathit{Safe}(a)$. Clearly, by definition of H and using (1) (see Section 3.2), we can compute $\lfloor\mathit{Safe}(a)\rfloor$ as follows:

$$\lfloor\mathit{Safe}(a)\rfloor=\left\lfloor\left\{\mathscr{C}\mid\forall h\in H,\exists q_f\in Q_f,\exists(q,\sigma)\in\mathscr{C}:(q,\sigma)\xrightarrow{h\overline{a}}(q_f,\epsilon)\right\}\right\rfloor. \quad (5)$$

4.4 Step 1 with Antichains: From $\lfloor\mathit{Safe}(u)\rfloor$ to $\lfloor\mathit{LSafe}(ua)\rfloor$

For the two steps, the goal is to adapt Proposition 1 so that it uses $\lfloor\mathit{Safe}(.)\rfloor$ instead of $\mathit{Safe}(.)$, and $\lfloor\mathit{LSafe}(.)\rfloor$ instead of $\mathit{LSafe}(.)$. We begin with Step 1. Implication ($\Rightarrow$) of Equivalence (3) can be directly adapted.

$$\mathscr{C}\in\lfloor\mathit{LSafe}(ua)\rfloor\implies\exists\mathscr{C}'\in\lfloor\mathit{Safe}(u)\rfloor,\ \mathscr{C}\in\mathit{Pred}_{\overline{a}}(\mathscr{C}'). \quad (6)$$

Implication (6) gives us a way to compute $\lfloor\mathit{LSafe}(ua)\rfloor$ from $\lfloor\mathit{Safe}(u)\rfloor$: it suffices to take all $\overline{a}$-predecessors of elements of $\lfloor\mathit{Safe}(u)\rfloor$ and then limit to those predecessors that are $\subseteq$-minimal. We can even only consider minimal $\overline{a}$-predecessors of $\lfloor\mathit{Safe}(u)\rfloor$ in the following sense: $\mathscr{C}$ is a *minimal* $\overline{a}$*-predecessor* of $\mathscr{C}'$ if for all $\mathscr{C}''$ $\overline{a}$-predecessor of $\mathscr{C}'$, $\mathscr{C}''\subseteq\mathscr{C}\implies\mathscr{C}''=\mathscr{C}$. We finally obtain:

$$\lfloor\mathit{LSafe}(ua)\rfloor=\lfloor\{\mathscr{C}\mid\mathscr{C}\text{ is a minimal }\overline{a}\text{-predecessor of }\mathscr{C}'\in\lfloor\mathit{Safe}(u)\rfloor\}\rfloor. \quad (7)$$

4.5 Step 2 with Antichains: From $\lfloor \boldsymbol{LSafe(ua)} \rfloor$ to $\lfloor \boldsymbol{Safe(ua)} \rfloor$

The second step for computing $\lfloor Safe(ua) \rfloor$ from $\lfloor Safe(u) \rfloor$ relies on the introduction of antichains in Equivalence (4). Implication (⇒) holds with antichains:

$$\mathscr{C} \in \lfloor Safe(ua) \rfloor \implies \forall h \in H_\Sigma,\ \exists \mathscr{C}' \in \lfloor LSafe(ua) \rfloor,\ \mathscr{C} \in Pred_{\overline{h}}(\mathscr{C}'). \tag{8}$$

Similarly to Implication (6), we can restrict h-predecessors to only consider minimal ones: $\mathscr{C}$ is a *minimal h-predecessor* of $\mathscr{C}'$ if for all $\mathscr{C}''$ h-predecessor of $\mathscr{C}'$, $\mathscr{C}'' \subseteq \mathscr{C} \implies \mathscr{C}'' = \mathscr{C}$. Moreover, by Lemma 2, we can limit the computations to hedges $h \in H$ where H is the finite set introduced in Section 4.1. Therefore, we obtain the next equality.

$$\lfloor Safe(ua) \rfloor =$$
$$\left\lfloor \left\{ \mathscr{C} \ \middle|\ \mathscr{C} = \bigcup_{h \in H} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor LSafe(ua) \rfloor \right\} \right\rfloor \tag{9}$$

4.6 Algorithm

We are now able to give our antichain-based algorithm (see Algorithm 2). It uses a stack $\mathcal{S}$ (initially empty) as recalled at the beginning of this section. The computation of $\lfloor H \rfloor$ is considered as a *preprocessing*, as its value only depends on $\mathcal{A}$ and not on $[t_0]$. The used results are mentioned inside the algorithm.

5 Improvements and Implementation

Section 4 resulted in a first algorithm for incrementally testing whether a VPA is right-universal. In this section, we show how this algorithm can be improved by limiting hedges to consider, and optimizing operators and predecessors to be computed. We also give the improved algorithm in a more detail way than in Algorithm 2, as well as the underlying data structures.

5.1 Minimal Hedges

A first improvement is obtained by further restricting hedges to consider. Indeed it suffices to consider *minimal hedges* wrt their function rel_h. Formally, let us write $h \leq h'$ whenever $\mathsf{rel}_h(q) \subseteq \mathsf{rel}_{h'}(q)$ for every $q \in Q$. We denote by $\lfloor H \rfloor$ the $\leq$-minimal elements of H. From the definition of h-predecessor, we have:

$$\mathscr{C}\ h\text{-predecessor of } \mathscr{C}' \text{ and } h \leq h' \implies \mathscr{C}\ h'\text{-predecessor of } \mathscr{C}' \tag{10}$$

This property can be used to replace $h \in H$ in (9) by $h \in \lfloor H \rfloor$:

$$\lfloor Safe(ua) \rfloor =$$
$$\left\lfloor \left\{ \mathscr{C} \ \middle|\ \mathscr{C} = \bigcup_{h \in \lfloor H \rfloor} \mathscr{C}_h \text{ with } \mathscr{C}_h \text{ a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor LSafe(ua) \rfloor \right\} \right\rfloor \tag{11}$$

Algorithm 2 Checking right-universality incrementally with antichains

```
function ANTICHAIN-BASED INCREMENTAL-RIGHT-UNIVERSALITY($\mathcal{A}$, $\lfloor H \rfloor$)
    PUSH($\mathcal{S}, \lfloor Init \rfloor$)    % by (2), $\mathcal{S}$ being a stack
    $u \leftarrow$ first letter of $[t_0]$
    $R \leftarrow \langle$Compute $Reach(u)\rangle$
    $\alpha \leftarrow \langle$Compute $\lfloor Safe(u) \rfloor\rangle$    % by (5)
    if $\exists \mathscr{C} \in \alpha : \mathscr{C} \subseteq R$ then
        return True    % Corollary 1
    end if
    while $[t_0]$ is not completely read do
        $a \leftarrow$ next letter of $[t_0]$
        $u \leftarrow ua$
        $R \leftarrow \langle$Compute $Reach(u)$ from $R\rangle$
        if $a \in \overline{\Sigma}$ then
            $\alpha \leftarrow$ POP($\mathcal{S}$)
        else
            PUSH($\mathcal{S}, \alpha$)
            $\alpha \leftarrow \langle$Compute $\lfloor Safe(u) \rfloor$ from $\alpha\rangle$    % by (7) and (9)
        end if
        if $\exists \mathscr{C} \in \alpha : \mathscr{C} \subseteq R$ then
            return True    % by Corollary 1
        end if
    end while
    return False
end function
```

and similarly for the starting point:

$$\lfloor Safe(a) \rfloor = \left\lfloor \left\{ \mathscr{C} \mid \forall h \in \lfloor H \rfloor, \exists q_f \in Q_f, \exists (q,\sigma) \in \mathscr{C} : (q,\sigma) \xrightarrow{h\overline{a}} (q_f,\epsilon) \right\} \right\rfloor \quad (12)$$

5.2 An Appropriate Operator

Equation (11) expresses that every set of configurations $\mathscr{C}$ in $\lfloor Safe(ua) \rfloor$ is the union of $\mathscr{C}_h$ with $h \in \lfloor H \rfloor$. We introduce a new operator to improve the readability and find new properties. Let S be a finite set, and $A, B \in 2^{2^S \setminus \{\emptyset\}}$. The set $A \sqcup B \in 2^{2^S}$ is defined by

$$A \sqcup B = \{a \cup b \ \mid \ a \in A \text{ and } b \in B\}.$$

Operator $\sqcup$ builds sets obtained by taking one set of each of its operands, and performing their union. It is obviously associative and commutative. Notice that the elements of A, B are supposed to be non empty sets. This is always the case in the algorithms using this operator.

When combined with operator $\lfloor . \rfloor$, operands of the $\sqcup$ operator can be splitted, so that $\sqcup$ is to be computed on smaller sets:

$$\lfloor A \sqcup B \rfloor = \lfloor (A \cap B) \cup (A \setminus B \ \sqcup \ B \setminus A) \rfloor \quad (13)$$

We experimentally observed that a good strategy to evaluate an expression $A_1 \sqcup \ldots \sqcup A_n$ is to process sets A_i with increasing cardinality.

Equation (11) can now be rewritten in a simpler way as follows.

$$\lfloor \mathit{Safe}(ua) \rfloor = \left\lfloor \bigsqcup_{h \in \lfloor H \rfloor} \{\mathscr{C}_h \mid \mathscr{C}_h \text{ is a minimal } h\text{-predecessor of } \mathscr{C}' \in \lfloor \mathit{LSafe}(ua) \rfloor\} \right\rfloor \quad (14)$$

We can go further by reconsidering the notions of minimal $\overline{a}$- and h-predecessor with the new operator $\sqcup$. From the definition of minimal predecessor, we immediately get that $\mathscr{C}$ is a minimal $\overline{a}$-predecessor of $\mathscr{C}'$ iff $\mathscr{C}$ belongs to the set $\left\lfloor \bigsqcup_{(q',\sigma') \in \mathscr{C}'} \left\{ \{(q,\sigma)\} \mid (q,\sigma) \xrightarrow{\overline{a}} (q',\sigma') \right\} \right\rfloor$, and similarly for h-minimal predecessors. A similar improvement can be provided for $\lfloor \mathit{Safe}(a) \rfloor$.

5.3 Using a SAT Solver to Find Minimal Predecessors

The operator $\sqcup$ allows to compute Step 1 and Step 2. Equation (13) accelerates these computations. In this section, we propose a method to compute Step 1 based on a SAT solver (a similar approach also works for Step 2).

A SAT solver is an algorithm used to efficiently test the satisfiability of a boolean formula φ, that is, to check whether there exists a valuation v of the boolean variables of φ that makes φ true. In this case we say that v is a *model* of φ, denoted by $v \models \varphi$.

In Step 1, the computation of set $\lfloor \mathit{LSafe}(ua) \rfloor$ from $\lfloor \mathit{Safe}(u) \rfloor$ is given in (7):

$$\lfloor \mathit{LSafe}(ua) \rfloor = \lfloor \{\mathscr{C} \mid \mathscr{C} \text{ is an } \overline{a}\text{-predecessor of } \mathscr{C}' \in \lfloor \mathit{Safe}(u) \rfloor\} \rfloor .$$

We recall that $\mathscr{C}$ is an $\overline{a}$-predecessor of $\mathscr{C}'$ if for all $(q',\sigma') \in \mathscr{C}'$, there exists $(q,\sigma) \in \mathscr{C}$ such that $(q,\sigma) \xrightarrow{\overline{a}} (q',\sigma')$. We also recall that $\mathscr{C} \in \lfloor \mathit{LSafe}(ua) \rfloor$ is a finite object such that $\mathscr{C} \subseteq Q \times \Gamma^k$ for some k. Let us associate a boolean variable x_c to each configuration $c \in Q \times \Gamma^k$.

We consider the following boolean formula $\varphi_{\overline{a}}$:

$$\varphi_{\overline{a}} = \bigvee_{\mathscr{C}' \in \lfloor \mathit{Safe}(u) \rfloor} \bigwedge_{c' \in \mathscr{C}'} \bigvee_{c \xrightarrow{\overline{a}} c'} x_c,$$

Let $v_{\mathscr{C}}$ be the valuation such that $v_{\mathscr{C}}(x_c) = 1$ iff $c \in \mathscr{C}$. We immediately obtain that:

$$v_{\mathscr{C}} \models \varphi_{\overline{a}} \qquad \text{iff} \qquad \mathscr{C} \in \mathit{LSafe}(ua) \cap Q \times \Gamma^k.$$

We define an ordering over valuations as follows, in a way to have a notion of minimal models equivalent to $\subseteq$-minimal elements of $\mathit{LSafe}(ua)$.

Let V be a set of boolean variables, let v and v' be two valuations over V. We define $v' \leq v$ iff for all variables $x \in V$, $v'(x) = 1 \implies v(x) = 1$. We denote $v' < v$ if $v' \leq v$ and $v' \neq v$. Given φ a boolean formula over V, we say that a model v of φ is *minimal* if for all model v' of φ, we have $v' \leq v \implies v' = v$. We get the next characterization.

Lemma 3. *$v_{\mathscr{C}}$ is a minimal model of $\varphi_{\overline{a}}$ iff $\mathscr{C} \in \lfloor LSafe(ua) \rfloor$.*

We can now explain how to compute all the minimal models of formula $\varphi_{\overline{a}}$. Let φ be a boolean formula over V.
First, we explain, knowing a model v of φ, how to compute a model v' of φ such that $v' < v$ (if it exists). Consider the next formula φ':

$$\varphi' = \varphi \wedge (\bigwedge_{x \in V_0} \neg x) \wedge (\bigvee_{x \in V_1} \neg x)$$

where V_0 (respectively V_1) is the set of all variables $x \in V$ such that $v(x) = 0$ (resp. $v(x) = 1$). If φ' has a model v', it follows from the definition of φ' that v' is a model of φ such that $v' < v$. Otherwise, v is a minimal model of φ. So from a model of φ we can compute a minimal model of φ by repeating the above procedure.
Second, let us explain how to compute all the minimal models of φ. Suppose that we already know some minimal model v of φ, and let V_1 be the set of variables $x \in V$ such that $v(x) = 1$. Consider the formula

$$\varphi' = \varphi \wedge (\bigvee_{x \in V_1} \neg x).$$

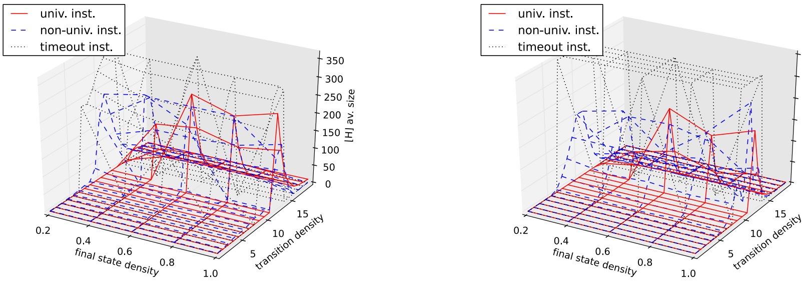
Then a model v' of φ', if it exists, is a model of φ such that neither $v' < v$ (since v is minimal) nor $v < v'$ (by definition of φ'). With the previous procedure, we thus get a minimal model of φ that is distinct from v. In this way we can compute all minimal models of φ.

5.4 Improved Algorithm and Data Structures

Let us come back to Algorithm 2 by indicating the underlying data structures and the improvements resulting from the previous three sections.

As explained in Section 5.1, we restrict the computations to the set $\lfloor H \rfloor$ of minimal hedges. Notice that the algorithm that computes the set $\{\mathsf{rel}_h \mid h \in H\}$ can be easily adapted to compute the set of its $\leq$-minimal elements. In the main loop of Algorithm 2, Steps 1 and 2 can be computed either with the new operator $\sqcup$ or with a SAT solver (see Sections 5.2 and 5.3 resp.).

Efficient data structures are used both for the relations associated to minimal hedges and for the antichains $\lfloor H \rfloor$, $\lfloor Safe(u) \rfloor$ and $\lfloor LSafe(u) \rfloor$. A relation rel_h, with $h \in \lfloor H \rfloor$, is stored as an array of bit-vectors. In this way the composition is computed efficiently using bit-operations, as well as the number of elements of the relation. A hash table is used to store each antichain, such that elements with different weights are stored in different lists. In the case of $\lfloor H \rfloor$, the weight is the number of elements of $\mathsf{rel}_h \in \lfloor H \rfloor$. In the case of $\lfloor Safe(u) \rfloor$ (resp. $\lfloor LSafe(u) \rfloor$), the weight is the number of elements of $\mathscr{C} \in \lfloor Safe(u) \rfloor$ (resp. $\mathscr{C} \in \lfloor LSafe(u) \rfloor$). In this way, comparing a new element with the elements of the antichain is made more efficient, by limiting the comparison with elements of the same weight.



(a) Size of $\lfloor H \rfloor$.

(b) Time to compute $\lfloor H \rfloor$.

Fig. 3. Space and time consumption for computing $\lfloor H \rfloor$ on 50 VPAs of various densities, $|Q| = 10$, $|\Sigma| = |\Gamma| = 3$, and timeout of 60s

6 Experiments

We have implemented Algorithm 2 in a prototype tool together with the data structures and improvements proposed in Section 5 (following both approaches, using operator $\sqcup$ and a SAT solver). We mainly use Python, with C binding to the Glucose SAT solver (first ranked in recent SAT competitions) [Glucose].

The experimental tests are performed on two different benchmarks, one composed of randomly generated VPAs, and another one based on the translation of XPath expressions to VPAs. The experiments were run on a PC equipped with an Intel i7 2.8GHz processor, 6 GB of RAM and running Linux Ubuntu 3.2.

6.1 Randomly Generated VPAs

During the random generation of VPAs $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$, $|Q_i|$ is fixed to 1, and several parameters vary: the sizes $|Q|, |\Sigma|$ and $|\Gamma|$, the density of final states $\frac{|Q_f|}{|Q|}$, and the transition density[5]. Our online algorithm for checking right-universality needs to compute the set $\lfloor H \rfloor$ of minimal hedges as a preprocessing. This set can be huge and thus lead to a timeout. In Figure 3a (resp. Figure 3b), we indicate the average size of $\lfloor H \rfloor$ (resp. the average execution time to compute it) for randomly generated VPAs with variable transition density (from 1 to 19) and variable final state density. In this test, we distinguish universal automata from non-universal ones since a universal VPA is right-universal w.r.t all $u \in \mathit{Pref}(T_\Sigma) \setminus \{\epsilon\}$. The two figures show that timeout happens for instances with transition density around 12, and that the density of final states has few influence.

In the next experiment, we study the behavior of our algorithm on 90 random instances of size 10 (resp. 20, 30), with fixed transition density 8 (resp.

[5] Equal to the number of outgoing transitions per state and per symbol.

Table 1. Number for each type (universal, right-universal, null rel_h, timeout) among 90 instances, with a timeout of 300s, and various number of states and transition densities. Average (preprocessing/online) time is given in parentheses (in s).

$\|Q\|$	universality		right-universality		null rel_h		timeout (preproc.)		timeout (online)	
	tr. density: 8	tr. density: 16	8	16	8	16	8	16	8	16
10	12 (0.59/0.01)	40 (1.19/0.01)	22 (16.62/9.36)	44 (11.00/1.10)	52 (0.67/0.01)	0	2	2	2	4
20	13 (9.35/0.01)	29 (0.86/0.01)	14 (20.44/30.85)	41 (5.18/1.07)	46 (9.62/0.01)	0	14	17	3	3
30	9 (20.22/0.01)	34 (13.39/0.01)	11 (21.23/5.85)	34 (27.31/3.43)	40 (1.23/0.01)	0	30	21	0	1

Table 2. Time and data structures size on random VPAs, with transition density of 16, $|Q| = 20$, $|\Sigma| = |\Gamma| \in \{2, 3, 4\}$, and timeout of 300s

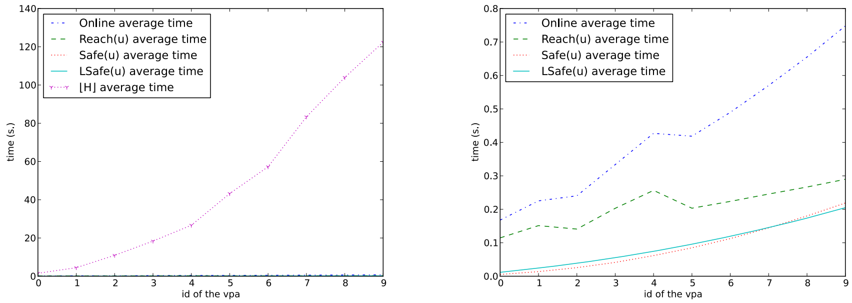
VPA id	Response	Time (s.)	$\|\lfloor H \rfloor\|$	$\|Reach(u)\|$	$\|\lfloor Safe(u) \rfloor\|$
q20-a02-x02-o16-c16-f0.5-00	right-universal w.r.t. a_1	0.03	3	4	33
q20-a02-x04-o16-c16-f0.5-02	right-universal w.r.t. a_1	0.07	3	8	53
q20-a02-x04-o16-c16-f0.5-06	right-universal w.r.t. a_0	0.06	3	5	54
q20-a03-x02-o16-c16-f0.5-06	right-universal w.r.t. a_2	0.13	12	4	32
q20-a03-x03-o16-c16-f0.5-06	right-universal w.r.t. a_0	0.37	23	4	59
q20-a03-x03-o16-c16-f0.5-09	right-universal w.r.t. a_2a_0	1.74	13	14	179
q20-a03-x04-o16-c16-f0.5-07	right-universal w.r.t. a_2	1.02	44	5	116
q20-a04-x02-o16-c16-f0.5-03	right-universal w.r.t. a_3	0.71	71	5	75
q20-a04-x02-o16-c16-f0.5-07	right-universal w.r.t. a_0	1.07	81	5	74
q20-a04-x03-o16-c16-f0.5-03	right-universal w.r.t. a_2	15.88	359	5	447

16) and fixed density 0.5 of final states[6]. The considered tree t_0 is a complete binary tree up to height 3 filled with randomly generated letters of Σ. We only comment the experiment using a SAT solver since it outperforms the approach with operator $\sqcup$. We observe several behaviors (see Table 1): many automata are either universal, or right-universal w.r.t. u with $|u| = 1$ (except 7 cases where $|u| = 2$), or exhibits a hedge h with rel_h being the null relation[7]; the number of timeout increases with $|Q|$. Table 2 indicates, for some representative VPAs, the execution time and the sizes of $\lfloor H \rfloor$, $Reach(u)$ and $\lfloor Safe(u) \rfloor$.

When it declares that a given VPA is right-universal w.r.t. u, our algorithm has the nice property that it reproduces the *same execution* (thus with the same time) for each tree t_0 such that u is prefix of $[t_0]$. This is clearly not the case for the *membership algorithm* that computes $Reach([t_0])$ and checks if it contains a final configuration. For instance, on a random VPA with size $|Q| = 20$ and transition density 8, our algorithm consumes less than 5s to declare that the automaton is right-universal w.r.t. a, whereas the membership algorithm takes more than 300s as soon as the height of a binary tree t_0 with an a-root is equal to 8.

[6] To deal with managable and not too small sets $\lfloor H \rfloor$.

[7] This means that the automaton is never right-universal w.r.t. u, for any proper prefix u of $[t_0]$. Therefore, in such cases, our algorithm is slower that the membership algorithm.



(a) With the $\lfloor H \rfloor$ average time curve. (b) Without the $\lfloor H \rfloor$ average time curve.

Fig. 4. Average time for computing *Safe*(u), *LSafe*(u) and *Reach*(u) and overall computation for 15 random trees

6.2 VPAs Resulting from XPath Translation

Our second benchmark is based on VPAs obtained from queries over XML documents expressed in the XPath language, and then translated into VPAs. This translation was performed by the QuiXProc tool, as described in [GN11]. This family of XPath expressions yields VPAs of linear size increase (VPA with id i has $16 + 11i$ states), and looks for some complex patterns in the tree. In Figure 4 we report the time used by our algorithm on randomly generated trees, for this family of VPAs. This shows that for real-world VPAs, the size of $\lfloor H \rfloor$ is outside the hardness threshold exhibited in Figure 3. For instance, for the VPA with id 9 and size $|Q| = 115$, $\lfloor H \rfloor$ is computed in about 120s. Moreover, Figure 4b shows the efficiency of our online algorithm (less than 0.8s).

Acknowledgements. We thank Grégoire Sutre for useful discussions, and the QuiXProc team for translating XPath expressions of our benchmark to VPAs. We also thank one of the reviewers who helped us to improve the presentation of this article. The second author is supported by a grant from FRIA. This work was also partially supported by CNRS SOSP project.

References

[AEM04] Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

[AM04] Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proc. STOC, pp. 202–211. ACM Press (2004)

[AM09] Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56, 1–43 (2009)

[BYFJ05] Bar-Yossef, Z., Fontoura, M., Josifovski, V.: Buffering in query evaluation over XML streams. In: Proc. PODS, pp. 216–227. ACM Press (2005)

[BJLW08] Benedikt, M., Jeffrey, A., Ley-Wild, R.: Stream Firewalling of XML Constraints. In: Proc. SIGMOD Conference, pp. 487–498. ACM-Press (2008)

[BHH$^+$08] Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)

[BDG12] Bruyère, V., Ducobu, M., Gauwin, O.: Visibly pushdown automata on trees: universality and u-universality, CoRR abs/1205.2841 (2012)

[BDG13] Bruyère, V., Ducobu, M., Gauwin, O.: Visibly pushdown automata: Universality and inclusion via antichains. In: Dediu, A.-H., Martín-Vide, C., Truthe, B. (eds.) LATA 2013. LNCS, vol. 7810, pp. 190–201. Springer, Heidelberg (2013)

[BYBC10] Bultan, T., Yu, F., Betin-Can, A.: Modular verification of synchronization with reentrant locks. In: Proc. MEMOCODE, pp. 59–68. IEEE Computer Society (2010)

[CA07] Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 279–283. Springer, Heidelberg (2007)

[CDG$^+$07] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://www.grappa.univ-lille3.fr/tata`

[DDHR06] De Wulf, M., Doyen, L., Henzinger, T., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)

[FJJ$^+$12] Fredrikson, M., Joiner, R., Jha, S., Reps, T., Porras, P., Saïdi, H., Yegneswaran, V.: Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 548–563. Springer, Heidelberg (2012)

[FKL13] Friedmann, O., Klaedtke, F., Lange, M.: Ramsey goes visibly pushdown. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 224–237. Springer, Heidelberg (2013)

[GN11] Gauwin, O., Niehren, J.: Streamable fragments of forward XPath. In: Bouchou-Markhoff, B., Caron, P., Champarnaud, J.-M., Maurel, D. (eds.) CIAA 2011. LNCS, vol. 6807, pp. 3–15. Springer, Heidelberg (2011)

[GNR08] Gauwin, O., Niehren, J., Roos, Y.: Streaming tree automata. Information Processing Letters 109, 13–17 (2008)

[GNT09] Gauwin, O., Niehren, J., Tison, S.: Earliest query answering for deterministic nested word automata. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 121–132. Springer, Heidelberg (2009)

[Glucose] Glucose, `www.lri.fr/~simon/?page=glucose`

[RCB08] Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008)

[TO12] Nguyen, T.V., Ohsaki, H.: On model checking for visibly pushdown automata. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 408–419. Springer, Heidelberg (2012)

Distributed Finite-State Runtime Monitoring with Aggregated Events*

Kevin Falzon[1], Eric Bodden[1], and Rahul Purandare[2]

[1] European Center for Security and Privacy by Design (EC-SPRIDE)
{kevin.falzon,eric.bodden}@ec-spride.de
[2] Department of Computer Science and Engineering
University of Nebraska-Lincoln
rpuranda@cse.unl.edu

Abstract. Security information and event management (SIEM) systems usually consist of a centralized monitoring server that processes events sent from a large number of hosts through a potentially slow network. In this work, we discuss how monitoring efficiency can be increased by switching to a model of *aggregated traces*, where monitored hosts buffer events into lossy but compact batches. In our trace model, such batches retain the number and types of events processed, but not their order.

We present an algorithm for automatically constructing, out of a regular finite-state property definition, a monitor that can process such aggregated traces. We discuss the resultant monitor's complexity and prove that it determines the set of possible next states without producing false negatives and with a precision that is optimal given the reduced information the trace carries.

1 Introduction

In this work, we consider a common scenario to which runtime monitoring is nowadays often applied, namely that of security information and event management (SIEM) systems [9]. Such systems, mainly designed for intrusion detection or the discovery of insider attacks, usually comprise a centralized monitoring server that processes events sent from a large number of hosts within a local company network. At peak times, these hosts might be slowed down significantly, as they block while trying to synchronously send off event information to an overloaded monitoring server [11].

We address this problem by proposing a trace model in which the monitored hosts can aggregate parts of the event stream, retaining the number and types of events processed, but not their order. Discarding ordering information allows event streams to be compressed effectively, whilst retaining event frequencies and types maintains a certain level of precision. In comparison to related work [5, 12], this trace model is not probabilistic and does not allow for "gaps" in the event stream—every occurring event is indeed accounted for. The aggregated trace rather provides an *over*-approximation that implicitly includes all permutations of the original trace it represents.

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE. (www.ec-spride.de).

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 94–111, 2013.

As the main contribution of this paper, we define an algorithm to automatically derive, from a finite-state property definition, a runtime monitor that can process such aggregated traces. We prove that the current formulation of our algorithm produces monitors that are guaranteed not to miss property violations. As we also show, the monitor processes the compressed event stream in bounded time, while not producing more false positives than a naïve monitor that traverses the original property state machine using all possible permutations of the original event trace. Our algorithm can be easily parameterized with different acceptance conditions that can decrease the number of warnings further while allowing for some missed violations.

To summarize, this paper presents the following original contributions:

- a general trace model in which trace producers supply, for certain periods of time, aggregated information about events that occurred during those periods,
- an algorithm that automatically constructs a monitor for such traces from a finite-state property definition,
- a proof that the algorithm always converges and that the resulting monitor is guaranteed not to miss any actual violations and is optimally precise given the aggregated trace information it receives, and
- a complexity estimation for the resulting monitoring algorithm.

The remainder of this paper is structured as follows. Section 2 presents a motivating example, leading to a description of *constraint automata* in Section 3. Section 4 defines a process for translating finite state automata into constraint automata, and Section 5 regards complexity and implementation considerations. The paper discusses related work in Section 6 and concludes in Section 7.

2 Introductory Example

Consider the automaton illustrated in Figure 1. We follow the model of a trace classifier, where different accepting states can cause different error messages, for instance as implemented through JavaMOP [8]. In this example, both the event sequences `logout` · `login` and `login` · `logout` are in the language of the automaton, yet they lead to different states, which in our model means that they would be classified differently.

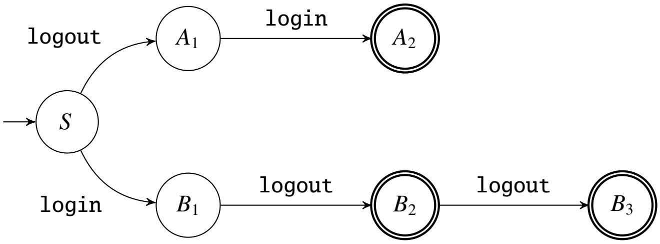


Fig. 1. A deterministic automaton

In this work, we consider the situation where monitored hosts in a distributed system wish to compress data to decrease the load on the network. One of the most effective ways to compress an event trace is to aggregate all occurring events in a data structure that captures the events' types and frequencies but discards their order. In our example, the host could retain the number of times that `login` and `logout` events were observed. The monitor would then face the challenge that, on receiving a compressed batch of two events, the next state would be uncertain (A_2 or B_2). Importantly, though, receiving a subsequent `logout` event would confirm that the automaton must be in state B_3, and would cause the automaton to converge onto a single state.

A naïve approach to processing such "compressed traces" with incomplete information about ordering constraints would be to define a special transition procedure over an unmodified finite-state property automaton. In such a model, on receiving an aggregated batch of events, one would be able to determine the possible next states by traversing the automaton with each legal permutation of events that satisfies the aggregated input, be it through brute-force generation of traces, or by using the automaton as a generator. Using the example illustrated in Figure 1, consider the case where the monitor has observed the following compressed batch of events:

$$\{\langle \texttt{logout}, 2\rangle, \langle \texttt{login}, 1\rangle\}$$

This signifies the arrival of two `logout` events and one `login` event, without any information on the order in which they were received. In a different scenario, one might consider aggregate traces that only record N, that is, the number of events that occurred, without even recording their type. For such a model, any state reachable within N steps of the current state is a potential next state. Preserving the type restricts the possible next state set to a subset of these states. In general, the larger the window and number of aggregated events, the greater the uncertainty of the end state, as the automaton will have potentially progressed to a greater depth.

The problem with determining the next state set via naïve traversal is that the running time will grow exponentially with the number of observed events. Thus, this work proposes an ahead-of-time automaton transformation of finite-state property automata into a data structure which, on being presented with a current state and an input of aggregated events, computes the set of possible and valid next states *efficiently*, within a time bounded by the size of the structure rather than that of the input.

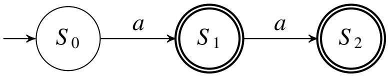
3 Constraint Automata

The following section introduces the notions of *constraints* and *constraint automata*, defining their structure, evaluation and traversal.

3.1 Overview

The transitions in a finite-state automaton determine the number, type and order of input elements required to move between states. In the scenario we consider, compressed inputs are *unordered*, containing information only pertaining to each element's frequency.

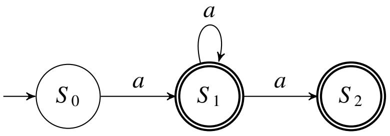
The problem of computing the set of next states can be reformulated in terms of reachability. Each state can be seen as having a set of associated conditions, or *constraints*, on the input. If a constraint evaluates to true, then the system can transition into that state. To compute a complete set of next states, one has to check these conditions for every reachable state.



State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v = 1$
S_2	$\#_a^v = 2$

Fig. 2. A linear automaton with constraints on a trace v received at S_0

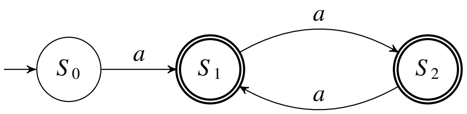
Figure 2 illustrates a very simple linear automaton, and the conditions required for entering each other state when starting from S_0. The function $\#_a^v$ returns the frequency with which symbol a appears in trace v. The constraints defined are strict and unambiguous, referring to specific frequencies. Thus, for example, if one a is observed at S_0, then the system can only be in state S_1.



State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v \geq 1$
S_2	$\#_a^v \geq 2$

Fig. 3. Modified constraints on introducing a loop at S_1

Precision becomes an issue once *loops* enter the equation. Figure 3 illustrates an automaton similar to that shown in Figure 2, yet the addition of a self-loop has weakened the conditions, which can now only place a lower bound on the number of observed events. However, as can be seen in the example illustrated in Figure 4, loops do not always introduce ambiguity; in this example, the automaton goes to a unique state under any input despite the presence of loops.



State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v \geq 1 \wedge (\#_a^v - 1) \bmod 2 = 0$
S_2	$\#_a^v \geq 2 \wedge (\#_a^v - 2) \bmod 2 = 0$

Fig. 4. A flip-flop automaton. Following a mandatory single input, the automaton oscillates between S_1 and S_2, with the final state depending on whether an even or odd number of inputs has been received

In general, fixed sequences of transitions precisely define the number of elements that must be observed for the end state to be reached. Loops consume elements in multiples of the number of elements along their path. For instance, the self-loop on S_1

in Figure 3 consumes a symbols in multiples of one, while the loop formed between S_1 and S_2 in Figure 4 consumes a symbols in multiples of two. In the presence of loops, fixed sequences outside of loops will set a lower bound on the number of elements that must be consumed.

3.2 The Constraint Automaton Model

The examples illustrated defined constraints with respect to a single start state. By considering every state as an initial state, one may construct a *constraint automaton* that accepts aggregated event sets instead of single event inputs. A constraint automaton has a state for each state in the original automaton, and a transition (and consequently a constraint) for each pair of connected states. This allows the constraint automaton to deduce the next states from any configuration, facilitating the processing of sequences of compressed traces.

A constraint automaton is evaluated as a subset automaton, i.e., the "next state" is modelled as a set of states, as the loss of event ordering may lead to multiple valid traversals. In general, shorter compressed traces leave less room for ambiguity. Varying the degree of compression may help re-converge the automaton in instances where the current state set is large (note that a sequence of aggregated event sets that are all singletons is essentially equivalent to a regular trace).

Losing event ordering makes analysis inherently incomplete. More specifically, if a trace leads to a final state in a given finite-state automaton, then its compressed version may lead to multiple states in the derived constraint automaton. As will be seen in Section 6, while certain alternative approaches apply statistical methods to determine the most likely next state, a loss of information will generally introduce uncertainty. Given that this incompleteness cannot be eliminated, it is more relevant to reason in terms of *relative precision*, that is, if the permutations of a given trace lead to a certain set of next states in a finite state automaton, then that trace's compressed version must return *exactly the same set* of next states in the constraint automaton.

Before progressing any further, we first define the notion of an *aggregated event set*, and transforming a trace into such a set, as follows:

Definition 1 (Aggregated Event Set). *An aggregated event set for an alphabet Σ is a set of pairs mapping elements to frequencies, with one pair defined for each element in Σ. The set of possible aggregated events for alphabet Σ, denoted by $\mathcal{A}_\Sigma$, is defined as:*

$$\begin{aligned}\mathcal{A}_\Sigma \overset{def}{=} \{s \mid s \in 2^{\Sigma \times \mathbb{N}}, |s| = |\Sigma|,\\ c_1 \in s, c_2 \in s, c_1 = \langle a_1, n_1 \rangle, c_2 = \langle a_2, n_2 \rangle, (c_1 \neq c_2) \rightarrow (a_1 \neq a_2)\}\end{aligned}$$

Definition 2 (Trace to Aggregated Count). *The aggregated event set for a trace v over events Σ, denoted by $\downarrow\#^v_\Sigma$, is defined as:*

$$\downarrow\#^v_\Sigma \overset{def}{=} \{\langle a, \#^v_a \rangle \mid a \in \Sigma\}$$

3.3 Regular Expressions as Constraints

Based on the defined constraint-automaton model, the next step is to devise a method for representing, deriving and evaluating the constraints. One possible constraint representation would be as *regular expressions*. For each pair of states in the original automaton, one would derive regular expressions that encompass all paths that lead from one state to the other. Several algorithms for deriving regular expressions from automata exist, such as the one described by Brzozowski [7]. Given any two states q and q' in an automaton, $\mathtt{regex}_{\mathcal{D}}(q, q')$ will return a regular expression for the set of strings which, starting from q, lead to q', or $\perp$ if there are no paths between the two states.

To evaluate a compressed trace against a regular expression constraint, one would check whether the expression matches at least one legal permutation of the elements in the compressed trace. However, this procedure is computationally expensive, making regular expressions inadequate for representing constraints in a monitoring context.

3.4 Constraints and Constraint Expressions

Given the inadequacy of regular expressions for representing constraints, the remainder of this section details *constraint expressions*, which are more amenable to direct comparisons with compressed traces. Section 4 then describes a rewriting system for converting regular expressions into such constraint expressions.

Definition 3 (Basic Constraint). *A* basic constraint $\mathbb{C}$ *is a tuple of the form* $2^{\mathbb{N}}{}_{\mathbb{N}\cdot\Sigma}$ *consisting of a set of* moduli*, a numerical* offset*, and the* symbol *being constrained.*

Example 1. The constraint on entering state S_1 from S_2 in the automaton illustrated in Figure 4 would be expressed as the following basic constraint:

$$\{2\}_{1\cdot a}$$

For an observed aggregate input v, this constraint would be true when at least one a symbol has been observed (denoted by the subscript), and when $\#_a^v - 1$ is a multiple of 2. The precise evaluation procedure will be described in Section 3.5.

Definition 4 (Well-formed Constraint Vector). *A* constraint vector $\overrightarrow{\mathbb{C}}$ *is a set of basic constraints. For a constraint vector to be* well-formed *with respect to an automaton with alphabet Σ, it must contain exactly one basic constraint for each element of Σ.*

Example 2. A *well-formed constraint vector* for an automaton with $\Sigma \overset{\text{def}}{=} \{a, b\}$.

$$\{\{2\}_{1\cdot a}, \emptyset_{3\cdot b}\}$$

Definition 5 (Constraint Expression). *A* constraint expression $\hat{\mathbb{C}}$ *is a logical formula of the form*

$$\hat{\mathbb{C}} := \overrightarrow{\mathbb{C}} \mid \hat{\mathbb{C}} \mathbin{\underline{\vee}} \hat{\mathbb{C}} \mid \hat{\mathbb{C}} \mathbin{\underline{\cdot}} \hat{\mathbb{C}} \mid \hat{\mathbb{C}}^{\underline{n}} \mid \hat{\mathbb{C}}^{\underline{*}} \mid \perp$$

The empty constraint expression is represented by $\perp$, and $\hat{\mathbb{C}} \mathbin{\underline{\vee}} \perp \equiv \perp \mathbin{\underline{\vee}} \hat{\mathbb{C}} \equiv \hat{\mathbb{C}} \mathbin{\underline{\cdot}} \perp \equiv \perp \mathbin{\underline{\cdot}} \hat{\mathbb{C}} \equiv \hat{\mathbb{C}}$, whereas $\perp^{\underline{*}} \equiv \perp^{\underline{n}} \equiv \perp$

Definition 6 (Disjunctive Constraint Expression). *A* Disjunctive Constraint Expression *(DCE) is a constraint expression consisting solely of well-formed constraint vectors and $\underline{\vee}$ operators.*

Example 3. Two examples of constraint expressions on an automaton with $\Sigma \overset{\text{def}}{=} \{a, b\}$, the latter being a DCE.

$$\{\{5\}_{2\cdot a}, \emptyset_{3\cdot b}\}^{\underline{*}} \tag{1}$$

$$\{\{5\}_{2\cdot a}, \emptyset_{3\cdot b}\} \veebar \{\emptyset_{3\cdot a}, \{2\}_{2\cdot b}\} \tag{2}$$

3.5 Evaluating Constraints

The function $\texttt{eval}_v(\dot{\mathbb{C}})$ evaluates a DCE $\dot{\mathbb{C}}$ on an aggregated event input v, and is defined as follows:

$$\begin{aligned}
\texttt{eval}_v(\bot) &\overset{\text{def}}{=} \mathit{false} \\
\texttt{eval}_v(\overrightarrow{\mathbb{C}}) &\overset{\text{def}}{=} \forall M_{i\cdot a} \in \overrightarrow{\mathbb{C}}.\ (M = \emptyset \wedge \#^v_a = i) \vee \\
&\qquad\qquad\qquad\quad (M \neq \emptyset \wedge \#^v_a \geq i \wedge \texttt{loops}(\#^v_a - i, M)) \\
\texttt{eval}_v(\overrightarrow{\mathbb{C}}_1 \veebar \overrightarrow{\mathbb{C}}_2 \veebar \ldots \veebar \overrightarrow{\mathbb{C}}_n) &\overset{\text{def}}{=} \texttt{eval}_v(\overrightarrow{\mathbb{C}}_1) \vee \texttt{eval}_v(\overrightarrow{\mathbb{C}}_2) \vee \ldots \vee \texttt{eval}_v(\overrightarrow{\mathbb{C}}_n)
\end{aligned}$$

Assuming that Mods_i returns the member of Mods at position i based on some ordering, `loops` is defined as:

$$\texttt{loops}(N, \mathit{Mods}) \overset{\text{def}}{=} \exists k \in \mathbb{N}^{|\mathit{Mods}|}.\ \sum_{i=1}^{|\mathit{Mods}|} \mathit{Mods}_i \times k_i = N$$

The predicate `loops` holds when there exists a set of coefficients such that, when multiplied by members of *Mods*, will result in the sum of the products being equal to N.

Example 4. $\texttt{loops}(7, \{2, 3\})$ is true, as $2k_1 + 3k_2 = 7$ for $k_1 = 2, k_2 = 1$.

3.6 Traversing a Constraint Automaton

Algorithm 1 details a general approach to traversing a constraint automaton. The algorithm is designed for online use, with the blocking `nextInputEventCount`() function returning aggregated event counts collected by the monitoring system. Naturally, this can readily be adapted for offline inputs.

When traversing an automaton, the algorithm must evaluate the constraint expression for each transition leaving the current state (line 9). Multiple constraint expressions may evaluate to *true* simultaneously, which results in a set of possible next states. Thus, the current automaton state must be modelled as a set of states, with the automaton potentially being in any of those states. This non-determinism may arise even if the original automaton was deterministic. For example, while the automaton illustrated in Figure 1 is deterministic, it has branches that accept two traces with differing order but equal event frequencies. Its constraint automaton (Figure 5) is thus afflicted by ambiguity, as an aggregated input of two events would lead to the automaton potentially being in either A_2 or B_2. A subsequent event would cause the automaton to converge onto B_3. As we detail in Section 5, determinizing the constraint automaton would not help, as the

Algorithm 1 On-line Traversal of a Constraint Automaton $\langle Q, q_0, \Sigma, F, \Gamma \rangle$

```
$Current \leftarrow \{q_0\}$                                   ▷ Start at initial state
loop                                                           ▷ Perpetual loop
    if ($Current \cap F \neq \emptyset$) then                  ▷ Check if potentially in a final state
        reportError($Current$)                                 ▷ Report current state set
    end if
    $v \leftarrow$ nextInputEventCount()                       ▷ Get next map of aggregated events
    $next_Q \leftarrow \emptyset$                              ▷ Reset next state set
    for all $c \in Current$ do                                 ▷ Determine next states for each current state
        $next_Q \leftarrow next_Q \cup \left\{c' \mid (c, \dot{\mathbb{C}}, c') \in \Gamma, \texttt{eval}_v(\dot{\mathbb{C}})\right\}$
    end for
    $Current \leftarrow next_Q$                                ▷ Update with computed next states
end loop
```

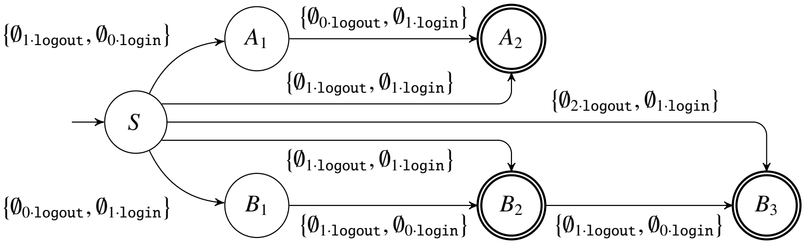

$\{\emptyset_{1\cdot\texttt{logout}}, \emptyset_{0\cdot\texttt{login}}\}$
A_1
$\{\emptyset_{0\cdot\texttt{logout}}, \emptyset_{1\cdot\texttt{login}}\}$
A_2
$\{\emptyset_{1\cdot\texttt{logout}}, \emptyset_{1\cdot\texttt{login}}\}$
$\{\emptyset_{2\cdot\texttt{logout}}, \emptyset_{1\cdot\texttt{login}}\}$
S
$\{\emptyset_{1\cdot\texttt{logout}}, \emptyset_{1\cdot\texttt{login}}\}$
$\{\emptyset_{0\cdot\texttt{logout}}, \emptyset_{1\cdot\texttt{login}}\}$
B_1
B_2
B_3
$\{\emptyset_{1\cdot\texttt{logout}}, \emptyset_{0\cdot\texttt{login}}\}$
$\{\emptyset_{1\cdot\texttt{logout}}, \emptyset_{0\cdot\texttt{login}}\}$

Fig. 5. The constraint automaton derived from the automaton in Figure 1

resulting deterministic automaton would still require a transition function that evaluates the same set of transition constraints.

The size of *Current* may grow as well as shrink, the latter occurring when parallel traversals converge onto a state, or when members of the set do not lead to valid next states under the observed input. As presented, the algorithm never halts, instead reporting an error whenever *Current* contains *some* final state. This policy over-approximates error states, which may lead to false alarms. To reduce them, one may consider using other policies, such as reporting errors only when *Current* is composed entirely of final states. Alternatively, one may augment the automaton with probabilistic information, terminating based on the likelihood that the system is in an actual error state.

4 Constructing a Constraint Automaton from a Property FSA

The following defines the process of deriving constraint automata from finite state properties. The transformation is performed in two phases. In the first phase, *regular expressions* are constructed for every pair of states in the property. In the second phase, each regular expression is subsequently transformed into a constraint on frequencies.

4.1 Translating Regular Expressions into Constraint Expressions

The process of translating a regular expression into a constraint expression involves two steps. The first step transforms the regular expression into an *initial constraint* *expression*, and is performed by applying the *regex-to-constraint expression* operator $\xrightarrow{\Sigma}$, defined as follows.

Definition 7 (Regex-To-CE). *Given a regular expression $\mathcal{R}$, one can derive a constraint expression $\hat{\mathbb{C}}$, whose vectors are well-formed with respect to an alphabet Σ. This is denoted by $\mathcal{R} \xrightarrow{\Sigma} \hat{\mathbb{C}} \overset{def}{=} \hat{\mathbb{C}} = [\![\mathcal{R}]\!]_{\Sigma}$, where $[\![]\!]_{\Sigma}$ is defined as:*

$$
\begin{aligned}
[\![\mathcal{R}_1\mathcal{R}_2]\!]_{\Sigma} &\rightarrow [\![\mathcal{R}_1]\!]_{\Sigma} \mathbin{\underline{:}} [\![\mathcal{R}_2]\!]_{\Sigma} & \qquad [\![\mathcal{R}_1 \mid \mathcal{R}_2]\!]_{\Sigma} &\rightarrow [\![\mathcal{R}_1]\!]_{\Sigma} \mathbin{\underline{\vee}} [\![\mathcal{R}_2]\!]_{\Sigma} \\
[\![\mathcal{R}^n]\!]_{\Sigma} &\rightarrow [\![\mathcal{R}]\!]^{\underline{n}}_{\Sigma} & [\![\mathcal{R}^*]\!]_{\Sigma} &\rightarrow [\![\mathcal{R}]\!]^{\underline{*}}_{\Sigma} \\
[\![a^n]\!]_{\Sigma} &\rightarrow \{\emptyset_{n\cdot a}\} \cup \bigcup_{e\in\Sigma\setminus\{a\}} \emptyset_{0\cdot e}
\end{aligned}
$$

The transformation replaces operators from the regex domain into that of constraint expressions, and transforms alphabetic symbols into well-formed constraint vectors.

Example 5. $a^2b \xrightarrow{\{a,b\}} (\{\emptyset_{2\cdot a}\} \cup \{\emptyset_{0\cdot b}\}) \mathbin{\underline{:}} (\{\emptyset_{1\cdot b}\} \cup \{\emptyset_{0\cdot a}\}) \equiv \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{:}} \{\emptyset_{0\cdot a}, \emptyset_{1\cdot b}\}$

4.2 From Constraint Expressions to DCEs

As can be seen in Example 5, the constraint expression produced by $\xrightarrow{\Sigma}$ will not necessarily be a DCE (in this case, because it contains a concatenation operator), yet the constraint evaluation function described in Section 3.5 is only defined for DCEs. The remainder of this section defines the $\twoheadrightarrow$ operator, which must be repeatedly applied to a constraint expression until a DCE is obtained. Before defining this operator, we first define the *mod-union* operator $\overset{m}{\cup}$ for two sets $M, N \in 2^{\mathbb{N}}$, as follows:

$$M \overset{m}{\cup} N \overset{\text{def}}{=} (M \cup N) \setminus \{m \mid m, n \in (M \cup N), m \bmod n = 0, m \neq n\}$$

When joining two sets under mod-union, values which are divisible by others in the resultant set are removed. This decreases the number of unnecessary computations that must be performed by `loops`, as multiples of values in a modulo set would produce the same verdict.

Example 6. The following are two applications of the mod-union operator, the second of which leads to a reduction in the resultant set.

$$\{7\} \overset{m}{\cup} \{3\} \overset{\text{def}}{=} \{7,3\} \setminus \{\} \equiv \{7,3\} \tag{1}$$

$$\{3\} \overset{m}{\cup} \{6\} \overset{\text{def}}{=} \{3,6\} \setminus \{6\} \equiv \{3\} \tag{2}$$

Definition 8 (Distribution of Concatenation over Disjunction). *We define $\twoheadrightarrow$ such that concatenation* distributes *over disjunctions of expressions:*

$$\hat{\mathbb{C}} \mathbin{\underline{:}} \left(\hat{\mathbb{C}}_1 \mathbin{\underline{\vee}} \hat{\mathbb{C}}_2 \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \hat{\mathbb{C}}_n\right) \twoheadrightarrow \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_1\right) \mathbin{\underline{\vee}} \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_2\right) \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_n\right)$$

Definition 9 (Concatenating Constraints). *Two constraint vectors can be concatenated by adding the corresponding basic constraints' offsets and modulo sets:*

$$\overrightarrow{\mathbb{C}}_1 \mathbin{\underline{:}} \overrightarrow{\mathbb{C}}_2 \twoheadrightarrow \left\{(m_1 \overset{m}{\cup} m_2)_{(n_1+n_2)\cdot a} \mid a \in \Sigma, {m_1}_{n_1\cdot a} \in \overrightarrow{\mathbb{C}}_1, {m_2}_{n_2\cdot a} \in \overrightarrow{\mathbb{C}}_2\right\}$$

Definition 10 (Bounded Repetition). *The reduction of expressions repeated for a fixed number of times is defined for a single constraint vector and a disjunction of constraint expressions, as follows:*

$$\overrightarrow{\mathbb{C}}^{\underline{k}} \twoheadrightarrow \left\{ m_{(n\times k)\cdot a} \mid m_{n\cdot a} \in \overrightarrow{\mathbb{C}} \right\}$$

$$\left(\hat{\mathbb{C}}_1 \mathbin{\underline{\vee}} \hat{\mathbb{C}}_2 \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \hat{\mathbb{C}}_n\right)^{\underline{k}} \twoheadrightarrow \underline{\bigvee_{i_1+i_2+\ldots+i_n=k}} \hat{\mathbb{C}}_1^{\underline{i_1}} \mathbin{\underline{\cdot}} \hat{\mathbb{C}}_2^{\underline{i_2}} \mathbin{\underline{\cdot}} \ldots \mathbin{\underline{\cdot}} \hat{\mathbb{C}}_n^{\underline{i_n}}$$

Definition 11 (Unbounded Repetition). *The reduction of expressions within unbounded repetition is defined for a single constraint vector and a disjunction of constraint expressions, as follows:*

$$\overrightarrow{\mathbb{C}}^{\underline{*}} \twoheadrightarrow \left\{ (m \overset{\mathtt{m}}{\cup} \{n\})_{0\cdot a} \mid m_{n\cdot a} \in \overrightarrow{\mathbb{C}} \right\}$$

$$\left(\hat{\mathbb{C}}_1 \mathbin{\underline{\vee}} \hat{\mathbb{C}}_2 \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \hat{\mathbb{C}}_n\right)^{\underline{*}} \twoheadrightarrow \left(\hat{\mathbb{C}}_1^{\underline{*}} \mathbin{\underline{\cdot}} \hat{\mathbb{C}}_2^{\underline{*}} \mathbin{\underline{\cdot}} \ldots \mathbin{\underline{\cdot}} \hat{\mathbb{C}}_n^{\underline{*}}\right)^{\underline{*}}$$

4.3 Building the Constraint Automaton

Based on the previous definitions, we can now define a construction for transforming a finite-state automaton into a constraint automaton.

Definition 12 (Regular Expression to Constraint Expression). *Given a regular expression $\mathcal{R}$, $C_\Sigma(\mathcal{R})$ will return a DCE $\dot{\mathbb{C}}$ whose vectors are well-formed with respect to Σ, such that $\mathcal{R} \xrightarrow{\Sigma} \hat{\mathbb{C}} \twoheadrightarrow^* \dot{\mathbb{C}}$.*

Definition 13 (FSA to CA). *Given a finite-state automaton $\mathcal{D} \overset{def}{=} \langle Q, q_0, \Sigma, F, \Gamma\rangle$ with Q states, initial state $q_0 \in Q$, alphabet Σ, final states $F \subseteq Q$, and $\Gamma \subseteq Q \times \Sigma \times Q$, one can construct a constraint automaton $\mathcal{CA} \overset{def}{=} \langle Q, q_0, \Sigma, F, \Gamma'\rangle$, where*

$$\Gamma' \overset{def}{=} \left\{ (q, \dot{\mathbb{C}}, q') \mid q, q' \in Q, \mathcal{R} = \mathtt{regex}_{\mathcal{D}}(q, q'), \mathcal{R} \neq \bot, \dot{\mathbb{C}} = C_\Sigma(\mathcal{R}) \right\}$$

The construction considers each pair of states, deriving the regular expressions and converting them into constraint expressions. Each state in $\mathcal{CA}$ will thus have a transition to every other state with the corresponding constraint expression, provided that a path between those states exists in $\mathcal{D}$.

4.4 Examples

Example 7. The following example shows the derivation of a constraint from state S_0 to S_2 in Figure 7, which involves a repeated set of identical transitions, equivalent to the bounded iteration of a group of regular expressions related via concatenation. As with multinomial expansion, raising a DCE with m terms to a power n will result in a constraint expression of $\binom{n+m-1}{n}$ terms. In this example, the terms are reduced further, yet in general, bounded iteration will produce long DCEs.

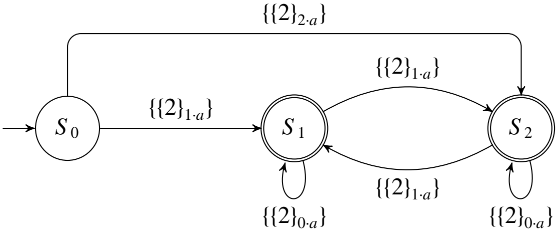


Fig. 6. Unambiguous constraint automaton for the flip-flop defined in Figure 4

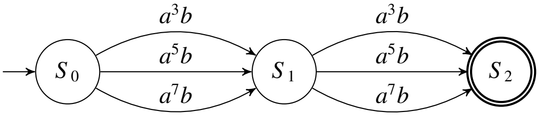


Fig. 7. Automaton for $(a^3b \mid a^5b \mid a^7b)^2$

$$(a^3b \mid a^5b \mid a^7b)^2$$

$$\xrightarrow{\Sigma} (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\})^{\underline{2}}$$

Regex-to-CE, Bounded Repetition, Catenation (7, 10, 9)

$$\begin{aligned}
\twoheadrightarrow\ & \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}}
\end{aligned}$$

Bounded Repetition (10)

$$\begin{aligned}
\twoheadrightarrow\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}) \mathbin{\underline{\vee}} \\
& (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}) \mathbin{\underline{\vee}} \\
& (\{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\})
\end{aligned}$$

Bounded Repetition, Power$^{\underline{0}}$ elimination (10)

$$\begin{aligned}
\twoheadrightarrow\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& \{\emptyset_{8\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{12\cdot a}, \emptyset_{2\cdot b}\}
\end{aligned}$$

Catenation (9)

$$\begin{aligned}
=\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& \{\emptyset_{8\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{12\cdot a}, \emptyset_{2\cdot b}\}
\end{aligned}$$

Removal of duplicates

Example 8. The following example shows the derivation of a constraint from state S_0 to S_2 in Figure 8a, involving a loop sandwiched between two compulsory single-element transitions, thus demonstrating the use of modulo sets.

$$a(a^5b^3)^*a$$

$$\xrightarrow{\Sigma} \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{3\cdot b}\}^{\underline{*}} \mathbin{\underline{\cdot}} \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\}$$

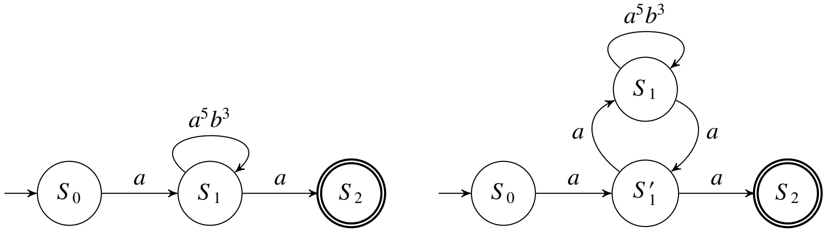
Regex-to-CE (7)

$$\twoheadrightarrow \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{\cdot}} \{\{5\}_{0\cdot a}, \{3\}_{0\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\}$$

Unbounded Repetition (11)

$$\twoheadrightarrow \{\{5\}_{2\cdot a}, \{3\}_{0\cdot b}\}$$

Catenation (9)



(a) Automaton for $a(a^5b^3)^*a$ (b) Automaton for $a(a(a^5b^3)^*a)^*a$

Fig. 8. Example automata showing loops and nested repetition

Example 9. The final example shows the derivation of a constraint from state S_0 to S_2 in Figure 8b, which showcases a nested repetition, demonstrating the effect of unbounded iteration on non-empty modulo sets.

$$
\begin{aligned}
& a(a(a^5b^3)^*a)^*a \\
\xrightarrow{\Sigma}\ & \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \underline{\cdot} (\{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \underline{\cdot} \{\emptyset_{5\cdot a}, \emptyset_{3\cdot b}\}^{\underline{*}})^{\underline{*}} && \text{Regex-to-CE, Catenation (7, 9)} \\
\twoheadrightarrow\ & \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \underline{\cdot} (\{\{5\}_{2\cdot a}, \{3\}_{0\cdot b}\})^{\underline{*}} && \text{Result of Example 8} \\
\twoheadrightarrow\ & \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \underline{\cdot} \{\{2,5\}_{0\cdot a}, \{3\}_{0\cdot b}\} && \text{Unbounded Repetition (11)} \\
\twoheadrightarrow\ & \{\{2,5\}_{2\cdot a}, \{3\}_{0\cdot b}\} && \text{Catenation (9)}
\end{aligned}
$$

5 Computational Complexity

The purpose of constraint automata is to determine the precise set of next states for unordered input traces *efficiently*. Thus, it is important to analyze the computational cost of using the involved structures.

Consider the conversion of an input automaton $\mathcal{D}$, with states Q and an alphabet Σ, into a constraint automaton $\mathcal{CA}$. The size of the resultant constraint automaton is influenced by three factors, namely (i) the *connectivity* of $\mathcal{D}$, with a fully connected automaton leading to an out-degree of $|Q|$ for each state in $\mathcal{CA}$, thus requiring a maximum of $|Q|$ constraint expressions to be evaluated with each step in the constraint automaton, (ii) the number of *choice operators* in the regular expressions derived from the automaton, which affects the number of constraint vectors in the derived constraint expressions, and (iii) the number of *cycles* in the regular expressions, which will cause basic constraints' modulo-set sizes to grow.

A sparsely-connected input automaton would tend to have fewer outgoing transitions per state (as fewer states would be reachable from other states), whereas a densely-connected automaton containing many loops of differing length would increase the size of constraints' modulo sets. As the constraint vectors in the automaton must be well-formed, they will each contain $|\Sigma|$ basic constraints.

Furthermore, as the current state is a set of possible states, the set of next states would have to be computed whilst taking each current state into consideration. The size of the current state set can be at most $|Q|$, which would only occur when the automaton is potentially in any state. Hence, the number of operations performed when computing the next state, assuming the worst-case scenario of a fully-connected aggregate automaton and a full current state set, is:

$$\underbrace{|Q|}_{\substack{\text{current}\\\text{states}}} \times \underbrace{|Q|}_{\substack{\text{constraint}\\\text{expressions}}} \times \underbrace{vecs}_{\substack{\text{vectors}\\\text{/CE}}} \times \underbrace{|\Sigma|}_{\substack{\text{basic}\\\text{constraints}\\\text{/CV}}} \times \Big(\underbrace{mods!}_{\substack{\text{modulo}\\\text{set size}\\\text{/BC}}} + \underbrace{1}_{\substack{\text{offset}\\\text{comparison}}} \Big)$$

As noted earlier, the magnitude of *mods* and *vecs* is dependent on the form of the extracted regular expressions, specifically the number of cycles and choice operators, respectively. Evaluating the modulo set requires calls to `loops`(), which has a worst-case running time that is factorial to the input size. As will be discussed in Section 6, in practice, one can lower the average running time by ordering the evaluation of constraints based on their weakness, and by using more sophisticated techniques for solving `loops`().

Implementation Considerations

A notable feature of constraint automata is that the running time is independent of the number of events encoded in an aggregated event burst. An indirect correlation may exist, as a large number of such events would be more likely to lead to multiple next states, which would consequently enlarge the current state set. The use of a set of current states could be eliminated by making the automaton deterministic, yet this generally results in an exponential growth in states, increasing the number of constraint expressions to be evaluated at every state by an equivalent degree. By maintaining a dynamic set of current states, one can thus reduce the average traversal time, as only the outgoing transitions from potential current states are evaluated.

In this work, we have opted to use regular expressions to produce an initial constraint automaton so as to modularize the transformation stages. It is possible that some performance gains may be obtained by generating constraint expressions directly from the original automaton. More specifically, this may allow the detection of sub-expressions that are shared across constraints, facilitating the caching and reuse of partial results during constraint evaluation. Optimizations could also extract common sub-expressions among constraint expressions emanating from states, rather than evaluating each outgoing transition in isolation. Such considerations could give rise to interesting future work.

A system implementing constraint automata would most likely benefit from changing the representation from the one used into one that is more amenable to comparisons. For example, constraints could be organized in a tree structure based on their offset values, speeding up evaluation by excluding branches which do not meet the minimum.

6 Related Work

Instrumentation is recognized as a source of overhead in runtime verification. This overhead can be reduced by decreasing the amount of instrumentation, or *sampling*, that is performed. Statistical methods can then be employed to infer the most probable sequence of state transitions that occurred during the time in which sampling was suspended. For instance, Stoller et al. [12] consider the scenario where instrumentation is suspended for some period of time, leaving a gap in the sequence of observed events. Their approach focused on reconstructing the missed events via probabilistic models, which estimate the next state based on traces that were observed previously. This approach differs from that explored in this work, as the gaps are devoid of information, not even specifying the number of missed events. In contrast, our work only considers unordered traces, and still requires that the number and type of events be logged.

Bodden et al. [5, 6] provide an implementation of efficient time-triggered automata, which consider gaps of events during monitoring. The approach explored can report false positives (but not false negatives) if continuously monitoring "skip" events that prevent an error state from being reached.

Bartocci et al. [2] extend the concept of probabilistically monitoring gaps in events, and introduce the notion of *criticality levels*, which vary based on the probability that a system reaches an error state. Criticality levels can then be used to determine the degree of instrumentation performed, with the system increasing sampling to determine the precise system state. A similar concept could be integrated into the construction examined in this work. For example, sampling could be increased on detecting that the current state set contains a final state.

Another approach, adopted by Basin et al. [3], is to handle the uncertainty brought about by incomplete traces using a three-valued logic, whereby the property's evaluation function is modified to also reason about indeterminate results.

Bauer et al. [4] present a multi-valued logic that is able to express not only whether a violation has taken place, but also whether a violation would occur if the trace terminated right now. One could easily combine such acceptance conditions with our approach.

The choice of algorithm when generating regular expressions from a finite-state automaton will affect the size and complexity of the resultant expressions [10]. Bounded iteration with choice produces constraint expressions with multiple constraint vectors. In broad terms, unbounded iteration will cause the offset value to be added to the modulo set. Subsequent nesting of an iterated expression within unbounded iterations will have no further effect on the constraint expression's size. Ideally, the algorithm employed would thus minimize the number of choice operators in the output expressions. Nevertheless, the upper-bound on running time remains independent of the number of missed events. In addition, by isolating the inefficient component of the constraints into modulo sets, one may choose to apply existing results and libraries addressing the *Satisfiability Modulo Theories* problem [1] to speed up the computation.

7 Conclusion

We have presented a trace model that allows for the monitoring of distributed systems by compressing partial event streams before they are sent to the monitoring server. We described an algorithm for constructing ahead-of-time a monitor that can deal with compressed event streams in such a way that it provably recognizes property violations without false negatives. We have further shown that the resulting automaton is as precise as possible, and has a complexity low enough to promise performance gains in practice.

References

1. Barrett, C., Stump, A., Tinelli, C.: The satisfiability modulo theories library (smt-lib) (April 2013), `http://smtlib.org/`
2. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S., Stoller, S., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 168–182. Springer, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-35632-2_18`
3. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring compliance policies over incomplete and disagreeing logs. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 151–167. Springer, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-35632-2_17`
4. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007), `http://dx.doi.org/10.1007/978-3-540-77395-5_11`
5. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 22–37. Springer, Heidelberg (2007), `http://www.bodden.de/pubs/bhl+07collaborative.pdf`
6. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. Oxford Journal of Logics and Computation (November 2008), `http://www.bodden.de/pubs/bhl+08collaborative.pdf`
7. Brzozowski, J.A.: Derivatives of regular expressions, vol. 11, pp. 481–494. ACM, New York (1964), `http://doi.acm.org/10.1145/321239.321249`
8. Chen, F., Roşu, G.: Mop: An efficient and generic runtime verification framework. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA 2007, pp. 569–588. ACM, New York (2007), `http://doi.acm.org/10.1145/1297027.1297069`
9. Miller, D., Pearson, B.: Security information and event management (SIEM) implementation. McGraw-Hill (2011)
10. Neumann, C.: Converting deterministic finite automata to regular expressions (March 2005), `http://neumannhaus.com/christoph/papers/2005-03-16.DFA_to_RegEx.pdf`
11. Steffens, S.: P3 consulting, personal communication, `http://www.p3-consulting.de/`
12. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-29860-8_15`

A Proofs

Theorem 1 (Convergence). *Given an arbitrary finite state automaton $\mathcal{D}$ with Q states, the transformation will always converge onto a constraint automaton whose constraint expressions are all DCEs.*

Proof. By definition, the function `regex` returns a regular expression of bounded size. As Q is finite, and `regex` is computed pairwise for each state, the constraint automaton will contain at most $|Q|^2$ transitions. The task is thus to show that the conversion of regular expressions into constraints always converges onto a DCE.

A regular expression $\mathcal{R}$ with N atoms of the form a^k, P power operators, and C concatenation operators, will produce an initial constraint expression $\hat{\mathbb{C}}$ under $\xrightarrow{\Sigma}$ with an equal number of P and C operators in the constraint expression domain, and each of the N atoms will be replaced with a constraint vector. The number of disjunctions in the expression is not directly relevant for the purposes of convergence.

The transformation $\twoheadrightarrow$ is defined for all constraint expression operators, only stopping once a constraint expression is a DCE. In addition, $\twoheadrightarrow$ will itself produce a constraint expression, thus showing that the system will always progress while there are operators left to be reduced. The next step is thus to demonstrate that repeated applications of $\twoheadrightarrow$ will reduce P and C to zero. This is done by case analysis on each constituent definition of the operator, as follows:

- **Concatenation** of two vectors will reduce C by 1.
- **Distributing $\underline{\cdot}$ over** $\underline{\vee}$ has the immediate effect of increasing C by the number of disjuncts, yet the composite expressions will subsequently undergo concatenation, giving an overall reduction in C of 1.
- **Bounded Repetition** on constraint vectors will reduce P. Bounded repetition on a disjunctions of constraint expressions will itself produce a disjunction of constraint expressions. The index is transferred to the individual, simpler disjuncts. These disjuncts would either consist of constraint vectors, in which case P would be reduced, or further disjunctions. Yet in the latter case, the nesting depth must be finite (due to the expression being finite).
- **Unbounded Repetition** of a constraint vector leads to an immediate reduction of 1 from P. For a disjunction of constraint vector, the transformation will result in each disjunct being raised to an unbounded power, and hence an initial increase in P and C by the number of disjuncts. Yet if the sub-terms are constraint vectors, then P and C will be reduced, as per the previous definitions. If, alternatively, the terms are themselves disjunctions of expressions, then the terms would be further expanded until concatenations of basic vectors are reached, at which point P and C will be reduced.

Theorem 2 (Equal Aggregate Event Vectors). *Given that Perms*(s) *returns the set of permutations of a trace s, if* $s \in \Sigma^*$, *then* $\forall p \in \mathit{Perms}(s).\ {\downarrow}\#^p_\Sigma = {\downarrow}\#^s_\Sigma$

Proof. As the alphabet Σ is fixed for all the permutations of s, the function will always return a set containing Σ maps, with one for each element of Σ. Since the permutation operation only affects the order of symbols within a trace, the symbol counts remain unaffected.

Theorem 3 (Equivalent Evaluations of Regex and Constraint Expressions). *The constraint automaton cannot produce false negatives, and is also maximally precise given the reduced information it receives, i.e., won't produce more false warnings than a solution based on the explicit automaton traversal using all string permutations. It holds that* $\forall s \in \Sigma^*.\ \left[\mathtt{eval}_{\downarrow\#^s_\Sigma}(C_\Sigma(\mathcal{R})) \leftrightarrow (\exists p \in \mathit{Perms}(s).\ \mathtt{match}(p,\mathcal{R}))\right]$

Each direction of the bi-implication is proven separately, and is presented as two proofs. Prior to the proofs, we state the following lemmas:

Lemma 1 (Permutations of a regular expression). *Given that* $Perms(\mathcal{R})$ *returns all permutations of the sub-expressions of a regular expression* $\mathcal{R}$ *such that any sub-expressions* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *can be reordered using* $\mathcal{R}_1\mathcal{R}_2 \equiv \mathcal{R}_2\mathcal{R}_1$ *and* $\mathcal{R}_1|\mathcal{R}_2 \equiv \mathcal{R}_2|\mathcal{R}_1$*, it holds that* $\exists p \in Perms(s).\ \mathtt{match}(p, \mathcal{R}) \leftrightarrow \exists p \in Perms(s).\ \exists r \in Perms(R).\ \mathtt{match}(p, r)$

Lemma 2 (Regex decomposition). *Decomposed strings match sub-expressions, that is,* $\forall s \in \Sigma^*.\ \mathtt{match}(s, \mathcal{R}_1\mathcal{R}_2) \rightarrow \exists p_1 p_2 = s.\ \mathtt{match}(p_1, \mathcal{R}_1) \wedge \mathtt{match}(p_2, \mathcal{R}_2)$*, and* $\forall s \in \Sigma^*.\ \mathtt{match}(s, \mathcal{R}_1|\mathcal{R}_2) \rightarrow \mathtt{match}(s, \mathcal{R}_1) \vee \mathtt{match}(s, \mathcal{R}_2)$

Proof (Evaluation → Match). Every basic constraint, constraint vector and DCE can be reconstructed into a regular expression, as follows:

$$\begin{aligned}
\mathtt{Rec}(\{m_1, m_2, \ldots, m_k\}_{n\cdot a}) &= a^n(a^{m_1^*}a^{m_2^*}\ldots a^{m_k^*})^* \\
\mathtt{Rec}(\{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k\}) &= \mathtt{Rec}(\mathbb{C}_1)\ \mathtt{Rec}(\mathbb{C}_2)\ \ldots\ \mathtt{Rec}(\mathbb{C}_k) \\
\mathtt{Rec}(\overrightarrow{\mathbb{C}}_1 \veebar \overrightarrow{\mathbb{C}}_2 \veebar \ldots \veebar \overrightarrow{\mathbb{C}}_k) &= \mathtt{Rec}(\overrightarrow{\mathbb{C}}_1)\,|\,\mathtt{Rec}(\overrightarrow{\mathbb{C}}_2)\,|\ \ldots\ |\,\mathtt{Rec}(\overrightarrow{\mathbb{C}}_k)
\end{aligned}$$

If $\dot{\mathbb{C}}$ is a DCE which holds for $\downarrow\#^s_\Sigma$, then one can rebuild a string $a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}$, for $a, b \ldots z \in \Sigma$, that will contain a symbol for each element in Σ with a frequency equal to its value in the aggregated input. Given the definition of Rec(), it follows that $\exists p \in Perms(a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}).\ \mathtt{match}(p, \mathtt{Rec}(\dot{\mathbb{C}}))$. By using Lemma 1, the fact that $Perms(a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}) = Perms(s)$, and by substituting $\dot{\mathbb{C}}$ with $C_\Sigma(\mathcal{R})$, the statement to be proven can be reformulated as:

$$\exists p \in Perms(s).\ \mathtt{match}(p, \mathtt{Rec}(C_\Sigma(\mathcal{R}))) \rightarrow \exists p \in Perms(s).\ \exists r \in Perms(\mathcal{R}).\ \mathtt{match}(p, r)$$

The relation will be demonstrated through case analysis on the different forms of regular expressions. Given that Lemma 2 holds, it is sufficient to show that the operators hold on the basic elements of regular expressions and then induce on the length of the regular expression.

$\mathcal{R}$	$C_\Sigma(\mathcal{R})$	$\mathtt{Rec}(C_\Sigma(\mathcal{R}))$	
$a^n b^m$	$\{\emptyset_{n\cdot a}, \emptyset_{m\cdot b}\}$	$a^n b^m$	$\in Perms(R)$
$a^n\|b^m$	$\{\emptyset_{n\cdot a}, \emptyset_{0\cdot b}\} \veebar \{\emptyset_{0\cdot a}, \emptyset_{m\cdot b}\}$	$a^n\|b^m$	$\in Perms(R)$
a^{n^*}	$\{\{n\}_{0\cdot a}\}$	$a^0 a^{n^*}$	$\in Perms(R)$
$(a^n\|b^m)^*$	$\{\{n\}_{0\cdot a}, \{m\}_{0\cdot b}\}$	$a^{n^*} b^{m^*}$	Matches $\mathcal{R}$ for permutation of s
$(a^{n^*}\|b^{m^*})^k$	$\{\{n\}_{0\cdot a}, \{m\}_{0\cdot b}\}$	$a^{n^*} b^{m^*}$	Matches $\mathcal{R}$ for permutation of s

Proof (Match → Evaluation). Recall that $C_\Sigma()$ is evaluated in two phases, first transforming the input into an initial constraint expression (Definition 7), and then iteratively reducing the expression into a DCE. The initial transformation simply changes the operators into those of the constraint-expression domain, whilst replacing alphabetic symbols into basic constraints. It is evident that $\forall a \in \Sigma.\ s \in \Sigma^*.\ \forall n \in \mathbb{N}.\ \mathtt{match}(s, a^n) \leftrightarrow \mathtt{eval}_{\downarrow\#^s_\Sigma}(\emptyset_{n\cdot a})$, since the LHS will only be true for sequences of as of length n, which also holds for the RHS. The task is thus to demonstrate that expressions obtained by reductions using $\twoheadrightarrow$ will also evaluate to true. This is done via case analysis on the operators, as follows:

- **Concatenation** of two constraint vectors is performed by summing the offsets and performing a union of modulo sets for every basic constraint. The result will thus consume at least the same amount of input events as the constituent vectors would in isolation.
- **Distributing** $\underline{\cdot}$ **over** $\underline{\vee}$ preserves truth due to the design of the `eval()` procedure, as each disjunct will also include the concatenated outer disjunct.
- **Bounded Repetition** of a constraint vector $\overrightarrow{\mathbb{C}}$ raised to a power k is equivalent to concatenating a sequence of k consecutive $\overrightarrow{\mathbb{C}}$ vectors. The modulo sets thus remains unchanged, whilst the offsets of each basic constraint are multiplied by a factor k. Bounded repetition of a disjunction of constraint is performed by expanding the disjunction as a *multinomial*, discarding the coefficients. This produces a disjunction of sequences with every combination of the terms, essentially unravelling the loop.
- **Unbounded Repetition** resolves to a concatenation of zero or more constraint expressions. If a single constraint vector is being raised to a power, then its offset is added to its modulo set, thus matching when the original expression is taken one or more times. In the case of an unbounded repetition of a DCE, each disjunct can be taken an unbounded number of times, forming a sequence of constraint expressions. Equivalent sequences can be formed by changing the disjunctions into concatenations, whilst simultaneously placing each concatenated expression in an unbounded loop.

Synthesising Correct Concurrent Runtime Monitors

(Extended Abstract)

Adrian Francalanza and Aldrin Seychell

Computer Science, ICT, University of Malta
{afra1,asey0001}@um.edu.mt

Abstract. We study the correctness of automated synthesis for concurrent monitors. We adapt sHML, a subset of the Hennessy-Milner logic with recursion, to specify safety properties of Erlang programs, and define an automated translation from sHML formulas to Erlang monitors so as to detect formula violations at runtime. We then formalise monitor correctness for our concurrent setting and describe a technique that allows us to prove monitor correctness in stages; this technique is used to prove the correctness of our automated monitor synthesis.

Keywords: runtime verification, monitor synthesis, concurrency, actors, Erlang.

1 Introduction

Runtime Verification (RV) [3], is a lightweight verification technique for determining whether the *current system run* observes a correctness property. Two requirements are crucial for the adoption of this technique. First, runtime *monitor overheads* need to be kept to a minimum so as not to degrade system performance. Second, instrumented monitors need to form part of the trusted computing base of a system by adhering to an agreed notion of *monitor correctness*; amongst other things, this normally includes a guarantee that runtime checking corresponds (in some sense) to the property being checked for. Monitor overheads and correctness are occasionally conflicting concerns. For instance, in order to lower monitoring overheads, engineers increasingly use *concurrent monitors* [11,22,28] so as to exploit better the underlying parallel and distributed architectures pervasive to today's computers. However concurrent monitors are also more susceptible to elusive errors such as non-deterministic monitor behaviour, deadlocks or livelocks which may, in turn, affect their correctness.

Ensuring monitor correctness is non-trivial. One prominent obstacle is the fact that system properties are typically specified using one formalism, *e.g.*, a high-level logic, whereas the respective monitors checking these properties are described using another formalism, *e.g.*, a programming language—this makes it hard to ascertain the semantic correspondence between the two descriptions. *Automated monitor synthesis* can mitigate this problem by standardising the translation from the property logic to the monitor formalism. It also gives more scope for a formal treatment of monitor correctness.

In this work, we investigate the correctness of *synthesised* monitors in a *concurrent setting*, whereby (*i*) the system executes concurrently with the synthesised monitor

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 112–129, 2013.

(*ii*) the system and the monitor themselves consist of concurrent sub-systems and sub-monitors. Previous work on correct monitor synthesis[17,27,4] abstracts away from the internal working of a system, representing it as a string of events/states (execution trace). It also focusses on a logic that is readily amenable to runtime analysis, namely Linear Temporal Logic (LTL)[8]. Moreover, it expresses synthesis in terms of *abstract* or *single-threaded* monitors—using pseudocode or automata—executing *wrt.* such trace. By contrast, we strive towards a more intensional formal definition of online correctness for synthesised concurrent monitors whereby, for arbitrary property φ, the synthesised monitor M_φ running concurrently *wrt.* some system S (denoted as $S \parallel M_\varphi$) observes the following condition:

$$S \text{ violates } \varphi \text{ in the current execution} \qquad \textit{iff} \qquad S \parallel M_\varphi \text{ detects the violation} \qquad (1)$$

The setting described in (1) brings to the fore a number of additional issues:

(i) Apart from the formal semantics of the source logic (used to specify the property φ), we also require a formal semantics for the target languages of *both* the system and the monitor executing in parallel, *i.e.*, $S \parallel M_\varphi$. In most cases, the latter may not always be available.

(ii) A property logic semantics is often defined over *systems* rather than *traces*, which may not lend itself well to the formulation of correctness runtime analysis outlined in condition (1) above. In the case of concurrent systems, this aspect is accentuated by the fact that systems may behave non-deterministically and typically have multiple execution paths as a result of different thread interleavings scheduled at runtime.

(iii) Concurrent monitors may also have multiple execution paths. Condition (1) thus requires stronger guarantees than those for single-threaded monitors so as to ensure that *all* these paths correspond to an appropriate runtime check of system property being monitored. Stated otherwise, correct concurrent monitors must *always* detect violations, irrespective of their runtime interleaving.

(iv) Online monitor correctness needs to ensure that monitor execution cannot be interfered by the system, and viceversa. Whereas adequate monitor instrumentation typically prevents direct interferences, condition (1) must consider indirect interferences such as system divergences [25,18], *i.e.*, infinite internal looping making the system unresponsive, which may prevent the monitors from progressing.

(v) Ensuring correctness along the lines of condition (1) can be quite onerous because *every* execution path of the monitor running concurrently with the monitored system, $S \parallel M_\varphi$, needs to be analysed so as to ensure consistent detections along every thread interleaving. Consequently, one needs to devise scalable techniques facilitating monitor correctness analysis.

We conduct our study in terms of actor-based [19] concurrent monitors written in Erlang [7,2], an industry strength language for constructing fault-tolerant systems; we also restrict ourselves to the monitoring of systems written in the same language. We limit ourselves to *reactive properties* describing *system interactions with the environment* and focus on the synthesis of *asynchronous* monitors, performing runtime analysis through the Erlang Virtual Machine (EVM)'s tracing mechanism. Despite the typical drawbacks

Actor Systems, Expressions, Values and Patterns

$$\begin{array}{rcl@{\qquad}rcl}
A,B,C \in \textsc{Actr} & ::= & i[e \triangleleft q]^m \mid A \parallel B \mid (\nu\, i)A & q,r \in \textsc{MBox} & ::= & \epsilon \mid v:q \\
e,d \in \textsc{Exp} & ::= & \multicolumn{4}{l}{v \mid \mathsf{self} \mid e!d \mid \mathsf{rcv}\, g\, \mathsf{end} \mid e(d) \mid \mathsf{spw}\, e \mid \mathsf{case}\, e\, \mathsf{of}\, g\, \mathsf{end} \mid x = e, d \mid \ldots} \\
v,u \in \textsc{Val} & ::= & x \mid i \mid a \mid \mu y.\lambda x.e \mid \{v,\ldots,v\} \mid l \mid \mathsf{exit} \mid \ldots & l,k \in \textsc{Lst} & ::= & \mathsf{nil} \mid v:l \\
p,o \in \textsc{Pat} & ::= & x \mid i \mid a \mid \{p,\ldots,p\} \mid \mathsf{nil} \mid p:x \mid \ldots & g,f \in \textsc{PLst} & ::= & \epsilon \mid p \rightarrow e; g
\end{array}$$

Fig. 1. Erlang Syntax

associated with asynchrony, *e.g.*, late detections, our monitoring setup is in line with the asynchrony advocated by the actor concurrency model, which facilitates scalable coding techniques such as fail-fast design patterns [7]. Asynchronous monitoring has also been used in other RV tools, *e.g.*, [9,12], and has proved to be less intrusive and easier to instrument than synchronous monitoring setups. It is also more feasible for monitoring distributed systems[14,11]. More importantly, though, we expect most of the issues investigated to carry over in straightforward fashion to purely synchronous settings.

As an expository logic for describing reactive properties, we consider an adaptation of sHML [1]—a syntactic subset of the more expressive logic μ-calculus, describing safety, *i.e.*, monitorable[21], properties. Our choice for this logic was, in part, motivated by the fact that the full μ-calculus had already been adapted to describe Erlang program behaviour in [16], albeit for model-checking purposes. Given the usual drawbacks associated with full-blown model checking, our work contributes towards an investigation of lightweight verification techniques for μ-calculus properties of Erlang programs. More importantly, though, it illustrates well the correctness monitoring issues arising in actual implementations, as discussed earlier for (1).

The rest of the paper is structured as follows. Sec. 2 discusses the formal semantics of our systems and monitor target language. Sec. 3 discusses reformulations to the logic facilitating the formulation of monitor correctness, discussed later in Sec. 4. Sec. 5 describes a synthesis algorithm for the logic and a tool built using the algorithm. Subsequently, Sec. 6 proves the correctness of this monitor synthesis. Sec. 7 concludes.

2 The Language

We require a formal semantics for both our monitor-synthesis target language, and the systems we intend to monitor. We partially address this problem by expressing both in terms of the same language, *i.e.*, Erlang, thus only requiring one semantics. However, we still need to describe the Erlang tracing semantics we intend to use for our asynchronous monitoring. Although Erlang semantic formalisations exist, *e.g.*, [30,16,6], none describe this tracing mechanism. We therefore define a calculus—following [30,16]—for modelling the *tracing semantics* of a (Turing-complete) subset of the Erlang language (we leave out distribution, process linking and fault-trapping mechanisms).

Figure 1 outlines the language syntax, assuming disjoint denumerable sets of process/actor identifiers $i, j, h \in \textsc{Pid}$, atoms $a, b \in \textsc{Atom}$, and variables $x, y, z \in \textsc{Var}$.

An executing Erlang program is made up of a *system of actors*, Actr, composed in *parallel*, $A \parallel B$, where some identifiers are local (scoped) to subsystems of actors, and thus not known to the environment, *e.g.*, i in a system $A \parallel (\nu\, i)B$. Individual actors, denoted as $i[e \triangleleft q]^m$, are uniquely identified by an identifier, i, and consist of an expression, e, executing *wrt.* a local mailbox, q (denoted as a list of values). Actor *expressions* typically consist of a sequence of variable binding $x_i = e_i,$ terminated by an expression, e_{final}:

$$x_1 = e_1,\ \ \ldots,\ x_n = e_n,\ e_{\text{final}}$$

An expression e_i in a binding $x_i = e_i, e_{i+1}$ is expected to evaluate to a value, v, which is then bound to x_i in the continuation expression e_{i+1}. When instead e_i generates an exception, exit, it aborts subsequent computations[1] in e_{i+k}. Apart from bindings, expressions may also consist of self references (to the actor's own identifier), self, outputs to other actors, $e_1!e_2$, pattern-matching inputs from the mailbox, $\mathsf{rcv}\, g\, \mathsf{end}$, or pattern-matching for case-branchings, $\mathsf{case}\, e\, \mathsf{of}\, g\, \mathsf{end}$ (where g is a list of expressions guarded by patterns, $p_i \to e_i$), function applications, $e_1(e_2)$, and actor-spawning, $\mathsf{spw}\, e$, amongst others. Values consist of variables, x, process ids, i, recursive functions,[2] $\mu y.\lambda x.e$, tuples $\{v_1, \ldots, v_n\}$ and lists, l, amongst others.

Remark 1. The functions $\text{fv}(A)$ and $\text{fId}(A)$ return the free variables and free process identifiers of A *resp.* and are defined in standard fashion. We write $\lambda x.e$ and d, e for $\mu y.\lambda x.e$ and $y = d, e$ *resp.* when $y \notin \text{fv}(e)$. In $p \to e$, we replace x in p with _ whenever $x \notin \text{fv}(e)$. We write $\mu y.\lambda(x_1, \ldots x_n).e$ for $\mu y.\lambda x_1.\ldots.\lambda x_n.e$. We elide mailboxes, $i[e]$, when empty, $i[e \triangleleft \epsilon]$, or when they do not change in the transition rules that follow.

Specific to our formalisation, we also subject each individual actor, $i[e \triangleleft q]^m$, to a *monitoring-modality*, $m, n \in \{\circ, \bullet, *\}$, where $\circ$, $\bullet$ and $*$ denote *monitored*, *unmonitored* and *tracing* actors *resp.* Modalities play a crucial role in our language semantics, defined as a labelled transition system over systems, $A \xrightarrow{\gamma} B$, where actions $\gamma \in \text{Act}_\tau$, include bound *output* labels, $(\tilde{h})i!v$, and *input* labels, $i?v$ and a distinguished *internal* label, τ. In line with the reactive properties we consider later, our formalisation only traces system interactions with the environment (send and receive messages) from *monitored* actors. Thus, whereas unmonitored, $\bullet$, and tracing, $*$, actors have standard input and output transition rules

$$\textsc{SndU}\ \frac{m \in \{\bullet, *\}}{j[i!v \triangleleft q]^m \xrightarrow{i!v} j[v \triangleleft q]^m} \qquad \textsc{RcvU}\ \frac{m \in \{\bullet, *\}}{i[e \triangleleft q]^m \xrightarrow{i?v} i[e \triangleleft q{:}v]^m}$$

actors with a monitored modality, $\circ$, *i.e.,* actors j and i in rules SndM and RcvM below, produce a residual message reporting the send and receive interactions ($\{\mathsf{sd}, i, v\}$ and $\{\mathsf{rv}, i, v\}$ *resp.*) at the tracer's mailbox *i.e.,* actor h with modality $*$ in the rules below; this models closely the tracing mechanism offered by the Erlang Virtual Machine (EVM) [7]. In our target language, the list of report messages at the tracer's mailbox constitutes the system trace to be used for asynchronous monitoring.

[1] Because of exit exceptions, variable bindings cannot be encoded as function applications.

[2] The preceding μy denotes the binder for function self-reference.

$$\textsc{SndM}\ \dfrac{}{j[i!v \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i!v} j[v \triangleleft q]^{\circ} \parallel h[d \triangleleft r{:}\{\mathsf{sd}, i, v\}]^{*}}$$

$$\textsc{RcvM}\ \dfrac{}{i[e \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i?v} i[e \triangleleft q{:}v]^{\circ} \parallel h[d \triangleleft r{:}\{\mathsf{rv}, i, v\}]^{*}}$$

Our LTS semantics assumes *well-formed* actor systems, whereby every actor identifier is *unique*; it is termed to be a *tracing semantics* because a distinguished *tracer* actor, identified by the monitoring modality $*$, receives messages recording external communication events by *monitored* actors. Formally, we write $A \xrightarrow{\gamma} B$ in lieu of $\langle A, \gamma, B\rangle \in \longrightarrow$, the least ternary relation satisfying the rules in Fig. 2. These rules employ *evaluation contexts*, denoted as C (described below) specifying which subexpressions are active. For instance, an expression is only evaluated when at the top level variable binding, $x = C, e$ or when the expression denoting the destination of an output has evaluated to a value, $v!C$; the other cases are also fairly standard.[3] We denote the application of a context C to an expression e as $C[e]$.

$$C \ ::= \quad [-] \mid C!e \mid v!C \mid C(e) \mid v(C) \mid \mathsf{case}\,C\,\mathsf{of}\,g\,\mathsf{end} \mid x = C, e \mid \ldots$$

Communication in actor systems happens in two stages: an actor receives messages, keeping them in order in its mailbox, and then selectively reads them at a later stage using pattern matching—rules Rd1 and Rd2 describe how mailbox messages are traversed in order to find the first one matching a pattern in the pattern list g, releasing the respective guarded expression e as a result. We choose only to record *external* communication at tracer processes, *i.e.*, between the system and the environment, and do not trace internally communication between actors within the system, irrespective of their modality (see Com); structural equivalence rules, $A \equiv B$, are employed to simplify the presentation of our rules—see rule Str and the corresponding structural rules. In Par, the side-condition enforces the *single-receiver* property, inherent to actor systems; for instance, it prevents a transition with an action $j!v$ when actor j is part of the actor system B. Finally, spawned actors inherit monitorability when launched by a monitored actor, but are launched as unmonitored otherwise (see Spw). The rest of the transition rules are fairly standard; consult [15] for details.

Remark 2. Our tracing semantics sits at higher level of abstraction than that offered by the EVM [7] because trace entries typically contain more information. For instance, the EVM records internal communication between monitored actors, as an output trace entry *immediately followed by* the corresponding input trace entry; we here describe *sanitised* traces whereby consecutive matching trace entries are filtered out.

Example 1 (Non-deterministic behaviour). Our systems exhibit non-deterministic behaviour through either internal or external choices [23,18]. Consider the actor system:

$$A \triangleq (\nu\, j_1, j_2, h)(\, i[\mathsf{rcv}\,x \to \mathsf{obs}!x\,\mathsf{end} \triangleleft \epsilon]^{\circ} \parallel j_1[i!v]^{\circ} \parallel j_2[i!u]^{\circ} \parallel h[e \triangleleft q]^{*}\,)$$

[3] In our formalisation, expressions are not allowed to evaluate under a spawn context, $\mathsf{spw}\,[-]$. This differs from standard Erlang semantics but allows a lightweight description of function application spawning; an adjustment in line with Erlang spawning would be straightforward.

$$\textsc{SndM}\,\frac{}{j[C[i!v] \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i!v} j[C[v] \triangleleft q]^{\circ} \parallel h[d \triangleleft (r{:}\{\mathsf{sd}, i, v\})]^{*}}$$

$$\textsc{RcvM}\,\frac{\mathrm{fv}(v) = \emptyset}{i[e \triangleleft q]^{\circ} \parallel h[d \triangleleft r]^{*} \xrightarrow{i?v} i[e \triangleleft q{:}v]^{\circ} \parallel h[d \triangleleft (r{:}\{\mathsf{rv}, i, v\})]^{*}}$$

$$\textsc{SndU}\,\frac{m \in \{\bullet, *\}}{j[C[i!v] \triangleleft q]^{m} \xrightarrow{i!v} j[C[v] \triangleleft q]^{m}} \qquad \textsc{RcvU}\,\frac{m \in \{\bullet, *\} \qquad \mathrm{fv}(v) = \emptyset}{i[e \triangleleft q]^{m} \xrightarrow{i?v} i[e \triangleleft q{:}v]^{m}}$$

$$\textsc{Scp}\,\frac{A \xrightarrow{\gamma} B}{(\nu\, j)A \xrightarrow{\gamma} (\nu\, j)B}\, j \notin (\mathrm{obj}(\gamma) \cup \mathrm{sbj}(\gamma)) \qquad \textsc{Opn}\,\frac{A \xrightarrow{(\tilde{h})i!v} B}{(\nu\, j)A \xrightarrow{(j,\tilde{h})i!v} B}\, i \neq j,\, j \in \mathrm{sbj}((\tilde{h})i!v)$$

$$\textsc{Com}\,\frac{}{j[C[i!v] \triangleleft q]^{m} \parallel i[e \triangleleft q]^{n} \xrightarrow{\tau} j[C[v] \triangleleft q]^{m} \parallel i[e \triangleleft q{:}v]^{n}}$$

$$\textsc{Par}\,\frac{A \xrightarrow{\gamma} A'}{A \parallel B \xrightarrow{\gamma} A' \parallel B}\, \mathrm{obj}(\gamma) \cap \mathrm{fId}(B) = \emptyset$$

$$\textsc{Rd1}\,\frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)]^{m} \xrightarrow{\tau} i[C[e] \triangleleft q]^{m}}$$

$$\textsc{Rd2}\,\frac{\mathrm{mtch}(g, v) = \bot \qquad i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft q]^{m} \xrightarrow{\tau} i[C[e] \triangleleft r]^{m}}{i[C[\mathsf{rcv}\, g\, \mathsf{end}] \triangleleft (v : q)]^{m} \xrightarrow{\tau} i[C[e] \triangleleft (v : r)]^{m}}$$

$$\textsc{Cs1}\,\frac{\mathrm{mtch}(g, v) = e}{i[C[\mathsf{case}\, v\, \mathsf{of}\, g\, \mathsf{end}]]^{m} \xrightarrow{\tau} i[C[e]]^{m}} \qquad \textsc{Cs2}\,\frac{\mathrm{mtch}(g, v) = \bot}{i[C[\mathsf{case}\, v\, \mathsf{of}\, g\, \mathsf{end}]]^{m} \xrightarrow{\tau} i[C[\mathsf{exit}]]^{m}}$$

$$\textsc{Ass}\,\frac{v \neq \mathsf{exit}}{i[C[x = v, e]]^{m} \xrightarrow{\tau} i[C[e\{{}^{v}\!/\!_{x}\}]]^{m}} \qquad \textsc{Ext}\,\frac{}{i[C[x = \mathsf{exit}, e]]^{m} \xrightarrow{\tau} i[C[\mathsf{exit}]]^{m}}$$

$$\textsc{App}\,\frac{}{i[C[\mu y.\lambda x.e\,(v)]]^{m} \xrightarrow{\tau} i[C[e\{{}^{\mu y.\lambda x.e}\!/\!_{y}\}\{{}^{v}\!/\!_{x}\}]]^{m}} \qquad \textsc{Slf}\,\frac{}{i[C[\mathsf{self}]]^{m} \xrightarrow{\tau} i[C[i]]^{m}}$$

$$\textsc{Spw}\,\frac{(m = \circ = n) \text{ or } (n = \bullet)}{i[C[\mathsf{spw}\, e] \triangleleft q]^{m} \xrightarrow{\tau} (\nu\, j)(i[C[j] \triangleleft q]^{m} \parallel j[e \triangleleft \epsilon]^{n})} \qquad \textsc{Str}\,\frac{A \equiv A' \xrightarrow{\gamma} B' \equiv B}{A \xrightarrow{\gamma} B}$$

$$\textsc{sCom}\,\frac{}{A \parallel B \equiv B \parallel A} \qquad \textsc{sAss}\,\frac{}{(A \parallel B) \parallel C \equiv A \parallel (B \parallel C)} \qquad \textsc{sCtxP}\,\frac{A \equiv B}{A \parallel C \equiv B \parallel C}$$

$$\textsc{sExt}\,\frac{i \notin \mathrm{fId}(A)}{A \parallel (\nu\, i)B \equiv (\nu\, i)(B \parallel A)} \qquad \textsc{sSwp}\,\frac{}{(\nu\, i)(\nu j)A \equiv (\nu j)(\nu\, i)A} \qquad \textsc{sCtxS}\,\frac{A \equiv B}{(\nu i)A \equiv (\nu i)B}$$

Fig. 2. Erlang Semantics for Actor Systems

Actors j_1, j_2 and h are local, thus not visible to the environment. The monitored actor i may receive value v from actor j_1, read it from its mailbox, and then output it to some environment actor obs, while recording this *external* output at h's mailbox (the tracer).

$$A \xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\mathsf{obs}!v} (\nu\, j_1, j_2, h)(\, i[v \triangleleft \epsilon]^{\circ} \parallel j_1[v] \parallel j_2[i!u] \parallel h[e \triangleleft q : \{\mathsf{sd}, \mathsf{obs}, v\}]^{*}\,)$$

But if actor j_2 sends its value to i before j_1, we observe a different external behaviour

$$A \xrightarrow{\tau} \cdot \xrightarrow{\tau} \cdot \xrightarrow{\mathsf{obs}!u} (\nu\, j_1, j_2, h)(\, i[u \triangleleft \epsilon]^{\circ} \parallel j_1[i!v] \parallel j_2[u] \parallel h[e \triangleleft q : \{\mathsf{sd}, \mathsf{obs}, u\}]^{*}\,)$$

i.e., A outputs u instead of v to obs (accordingly monitor h would hold the entry $\{\mathsf{sd}, \mathsf{obs}, u\}$ instead); these behaviours amounts to an *internal choice*.

External choice results when A receives different external inputs: we can derive $A \xrightarrow{i?v_1} B_1$, but also $A \xrightarrow{i?v_2} B_2$. Subsequently, B_1 can only produce the external output $B_1 \xrightarrow{\tau}^{*} \xrightarrow{\mathsf{obs}!v_1}$ whereas from B_2 can only produce $B_2 \xrightarrow{\tau}^{*} \xrightarrow{\mathsf{obs}!v_2}$. Note that, in the first case, h's mailbox is appended by entries $\{\mathsf{rv}, i, v_1\} : \{\mathsf{sd}, \mathsf{obs}, v_1\}$ whereas, in the second case, it is appended by $\{\mathsf{rv}, i, v_2\} : \{\mathsf{sd}, \mathsf{obs}, v_2\}$. ■

3 The Logic

To specify reactive properties of the systems we consider an adaptation of SafeHML[1] (sHML) , a sub-logic of the Hennessy-Milner Logic (HML) with recursion.[4] It assumes a denumerable set of formula variables, $X, Y \in \textsc{LVar}$, and is defined by the grammar:

$$\varphi, \psi \in \text{sHML} \;::=\; \mathsf{ff} \;\mid\; \varphi\wedge\psi \;\mid\; [\alpha]\varphi \;\mid\; X \;\mid\; \mathsf{max}(X, \varphi)$$

The formulas for falsity, ff, conjunction, $\varphi\wedge\psi$, and action necessity, $[\alpha]\varphi$, are inherited from HML[18], whereas variables X and the recursion construct $\mathsf{max}(X, \varphi)$ are used to define *maximal* fixpoints; as expected, $\mathsf{max}(X, \varphi)$ is a binder for the free variables X in φ, inducing standard notions of open and closed formulas. We only depart from the logic of [1] by limiting formulas to *basic actions* $\alpha, \beta \in \textsc{BAct}$, including just input, $i?v$, and *unbound* outputs, $i!v$, so as to keep our technical development manageable.

Remark 3. The handling of bounded output actions, $(\tilde{h})i!v$, is well understood [24] and does not pose problems to monitoring, apart from making action pattern matching cumbersome; it also complicates instrumentation (see Sec. 4 and 5). Silent τ labels can also be monitored using minor adaptations; they however increase substantially the size of the traces recorded, unnecessarily cluttering the tracing semantics of Section 2.

The semantics of our logic is defined for closed formulas, using the operation $\varphi\{\psi/X\}$, which substitutes free occurrences of X in φ with ψ without introducing any variable capture. It is specified as the satisfaction relation of Def. 1 (adapted from [1]). In what follows, we write *weak* transitions $A \Longrightarrow B$ and $A \overset{\alpha}{\Longrightarrow} B$, for $A \xrightarrow{\tau}^{*} B$ and $A \xrightarrow{\tau}^{*} \cdot \xrightarrow{\alpha} \cdot \xrightarrow{\tau}^{*} B$ *resp.* We let $s, t \in (\textsc{BAct})^{*}$ range over *lists of basic actions* and write sequences of weak actions $A \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} B$, where $s = \alpha_1, \ldots, \alpha_n$, as $A \overset{s}{\Longrightarrow} B$ (or as $A \overset{s}{\Longrightarrow}$ when B is unimportant).

[4] HML with recursion has been shown to be equally expressive to the μ-calculus[20].

Definition 1 (Satisfiability). *A relation $\mathcal{R} \in$ Actr × sHML is a satisfaction relation iff:*

$$\begin{array}{rl}
(A, \mathsf{ff}) \in \mathcal{R} & \textit{never} \\
(A, \varphi \wedge \psi) \in \mathcal{R} & \textit{implies } (A, \varphi) \in \mathcal{R} \textit{ and } (A, \psi) \in \mathcal{R} \\
(A, [\alpha]\varphi) \in \mathcal{R} & \textit{implies } (B, \varphi) \in \mathcal{R} \textit{ whenever } A \overset{\alpha}{\Longrightarrow} B \\
(A, \mathsf{max}(X, \varphi)) \in \mathcal{R} & \textit{implies } (A, \varphi\{\mathsf{max}(X, \varphi)/X\}) \in \mathcal{R}
\end{array}$$

Satisfiability, $\models_s$, is the largest *satisfaction relation; we write $A \models_s \varphi$ for $(A, \varphi) \in \models_s$.*[5]

Example 2 (Satisfiability). Consider the safety formula

$$\varphi_{\text{safe}} \triangleq \mathsf{max}(X,\ [\alpha][\alpha][\beta]\mathsf{ff} \wedge [\alpha]X\,) \tag{2}$$

stating that a satisfying actor system cannot perform a sequence of two external actions α followed by the action β (through the subformula $[\alpha][\alpha][\beta]\mathsf{ff}$), and that this needs to hold after every α action (through $[\alpha]X$); effectively the formula states that sequences of α-actions *greater than two* cannot be followed by a β-action.

A system A_1 exhibiting (just) the behaviour $A_1 \overset{\alpha\beta}{\Longrightarrow}$ satisfies φ_{safe}, as would a system A_2 with just the (infinite) behaviour $A_2 \overset{\alpha}{\Longrightarrow} A_2$. System A_3 with a trace $A_3 \overset{\alpha\alpha\beta}{\Longrightarrow}$ *does not* satisfy φ_{safe}. However, if at runtime, A_3 exhibits the alternate behaviour $A_3 \overset{\beta}{\Longrightarrow}$ (through an internal choice) we *would not be able to* detect the fact that $A_3 \not\models_s \varphi_{\text{safe}}$. ■

Since actors may violate a property along one execution but satisfy it along another, the inverse of $\models_s$, *i.e.*, $A \not\models_s \varphi$, is too coarse to be used for a definition of monitor correctness along the lines of (1) discussed earlier. We thus define a *violation relation*, Def. 2, characterising actors violating a property along a specific execution trace.

Definition 2 (Violation). *The violation relation, denoted as $\models_v$, is the* least *relation of the form* (Actr × BAct* × sHML) *satisfying the following rules:*[6]

$$\begin{array}{rl}
A, s \models_v \mathsf{ff} & \textit{always} \\
A, s \models_v \varphi \wedge \psi & \textit{if } A, s \models_v \varphi \textit{ or } A, s \models_v \psi \\
A, \alpha s \models_v [\alpha]\varphi & \textit{if } A \overset{\alpha}{\Longrightarrow} B \textit{ and } B, s \models_v \varphi \\
A, s \models_v \mathsf{max}(X, \varphi) & \textit{if } A, s \models_v \varphi\{\mathsf{max}(X, \varphi)/X\}
\end{array}$$

Example 3 (Violation). Recall the safety formula φ_{safe} defined in (2). Actor A_3, from Ex. 2, together with the witness violating trace $\alpha\alpha\beta$ violate φ_{safe}, *i.e.*, $(A_3, \alpha\alpha\beta) \models_v \varphi_{\text{safe}}$. However, A_3 together with trace β do not violate φ_{safe}, *i.e.*, $(A_3, \beta) \not\models_v \varphi_{\text{safe}}$. Def. 2 relates a violating trace with an actor *only when* that trace leads the actor to a violation: if A_3 cannot perform the trace $\alpha\alpha\alpha\beta$, by Def. 2, we have $(A_3, \alpha\alpha\alpha\beta) \not\models_v \varphi_{\text{safe}}$, even though the trace is prohibited by φ_{safe}. A violating trace may also lead a system to a violation before its end, *e.g.*, $(A_3, \alpha\alpha\beta\alpha) \models_v \varphi_{\text{safe}}$ according to Def. 2. ■

[5] It follows from standard fixed-point theory that the implications of satisfaction relation are bi-implications for Satisfiability.

[6] We write $A, s \models_v \varphi$ in lieu of $(A, s, \varphi) \in \models_v$. It also follows from standard fixed-point theory that the constraints of the violation relation are bi-implications.

Despite the technical discrepancies between the two definitions, *e.g.*, maximal versus minimal fixpoints, a different model semantics *etc.*, we show that Def. 2 corresponds, in some sense, to the dual of Def. 1.

Theorem 1 (Correspondence). $\exists s.(A, s) \models_v \varphi \quad \textit{iff} \quad A \not\models_s \varphi$

Proof. For the *if* case we prove the contrapositive, *i.e.*, that $\forall s.A, s \not\models_{\mathrm{v}} \varphi$ implies $A \models_{\mathrm{s}} \varphi$ by co-inductively showing that $\mathcal{R} = \{(A, \varphi) \mid \forall s.A, s \not\models_{\mathrm{v}} \varphi\}$ is a satisfaction relation. For the *only-if* case we prove $\exists s.A, s \models_{\mathrm{v}} \varphi$ implies $A \not\models_{\mathrm{s}} \varphi$ by rule induction on $A, s \models_{\mathrm{v}} \varphi$. See [15] for details.

4 Correctness

Specifying online monitor correctness is complicated by the fact that, in general, we have limited control over the behaviour of the systems being monitored. For starters, a system that does not satisfy a property may still exhibit runtime behaviour that does not violate it, as discussed earlier in the case of system A_3 of Ex. 2 and Ex. 3. We deal with system non-determinism by only requiring monitor detection when the system performs a violating execution: this can be expressed through the violation relation of Def. 2.

At runtime, a system may also interfere with the execution of monitors. Appropriate *instrumentation* can limit system effects on the monitors. In our asynchronous actor setting, *direct* interferences from the system to the monitors can be precluded by (*i*) locating the monitors at process identifiers *not* known to the system (*ii*) preventing the monitors from communicating these identifiers to the system. These measures inhibit the system's ability to send messages to the monitors.

A system may also interfere with monitor executions indirectly by *diverging*, *i.e.*, infinite internal computation (τ-transitions) without external actions. This can prevent the monitors from executing and thus postpone indefinitely violation detections [25]. In our case, divergence is handled, in part, by the EVM itself, which guarantees fair executions for concurrent actors [7]. In settings where fair executions may be assumed, it suffices to require a weaker property from monitors, reminiscent of the condition in fair/should-testing[26]. Def. 3 states that, for an arbitrary basic action α, an actor system A satisfies the predicate *should-*α if, for *any* sequence of internal actions, there always *exists* an execution that can produce the action α; in the case of monitors, the external should-action is set to a reserved violation-detection action, *e.g.*, fail!.

Definition 3 (Should-α). $A \Downarrow_\alpha \overset{\mathit{def}}{=} \left(A \Longrightarrow B \quad \textit{implies} \quad B \overset{\alpha}{\Longrightarrow} \right)$

We limit monitoring to *monitorable* systems, where all actors are subject to a monitorable modality.

$$A \equiv (\nu\,\tilde{h})(i[e \triangleleft q]^m \parallel B) \quad \text{implies} \quad m = \circ$$

This guarantees that (*i*) they can be composed with a tracer actor (*ii*) all the basic actions produced by the system are recorded as trace entries at the tracer's mailbox.[7] Monitor correctness is defined for (unmonitored) *basic* systems, satisfying the condition:

[7] Due to asynchronous communication, even scoped actors can produce visible actions by sending messages to environment actors.

$$A \equiv (\nu\,\tilde{h})(i[e \lhd q]^{m} \parallel B) \quad \text{implies} \quad m = \bullet$$

which are instrumented to execute in parallel with the monitor. Our instrumentation is defined through the operation $\lceil - \rceil$, Def. 4, converting basic systems to monitorable ones using `trace/2` and `set_on_spawn` Erlang commands [7]; see Lemma 1. Importantly, instrumentation does not affect the visible behaviour of a basic system; see Lemma 2.

Definition 4 (Instrumentation). $\lceil - \rceil :: \textsc{Actr} \rightarrow \textsc{Actr}$ *is defined inductively as:*

$$\lceil i[e \lhd q]^{m} \rceil \overset{\textit{def}}{=} i[e \lhd q]^{\circ} \qquad \lceil B \parallel C \rceil \overset{\textit{def}}{=} \lceil B \rceil \parallel \lceil C \rceil \qquad \lceil (\nu\, i)B \rceil \overset{\textit{def}}{=} (\nu\, i)\lceil B \rceil$$

Lemma 1. *If A is a basic system then $\lceil A \rceil$ is monitorable.*

Lemma 2. *For all basic actors A where $i \notin \mathrm{fId}(A)$:*

$$A \xrightarrow{\alpha} B \textit{ iff } \begin{cases} (\nu\, i)(\lceil A \rceil \parallel i[e \lhd q]^{*}) \xrightarrow{j!v} (\nu\, i)(\lceil B \rceil \parallel i[e \lhd q : \{\mathsf{sd}, j, v\}]^{*}) & \textit{if } \alpha = j!v \\ (\nu\, i)(\lceil A \rceil \parallel i[e \lhd q]^{*}) \xrightarrow{j?v} (\nu\, i)(\lceil B \rceil \parallel i[e \lhd q : \{\mathsf{rv}, j, v\}]^{*}) & \textit{if } \alpha = j?v \\ (\nu\, i)(\lceil A \rceil \parallel i[e \lhd q]^{*}) \xrightarrow{\tau} (\nu\, i)(\lceil B \rceil \parallel i[e \lhd q]^{*}) & \textit{if } \alpha = \tau \end{cases}$$

We are now in a position to state monitor correctness, for some predefined violation-detection monitor action $\mathsf{fail}!$, Def. 5; in what follows, fail is always assumed to be fresh. We restrict our definition to expressions e located at a fresh scoped location i (not used by the system, *i.e.*, $i \notin \mathrm{fId}(A)$) with an empty mailbox, ϵ; expression e may then spawn concurrent submonitors while executing. The definition can be extended to generic concurrent monitors, *i.e.*, multiple expressions, in straightforward fashion.

Definition 5 (Correctness). *$e \in \textsc{Exp}$ is a correct monitor for $\varphi \in$ sHML iff for any basic actors $A \in \textsc{Actr}$, $i \notin \mathrm{fId}(A)$, and execution traces $s \in (\textsc{Act} \setminus \{\mathsf{fail}!\})^{*}$:*

$$(\nu\, i)(\lceil A \rceil \parallel i[e]^{*}) \overset{s}{\Longrightarrow} B \quad \textit{implies} \quad (A, s \models_{v} \varphi \quad \textit{iff} \quad B \Downarrow_{\mathsf{fail}!})$$

Def. 5 states that e correctly monitors property φ whenever, for any trace of environment interactions, s, of a monitored system, $(\nu\, i)(\lceil A \rceil \parallel i[e \lhd \epsilon]^{*})$, yielding system B, if s leads A to a violation of φ, then system B should always detect it, and viceversa.

5 Automated Monitor Synthesis

We define a translation from sHML formulas to Erlang *monitors* that asynchronously analyse a system and flag an alert whenever they detect violations by the current system execution (for the respective sHML formula). This translation describes the core algorithm for a tool automating monitor synthesis from sHML formulas [29].

Despite its relative simplicity, the sHML provides opportunities to perform concurrent monitoring. The most obvious case is the translation of conjunction formulas, $\varphi_1 \wedge \varphi_2$, whereby the resulting code needs to check *both* sub-formulas φ_1 and φ_2 so as to ensure that *neither* is violated.[8] A translation in terms of two concurrent (sub)monitors, each analysing different parts of the trace so as to ensure the observation of its respective sub-formula, constitutes a natural synthesis of the conjunction formula in our target language: it adheres to recommended Erlang practices advocating for concurrency wherever possible [7], but also allows us to benefit from the advantages of concurrent monitors discussed in the Introduction.

Example 4 (Conjunction Formulas). Consider the two sHML formulas

$$\varphi_{\text{no\_dup\_ans}} \triangleq [\alpha_{\text{call}}]\,(\,\mathsf{max}(X,\ [\beta_{\text{ans}}]\,[\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\beta_{\text{ans}}]\,[\alpha_{\text{call}}]\,X)\,)$$
$$\varphi_{\text{react\_ans}} \triangleq \mathsf{max}(Y,\ [\beta_{\text{ans}}]\,\mathsf{ff} \wedge [\alpha_{\text{call}}]\,[\beta_{\text{ans}}]\,Y\,)$$

Formula $\varphi_{\text{no\_dup\_ans}}$ requires that call actions α_{call} are at most serviced by a *single* answer action β_{ans}, whereas formula $\varphi_{\text{react\_ans}}$ requires that answer actions are only produced *in response to* call actions. Although one can rephrase the conjunction of the two formulas as a formula without a top-level conjunction, it is more straightforward to use two concurrent monitors executing in parallel (one for each sub-formula in $\varphi_{\text{no\_dup\_ans}} \wedge \varphi_{\text{react\_ans}}$). There are also other reasons why it would be beneficial to keep the sub-formulas separate: for instance, keeping the formulas disentangled improves maintainability and separation of concerns when subformulas originate from distinct specifying parties. ■

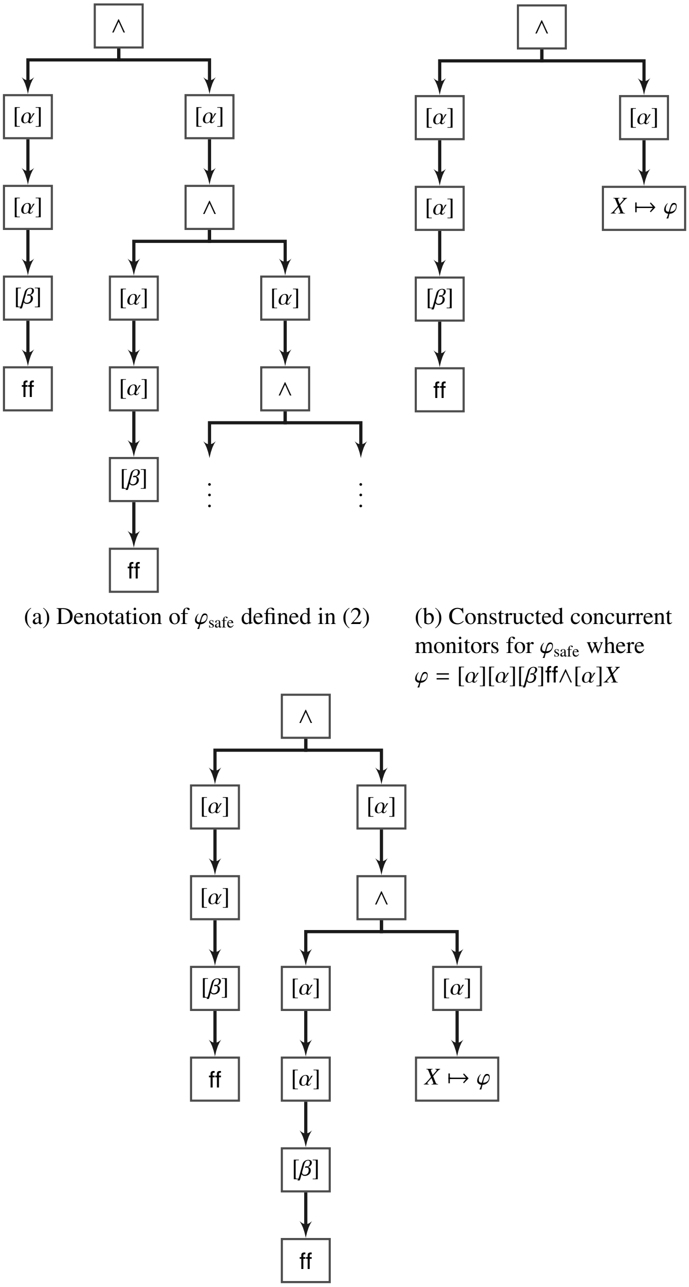
Multiple conjunctions also arise indirectly when used under fix-point operators. When synthesising concurrent monitors analysing separate branches of such recursive properties, it is important to generate monitors that can dynamic spawn further submonitors themselves as required at runtime, so as to keep the monitoring overheads to a minimum.

Example 5 (Conjunctions and Fixpoints). Recall φ_{safe}, from (2) in Ex. 2. Semantically, the formula represents the infinite-depth tree with an infinite number of conjunctions, depicted in Fig. 3(a). Although in practice, we cannot generate an infinite number of concurrent monitors, φ_{safe} will translate into possibly more than two concurrent monitors executing in parallel. ■

Our monitor synthesis, $[\![-]\!]^{\mathbf{m}}$ :: sHML → Exp , takes a *closed, guarded*[9] sHML formula and returns an Erlang function that is then parameterised by a map (encoded as a list of tuples) from formula variables to other synthesised monitors of the same form. The map encodes the variable bindings introduced by the construct $\mathsf{max}(X, \varphi)$; it is used for *lazy* recursive unrolling of formulas so as to minimize monitoring overhead. For instance, when synthesising formula φ_{safe} from Ex. 2, the algorithm initially spawns only two concurrent submonitors, one checking for the subformula $[\alpha][\alpha][\beta]\mathsf{ff}$, and another one checking for the formula $[\alpha]X$, as is depicted in Fig. 3(b). Whenever the rightmost submonitor in Fig. 3(b) observes the action α and reaches X, it unfolds X and spawns an additonal submonitor as depicted in Fig. 3(c), thereby increasing the monitor overheads incrementally.

[8] Since conjunctions are found in many monitoring logics, the concepts discussed here extend directly to other RV settings.

[9] In guarded sHML formulas, variables appear only as a sub-formula of a necessity formula.



(a) Denotation of φ_{safe} defined in (2)

(b) Constructed concurrent monitors for φ_{safe} where $\varphi = [\alpha][\alpha][\beta]\mathsf{ff}\wedge[\alpha]X$

(c) First expansion of the constructed monitor for φ_{safe}

Fig. 3. Monitor Combinator generation for φ_{safe} of Ex. 2

Definition 6 (Synthesis). $\llbracket - \rrbracket^{\mathbf{m}}$ *is defined on the structure of the* sHML *formula:*

$$\begin{aligned}
\llbracket \mathsf{ff} \rrbracket^{\mathbf{m}} &\overset{\mathit{def}}{=} \lambda x_{env}.\mathit{fail}! \\
\llbracket \varphi \wedge \psi \rrbracket^{\mathbf{m}} &\overset{\mathit{def}}{=} \begin{cases} \lambda x_{env}.\, y_{pid1} = \mathsf{spw}\,(\llbracket \varphi \rrbracket^{\mathbf{m}}(x_{env})),\ y_{pid2} = \mathsf{spw}\,(\llbracket \psi \rrbracket^{\mathbf{m}}(x_{env})), \\ \qquad \mathit{fork}(y_{pid1}, y_{pid2}) \end{cases} \\
\llbracket [\alpha]\varphi \rrbracket^{\mathbf{m}} &\overset{\mathit{def}}{=} \lambda x_{env}.\mathsf{rcv}\,(\,\mathrm{tr}(\alpha) \rightarrow \llbracket \varphi \rrbracket^{\mathbf{m}}(x_{env});\ \ \_ \rightarrow \mathit{stop}\,)\,\mathsf{end} \\
\llbracket \mathsf{max}(X, \varphi) \rrbracket^{\mathbf{m}} &\overset{\mathit{def}}{=} \lambda x_{env}.\, y_{mon} = \llbracket \varphi \rrbracket^{\mathbf{m}}, y_{mon}(\{'X', y_{mon}\} : x_{env}) \\
\llbracket X \rrbracket^{\mathbf{m}} &\overset{\mathit{def}}{=} \lambda x_{env}.\, y_{mon} = \mathit{lookUp}('X', x_{env}),\ y_{mon}(x_{env})
\end{aligned}$$

Auxiliary Function definitions and meta-operators:

$$\mathit{fork} \overset{\mathit{def}}{=} \mu y_{rec}.\lambda(x_{pid1}, x_{pid2}).\mathsf{rcv}\, z \rightarrow (x_{pid1}!z,\ x_{pid2}!z)\,\mathsf{end},\ y_{rec}(x_{pid1}, x_{pid2})$$

$$\mathit{lookUp} \overset{\mathit{def}}{=} \begin{cases} \mu y_{rec}.\lambda(x_{var}, x_{map}).\mathsf{case}\ x_{map}\ \mathsf{of}\ (\{x_{var}, z_{mon}\} : \_) \rightarrow z_{mon} \\ \qquad\qquad\qquad\qquad\qquad\qquad \_ : z_{tl} \rightarrow y_{rec}(x_{var}, z_{tl}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{nil} \rightarrow \mathit{exit} \\ \qquad\qquad\qquad \mathsf{end} \end{cases}$$

In Def. 6, the monitor for the formula ff immediately reports a violation to some supervisor actor identified as fail, handling the violation. Conjunction, $\varphi_1 \wedge \varphi_2$, translates into spawning the respective monitors for φ_1 and φ_2 and subsequently forwarding any trace messages to these spawned monitors through the auxiliary function fork. The translated monitor for $[\alpha]\varphi$ behaves as the monitor translation for φ once it receives a trace message encoding the occurrence of action α, using the function:

$$\mathrm{tr}(i?v) \overset{\mathrm{def}}{=} \{\mathsf{rv}, i, v\} \qquad\qquad \mathrm{tr}(i!v) \overset{\mathrm{def}}{=} \{\mathsf{sd}, i, v\}$$

Importantly, the monitor for $[\alpha]\varphi$ terminates if the trace message does not correspond to α. The translations of $\mathsf{max}(X, \varphi)$ and X are best understood together. The monitor for $\mathsf{max}(X, \varphi)$ behaves like that for φ, under the extended map where X is mapped to the monitor for φ, effectively modelling the formula unrolling $\varphi\{^{\mathsf{max}(X,\varphi)}/_X\}$ from Def. 2. The monitor for X retrieves the respective monitor translation bound to X in the map using function lookUp, and behaves like this monitor. *Closed* formulas guarantee that map entries are always found by lookUp, whereas *guarded* formulas guarantee that formula variables, X, are guarded by necessity conditions, $[\alpha]\varphi$ — this implements the *lazy* recursive unrolling of formulas and prevents infinite bound-variable expansions.

$$\begin{aligned}
\mathsf{Mon} &\overset{\mathrm{def}}{=} \lambda x_{\mathrm{frm}}.z_{\mathrm{pid}} = \mathsf{spw}\,(\llbracket x_{\mathrm{frm}} \rrbracket^{\mathbf{m}}(\mathsf{nil})), \mathsf{mLoop}(z_{\mathrm{pid}}) \\
\mathsf{mLoop} &\overset{\mathrm{def}}{=} \mu y_{\mathrm{rec}}.\lambda x_{\mathrm{pid}}.\mathsf{rcv}\, z_{\mathrm{msg}} \rightarrow (x_{\mathrm{pid}}!z_{\mathrm{msg}})\,\mathsf{end}, y_{\mathrm{rec}}(x_{\mathrm{pid}})
\end{aligned}$$

Monitor instrumentation, performed through the function Mon (defined above), spawns the synthesised function initialised to the empty map, nil, and then acts as a message forwarder to the spawned process, through the function mLoop (defined above), for any trace messages it receives through the tracing semantics discussed in Sec. 2.

We have constructed a tool [29] that implements the monitor synthesis of Def. 6: given an sHML formula it generates a monitor that can be instrumented with minimal changes to the system code, as discussed earlier in Sec. 4. The performance of our synthesised monitor was evaluated through a simulated server that launches individual workers to handle a series of requests from individual clients; we also injected faults making certain workers non-deterministically behave erratically. We synthesised monitors to check that each worker observes the *no-duplicate-reply* property from Ex. 4:

$$\varphi_{wrkr} \triangleq [wrk?req]\,(\,\mathsf{max}(X,\ [clnt!rply]\,[clnt!rply]\,\mathsf{ff} \wedge [clnt!rply]\,[wrk?req]\,X)\,)$$

and calculated the overheads incurred for varying number of client requests (*i.e.,* concurrent workers); we also compared this with the performance a monitor that checks for property violations in sequential fashion. Tests were carried out on an Intel Core i7 processor with 8GB of RAM, running Microsoft Windows 8 and EVM version R15B02. The result, summarised in the table below, show that our synthesised concurrent monitoring yields acceptable overheads that are consistently lower than those of a sequential monitor. We conjecture that this discrepancy can be increased further when monitoring for recursive properties with longer chains of necessity formulas.

	Unmonitored	**Sequential**		**Concurrent**		
No of. Reqs.	**Time**(μs)	**Time** (μs)	**Ovhd**(%)	**Time**(μs)	**Ovhd.**(%)	**Improv.**(%)
250	117.813	121.667	3.27	118.293	0.40	2.86
350	185.232	202.500	9.32	194.793	5.16	4.16
450	237.606	248.333	4.51	242.380	2.01	2.51
550	286.461	319.167	11.42	308.853	7.82	3.60
650	345.543	372.232	7.72	354.333	2.54	5.18

6 Proving Correctness

The preliminary results obtained in Sec. 5 advocate for the feasibility of using concurrent monitors. We however still need to show that the monitors synthesised are correct. Def. 5 allows us to state one of the main results of the paper, Theorem 2.

Theorem 2 (Correctness). *For all* $\varphi \in$ sHML*,* $\mathsf{Mon}(\varphi)$ *is a correct monitor for* φ*.*

Proving Theorem 2 directly can be an arduous task: for *any* sHML formula, it requires reasoning about *all* the possible execution paths of *any* monitored system in parallel with the instrumented monitor. We propose a formal technique for alleviating the task of ascertaining the monitor correctness of Def. 5 by teasing apart *three* separate (weaker) monitor-conditions: they are referred to as *Violation Detectability*, *Detection Preservation* and *Monitor Separability*. These conditions are important properties in their own right— for instance, Detection Preservation requires the monitor to behave *deterministically wrt.* violation detections. Moreover, the three conditions pose advantages to the checking of monitor correctness: since these conditions are independent to one another, they can be checked in parallel by distinct analysing entities; alternatively, the conditions that are inexpensive to check may be carried out before the more expensive ones, thus acting as vetting phases that abort early and keep the analysis cost to a minimum. More importantly though, the three conditions together imply our original monitor-correctness criteria.

The first sub-property is *Violation Detectability*, Lemma 3, guaranteeing that every violating trace s of formula φ is detectable by the respective synthesised monitor,[10] (the *only-if* case) and that there are no false detections (the *if* case). This property is easier to verify than Theorem 2 since it requires us to consider the execution of the monitor in isolation and, more importantly, requires us to verify the *existence of an execution path* that detects the violation; concurrent monitors typically have multiple execution paths.

Lemma 3 (Violation Detectability). *For basic $A \in$ Actr and $i \notin \mathrm{fId}(A)$, $A \overset{s}{\Longrightarrow}$ implies:*

$$A, s \models_v \varphi \quad \textit{iff} \quad i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \xrightarrow{\mathsf{fail!}}$$

Detection Preservation (Lemma 4), the second sub-property, is not concerned with relating detections to actual violations. Instead it guarantees that if a monitor can potentially detect a violation, further reductions do not exclude the possibility of this detection. In the case where monitors always have a finite reduction *wrt.* their mailbox contents (as it turns out to be the case for monitors synthesised by Def. 6) this condition guarantees that the monitor will deterministically detect violations.

Lemma 4 (Detection Preservation). *For all $\varphi \in$ sHML, $q \in$ Val**

$$\Big(i[\mathsf{Mon}(\varphi) \triangleleft q]^* \xRightarrow{\mathsf{fail!}} \;\; \textit{and} \;\; i[\mathsf{Mon}(\varphi) \triangleleft q]^* \Longrightarrow B\Big) \quad \textit{implies} \quad B \xRightarrow{\mathsf{fail!}}$$

The third sub-property is Separability, Lemma 5, which implies that the behaviour of a (monitored) system *is independent of* the monitor and, dually, the behaviour of the monitor depends, *at most*, on the trace generated by the system.

Lemma 5 (Monitor Separability). *For all basic $A \in$ Actr, $i \notin \mathrm{fId}(A)$, $\varphi \in$ sHML, and $s \in (\mathrm{Act} \setminus \{\mathsf{fail!}\})^*$,*

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \textit{ implies } \exists B', B'' \textit{s.t.}$$

$$B \equiv (\nu\, i)(B' \parallel B'') \quad \textit{and} \quad A \overset{s}{\Rightarrow} A' \textit{ s.t. } B' = \lceil A' \rceil \quad \textit{and} \quad i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''$$

These three properties suffice to show monitor correctness.

Theorem 2 (Correctness). *For all $\varphi \in$ sHML, $\mathsf{Mon}(\varphi)$ is a correct monitor for φ.*

Proof. According to Def. 5 we have to show:

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \quad \text{implies} \quad (A, s \models_{\mathrm{v}} \varphi \quad \text{iff} \quad B \Downarrow_{\mathsf{fail!}})$$

For the *only-if* case, we assume

$$(\nu\, i)(A \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \tag{3}$$

$$A, s \models_{\mathrm{v}} \varphi \tag{4}$$

[10] We elevate tr to basic action sequences s in pointwise fashion, $\mathrm{tr}(s)$, where $\mathrm{tr}(\epsilon) = \epsilon$.

To show $B \Downarrow_{\mathsf{fail!}}$, by Def. 3 we also assume $B \Longrightarrow B'$, for arbitrary B', and then be required to prove that $B' \overset{\mathsf{fail!}}{\Longrightarrow}$. From (3), $B \Longrightarrow B'$ and Lemma 5 we know

$$\exists B'', B''' s.t.\ B' \equiv (\nu\, i)(B'' \parallel B''') \tag{5}$$

$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{6}$$

$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''' \tag{7}$$

From (6), (4) and Lemma 3 we obtain $i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \overset{\mathsf{fail!}}{\Longrightarrow}$, and from (7) and Lemma 4 we get $B''' \overset{\mathsf{fail!}}{\Longrightarrow}$. Hence, by (5), and standard transition rules for parallel composition and scoping, Par and Scp, we can reconstruct $B' \overset{\mathsf{fail!}}{\Longrightarrow}$, as required.
For the *if* case we assume:

$$(\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B \tag{8}$$

$$B \Downarrow_{\mathsf{fail!}} \tag{9}$$

and have to prove $A, s \models_{\mathrm{v}} \varphi$. From (9) we know $B \overset{\mathsf{fail!}}{\Longrightarrow}$. Together with (8) this implies

$$\exists B'\ s.t.\ (\nu\, i)(\lceil A \rceil \parallel i[\mathsf{Mon}(\varphi)]^*) \overset{s}{\Longrightarrow} B' \xrightarrow{\mathsf{fail!}} \tag{10}$$

From Lemma 5 and (10) we obtain

$$\exists B'', B''' s.t.\ B' = (\nu\, i)(B'' \parallel B''') \tag{11}$$

$$A \overset{s}{\Longrightarrow} A' \text{ for some } A' \text{ where } \lceil A' \rceil = B'' \tag{12}$$

$$i[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)]^* \Longrightarrow B''' \tag{13}$$

From (10), (11) and the freshness of fail! to A we deduce that $B''' \xrightarrow{\mathsf{fail!}}$, and subsequently, by (13), we get $i_{\mathrm{mtr}}[\mathsf{Mon}(\varphi) \triangleleft \mathrm{tr}(s)] \overset{\mathsf{fail!}}{\Longrightarrow}$. Therefore, by (12) and Lemma 3 we obtain $A, s \models_{\mathrm{v}} \varphi$, as required. □

7 Conclusion

We have studied a more intensional notion of correctness for monitor synthesis in a concurrent online setting; we worked close to the actual implementation level of abstraction so as to enhance our confidence in the correctness of our instrumented monitors. More precisely, we have identified a number of additional issues raised when proving monitor correctness in this concurrent setting, illustrating them through a case study that builds a tool [29] automating monitor synthesis from a reactive property logic (sHML) to asynchronous monitors in a concurrent language (Erlang). The specific contributions of the paper, in order of importance, are:

1. A novel formal *definition of monitor correctness*, Def. 5, dealing with issues such as system non-determinism and system interference.

2. A *proof technique* teasing apart aspects of the monitor correctness definition, Lem. 3, Lem. 4 and Lem. 5, allowing us to prove correctness in stages. We subsequently apply this technique to prove the *correctness of our tool*, Thm. 2.
3. An alternative *violation* characterisation of the logic, sHML, that is more amenable to runtime analysis and reasoning about monitor correctness, together with a proof of correspondence for this reformulation, Thm. 1.
4. An extension of a formalisation for Erlang describing its tracing semantics, Sec. 2.
5. A formal monitor synthesis definition from sHML formulas to Erlang code, Def. 6.

Related Work: The aforementioned work, [17,27,4], discusses monitor synthesis from a different logic, namely LTL, to either pseudocode, automata or Büchi automata; none of this work considers online concurrent monitoring, circumventing issues associated with concurrency and system interference. There is considerable work on runtime monitoring of web services, *e.g.*, [13,5] verifying the correctness of reactive (communication) properties, similar to those expressed through sHML; to the best of our knowledge, none of this work tackles correct monitor synthesis from a specified logic. In [9], Colombo *et al.* develop an Erlang RV tool using the EVM tracing mechanism but do not consider the issue of correct monitor generation. Fredlund [16] adapted a variant of HML to specify correctness properties in Erlang, albeit for model checking purposes. There is also work relating HML formulas with tests, namely [1]. Our monitors differ from tests, as in [1], in a number of ways: (*i*) they are defined in terms of *concurrent* actors, as opposed to *sequential* CCS processes; (*ii*) they analyse systems *asynchronously*, acting on traces, whereas tests interact with the system *directly*, forcing certain system behaviour; (*iii*) they are expected to *always* detect violations when they occur whereas tests are only required to have *one* possible execution that detects violations.

Future Work: The monitoring semantics of Section 2 can be used as a basis to formally prove existing Erlang monitoring tools such as [9,10]. sHML can also be extended to handle limited, monitorable forms of liveness properties (often termed co-safety properties [21]). It is also worth exploring mechanisms for synchronous monitoring, as opposed to asynchronous variant studied in this paper. Erlang also facilitates monitor *distribution* which can be used to lower monitoring overheads [11]. Distributed monitoring can also be used to increase the expressivity of our tool so as to handle correctness properties for distributed programs. The latter extension, however, poses a departure from our setting because the unique trace described by our framework would be replaced by separate independent traces at each location, where the lack of a total ordering of events may prohibit the detection of certain violations [14].

References

1. Aceto, L., Ingólfsdóttir, A.: Testing Hennessy-Milner Logic with Recursion. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 41–55. Springer, Heidelberg (1999)
2. Armstrong, J.: Programming Erlang. The Pragmatic Bookshelf (2007)
3. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)

4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. 20, 14:1–14:64 (2011)
5. Cao, T.-D., Phan-Quang, T.-T., Felix, P., Castanet, R.: Automated runtime verification for web services. In: ICWS, pp. 76–82. IEEE (2010)
6. Carlsson, R.: An introduction to core erlang. In: PLI 2001 (Erlang Workshop) (2001)
7. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly (2009)
8. Clarke Jr., E., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
9. Colombo, C., Francalanza, A., Gatt, R.: Elarva: A monitoring tool for erlang. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 370–374. Springer, Heidelberg (2012)
10. Colombo, C., Francalanza, A., Grima, I.: Simplifying contract-violating traces. In: FLACOS. EPTCS, vol. 94, pp. 11–20 (2012)
11. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: polyLARVA: Runtime verification with configurable resource-aware monitoring boundaries. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 218–232. Springer, Heidelberg (2012)
12. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: TIME. IEEE (2005)
13. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 204–220. Springer, Heidelberg (2011)
14. Francalanza, A., Gauci, A., Pace, G.: Distributed system contract monitoring. JLAP (to appear, 2013)
15. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors in erlang. Technical Report CS2013-01, University of Malta (January 2013), `http://www.cs.um.edu.mt/svrg/papers.html` (accessible)
16. Fredlund, L.-Å.: A Framework for Reasoning about Erlang Code. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (2001)
17. Geilen, M.: On the construction of monitors for temporal logic properties. ENTCS 55(2), 181–199 (2001)
18. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32(1), 137–161 (1985)
19. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI, pp. 235–245. Morgan Kaufmann (1973)
20. Kozen, D.: Results on the propositional μ-calculus. TCS 27, 333–354 (1983)
21. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC, pp. 377–410. ACM (1990) (invited paper, 1989)
22. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14(3), 249–289 (2012)
23. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
24. Milner, R., Parrow, J., Walker, D.: Modal logics for mobile processes. Theoretical Computer Science 114, 149–171 (1993)
25. Nicola, R.D., Hennessy, M.C.B.: Testing equivalences for processes. TCS, 83–133 (1984)
26. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. 205(2), 125–198 (2007)
27. Sen, K., Roşu, G., Agha, G.: Generating optimal linear temporal logic monitors by coinduction. In: Saraswat, V.A. (ed.) ASIAN 2003. LNCS, vol. 2896, pp. 260–275. Springer, Heidelberg (2003)
28. Sen, K., Vardhan, A., Agha, G., Roşu, G.: Efficient decentralized monitoring of safety in distributed systems. In: ICSE, pp. 418–427 (2004)
29. Seychell, A.: DetectEr, `http://www.cs.um.edu.mt/svrg/Tools/detectEr`
30. Svensson, H., Fredlund, L.-Å., Benac Earle, C.: A unified semantics for future erlang. In: Erlang Workshop, pp. 23–32. ACM (2010)

Practical Interruptible Conversations
Distributed Dynamic Verification with Session Types and Python

Raymond Hu[1], Rumyana Neykova[1], Nobuko Yoshida[1],
Romain Demangeon[1], and Kohei Honda[2]

[1] Imperial College London
[2] Queen Mary, University of London

Abstract. The rigorous and comprehensive verification of communication-based software is an important engineering challenge in distributed systems. Drawn from our industrial collaborations [33,28] on Scribble, a choreography description language based on multiparty session types, this paper proposes a dynamic verification framework for structured interruptible conversation programming. We first present our extension of Scribble to support the specification of asynchronously interruptible conversations. We then implement a concise API for conversation programming with interrupts in Python that enables session types properties to be dynamically verified for distributed processes. Our framework ensures the global safety of a system in the presence of asynchronous interrupts through independent runtime monitoring of each endpoint, checking the conformance of the local execution trace to the specified protocol. The usability of our framework for describing and verifying choreographic communications has been tested by integration into the large scientific cyberinfrastructure developed by the Ocean Observatories Initiative. Asynchronous interrupts have proven expressive enough to represent and verify their main classes of communication patterns, including asynchronous streaming and various timeout-based protocols, without requiring additional synchronisation mechanisms. Benchmarks show conversation programming and monitoring can be realised with little overhead.

1 Introduction

The main engineering challenges in distributed systems include finding suitable specifications that model the range of states exhibited by a system, and ensuring that these specifications are followed by the implementation. In message passing applications, rigorous specification and verification of communication protocols is particularly crucial: a protocol is the interface to which concurrent components should be independently implementable while ensuring their composition will form a correct system as a whole. Multiparty Session Types (MPST) [17,6] is a type theory for communication-oriented programming, originating from works on types for the π-calculus, towards tackling this challenge. In the original MPST setting, protocols are expressed as types and static type checking verifies that the system of processes engaged in a communication session (also referred to as a *conversation*) conforms to a globally agreed protocol. The properties enjoyed by well-typed processes are communication safety (no unexpected messages or races during the execution of the conversation) and deadlock-freedom.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 130–148, 2013.

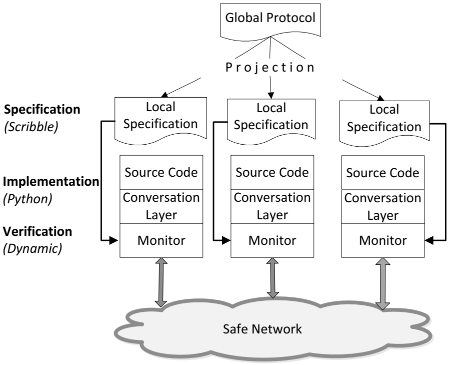


Fig. 1. Scribble methodology from global specification to local runtime verification

In this paper, we present the design and implementation of a framework for dynamic verification of protocols based on MPST, developed from our collaboration with industry partners [33,28] on the application of MPST theory. In this ongoing work, we are motivated to adapt MPST to dynamic verification for several reasons. First, session type checking is typically designed for languages with first-class communication and concurrency primitives, whereas our collaborations use mainstream engineering languages, such as Python and Java, that lack the features required to make static session typing tractable. Distributed systems are also often heterogeneous in nature, meaning that different languages and techniques (e.g. the control flow of an event-driven program is tricky to verify statically) may be used in the implementation of one system. Dynamic verification by communication monitoring allows us to verify MPST safety properties directly for mainstream languages in a more scalable way. Second, a system may use third-party components or services for which the source code is unavailable for type checking. Third, certain protocol specification features, such as assertions on specific message values, can be precisely evaluated at runtime, while static treatments would usually be more conservative.

Framework Overview. Figure 1 illustrates the methodology of our framework. The development of a communication-oriented application starts with the specification of the intended interactions (the choreography) as a *global* protocol using the Scribble protocol description language [34], an engineering incarnation of the formal MPST type language. The core features of Scribble include multicast message passing and constructs for branching, recursive and parallel conversations. These features support the specification of a wide range of protocols, from domains such as standard Internet applications [18], parallel algorithms [27] and Web services [12].

Our toolchain validates that the global protocol satisfies certain well-formedness properties, such as coherent branches (no ambiguity between participants about which branch to follow) and deadlock-freedom (between parallel flows). From a well-formed global protocol, the toolchain mechanically generates (projects) Scribble *local* protocols for each participant (role) defined in the protocol. A local protocol is essentially a view of the global protocol from the perspective of one role, and provides a more direct specification for endpoint implementation than the global protocol.

When a conversation is initiated at runtime, the monitor at each endpoint generates a finite state machine (FSM) representation of the local communication behaviour from the local protocol for its role. In our implementation, the FSM generation is an extension of the correspondence between MPST and communication automata in [13] to support *interruptible* sessions (discussed below) and optimised to avoid parallel state explosion. The monitor tracks the communication actions performed by the endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the FSM. Each monitor thus works to protect both the endpoint from illegal actions by the environment, and the network from bad endpoints. In this way, our framework is able to ensure from the local verification of each endpoint that the global progress of the system as a whole conforms to the original global protocol [7], and that unsafe actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints.

This MPST monitoring framework has been integrated into the Python-based runtime platform developed by the Ocean Observatories Initiative (OOI) [28]. The OOI is a project to establish a cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. Their architecture relies on the combination of high-level protocol specifications of network services (expressed as Scribble protocols [29]) and distributed runtime monitoring to regulate the behaviour of third-party applications within the system [31]. Although this work is in collaboration with the OOI, our implementation can be used orthogonally as a standalone monitoring framework for distributed Python applications.

Contributions and Summary. This paper demonstrates the application of multiparty session types, through the Scribble protocol language, to industry practice by presenting (1) the first implementation of MPST-based dynamic protocol verification (as outlined above) that offers the same safety guarantees as static session type checking, and (2) a use case motivated extension of Scribble to support the first construct for the verification of asynchronous communication interrupts in multiparty sessions.

We developed the extension of Scribble with asynchronous interrupts to support a range of OOI use cases that feature protocol structures in which one flow of interactions can be asynchronously interrupted by another. Examples include various service calls (request-reply) with timeout, and publish-subscribe applications where the consumer can request to pause, resume and stop externally controlled sensor feeds. Although the existing features of Scribble (i.e. those previously established in MPST theory) are sufficiently expressive for many communication patterns, we observed that these important structures could not be directly or naturally represented without interrupts.

We outline the structure of this paper, summarising the contributions of each part:

§ 2 presents a use case for the extension of Scribble with asynchronous interrupts. This is a new feature in MPST, giving the first general mechanism for nested, multiparty session interrupts. We explain why implementing this feature is a challenge in session types. The previous works on exceptions in session types are purely theoretical, and are either restricted to binary session types (i.e. not multiparty) [11], do not support nesting and continuations [11,10], or rely on additional implicit synchronisation [9]. A formal proof of the correctness of our design is given in § 5.

§ 3 discusses the Python implementation of our MPST monitoring framework that we have integrated into the OOI project, and demonstrates the global-to-local projection of Scribble protocols, endpoint implementation, and local FSM generation. § 3.1 describes a concise API for conversation programming in Python. The API decorates conversation messages with the runtime session information required by the monitors to perform the dynamic verification. § 3.2 discusses the monitor implementation, how asynchronous interrupts are handled, and the key architectural requirements of our framework.

§ 4 evaluates the performance of our monitor implementation through a collection of benchmarks. The results show that conversation programming and monitoring can be realised with low overhead.

The source code of our Scribble toolchain, conversation runtime and monitor, performance benchmarks and further resources are available from the project page [35].

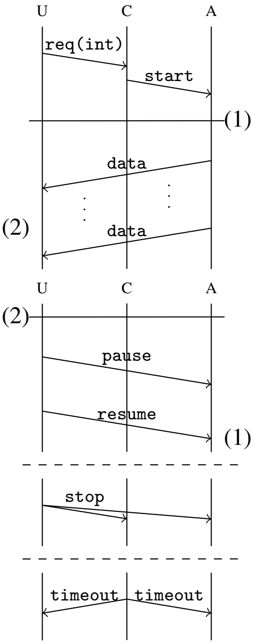
2 Communication Protocols with Asynchronous Interrupts

This section expands on why and how we extend Scribble to support the specification and verification of asynchronous session interrupts, henceforth referred to as just interrupts. Our running example is based on an OOI project use case, which we have distilled to focus on session interrupts. Using this example, we outline the technical challenges of extending Scribble with interrupts.

Resource Access Control (RAC) Use Case. As is common practice in industry, the cyberinfrastructure team of the OOI project [28] manages communication protocol specifications through a combination of informal sequence diagrams and prose descriptions. Figure 2 (left) gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control use case [29], regarding access control of users to sensor devices in the ION Cyberinfrastucture for data acquisition. In the ION setting, a User interacts with a sensor device via its Agent proxy (which interacts with the device via a separate protocol outside of this example). ION Controller agents manage concerns such as authentication of users and metering of service usage.

For brevity, we omit from the diagram some of the data types to be carried in the messages and focus on the *structure* of the protocol. The depicted interaction can be summarised as follows. The protocol starts at the top of the left-hand diagram. User sends Controller a `request` message to use a sensor for a certain amount of time (the `int` in parentheses), and Controller sends a `start` to Agent. The protocol then enters a phase (denoted by the horizontal line) that we label (1), in which Agent streams `data` messages (acquired from the sensor) to User. The vertical dots signify that Agent produces the stream of data freely under its own control, i.e. without application-level control from User. User and Controller, however, have the option at any point in phase (1) to move the protocol to the phase labelled (2), below.

Phase (2) comprises three alternatives, separated by dashed lines. In the upper case, User *interrupts* the stream from Agent by sending Agent a `pause` message. At some subsequent point, User sends a `resume` and the protocol returns to phase (1). In the middle case, User interrupts the stream, sending both Agent and Controller a `stop` message. This is the case where User does not want any more sensor data, and ends the protocol for all three participants. Finally, in the lower case, Controller interrupts the stream by



```
1   global protocol ResourceAccessControl(role User as U,
2             role Controller as C, role Agent as A) {
3     req(duration:int) from U to C;
4       // U requests the device for some duration
5     start() from C to A;
6     interruptible { // U, C and A in scope
7       rec X {
8         interruptible { // U and A in scope
9           rec Y {
10            data() from A to U;
11            continue Y;
12          }
13        } with { // Interrupts A in Y
14          pause() by U;
15        }
16        resume() from U to A;
17        continue X;
18      }
19    } with { // Interrupts A and C/U in X
20      stop() by U; // Any time within the duration
21      timeout() by C; // Duration is up
22    }
23  }
```

Fig. 2. Sequence diagram (left) and Scribble protocol (right) for the RAC use case

sending a `timeout` message to User and Agent. This is the case where, from Controller's view, the session has exceeded the requested duration, so Controller interrupts the other two participants to end the protocol. Note this diagram actually intends that `stop` (and `timeout`) can arise anytime after (1), e.g. between `pause` and `resume` (a notational ambiguity that is compensated by additional prose comments in the specification).

Interruptible Multiparty Session Types. Figure 2 (right) shows a Scribble protocol that formally captures the structure of interaction in the Resource Access Control use case and demonstrates the uses of our new extension for asynchronous interrupts. Besides the formal foundations, we find the Scribble specification is more explicit and precise, particularly regarding the combination of compound constructs such as choice and recursion, than the sequence diagram format, and provides firmer implementation guidelines for the programmer (demonstrated in § 3.1).

A Scribble protocol starts with a header declaring the protocol name (in Figure 2, `ResourceAccessControl`) and role names for the participants (three roles, aliased in the scope of this protocol definition as `U`, `C` and `A`). Lines 3 and 5 straightforwardly correspond to the first two communications in the sequence diagram. The Scribble syntax for message signatures, e.g. `req(duration:int)`, means a message with *operator* (i.e. header, or label) `req`, carrying a *payload* `int` annotated as `duration`. The `start()` message signature means operator `start` with an empty payload.

We now come to "phase" (1) of the sequence diagram. The new `interruptible` construct captures the informal usage of protocol phases in disciplined manner, making explicit the interrupt messages and the *scope* in which they apply. Although the syntax has been designed to be readable and familiar to programmers, `interruptible` is an advanced construct that encapsulates several aspects of asynchronous interaction, which we discuss at the end of this section.

The intended communication protocol in our example is clarified in Scribble as two nested `interruptible` statements. The outer statement, on lines 6–22, corresponds to the options for User and Controller to end the protocol via the `stop` and `timeout` interrupts. An `interruptible` consists of a main body of protocol actions, here lines 7–18, and a set of interrupt message signatures, lines 19–22. The statement stipulates that each participant behaves by either (a) following the protocol specified in the body until finished for their role, or (b) raising or detecting a specified interrupt at any point during (a) and exiting the statement. Thus, the outer `interruptible` states that `U` can interrupt the body (and end the protocol) by a `stop()` message, and `C` by a `timeout()`.

The body of the outer `interruptible` is a labelled recursion statement with label `X`. The `continue X;` inside the recursion (line 17) causes the flow of the protocol to return to the top of the recursion (line 7). This recursion corresponds to the loop implied by the sequence diagram that allows User to pause and resume repeatedly. Since the recursion body always leads to the `continue`, Scribble protocols of this form state that the loop should be driven indefinitely by one role, until one of the interrupts is raised by *another* role. This communication pattern cannot be expressed in multiparty session types without `interruptible`.

The body of the `X`-recursion is the inner `interruptible`, which corresponds to the option for User to pause the stream. The stream itself is specified by the `Y`-recursion, in which `A` continuously sends `data()` messages to `U`. The inner `interruptible` specifies that `U` may interrupt the `Y`-recursion by a `pause()` message, which is followed by the `resume()` message from `U` before the protocol returns to the top of the `X`-recursion.

Challenges of Asynchronous Interrupts in MPST. The following summarises our observations from the extension and usage of MPST with asynchronous interrupts. We find the basic operational meaning of `interruptible`, as illustrated in the above example, is readily understood by architects and developers, which is a primary consideration in the design of Scribble. The challenges in this extension are in the design of the supporting runtime and verification techniques to preserve the desired safety properties in the presence of `interruptible`. The challenges stem from the fact that `interruptible` combines several tricky, from a session typing view, aspects of communication

```
// Well-formed, but incorrect semantics:
// the recursion cannot be stopped
par {
    rec Y {
        data() from A to U;
        continue Y; }
} and {
    // Does not stop the recursion
    pause() from U to A;
}
resume() from U to A;
```

```
// Naive mixed-choice is not well-formed
choice at A {
   // A should make the choice..
   rec Y {
        data() from A to U;
        continue Y; }
} or {
    // ..not U
    pause() from U to A;
}
resume() from U to A;
```

Fig. 3. Naive, incorrect interruptible encoding attempts using parallel (left) and choice (right)

Conversation API operation	Purpose
`create(protocol_name, invitation_config.yml)`	Initiate conversation, send invitations
`join(self, role, principal_name)`	Accept invitation
`send(role, op, payload)`	Send message with operation and payload
`recv(role)`	Receive message from role
`recv_async(self, role, callback)`	Asynchronous receive
`scope(msg)`	Create a conversation scope
`close()`	Close the connection to the conversation

Fig. 4. The core Python Conversation API operations

behaviours that session type systems traditionally aim to prohibit, in order to prevent communication races and thereby ensure the desired safety properties.

A key aspect, due to asynchrony, is that an interrupt may occur in parallel to the actions of the roles being interrupted (e.g. `pause` by `U` to `A` while `A` is streaming `data` to `U`). Although standard MPST (and Scribble) support parallel protocol flows, the interesting point here is that the nature of an interrupt is to preclude further actions in another parallel flow under the control of a different role, whereas the basic MPST parallel does not permit such interference. Figure 3 (left) is a naively incorrect attempt to express this aspect without interruptible: the second parallel path is never able to intefere with the first to actually stop the recursion.

Another aspect is that of mixed choice in the protocol, in terms of both communication direction (e.g. `U` may choose to either receive the next `data` or send a `stop`), and between different roles (e.g. `U` and `C` independently, and possibly concurrently, interrupt the protocol) due to multiparty. Moreover, the implicit interrupt choice is truly optional in the sense that it may never be selected at runtime. The basic choice in standard MPST (e.g. as defined in [17,13]) is inadequate because it is designed to safely identify a single role as the decision maker, who communicates exactly one of a set of message choices unambiguously to all relevant roles. Figure 3 (right) demonstrates a naive mixed choice that is not well-formed (it breaks the unique sender condition in [13]).

Due to the asynchronous setting, it is also important that `interruptible` does not require implicit synchronisations to preserve communication safety. The underlying mechanisms are formalised and the correctness of our extension is proved in § 5.

3 Runtime Verification

This section discusses implementation details of our monitoring framework and the accompanying Python API (Conversation API) for writing monitorable, distributed MPST programs. This work is the first implementation of the theory in [7] in practice, and is the first (theory or practice) to support a general, asynchronous MPST interrupt mechanism in the protocol language and API for endpoint implementation.

We first outline the verification methodology of our framework to clarify the purpose of the main components. Developers write endpoint programs in native Python using the Conversation API, an MPST-based message passing library that supports the core MPST primitives for communication programming. The execution of these

```
1  local protocol ResourceAccessControl
2    at User as U (role Controller as C,
3      role Agent as A) {
4   req(duration:int) to C;
5   interruptible {
6     rec X {
7      interruptible {
8       rec Y {
9        data() from A;
10       continue Y;
11      }
12     } with {
13      pause() by U;
14     }
15     resume() to A;
16     continue X;
17    }
18  } with {
19    stop() by U;
20    timeout() by C;
21  }
22 }
```

```
1  class UserApp(BaseApp):
2   user, controller, agent =
3     ['User', 'Controller', 'Agent']
4   def start(self):
5    conv = Conversation.create(
6      'RACProtocol', 'config.yml')
7    c = conv.join(user, 'alice')
8    # request 1 hour access
9    c.send(controller, 'req', 123)
10   with c.scope('timeout', 'stop')
11     as c1:
12    while not self.limit_reached():
13     with c1.scope('pause') as c2:
14      while not buffer.full:
15       resource = c2.recv(controller)
16       buffer.append(resource)
17      c2.send_interrupt('pause')
18     # sleep before resume
19     c1.send(agent, 'resume')
20    if self.should_stop():
21     c1.send_interrupt('stop')
22   c.close()
```

Fig. 5. Scribble local protocol (left) and Python implementation (right) for the User role

operations at each endpoint is performed by the local conversation library runtime. The full runtime includes infrastructure for inline monitoring of conversation actions, while the lightweight version is used with an outline (i.e. externally hosted) monitor. In both cases, the API enables MPST verification of message exchanges by the monitor by embedding a small amount of MPST meta data (e.g. conversation identifier, message kind and operator, source and destination roles), based on the actions and current state of the endpoint, into the message payload. For each conversation initiated or joined by an endpoint, the monitor generates an FSM from the local protocol for the role of the endpoint. The monitor uses the FSM to track the progress of this conversation according to the protocol, validating each message (via the meta data) as it is sent or received.

3.1 Conversation API

The Python Conversation API offers a high-level interface for safe conversation programming, mapping the interaction primitives of session types to lower-level communication actions on concrete transports. Our current implementation is built over an AMQP [2] transport. In summary, the API provides the functionality for (1) session initiation and joining, (2) basic send/receive and (3) *conversation scope* management for handling interrupt messages. Figure 4 lists the core API operations. The invitation operations (`create` and `join`) have not been captured in standard MPST systems, but have formal counterparts in the literature in formalisms such as [11].

We demonstrate the usage of the API in a Python implementation of the local protocol projected for the User role. Figure 5 gives the local protocol and its implementation.

Conversation Initiation. First, the `create` method of the Conversation API (line 5, right) initiates a new conversation instance of the `ResourceAccessControl` (Figure 2) protocol, and returns a token that can be used to join the conversation locally. The config.yml file specifies which network principals will play which roles in this session and the runtime sends invitation messages to each. The join method confirms that the endpoint is joining the conversation as the principal alice playing the role User, and returns a conversation channel object for performing the subsequent communication operations. Once the invitations are sent and accepted (via Conversation.join), the conversation is established and the intended message exchanges can proceed. As a result of the initiation procedure, the runtime at every participant has a mapping (conversation table) between each role and their AMQP addresses.

Conversation Message Passing. Following its local protocol, the User program sends a request to the controller, stating the duration for which it requires access to agent. The send method called on the conversation channel c takes, in this order, the destination role, message operator and payload values as arguments. This information is embedded into the message payload as part of the conversation meta data, and is later used by the monitor in the runtime verification. The recv method can take the source role as a single argument, or additionally the operator of the desired message. Send is asynchronous, meaning that the operation does not block on the corresponding receive; however, the basic receive does block until the complete message has been received. For asynchronous, non-blocking receives, the API provides recv_async to be used in an event-driven style.

Interrupt Handling via Conversation Scopes. This example demonstrates a way of handling conversation interrupts by combining conversation scopes with the Python with statement (an enhanced try-finally construct). We use with to conveniently capture interruptible conversation flows and the nesting of interruptible scopes, as well as automatic close of interrupted channels in the standard manner, as follows. The API provides the c.scope() method, as in line 10, to create and enter the scope of an interruptible Scribble block (here, the outer interruptible of the RAC protocol). The timeout and stop arguments associate these message signatures as interrupts to this scope. The conversation channel c1 returned by scope is a wrapper of the parent channel c that (1) records the current scope of every message sent in its meta data, (2) ensures every send and receive operation is guarded by a check on the local interrupt queue, and (3) tracks the nesting of scope contexts through nested with statements. The interruptible scope of c1 is given by the enclosing with (lines 10–21); if, e.g., a timeout is received within this scope, the control flow will exit the with to line 22. The inner with (lines 13–17), corresponding to the inner interruptible block, is associated with the pause interrupt. When an interrupt, e.g. pause in line 17, is thrown (send_interrupt) to the other conversation participants, the local and receiver runtimes each raise an internal exception that is either handled or propagated up, depending on the interrupts declared at the current scope level, to direct the interrupted control flow accordingly. The delineation of interruptible scopes by the global protocol, and its projection to each local protocol, thus allows interrupted control flows to be coordinated between distributed participants in a structured manner.

The scope wrapper channels are closed (via the with) after throwing or handling an interrupt message. For example, using c1 (outside its parent scope) after a timeout is received will be flagged as an error. By identifying the scope of every message from its

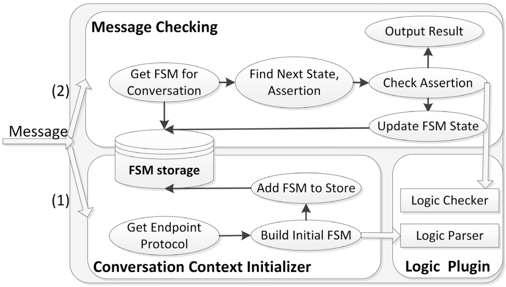


Fig. 6. Monitor workflow for (1) invitation and (2) in-conversation messages

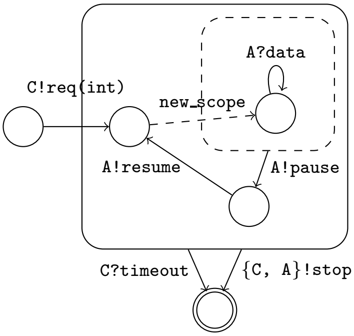


Fig. 7. Nested FSM generated from the User local protocol

meta data, the conversation runtime (and monitor) is able to compensate for the inherent discrepancies in protocol synchronisation, due to asynchronous interrupts between distributed endpoints, by safely discarding out-of-scope messages. In our example, the User runtime discards `data` messages that arrive after `pause` is thrown. To prevent the loss of such messages in the application logic when the stream is resumed, we could extend the protocol to simply carry the id of the last received resource in the payload of the `resume` (in line 21). The API can also make the discarded data available to the programmer through secondary (non-monitored) operations.

An alternative event-driven implementation using `receive_asyc` and callbacks (that can, however, be safely monitored against the same local protocol) is given in [35].

3.2 Monitoring Architecture

Inline and Outline Monitoring. In order to guarantee global safety, our monitoring framework imposes *complete mediation* of communications: no communication action should have an effect unless the message is mediated by the monitor. This principle requires that all outgoing messages from a principal before reaching the destination, and all incoming messages before reaching the principal, are routed through the monitor.

The monitor implementation (and the accompanying theory [7]) is compatible with a range of monitor configurations. At one end of the spectrum is *inline monitoring*, where the monitor is embedded into the endpoint code. Then there are various configurations for *outline monitoring*, where the monitor is positioned externally to its component. In the OOI project, our focus has been to integrate our framework for inline monitoring due to the architecture of the OOI message interceptor stack [31].

Monitor Implementation. Figure 6 depicts the main components and internal workflow of our prototype monitor. The lower part relates to conversation initiation. The *invitation* message carries (a reference to) the local protocol for the invitee and the conversation id (global protocols can also be exchanged if the monitor has the facility for projection.) The monitor generates the FSM from the local protocol following [13]. Our implementation differs from [13] in the treatment of parallel subprotocols (i.e. unordered message sequences), and additionally supports interrupts. For efficiency, we extend [13] to generate a nested FSM for each conversation thread, avoiding the potential state explosion that comes from constructing their product. This allows FSM generation in polynomial time and space in the length of the local protocol. The (nested) FSMs are stored in a hash table with conversation id as the key. Transition functions are similarly hashed, each entry having the shape: $(\mathit{current\_state},\ \mathit{transition}) \mapsto (\mathit{next\_state},\ \mathit{assertion},\ \mathit{var})$, where *transition* is a triple $(\mathit{label}, \mathit{sender}, \mathit{receiver})$ and *var* is the variable binder for the message payload. Due to standard MPST well-formedness (message label distinction), any nested FSM is uniquely identifiable from any unordered message, i.e. message-to-transition matching in a conversation FSM is deterministic.

The upper part of Figure 6 relates to *in-conversation* messages, which carry the conversation id (matching an entry in the FSM hash table), sender and receiver fields, and the message label and payload. This information allows the monitor to retrieve the corresponding FSM (by matching the message signature to the FSM's transition function). Assertions associated to communication actions are evaluated by invoking an external logic engine; a monitor can be configured to use various logic engines, such as for the validation of assertions, automata-based specifications (e.g. security automata), or other stateful properties. Our current implementation uses a basic Python predicate evaluator, which is sufficient for the use case protocols we have developed so far.

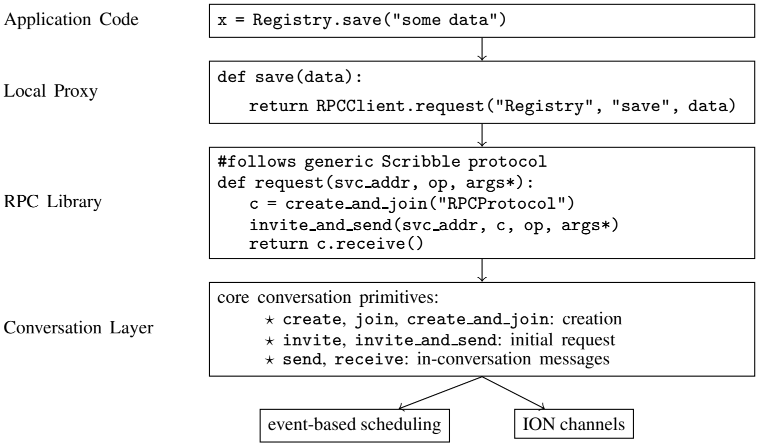
Monitoring Interrupts. FSM generation for interruptible local protocols again makes use of nested FSMs. Each `interruptible` induces a nested FSM given by the main interruptible block, as illustrated in Figure 7 for the User local protocol. The monitor internally augments the nested FSM with a scope id, derived from the signature of the interruptible block, and an interrupt table, which records the interrupt message signatures that may be thrown or received in this scope. Interrupt messages are marked via the same meta data field used to designate invitation and in-conversation messages, and are are validated in a similar way except that they are checked against the interrupt table. However, if an interrupt arrives that does not have a match in the interrupt table of the immediate FSM(s), the check searches upwards through the parent FSMs; the interrupt is invalid if it cannot be matched after reaching the outermost FSM is reached.

4 Evaluation

Our dynamic MPST verification framework has been implemented and integrated into the current release of the Ocean Observatories infrastructure [30]. This section reports on our integration efforts and the performance of our framework.

4.1 Experience: OOI Integration

The current release of OOI is based on a Service-Oriented Architecture, with all of the distributed system services accessible by RPC. As part of their efforts to move to agent-based systems in the next release, and to support distributed governance for more than just individual RPC calls, we engineered the following step-by-step transition. The first step was to add our Scribble monitor to the message interceptor stack of their middleware [31]. The second was to propose our conversation programming interface to



```
x = Registry.save("some data")
```

```
def save(data):
   return RPCClient.request("Registry", "save", data)
```

```
#follows generic Scribble protocol
def request(svc_addr, op, args*):
   c = create_and_join("RPCProtocol")
   invite_and_send(svc_addr, c, op, args*)
   return c.receive()
```

core conversation primitives:
- ⋆ create, join, create_and_join: creation
- ⋆ invite, invite_and_send: initial request
- ⋆ send, receive: in-conversation messages

Fig. 8. Translation of an RPC command into lower-level conversation calls

the OOI developers. To facilitate the use of session types without obstructing the existing application code, we preserved the interface of the RPC libraries but replaced the underlying machinery with the distributed runtime for session types (as shown in Figure 8, the RPC library is now realised on top of the Conversation Layer). As wrappers to the conversation primitives, all RPC calls are now automatically verified by the inline MPST monitors. This approach was feasible because no changes were required to existing application code, but at the same time, developers now have the option to use the Conversation API directly for conversations more complex than RPC. The next step in this ongoing integration work involves porting higher-level and more complex OOI application protocols, such as distributed agent negotiation [29], to Scribble specifications and Conversation API implementations.

4.2 Benchmarks

The potential performance overhead that the Conversation Layer and monitoring could introduce to the system is an important consideration. The following performance measurements for the current prototype show that our framework can be realised at reasonable cost. Table 1 presents the execution time comparing RPC calls using the original OOI RPC library implementation and the conversation-based RPC with and without monitor verification. On average, 13% overhead is recorded for conversations of 10 consecutive RPCs, mostly due to the FSM generation from the textual local Scribble protocol (our implementation currently uses Python ANTLR); the cost of message validation itself is negligible in comparison. We plan to experiment with optimisations such as pre-generating or caching FSMs to reduce the monitor initialisation time.

The second benchmark gives an idea of how well our framework scales beyond basic RPC patterns. Table 2 shows that the overall verification architecture (Conversation Layer and inline monitor) scales reasonably with increasing session length (number of message exchanges) and increasing parallel states (nested FSM size): “Rec States” is

Table 1. Original OOI RPC vs. conversation-based RPC

	10 RPCs (s)
RPC Lib	0.103
No Monitor	0.108 +4%
Monitor	0.122 +13%

Table 2. Conversation execution time for an increasing number of sequential and parallel states

Rec States	NoM (s)	Mon (s)	
10	0.92	0.95	+3.2%
100	8.13	8.22	+1.1%
1000	80.31	80.53	+0.8%

Par States	NoM (s)	Mon (s)	
10	0.45	0.49	+8%
100	4.05	4.22	+4.1%
1000	40.16	41.24	+2.7%

the number of states passed through sequentially by a simple recursive protocol (used to parameterise the length of the conversation), and "Par States" the number of parallel states in a parallel protocol. Two benchmark cases are compared. The main case "Monitor" (Mon) is fully monitored, i.e. FSM generation and message validation are enabled for both the client and server. The base case for comparison "No Monitor" (NoM) has the client and server in the same configuration, but monitors are disabled (messages do not go through the interceptor stack). As above, we found that the overhead introduced by the monitor when executing conversations of increasing number of recursive and parallel states is again mostly due to the cost of the initial FSM generation. We also note that the relative overhead decreases as the session length increases, because the one-time FSM generation cost becomes less prominent. For dense FSMs, the worse case scenario results in linear overhead growth wrt. the number of parallel branches.

In both of the above tables, the presented figures are the mean time for the client and server, connected by a single-broker AMQP network, to complete one conversation after repeating the benchmark 100 times for each parameter configuration. The client and server Python processes (including the conversation runtime and monitor) and the AMQP broker were each run on separate machines (Intel Core2 Duo 2.80 GHz, 4 GB memory, 64-bit Ubuntu 11.04, kernel 2.6.38). Latency between each node was measured to be 0.24 ms on average (ping 64 bytes). The full source code of the benchmark protocols and applications and the raw data are available from the project page [35].

4.3 Use Cases

We conclude our evaluation with some remarks on use cases we have examined. Table 3 features a list of protocols, sourced from both the research community and our industry use cases, that we have written in Scribble and used to test our monitor implementation on more realistic protocol specifications. A natural question for our methodology, being based on explicit specification of protocols, is the overhead imposed on developers wrt. writing protocols, given that a primary motivation for the development of Scribble is to reduce the design and testing effort for distributed systems. Among these use cases, we found the average Scribble global protocol is roughly 10 LOC, with the longest one at 26 LOC, suggesting that Scribble is reasonably concise.

The main factors that may affect the performance and scalability of our monitor implementation, and which depend on the shape of a protocol, are (i) the time required for the generation of FSMs and (ii) the memory overhead that may be induced by the generation of nested FSMs in case of parallel blocks and interrupts. Table 3 measures

Table 3. Use case protocols implemented in Scribble

Use Cases from research papers	Global Scribble (LOC)	FSM Memory (B)	Generation Time (s)
A vehicle subsystem protocol [21]	8	840	0.006
Map web-service protocol [15]	10	1040	0.010
A bidding protocol [24]	26	1544	0.020
Amazon search service [16]	12	1088	0.010
SQL service [32]	8	1936	0.009
Online shopping system [14]	10	1024	0.008
Travel booking system [14]	16	1440	0.013
Use Cases from OOI and Savara			
A purchasing protocol [20]	11	1088	0.010
A banking example [29]	16	1564	0.013
Negotiation protocol [29]	20	1320	0.014
RPC with timeout [29]	11	1016	0.013
Resource Access Control [29]	21	1854	0.018

these factors for each of the listed protocols. The time required for FSM generation remains under 20 ms, measuring on average to be around 10 ms. The memory overhead also remains within reasonable boundaries (under 1.5 KB), indicating that FSM caching is a feasible optimisation approach. The full Scribble protocols can be found at [35].

From our experience of running our conversation monitoring framework within the OOI system, we expect that, in many large distributed systems, the cost of a decentralised monitoring infrastructure would be largely overshadowed by the raw cost of communication (latency, routing) and other services running at the same time. Considering the presented results, we thus believe the important benefits in terms of safety and management of high-level applications come at a reasonable cost and would be a realistic mechanism in many distributed systems.

5 Interruptible Session Type Theory and Related Work

5.1 Session Type Theory for Interrupts

In this subsection, we sketch the underlying session type theory with interrupts and its correctness result, *session fidelity*, justifying our design choices. We build over the multiparty session theory [17], adding syntax and semantics for interrupts. In our theory, global types correspond to session specifications whereas local types are used to express monitored behaviours of processes [7]. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that realises, through an explicit identifier, the domain of interrupts. Our scope-based session types can handle nested interrupts and multiparty continuations to interruptible blocks, allowing us to model truly asynchronous exceptions implemented in this paper (these features have not been modelled in existing MPST theories for exceptions [11,10,9]). The full definitions and proofs are available from [35].

Global types (G) below correspond to Scribble protocols. Scopes are made explicit by the use of scope variables S, corresponding to the dynamic scope generation present in the implementation in § 3.1. Roles in types are denoted by $\mathtt{r}$, and labels with l.

$$\begin{array}{ll} G ::= & \mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I} \mid G|G \mid \{\!|G|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G' \mid \mu\mathbf{x}.G \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend} \\ T ::= & \mathtt{r}!\{l_i.T_i\}_{i\in I} \mid \mathtt{r}?\{l_i.T_i\}_{i\in I} \mid T|T \\ & \mid \{\!|T|\!\}^S \triangleleft \langle \mathtt{r}!l\rangle;T' \mid \{\!|T|\!\}^S \triangleright \langle \mathtt{r}?l\rangle;T' \mid \mu\mathbf{x}.T \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend} \end{array}$$

The main primitive is the interaction with directed choice: $\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}$ is a communication between the sender $\mathtt{r}$ and the receiver $\mathtt{r}'$ which involves a choice between several labels l_i, the corresponding continuations are denoted by the G_i. Parallel composition $G_1|G_2$ allows the execution of interactions not linked by causality.

Our types feature a new interrupt mechanism by explicit interruptible scopes: we write $\{\!|G|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G'$ to denote a creation of an interruptible block identified by scope S, containing protocol G, that can be interrupted by a message l from $\mathtt{r}$ and continued after completion (either normal or exceptional) with protocol G'. This construct corresponds to the `interruptible` of Scribble, presented in § 2. Note that we allow interruptible scopes to be nested. This syntax (and the related properties) can be easily extended to multiple messages from different roles. We use `Eend` (resp. `end`) to denote the exceptional (resp. normal) termination of a scope.

The local type syntax (T) given above follows the same pattern, but the main difference is that the interruptible operation is divided into two sides, one $\triangleleft$ side for the roles which can send an interrupt $\{\!|T|\!\}^S \triangleleft \langle \mathtt{r}!l\rangle;T'$, and the $\triangleright$ side for the roles which should expect to receive an interrupt message $\{\!|T|\!\}^S \triangleright \langle \mathtt{r}?l\rangle;T'$.

$$\begin{array}{rl} G_{\mathtt{ResCont}} = & \mathtt{U}{\to}\mathtt{C}:\text{req};\mathtt{C}{\to}\mathtt{A}:\text{start}\{\!|\ \mu X.\{\!|\mu Y.\mathtt{A}{\to}\mathtt{U}:\text{data};Y|\!\}^{S_2}\langle \text{pause by } \mathtt{U}\rangle; \\ & \mathtt{U}{\to}\mathtt{A}:\text{resume};X|\!\}^{S_1}\langle \text{stop by } \mathtt{U}, \text{timeout by } \mathtt{C}\rangle;\mathtt{end} \end{array}$$

Above we describe a global type which corresponds to the Scribble protocol in Figure 2. The explicit naming of the scopes, S_1 and S_2, correspond to the dynamic scope generations in § 3.1, and are required to formalise the semantics of local types.

We define the relation $G \leadsto G'$ as:

$$\begin{array}{c} \mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I} \leadsto G_i \qquad \{\!|G|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G_0 \leadsto \{\!|\mathtt{Eend}|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G_0 \\ G \leadsto G' \text{ implies } \{\!|G|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G_0 \leadsto \{\!|G'|\!\}^S\langle l \text{ by } \mathtt{r}\rangle;G_0 \quad G \leadsto G' \text{ implies } G \mid G_0 \leadsto G' \mid G_0 \end{array}$$

and say G' is a *derivative* of G if $G \leadsto^* G'$. We define *configurations* Δ,Σ as a pair of a mapping from a session channel to a local type and a collection of queues (a mapping from a session channel to a vector of the values). Configurations model the behaviour of a network of monitored agents. We say a configuration Δ,Σ corresponds to a collection of global types $G_1,\dots,G_l$ whenever Σ is empty and the environment Δ is a projection of $G_1,\dots,G_l$. The reduction semantics of the configuration ($\Delta,\Sigma \to \Delta',\Sigma'$) is defined using the contexts with the scopes. Formal definitions can be found in [35].

The correctness of our theory is ensured by Theorem 1, which states a local enforcement implies global correctness: if a network of monitored agents (modelled as a configuration) corresponds to a collection of well-formed specifications and makes some steps by firing messages, then the network can perform reductions (consuming these messages) and eventually reaches a state that corresponds to a collection of well-formed specifications, obtained from the previous one. This property guarantees that the network is always linked to the specification, and proves, with the previous dynamic monitoring process theory [7], that the introduction of interruptible blocks to the syntax and semantics yields a sound theory. The proofs can be found in [35].

Theorem 1 (Session fidelity). *If Δ corresponds to $G_1, \ldots, G_n$ and $\Delta_0, \varepsilon \rightarrow^* \Delta, \Sigma$, there exists $\Delta, \Sigma \rightarrow^* \Delta', \varepsilon$ such that Δ' corresponds to $G'_1, \ldots, G'_n$ which are derivatives of $G_1, \ldots, G_n$.*

5.2 Related Work

Distributed Runtime Verification. The work in [3] explores runtime monitoring based on session types as a test framework for multi-agent systems (MAS). A global session type is specified as cyclic Prolog terms in Jason (a MAS development platform). Their global types are less expressive in comparison with the language presented in this paper (due to restricted arity on forks and the lack of session interrupts). Their monitor is centralised (thus no projection facilities are discussed), and neither formalisation, global safety property nor proof of correctness is given in [3].

Other works, notably from the multi-agent community, have studied distributed enforcement of global properties through monitoring. A distributed architecture for local enforcement of global laws is presented by Zhang et al. [36], where monitors enforce *laws* expressed as event-condition-action. In [26], monitors may trigger sanctions if agents do not fulfil their obligations within given deadlines. Unlike such frameworks, where all agents belonging to a group obey the same set of laws, our approach asks agents to follow personalised laws based on the role they play in each session.

In runtime verification for Web services, the works [24,25] propose FSM-based monitoring using a rule-based declarative language for specifications. These systems typically position monitors to protect the safety of service interfaces, but do not aim to enforce global network properties. Cambronero et al. [8] transform a subset of Web Services Choreography Description Language into timed-automata and prove their transformation is correct with respect to timed traces. Their approach is model-based, static and centralised, and does not treat either the runtime verification or interrupts. Baresi et al. [5] develop a runtime monitoring tool for BPEL with assertions. A major difference is that BPEL approaches do not treat or prove global safety. BPEL is expressive, but does not support distribution and is designed to work in a centralised manner. Kruger et al. [22] propose a runtime monitoring framework, projecting MSCs to FSM-based distributed monitors. They use aspect-oriented programming techniques to inject monitors into the implementation of the components. Our outline monitoring verifies conversation protocols and does not require such monitoring-specific augmentation of programs. Gan [14] follows a similar but centralised approach of [22]. As a language for protocol specification, a main advantage of Scribble (i.e. MPST) over alternatives, such as message sequence charts (MSC), CDL and BPML, is that MPST has both a formal basis and an in-built mechanism (projection) for decentralisation, and is easily integrated with the language framework as demonstrated for Python in this paper.

Language-Based Monitoring Tools. Jass [19] is a precompiler tool for monitoring the dynamic behaviour of sequential objects and the ordering of method invocations by annotating Java programs with specifications that can be checked at runtime. Other approaches to runtime verification of program execution by monitors generated from language-based specifications include: aspect-oriented programming [23]; other works that use process calculi formalisms, such as CSP [19]; monitors based on FSM skeletons associated to various forms of underlying patterns [1,4]; and the analysis of dynamic parametric traces [4]. Our monitor framework has been influenced by these works and shares similarities with some of the presented RV techniques. However, the target program domain and focus of our work are different. Our framework is specifically designed for decentralised monitoring of distributed programs with diverse participants and interleaving sessions, as opposed to monitoring the execution of a single program and verifying its local properties. The basis of our design and implementation is the theory of multiparty session types, over which we have developed practically motivated extensions to the type language and the methodology for runtime verification.

6 Conclusion

We have implemented the first dynamic verification of distributed communications based on multiparty session types and shown that a new feature for interruptible conversations is effective in the runtime verification of message exchanges in a large cyberinfrastructure [28] and Web services [33,34]. Our implementation automates distributed monitoring by generating FSMs from local protocol projections. We sketched the formalisation of asynchronous interruptions with conversation scopes, and proved the correctness of our design through the session fidelity theorem. Future work includes the incorporation of more elaborate handling of error cases into monitors and automatic generation of service code stubs. Although our implementation work is ongoing through industry collaborations, the results already confirm the feasibility of our approach. We believe this work contributes towards methodologies for better specification and more rigorous governance of network conversations in distributed systems.

Acknowledgements. We thank Gary Brown and the Scribble team for discussions and collaborations. This work has been partially sponsored by the Ocean Observatories Initiative, VMWare, Cognizant and EPSRC EP/K034413/1, EP/K011715/1, EP/G015635/1 and EP/G015481/1.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. SIGPLAN Not 40(10), 345–364 (2005)
2. Advanced Message Queuing protocols (AMQP) homepage, `http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`
3. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic generation of self-monitoring mass from multiparty global session types in Jason. In: Baldoni, M., Dennis, L., Mascardi, V., Vasconcelos, W. (eds.) DALT 2012. LNCS, vol. 7784, pp. 76–95. Springer, Heidelberg (2013)
4. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. SIGPLAN Not 42(10), 589–608 (2007)

5. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: ICSOC 2004, pp. 193–202 (2004)
6. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
7. Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 50–65. Springer, Heidelberg (2013)
8. Cambronero, M.-E., et al.: Validation and verification of web services choreographies by using timed automata. J. Log. Algebr. Program. 80(1), 25–49 (2011)
9. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty session. In: FSTTCS 2010. LIPICS, vol. 8, pp. 338–351 (2010)
10. Carbone, M.: Session-based choreography with exceptions. Electr. Notes Theor. Comput. Sci. 241, 35–55 (2009)
11. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
12. W3C WS-CDL, `http://www.w3.org/2002/ws/chor/`
13. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
14. Gan, Y., et al.: Runtime monitoring of web service conversations. In: CASCON 2007, pp. 42–57. ACM (2007)
15. Ghezzi, C., Guinea, S.: Run-time monitoring in service-oriented architectures. In: Test and Analysis of Web Services, pp. 237–264. Springer (2007)
16. Hallé, S., Bultan, T., Hughes, G., Alkhalaf, M., Villemaire, R.: Runtime verification of web service interface contracts. Computer 43(3), 59–66 (2010)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
18. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-safe eventful sessions in java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
19. Jass Home Page, `http://modernjass.sourceforge.net/`
20. Jboss Savara project, `http://www.jboss.org/savara/downloads`
21. Krüger, I.H., Meisinger, M., Menarini, M.: Runtime verification of interactions: from mscs to aspects. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 63–74. Springer, Heidelberg (2007)
22. Krüger, I.H., Meisinger, M., Menarini, M.: Interaction-based runtime verification for systems of systems integration. J. Log. Comput. 20(3), 725–742 (2010)
23. LAVANA project, `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`
24. Li, Z., Han, J., Jin, Y.: Pattern-based specification and validation of web services interaction properties. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 73–86. Springer, Heidelberg (2005)
25. Li, Z., Jin, Y., Han, J.: A runtime monitoring and validation framework for web service interactions. In: ASWEC 2006. IEEE (2006)
26. Minsky, N.H., Ungureanu, V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. TOSEM 9, 273–305 (2000)
27. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe Parallel Programming with Message Optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012)
28. Ocean Observatories Initative, `http://www.oceanobservatories.org/`

29. OOI, https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI
30. OOI codebase, https://github.com/ooici/pyon
31. OOI COI governance framework, https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Framework
32. Salaün, G.: Analysis and verification of service interaction protocols - a brief survey. In: TAV-WEB. EPTCS, vol. 35, pp. 75–86 (2010)
33. JBoss Savara Project, http://www.jboss.org/savara
34. Scribble Project homepage, http://www.scribble.org
35. Full version of this paper, http://www.doc.ic.ac.uk/~rn710/mon
36. Zhang, W., Serban, C., Minsky, N.: Establishing global properties of multi-agent systems via local laws. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 170–183. Springer, Heidelberg (2007)

Runtime Verification with Particle Filtering

Kenan Kalajdzic[1], Ezio Bartocci[1], Scott A. Smolka[2],
Scott D. Stoller[2], and Radu Grosu[1,2]

[1] Faculty of Informatics, Vienna University of Technology, Austria
[2] Department of Computer Science, Stony Brook University, USA

Abstract. We introduce *Runtime Verification with Particle Filtering* (RVPF), a powerful and versatile method for controlling the tradeoff between uncertainty and overhead in runtime verification. Overhead and accuracy are controlled by adjusting the frequency and duration of *observation gaps*, during which program events are not monitored, and by adjusting the number of particles used in the RVPF algorithm. We succinctly represent the program model, the program monitor, their interaction, and their observations as a dynamic Bayesian network (DBN). Our formulation of RVPF in terms of DBNs is essential for a proper formalization of *peek events*: low-cost observations of parts of the program state, which are performed probabilistically at the end of observation gaps. Peek events provide information that our algorithm uses to reduce the uncertainty in the monitor state after gaps.

We estimate the internal state of the DBN using particle filtering (PF) with sequential importance resampling (SIR). PF uses a collection of conceptual particles (random samples) to estimate the probability distribution for the system's current state: the probability of a state is given by the sum of the importance weights of the particles in that state. After an observed event, each particle chooses a state transition to execute by sampling the DBN's joint transition probability distribution; particles are then redistributed among the states that best predicted the current observation. SIR exploits the DBN structure and the current observation to reduce the variance of the PF and increase its performance.

We experimentally compare the overhead and accuracy of our RVPF algorithm with two previous approaches to runtime verification with state estimation: an exact algorithm based on the forward algorithm for HMMs, and an approximate version of that algorithm, which uses precomputation to reduce runtime overhead. Our results confim RVPF's versatility, showing how it can be used to control the tradeoff between execution time and memory usage while, at the same time, being the most accurate of the three algorithms.

1 Introduction

Runtime verification does not come for free. It introduces runtime overhead, thereby altering the timing-related behavior of the program under scrutiny. In applications with realtime constraints, overhead control may be necessary to reduce overhead to an acceptable level.

In previous work [5], we introduced *Software Monitoring with Controllable Overhead* (SMCO), an overhead-control technique that selectively turns monitoring on and

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 149–166, 2013.

off, such that the use of a short- or long-term overhead budget is maximized and never exceeded. Gaps in monitoring, however, introduce uncertainty in the monitoring results.

To quantify the uncertainty, one can estimate the current state of the program. We developed a framework for this, called *Runtime Verification with State Estimation* (RVSE) [10], in which a hidden Markov model (HMM) is used to succinctly model the program and compute the uncertainty in predictions due to incomplete information.

While monitoring is on, the observed program events drive the transitions of the property checker, modeled as a deterministic finite automaton (DFA). They also provide information used to help correct the state estimates (specifically, state probability distributions) computed from the HMM transition probabilities, by comparing the output probabilities in each state with the observed outputs. When monitoring is off, the transition probabilities in the HMM alone determine the updated state estimate after the gap, and the output probabilities in the HMM drive the transitions of the DFA. Each gap is characterized by a *gap length distribution*, which is a probability distribution for the number of missed observations during that gap.

Our algorithm was based on an optimal state estimation algorithm, known as the forward algorithm, extended to handle gaps. Unfortunately, this algorithm incurs high overhead, especially for longer sequences of gaps, because it involves repeated matrix multiplications using the observation-probability and transition-probability matrices. In our measurements, this was often more than a factor of 10 larger than the overhead of monitoring the events themselves!

To reduce the runtime overhead, we developed a version of the algorithm, which we call *approximate precomputed RVSE* (AP-RVSE), that precomputes the matrix calculations and stores the results in a table [1]. Essentially, AP-RVSE precomputes a potentially infinite graph unfolding, where nodes are labeled with state probability distributions, and edges are labeled with transitions. To ensure the table is finite, we introduced an approximation in the calculations, controlled by an accuracy parameter ϵ: if a newly computed matrix differs from the matrix on an existing node by at most ϵ according to the L^1-norm, then we re-use the existing node instead of creating a new one. With this algorithm, the runtime overhead is low, independent of the desired accuracy, but higher accuracy requires larger tables, and the memory requirements could become problematic. Also, if the set of gap length distributions that may appear in an execution is not known in advance, precomputation is infeasible.

This paper introduces an alternative approach, called *Runtime Verification with Particle Filtering* (RVPF), to control the balance among runtime overhead, memory usage, and prediction accuracy. In particle filtering (PF) [7], the probability distribution of states is approximated by the proportion of particles in each state. The particle filtering process works in three recurring steps. First, the particles are advanced to their successor states by sampling from the HMM's transition probability distribution. Second, each particle is assigned a weight corresponding to the output probability of the observed program event. Third, the particles are resampled according to the normalized weights from the second step; this has the effect of redistributing the particles so that they provide a better prediction of the program events.

To reduce the variance of PF, we exploit the knowledge of the current program event and the particular structure of the DBN and employ a variant of PF known as *sequential* *importance resampling* (SIR). The resampling step (which is a performance bottleneck) does not have to be performed in each round, and the particles are advanced to their successor states by sampling from the HMM's transition probability distribution *conditioned by* the current observation. While this conditional probability distribution cannot be computed in general, it can be computed for HMMs.

To handle gaps, we extend PF in a manner that is consistent with the one we devised for the forward algorithm: as long as gaps are the only observations, the particles are advanced to their successor states by sampling from the HMM's transition probability distribution conditioned on the most probable output event. Such output events are chosen by sampling from the output probability distribution of the HMM conditioned on the previous HMM state. These events are used to drive the DFA transitions.

In contrast to our previous work [10,1], we model the HMM, the DFA, and their composition in a more elegant and succinct way as a dynamic Bayesian network (DBN). This allows us to properly formalize a new kind of event, called *peek events*, which are inexpensive observations of part of the program state. In many applications, program states and monitor states are correlated, and hence peek events can be used to narrow down the possible states of the monitor DFA. We use peek events at the end of monitoring gaps to refocus the HMM and DFA states. Our combination of these two kinds of observations, program events and peek events, is akin to *sensor fusion* in robotics.

Adjusting the number of particles used by RVPF provides a versatile way to tune the memory requirements, runtime overhead, and prediction accuracy. With larger numbers of gaps, the particles get more widely dispersed in the state space, and more particles are needed to cover all of the interesting states. To evaluate the performance and accuracy of RVPF, we implemented it along with our previous two algorithms in C and compared them through experiments based on the benchmarks used in [1]. Our results confirm RVPF's versatility. Specifically, we demonstrate in Section 6 that, with the right choice of the number of particles, RVPF consumes 80–100 times less memory than AP-RVSE while being twice as fast as RVSE, and the most accurate of the three algorithms.

The rest of the paper is organized as follows. Section 2 provides background. Sections 3 and 4 define the runtime verification problem we are addressing and system model, respectively. Section 5 presents the RVPF algorithm. Section 6 describes our evaluation methodology and the results of our experiments. Section 7 discusses related work. Section 8 offers concluding remarks and directions for future work.

2 Background

This section provides background information on Bayesian networks, dynamic Bayesian networks, particle filtering, and runtime verification with state estimation.

A *Bayesian network* is a directed acyclic graph in which each node corresponds to a (discrete or continuous) random variable. An edge from node X to node Y indicates that X has a direct influence on Y, and X is called a *parent* of Y. Let B be a Bayesian network over variables $X_1, \ldots, X_n$. Each X_i has a conditional probability distribution $P(X_i \mid \mathit{Parents}(X_i))$ that quantifies the influence of the parents on the node [7].

The meaning of B is a joint distribution over its variables. Let $P(x_1, \ldots, x_n)$ abbreviate $P(X_1 = x_1 \wedge \cdots \wedge X_n = x_n)$, i.e., the conjunction of particular assignments to each variable. Then $P(x_1, \ldots, x_n) = \prod P(x_i \mid parents(X_i))$, where $parents(X_i)$ denotes the values of $Parents(X_i)$ that appear in $x_1, \ldots, x_n$.

A *dynamic Bayesian Network* (DBN) is a Bayesian network that relates random variables to each other over adjacent time steps. Moreover, some variables are observable, and some are not. Let $\mathbf{X_t}$ denote the set of *state variables* at time t. State variables are assumed to be unobservable. Let $\mathbf{O_t}$ denote the set of *observable variables* at time t. The observation at time t is $\mathbf{O_t} = \mathbf{o_t}$ for some set of values $\mathbf{o_t}$.

A *Hidden Markov Model* (HMM) is a special kind of DBN; specifically, an HMM is a DBN with a single state variable and a single observable variable. We refer to a value of the observable variable of an HMM as an *observable action*.

To construct a DBN, one must specify the *prior distribution* $\mathbf{P}(\mathbf{X}_0)$, capturing the initial state distribution; the *transition model* $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$, capturing the dependency of the next state on the current state; and the *observation model* $\mathbf{P}(\mathbf{O}_t \mid \mathbf{X}_t)$, encoding the dependency of the observation on the current state. The transition and observation models are represented as a Bayesian network.

Particle filtering (PF) is a sequential Monte Carlo method that can be used to perform state estimation in a Bayesian network [7]. PF can be used to estimate the state probability distribution $\mathbf{P}(\mathbf{X}_t)$, given an observation sequence $o_{1:t}$. In one of the most commonly used forms of particle filtering, known as *sequential importance resampling* (SIR), a population of N_p particles is first created and assigned initial states by sampling from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. A three-step update cycle is then repeated for each time step: (i) each particle is propagated forward by sampling the new state value $\mathbf{x}_t$ given the previous state $\mathbf{x}_{t-1}$ of the particle, based on the transition model $\mathbf{P}(\mathbf{X}_t \mid \mathbf{x}_{t-1})$; (ii) each particle is weighted by the probability it assigns to the new evidence, $\mathbf{P}(\mathbf{o}_t \mid \mathbf{x}_t)$; (iii) the population is *resampled*, i.e., a new population of N_p (unweighted) particles is created, where each new particle is selected from the current population, and the probability that a particular particle is selected is proportional to its weight. Resampling focuses the particles on the high-probability regions of the state space, by probabilistically discarding particles with low weight and duplicating particles with high weight.

One can reduce the variance of PF by using evidence $\mathbf{o}_t$ in the first step of the update cycle by sampling the next state $\mathbf{x}_t$ from the conditional probability distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{x}_{t-1}, \mathbf{o}_t)$. As we show in Section 5, this probability distribution can be computed as $P(x_t \,|\, x_{t-1}, o_t) = P(x_t \,|\, x_{t-1}) \cdot P(o_t \,|\, x_t) / P(o_t \,|\, x_{t-1})$. Precomputing $P(o_t \,|\, x_{t-1})$ is possible if the HMM transition probabilities and observation probabilities are given explicitly. By reducing the variance, the resampling frequency (which is a considerable performance bottleneck) can also be reduced.

PF approximates $\mathbf{P}(\mathbf{x}_t \mid \mathbf{o}_{1:t})$, the probability of state $\mathbf{x}_t$ after observation sequence $\mathbf{o}_{1:t}$, by $\frac{1}{N_p} \sum_{i=1}^{N(\mathbf{x}_t | \mathbf{o}_{1:t})} w_i$, where $N(\mathbf{x}_t \mid \mathbf{o}_{1:t})$ is the number of particles in state $\mathbf{x}_t$ after processing observations $\mathbf{o}_1, \ldots, \mathbf{o}_t$ and w_i are the weights of the individual particles which are in state x_t.

Runtime Verification with State Estimation (RVSE) [10] is an algorithm for runtime verification in the presence of observation gaps. In RVSE, a Hidden Markov Model of the monitored program is constructed, monitored event sequences are treated as observation sequences of the HMM, and an extension of the optimal *forward algorithm* for HMM state estimation [7] is used to estimate the state of the HMM and the monitor DFA. We extended the forward algorithm to handle observation gaps, by using the HMM to estimate the unobserved states and events. The time complexity of the RVSE algorithm for a single observation is $O(N_h^2 \cdot N_d)$ for a non-gap event and $O(N_h^2 \cdot N_d^2)$ for a gap event, where N_h and N_d are the numbers of states of the HMM and the DFA, respectively. The *approximately precomputed RVSE* (AP-RVSE) algorithm, described briefly in Section 1, significantly reduces the runtime overhead of RVSE by precomputing and storing the results of the matrix calculations performed by RVSE [1].

3 Problem Statement

The problem statement is based closely on [10]. A problem instance is defined by an HMM H modeling the monitored system, an observation sequence $o_{1:T}$, and a temporal property ϕ over sequences of actions of the monitored system.

The observation sequence contains events that are occurrences of actions performed by the monitored system. In addition, it may contain the symbol $gap(L)$ denoting a possible gap whose length is drawn from a length distribution L, a probability distribution over the natural numbers: $L(\ell)$ is the probability that the gap has length ℓ. In the rest of this paper, we consider a simpler definition of gaps, with gap symbols of form $gap(\ell)$, where the length ℓ of each gap is encoded in the trace.

The HMM H models the monitored system. The HMM need not be an exact model of the system; it simply embodies the available information about the system's behavior. It can be learned automatically from complete traces using standard learning algorithms [7]. Let S_H denote the set of states of the HMM, i.e., the set of possible values of its state variable.

The property ϕ is represented by a DFA $M = \langle S_M, m_{init}, A, \delta, F\rangle$, consisting of a set S_M of states, an initial state m_{init}, an alphabet A, a transition relation δ, and a set F of final (also called "accepting") states. The alphabet A is a subset of the observable actions of the HMM; actions not in A leave the DFA's state unchanged. [1]

The goal is to compute $P(\phi \mid o_{1:T})$, that is, the probability that the system's behavior satisfies ϕ, given observation sequence $o_{1:T}$. This probability is computed from the probability distribution on composite states, where a *composite state* (x, s) is a pair containing an HMM state x and a DFA state s. Specifically,

$$P(\phi \mid o_{1:t}) = \textstyle\sum_{x_t \in S_H, s_t \in F} P(x_t, s_t \mid o_{1:t}) / \sum_{x_t \in S_H, s_t \in S_M} P(x_t, s_t \mid o_{1:t})$$

where $P(x_t, s_t \mid o_{1:t})$ is the probability that the HMM is in state x_t and the DFA is in state s_t after observation sequence $o_{1:t}$.

4 System Model

The composition of the HMM H modeling the monitored system and the monitor DFA M defines a DBN D representing the entire system. This DBN is illustrated in Figure 1. It shows dependencies among the state variables and observation variables during the t'th time step as well as the dependencies of the state variables X_t and S_t from the

[1] In Section 5 we use a different, HMM-like notation for the DFA.

previous states X_{t-1} and S_{t-1}, respectively. These relationships hold for consecutive observations, without gaps.

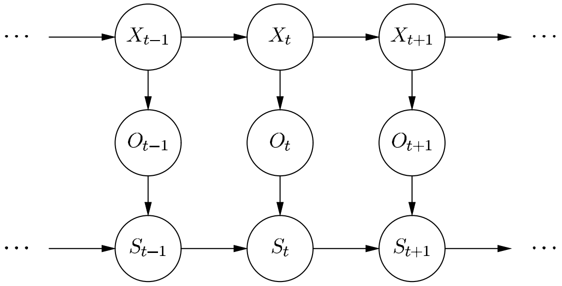


Fig. 1. DBN D composed from the HMM H and the monitor DFA M. X_t and O_t denote the state and observation variables of H at time t, respectively, and S_t denotes the state variables of M, at time t. Note that O_t is also M's input.

4.1 Peek Operations

When a gap occurs, the missing observations cause uncertainty in the state of the DFA. Our algorithm performs a peek operation, which is a lightweight procedure that inspects a part of the monitored system's state immediately after a gap, and can be regarded as an event which is used to reduce the uncertainty in the state of the DFA. Which part of the program state is considered during a peek operation depends on the particular problem and is built into the definition of the procedure that implements the peek operation.

Specifically, peek events are useful for applications in which certain DFA states are known to be inconsistent with certain program states. In such situations, the probabilities associated with composite states containing DFA states which are inconsistent with the partial program state provided by the peek operation can be zeroed, after which the probabilities associated with other composite states are renormalized so that they sum to 1. The additional dependencies between the variables are represented by the DBN in Figure 2.

Because our algorithm uses peek events to reduce uncertainty in the DFA state, we characterize the result of a peek operation q_t by a probability distribution $P(Q_t \mid S_t)$, which is the probability that a peek operation returns Q_t given that the DFA is in state S_t. Using Bayes' rule, after a peek operation that returns q_t after a gap, the probability that the DFA is in state s_t is $P(s_t \mid q_t) = \alpha P(q_t \mid s_t)P(s_t)$, where α is a constant factor used for normalization, and $P(s_t)$ is the probability that the DFA is in state s_t after processing the gap and before processing the peek event.

We do not directly use peek events to reduce uncertainty in the state of the HMM, because generally we do not know a correspondence between concrete program states (provided by peek events) and states of the HMM. This is because the HMM is typically an abstract model learned automatically from traces. However, if such a correspondence is known, then peek events can be used to reduce uncertainty in the state of the HMM, in the same way they are used to reduce uncertainty in the state of the DFA.

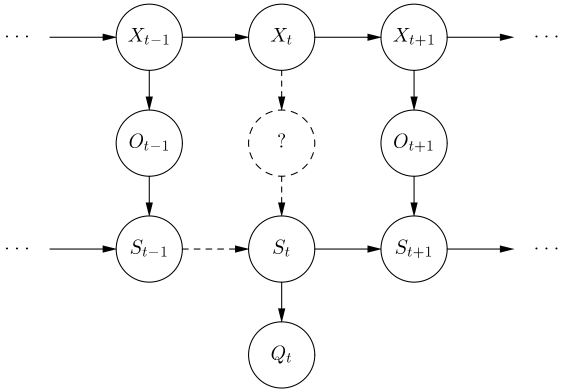


Fig. 2. DBN D composed from the HMM H and the monitor DFA M, when observation o_t is missing due to a gap, and peek event q_t provides information about possible states of the DFA at time t

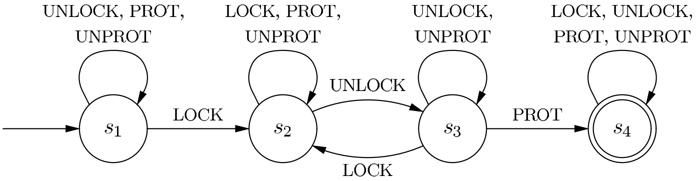


Fig. 3. DFA used to detect violations of a locking discipline

4.2 Running Example

Consider a monitor for a locking discipline for a structure type S in a program. The structure type S contains a lock field (i.e., a field that refers to a lock), protected fields, and unprotected fields. There is a monitor instance for each combination of a thread and a structure of type S. The monitor checks that, when the thread accesses a protected field of the instance of S, the thread holds the lock associated with the instance. The DFA is shown in Figure 3; the parameterization by a thread and a structure is implicit.

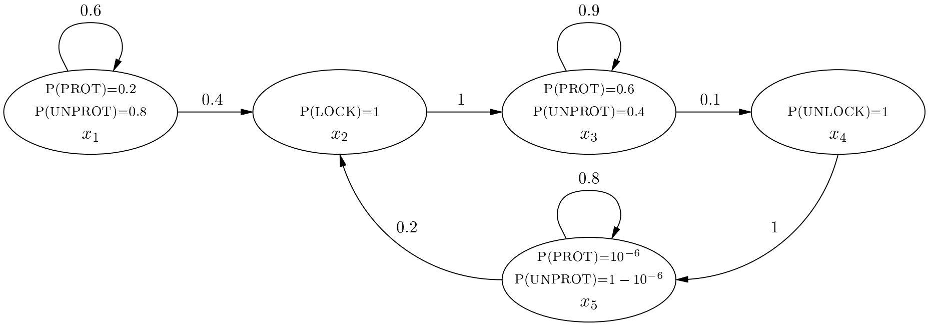


Fig. 4. Graphical representation of the transition and observation probability distributions of an HMM model of a system that usually follows the locking discipline

The alphabet contains four types of events: LOCK, UNLOCK, PROT (representing an access to a protected field) and UNPROT (representing an access to an unprotected field). The states of the monitor have the following interpretation: s_1 – initial state, s_2 – lock is held, s_3 – lock is not held, s_4 – error state (i.e., violation of locking discipline has been detected).

In general, after a gap, the joint probability distribution $P(X_t, S_t)$ may contain non-zero probabilities for all composite states, reflecting uncertainty in the current state of the DFA. The monitor can, however, quickly peek at the state of the lock to check whether it is held by the associated thread. If so, the DFA can only be in states s_2 or s_4, so probabilities of composite states containing s_1 or s_3 can be set to zero. If not, the DFA can only be in states s_1, s_3, or s_4, so the probabilities of composite states containing s_2 can be set to zero. For example, some sample entries in the probability distribution for peek events are $P(s_2 \mid \text{held}) = 1$ and $P(s_2 \mid \text{notHeld}) = 0$.

Figure 4 shows in a graphical way the transition and observation probability distributions of an HMM model of a system that usually follows this locking discipline but has a small chance of violating it.

5 RVPF Algorithm

This section describes our RuntimeVerificationParticleFiltering (RVPF) algorithm, which performs approximate state estimation based on particle filtering. Like the original RVSE algorithm [10], RVPF estimates the probability that the system is in a composite state (x_t, s_t) at time t. Let $(x_t^{(i)}, s_t^{(i)})$ denote the state, also called the "position", of the i'th particle at time t.

5.1 The Precomputation Phase

In Line 1 of the RVPF algorithm, whose pseudo code is given on page 157, the probabilities $P(O_t \mid X_{t-1})$ and $P(X_t \mid X_{t-1}, O_t)$ are precomputed so they can be accessed quickly by the rest of the algorithm. The exact details of the precomputation are shown in algorithm PrecomputeProbabilities.

Algorithm PrecomputeProbabilities

```
Input:   System Model HMM $H = (X, O, P(X_t \mid X_{t-1}), P(O_t \mid X_t), P(X_0))$
Output:  $P(O_t \mid X_{t-1})$, $P(X_t \mid X_{t-1}, O_t)$

$P(O_t \mid X_{t-1}) = P(X_t \mid X_{t-1}) P(O_t \mid X_t)$
for $i = 1$ to $|\mathrm{dom}(X)|$ do
    for $j = 1$ to $|\mathrm{dom}(X)|$ do
        for $k = 1$ to $|\mathrm{dom}(O)|$ do
            $P(X_t = x_i \mid X_{t-1} = x_j, O_t = o_k) =$
                $P(X_t = x_i \mid X_{t-1} = x_j) \cdot P(O_t = o_k \mid X_t = x_i) \,/\, P(O_t = o_k \mid X_{t-1} = x_j)$
        end
    end
end
return $[\, P(O_t \mid X_{t-1}),\ P(X_t \mid X_{t-1}, O_t) \,]$
```

```
Algorithm RuntimeVerificationParticleFiltering
Input:  System Model HMM $H = (X, O, P(X_t \mid X_{t-1}), P(O_t \mid X_t), P(X_0))$,
        Monitor DFA $M = (S, Q, P(S_t \mid S_{t-1}), P(Q_t \mid S_t), P(S_0), F)$,
        Program Events $\boldsymbol{o}_{1:T}$, Peek Events $\boldsymbol{q}_{1:T}$, Number of particles $N_p$
Output: Joint probability distribution $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ after seeing $\boldsymbol{o}_{1:T}$ and $\boldsymbol{q}_{1:T}$

$[\,P(O_t \mid X_{t-1}),\ P(X_t \mid X_{t-1}, O_t)\,]$ = PrecomputeProbabilities($H$)
$(\boldsymbol{x}_0, \boldsymbol{s}_0, \boldsymbol{w}_0)$ = InitializeParticleDistribution($P(X_0)$, $P(S_0)$, $N_p$)
for $t = 1$ to $T$ do
    if $o_t \neq gap$ then
        for $i = 1$ to $N_p$ do
            SAMPLE $x_t^{(i)}$ FROM $P(X_t \mid x_{t-1}^{(i)}, o_t)$
            SAMPLE $s_t^{(i)}$ FROM $P(S_t \mid s_{t-1}^{(i)}, o_t)$
            $w_t^{(i)} = w_{t-1}^{(i)} \cdot P(o_t^{(i)} \mid x_{t-1}^{(i)})$
        end
    else
        $\ell$ = LENGTH OF GAP
        $(\boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{w}_{t-1})$ = Resample($\boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{w}_{t-1}$)
        for $i = 1$ to $N_p$ do
            $(x'_0, s'_0, w'_0) = (x_{t-1}^{(i)}, s_{t-1}^{(i)}, w_{t-1}^{(i)})$
            for $k = 1$ to $\ell$ do
                SAMPLE $o'_k$ FROM $P(O'_k \mid x'_{k-1})$
                SAMPLE $x'_k$ FROM $P(X'_k \mid x'_{k-1}, o'_k)$
                SAMPLE $s'_k$ FROM $P(S'_k \mid s'_{k-1}, o'_k)$
                $w'_k = w'_{k-1} \cdot P(o'_k \mid x'_{k-1})$
            end
            $(x_t^{(i)}, s_t^{(i)}, w_t^{(i)}) = (x'_k, s'_k, w'_k)$
        end
        for $i = 1$ to $N_p$ do $w_t^{(i)} = w_t^{(i)} \cdot P(q_t \mid s_t^{(i)})$      /* handling a peek event $q_t$ */
    end
    NORMALIZE WEIGHTS $w_t$
    $m = 0$
    for $i = 1$ to $N_p$ do $m = m + w_i^2$
    if $1/m \ll N_p \vee q_t \neq \emptyset$ then $(\boldsymbol{x}_t, \boldsymbol{s}_t, \boldsymbol{w}_t)$ = Resample($\boldsymbol{x}_t, \boldsymbol{s}_t, \boldsymbol{w}_t$)
end
INITIALIZE MATRIX $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ WITH ZEROS
for $i = 1$ to $N_p$ do $P(x_T^{(i)}, s_T^{(i)} \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T}) = P(x_T^{(i)}, s_T^{(i)} \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T}) + w_T^{(i)}$
return $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$
```

On Line 1 of PrecomputeProbabilities, the matrix $P(O_t \mid X_{t-1})$ is obtained through a straightforward matrix multiplication of $P(X_t \mid X_{t-1})$ and $P(O_t \mid X_t)$. This is followed by the construction of the 3D-array $P(X_t \mid X_{t-1}, O_t)$ in Lines 2–8.

$P(X_t \mid X_{t-1}, O_t)$ can be best thought of as an array of transition probability matrices $P(X_t \mid X_{t-1})$, one for each observation symbol o_t. This layout makes it possible for the RVPF algorithm to choose the appropriate transition probability distribution depending on the observation symbol generated by the HMM.

5.2 Initial Particle Distribution

The function INITIALIZEPARTICLEDISTRIBUTION, which is invoked on Line 2 of the RVPF algorithm, distributes N_p particles in the state space based on the initial probability distributions $P(X_0)$ and $P(S_0)$ of the HMM and DFA, respectively.

In the code for this function, variable $D_{i,j}$ holds the number of particles in HMM state x_i and DFA state s_j. The rationale for using $\lceil N_p \cdot P(x_0^{(i)}) \cdot P(s_0^{(j)}) \rceil$ on Line 3 is to guarantee that every state with a non-zero initial probability will contain at least one particle. The code in Lines 1–5 is guaranteed to generate at least N_p particles. If the number of generated particles exceeds N_p, the number is reduced in Lines 6–9 by removing individual particles from the richest states.

Algorithm INITIALIZEPARTICLEDISTRIBUTION

Input: Initial probability distributions $P(X_0)$ and $P(S_0)$ of the HMM and DFA, respectively, Number of particles N_p

Output: Initial positions $\boldsymbol{x}_0, \boldsymbol{s}_0$ and weights $\boldsymbol{w}_0$ of particles

```
1   for $i = 1$ to $|\operatorname{dom}(X_0)|$ do
2       for $j = 1$ to $|\operatorname{dom}(S_0)|$ do
3           $D_{i,j} = \lceil N_p \cdot P(x_0^{(i)}) \cdot P(s_0^{(j)}) \rceil$
4       end
5   end
6   while $\sum_{i=1}^{|\operatorname{dom}(X_0)|} \sum_{j=1}^{|\operatorname{dom}(S_0)|} D_{i,j} > N_p$ do
7       FIND $a, b$ FOR WHICH $D_{a,b} = \max(D)$
8       $D_{a,b} = D_{a,b} - 1$
9   end
10  $n = 1$
11  for $i = 1$ to $|\operatorname{dom}(X_0)|$ do
12      for $j = 1$ to $|\operatorname{dom}(S_0)|$ do
13          for $k = 1$ to $D_{i,j}$ do
14              $(x_0^{(n)}, s_0^{(n)}, w_0^{(n)}) = (x_i, s_j, 1/N_p)$
15              $n = n + 1$
16          end
17      end
18  end
19  return $(\boldsymbol{x}_0, \boldsymbol{s}_0, \boldsymbol{w}_0)$
```

5.3 Deriving the Optimal Importance Density Function

The simplest form of the SIR particle filter, known as the *bootstrap filter*, uses the transition prior $P(X_t \mid x_{t-1})$ as the importance density function, i.e., the probability distribution from which new particle positions are drawn. Subsequent weight calculations are performed based on the observation probabilities $P(o_t \mid x_t)$, and the particles are then moved to the interesting regions of the state space through resampling. This approach, however, gives poor results in our setting.

The probability distributions of our learned HMMs often have large transition probabilities of x_{t-1} associated with small observation probabilities of x_t, and small transition probabilities of x_{t-1} associated with large observation probabilities of x_t. As a consequence, if the observation o_t corresponds to a low probability transition in x_{t-1}, drawing particles from $P(X_t \,|\, x_{t-1})$ moves all particles in the "wrong" direction (i.e., contrary to the information provided by the observation), and resampling will have a hard time to move them back to the interesting states.

The solution is to draw new particle positions from an importance density function that takes the observation o_t into account. It has been shown in [3] that the target distribution $P(X_t \,|\, x_{t-1}, o_t)$ minimizes the variance of importance weights $\boldsymbol{w}_t$ conditioned on $\boldsymbol{x}_{1:t-1}$ and $\boldsymbol{o}_{1:t}$. In practice, it is often difficult to sample from $P(X_t \,|\, x_{t-1}, o_t)$. Fortunately, in our case, it is possible to obtain $P(X_t \,|\, X_{t-1}, o_t)$ in closed form [4], which leads to an optimal filter.

5.4 The Forward Step

The loop in Lines 3–29 of the RVPF algorithm estimates the state of the system after each observation. Lines 5–9 handle the regular case in which an observation o_t is available. Lines 11–23 handle gaps.

Handling Program Events. If an observation o_t is available, each particle currently in (x_{t-1}, s_{t-1}) is moved first to (x_t, s_{t-1}) by sampling from $P(X_t \,|\, x_{t-1}, o_t)$ in Line 6. In next step, the particle is moved to (x_t, s_t) by sampling from $P(S_t \,|\, s_{t-1}, o_t)$. Note that $P(S_t \,|\, s_{t-1}, o_t)$ is a conditional probability table which corresponds to the DFA transition function δ. Therefore, the sampling step in Line 7 is guaranteed to return $s_t = \delta(s_{t-1}, o_t)$. Subsequently, in Line 8, the importance weight of each particle is updated by multiplying its current weight with the value from the precomputed matrix $P(O_t \,|\, X_{t-1})$, where $O_t = o_t$ and $X_{t-1} = x_t$. If the number of particles with significant weights becomes too low (which is estimated in Lines 26–28), the particle positions are resampled in Line 28, based on the weight distribution $\boldsymbol{w}$. This concentrates the particles in the more probable regions of the state space.

Handling Gaps. Upon encountering a gap of length ℓ in the trace, the RVPF algorithm moves each particle, from current state (x_{t-1}, s_{t-1}), ℓ steps forward to state $(x_{t+\ell-1}, s_{t+\ell-1})$, following the probability distributions in the HMM. A single step consists of: sampling an observation o_t from $P(O_t \,|\, x_{t-1})$ in Line 16; sampling next HMM state x_t from $P(X_t \,|\, x_{t-1}, o_t)$ in Line 17; sampling next DFA state s_t from $P(S_t \,|\, s_{t-1}, o_t)$ in Line 18 (which, again, corresponds to advancing the DFA using $s_t = \delta(s_{t-1}, o_t)$); and updating the particle weight on Line 19 using the same equation as on Line 8.

Handling Peek Events. Peek events help correct the movement errors introduced by using the HMM model during gaps. After each gap, a peek operation inspects a variable or a set of variables in the program state and returns an observation q_t. On Line 23, each particle i is weighted by $P(q_t \,|\, s_t^{(i)})$, the probability DFA state $s_t^{(i)}$ assigns evidence q_t. Particles with impossible DFA states are assigned a weight of zero, and particles with possible DFA states are assigned a weight of 1. The resampling in Line 28 redistributes all particles across the possible DFA states.

5.5 Resampling

Resampling plays a crucial role in maintaining the diversity among particles. Resampling relies on drawing particles from their corresponding weight distribution. To do so, Lines 1–5 of algorithm RESAMPLE compute a distribution containing prefix sums of particle weights: C_i is the sum of weights $w_1, w_2, ..., w_i$. In each iteration of the loop in Lines 6–9, new particle positions are drawn by sampling from C.

Algorithm RESAMPLE

Input: Particle positions and weights $(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{w})$
Output: Particle positions and weights after resampling $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}}, \tilde{\boldsymbol{w}})$

```
1  $C_0 = 0$
2  $N_p = \dim \boldsymbol{x}$
3  **for** $i = 1$ **to** $N_p$ **do**
4      $C_i = C_{i-1} + w^{(i)}$
5  **end**
6  **for** $i = 1$ **to** $N_p$ **do**
7      SAMPLE PARTICLE INDEX $k$ FROM $C$
8      $(\tilde{x}^{(i)}, \tilde{s}^{(i)}, \tilde{w}^{(i)}) = (x^{(k)}, s^{(k)}, 1/N_p)$
9  **end**
10 **return** $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}}, \tilde{\boldsymbol{w}})$
```

5.6 Calculating the Probability Distribution

After T time steps, the probability distribution $P(X_T, S_T \mid \boldsymbol{o}_{1:T}, \boldsymbol{q}_{1:T})$ is estimated on Lines 30–31 by summing the weights of particles in each state.

6 Evaluation

In this section, we evaluate the performance of the RVPF algorithm by comparing it to the RVSE and AP-RVSE algorithms. We conducted multiple experiments focusing on three important factors: *execution time*, *memory usage*, and state-estimation *accuracy*. All our experiments were carried out under Fedora Linux 17 on a computer with 4GB of RAM and a quad-core Intel® Core$^{\text{TM}}$ i5-2500 CPU running at 3.3GHz. For these experiments, we adapted the existing implementation of AP-RVSE and created new implementations of RVSE and RVPF, reusing relevant parts from the code for AP-RVSE. All three programs are written in C.

The *micro-benchmark* is a multi-threaded application developed for the purpose of experimental evaluation of the AP-RVSE algorithm [1]. It consists of five threads concurrently accessing 100 objects. Each thread can perform four possible operations on any of the objects: LOCK, UNLOCK, PROT, and UNPROT. Threads choose which of these operations to execute according to the HMM in Figure 4. The DFA in Figure 3 is used to check for proper access to the protected fields of each object.

To offer a fair comparison, all three algorithms were evaluated on the same set of micro-benchmark-generated traces (event sequences), which contain gaps of varying frequency (measured as the percentage of trace elements that are gap symbols) and length.

Moreover, each of the algorithms performed state estimation using the same HMM that was used to generate the traces in the first place (i.e., the HMM of Figure 4). In this way, we eliminated the imprecision that might have occurred as a result of re-learning an HMM from the traces, thereby giving each algorithm the opportunity to perform state estimation as accurately as it can. The fact that we used a predefined HMM to drive the micro-benchmark from which we collected the traces allowed us to also log the current state of the HMM and the DFA along with each emitted observation symbol.

6.1 Execution Time and Memory Usage

We conducted experiments aimed at understanding how the number of particles affects the execution time and memory usage of the RVPF algorithm, and how RVPF compares to RVSE and AP-RVSE in terms of these performance measures. For all our experiments, we used both the original HMM (Figure 4) and an additional 10-state HMM learned from the micro-benchmark-generated traces using the Baum-Welch algorithm.

One of our first tests was to measure the execution time of AP-RVSE for different gap lengths (GLs) and gap frequencies (GFs). As expected, in all cases, AP-RVSE always had nearly the same execution time, and was faster than RVPF and RVSE. This was true even when there were no gaps and only two particles were used by RVPF.

The speed of AP-RVSE, however, comes at a price of high memory usage, which is several orders of magnitude higher than that of RVSE and RVPF.

Table 1. Memory consumption in bytes of RVSE, AP-RVPF (with accuracy parameter $\epsilon = 0.1$) and RVPF (for 150 and 350 particles)

Algorithm	RVSE	AP-RVSE	RVPF ($N_p = 150$)	RVPF ($N_p = 350$)
Original 5-state HMM	480	361,240	3,380	6,580
Learned 10-state HMM	960	764,560	5,960	9,160

As Table 1 shows, RVSE uses a relatively small amount of memory, only for storing the HMM and DFA matrices. For RVPF, the amount of required memory is a linear function of the number of particles N_p and was measured to be $16 \cdot N_p + 980$ bytes in case of the original 5-state HMM and $16 \cdot N_p + 3560$ bytes in case of the learned 10-state HMM. In case of the original HMM, with 150 particles RVPF requires around 100 times less memory than AP-RVSE. For the learned HMM and 350 particles, the memory consumption of RVPF is still around 80 times lower than that of AP-RVSE.

We also compared the speed of RVPF to the speed of RVSE. Instead of reporting absolute execution times, we used the execution time of RVSE as the basis for the comparison and determined the number of particles for which RVPF runs exactly as fast as RVSE. For varying GFs and GLs 1-3, we first measured the execution time of RVSE. We then measured the execution time of RVPF with an increasing number of particles until we found the number of particles for which RVPF is exactly as fast as RVSE.

Figure 5 shows that the execution time of RVPF relative to RVSE improves monotonically with respect to the GF, leveling off and reaching a maximum value at a GF of

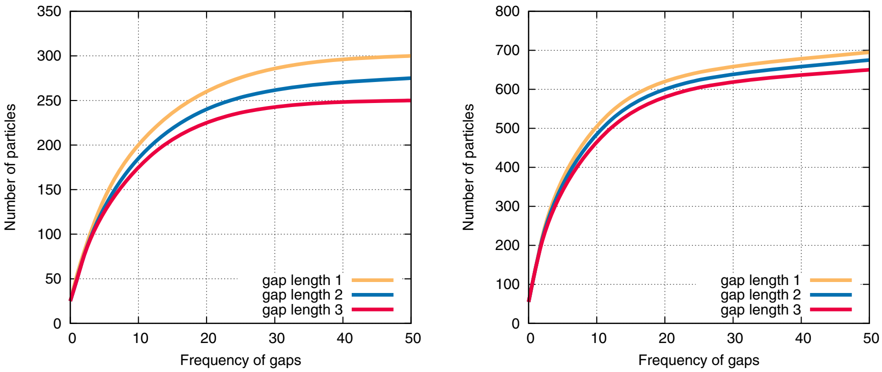


Fig. 5. The number of particles for which RVPF is exactly as fast as RVSE, measured for different GFs and GLs. The figure on the left shows the results for the original 5-state HMM. The figure on the right shows the results for the learned 10-state HMM.

50%. These results also provide a useful guide for choosing the number of particles that maximizes RVPF's accuracy while maintaining its performance advantage over RVSE.

Figure 5 justifies our choice of 150 and 350 particles in Table 1. Namely, with 150 particles RVPF outperforms RVSE for all GLs from 10% to 50% in case of the original 5-state HMM. The same is true for 350 particles when the learned 10-state HMM is used instead.

6.2 Accuracy of State Estimation

Since we recorded the HMM and DFA states in our traces, we can use these values to determine the accuracy of each algorithm's state estimation. We consider first the DFA state. Figure 6 contains our results for estimating the probability for DFA state s_2. The gray line in the graphs serves as a reference value, showing exactly when the DFA was in state s_2. These results are for the worst-case scenario in which there is a gap after each observation symbol (GF = 50%).

The number of particles used for RVPF in obtaining these results was determined as follows. To guarantee that RVPF would always be about twice as fast as RVSE, a significant speed-up, we used the results of Figure 5 to choose the number of particles for RVPF to be half of the value for which it matched RVSE's execution time.

Although we performed state estimation for each DFA state, for presentation purposes, we show only the results for the estimation of DFA state s_2. Even though we are usually interested in state s_4 of the DFA, which is the error state, this state has a very low probability of being reached. The estimated probability of s_4 is therefore almost always zero and rises very slowly. Also, state s_4 is a trap-state, meaning that once entered, the DFA will remain in the state forever. These considerations make s_4 less suitable for measuring accuracy of state estimation. In contrast, s_2 is entered and exited frequently and is thus much more suitable for measuring accuracy of state estimation.

The effect of a 50% GF can be seen in Figure 6 as a form of jitter in the graphs for all three algorithms. Each available observation symbol helps the algorithms increase their certainty, whereas each gap introduces uncertainty. The repeated alternation between visible symbols and gap symbols thus causes the estimated probability to oscillate.

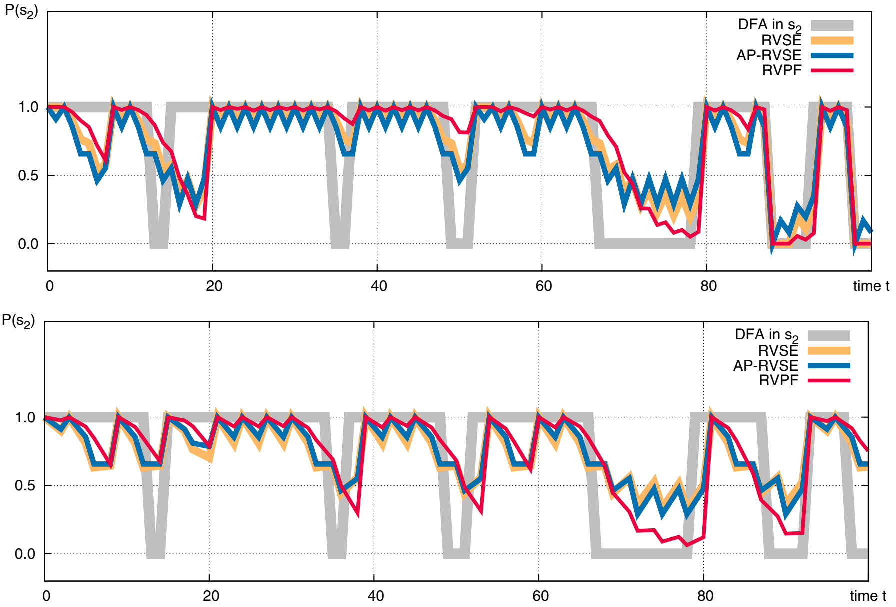


Fig. 6. Measuring accuracy of RVSE, AP-RVSE and RVPF in estimating probability of DFA state s_2 for GF = 50% and GL = 1 (top) and GL = 2 (bottom)

Table 2. Accuracy of RVPF, AP-RVPF and RVSE in estimating probability of DFA state s_2 expressed as L^1-norm of the distance between estimated probability and actual probability at 100 consecutive points in the trace

Algorithm	RVSE	AP-RVSE	RVPF
Gap length 1 (Figure 6 (top))	19.9740	22.5312	17.6149
Gap length 2 (Figure 6 (bottom))	27.1269	24.4361	18.2829
Gap length 2 with peek events (Figure 7)	10.6527	10.2417	8.2252

For each algorithm, we also calculated the L^1-norm of the difference between the estimated probability and the actual probability of DFA state s_2, at 100 consecutive points in the trace. The results are summarized in rows 1 and 2 of Table 2. As the table shows, RVPF gives more accurate results than both RVSE and AP-RVSE in both considered cases (gap length 1 and 2). The reason for this lies in the fact that RVSE and AP-RVSE tend to spread their estimates across all of the states, whereas the limited number of samples drives the estimates of RVPF to the most probable parts of the state space. The RVPF curves in Figure 6 thus show much less jitter and follow the reference curve better than those of RVSE and AP-RVSE. The spreading of estimates across the entire state space in case of RVSE significantly reduces its accuracy as the gap length grows. This can be observed by comparing the results for RVSE and AP-RVSE in the upper two rows of Table 2.

6.3 Estimation Accuracy with Peek Events

To show how peek events help correct estimation errors due to gaps, consider again the results of Figure 6 (bottom), where monitoring is turned off two thirds of the time; i.e., each observation symbol is followed by a gap of length 2. Since, in general, none of the algorithms performs well in this case, we repeated the same test, this time allowing each algorithm to perform a peek operation on the lock after each gap. The noticeable improvements are visible in Figure 7 and quantified in row 3 of Table 2. As expected, peeking results in nearly the same accuracy for RVSE and AP-RVSE, with RVPF being more accurate than both of them for the same reasons given in the previous section (i.e. less jitter and concentration of the estimates in most probable parts of the state space).

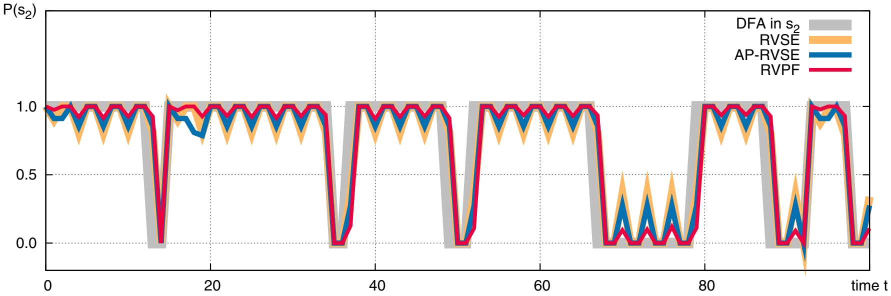


Fig. 7. Estimation of probability of DFA state s_2 for GF = 50% and GL = 2 after correction through peek operations

Peek operations also allow RVPF to correct the estimation of the current state of the HMM. Even though peek events are used only to exclude DFA states from the belief space, the link between a peek observation and the state of the HMM is established through the DBN (Figure 2). This connection allows the peek observation to affect (correct) the estimated probability of the state of the HMM. Figure 8 illustrates the effect of peeking on the estimation of an HMM state by all three algorithms. Recall that we recorded the actual state of the HMM in the traces, depicted in the figures as the gray (reference) curve.

7 Related Work

Particle filtering (PF) has recently been applied to hybrid systems for monitoring and diagnosis purposes, and in particular to estimate the hidden hybrid discrete-continuous state from a set of available measurements [6,2,8,9]. In [6], PF is applied to a class of distributed hybrid systems with autonomous transitions, non-linear system dynamics, and non-Gaussian noise. They demonstrate their approach on a cryogenic propulsion system. In [2], the authors present a PF-based method for discrete-time stochastic hybrid systems, where each particle has two components: a Euclidean component representing the continuous state and a discrete component representing the mode. Their approach combines exact conditional mode probabilities, given the observations, with

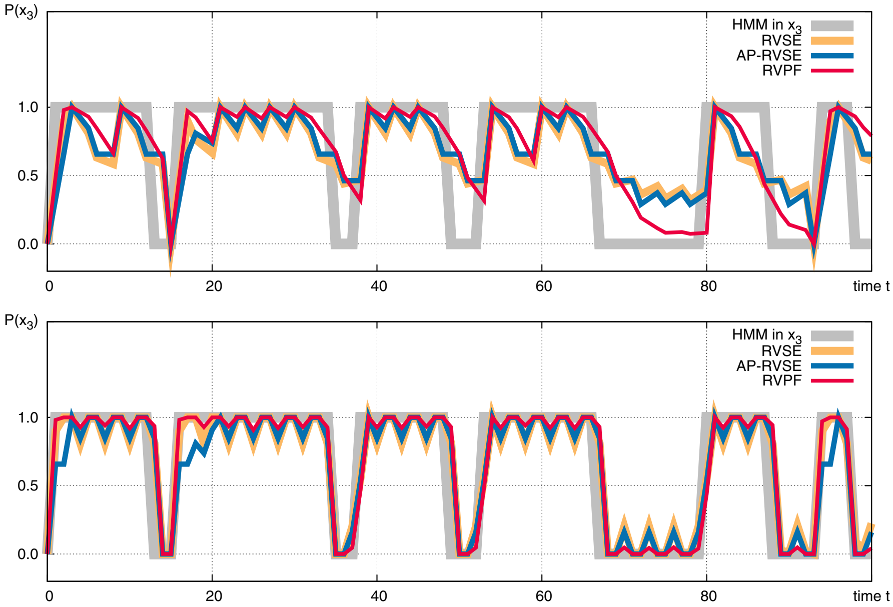


Fig. 8. Estimation of probability of HMM state x_3 with GF = 50% and GL = 1 before and after correction through peek operations

the use of particles to estimate the Euclidean component, showing that this technique works significantly better than standard PF.

Sistla *et al.* use PF to investigate the effectiveness of algorithms for monitorability and strong monitorability of partially observable stochastic systems [8,9]. Familiarity with PF is assumed and no further details, except for the number of particles used, are provided. This application of PF is closest in nature to RVPF but there are significant differences, as witnessed by the contrasting goals of RVPF. In particular, we seek to show how PF can be a highly effective technique for runtime verification, and give a detailed presentation of the RVPF algorithm and its experimental evaluation. Furthermore, we extend PF to handle gaps and peek events. Our experimental results, which compare the accuracy and overhead of RVPF with those of RVSE [10] and approximate precomputed RVSE [1], confirm RVPF's versatility.

The problem we consider—estimating the probability that a safety property is violated by a program execution when monitoring gaps may be present—was introduced in [10]. There an optimal but compute-intensive solution based on the forward algorithm was given. In this paper, we additionally consider *peek events*, which required us to reformulate the problem in terms of DBNs. We also show how to enhance our RVPF algorithm with the *sequential importance resampling* (SIR) strategy using an optimal importance density function, to reduce the variance in state estimation in our setting.

8 Conclusions

This paper introduces RVPF, a versatile method for runtime verification with state estimation in which the balance among runtime overhead, memory usage, and prediction accuracy can be controlled by varying the number of particles RVPF uses for state estimation. Our benchmarking results confirm RVPF's flexibility and its superiority over RVSE and AP-RVSE in terms of state-estimation accuracy.

Although RVPF cannot match the speed of AP-RVSE, its relatively low memory footprint gives it an advantage in the context of embedded systems, where memory resources are limited. Our results also show that RVPF can be configured to outperform RVSE without significantly impacting the accuracy of state estimation.

As future work, we are developing a version of RVPF where the number of particles used for state estimation can vary at runtime. This would allow for dynamic control of the tradeoff involving estimation accuracy, memory consumption, and speed.

Acknowledgements. We thank Justin Seyster for introducing the concept of peek events and developing the micro-benchmarking facility on which our experimental results are based.

References

1. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 168–182. Springer, Heidelberg (2013)
2. Blom, H.A.P., Bloem, E.: Particle filtering for stochastic hybrid systems. In: Proceedings of 43rd IEEE Conference on Decision and Control, CDC 2004, vol. 3, pp. 3221–3226 (2004)
3. Doucet, A.: Monte Carlo Methods for Bayesian Estimation of Hidden Markov Models. Application to Radiation Signals. Ph.D. Thesis (1997)
4. Doucet, A.: On sequential simulation-based methods for bayesian filtering. Technical Report CUED-F-ENG-TR310, University of Cambridge, Department of Engineering (1998)
5. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. STTT 14(3), 327–347 (2012)
6. Koutsoukos, X.D., Kurien, J., Zhao, F.: Estimation of distributed hybrid systems using particle filtering methods. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 298–313. Springer, Heidelberg (2003)
7. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall (2010)
8. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012)
9. Sistla, A.P., Žefran, M., Feng, Y.: Monitorability of stochastic dynamical systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 720–736. Springer, Heidelberg (2011)
10. Stoller, S., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)

An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs

Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Runtime assertion checking provides a powerful, highly automatizable technique to detect violations of specified program properties. However, monitoring of annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.) is not straightforward and requires systematic instrumentation and monitoring of memory-related operations.

This paper describes the runtime memory monitoring library we developed for execution support of E-ACSL, executable specification language for C programs offered by the FRAMA-C platform for analysis of C code. We present the global architecture of our solution as well as various optimizations we realized to make memory monitoring more efficient. Our experiments confirm the benefits of these optimizations and illustrate the bug detection potential of runtime assertion checking with E-ACSL.

Keywords: runtime assertion checking, memory monitoring, executable specification, invalid pointers, memory-related errors, FRAMA-C, E-ACSL.

1 Introduction

Memory related errors, including invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, are very common. For example, the study for IBM MVS software in [1] reports that about 50% of detected software errors were related to pointers and array accesses. This is particularly an issue for a programming language like C that is paradoxically both the most commonly used for development of system software with various critical components, and one of the most poorly equipped with adequate protection mechanisms. The C developer is responsible for correct allocation and deallocation of memory, pointer dereferencing and manipulation (like casts, offsets, etc.), as well as for the validity of indices in array accesses.

Among the most useful techniques for detecting and locating software errors, *runtime assertion checking* is now a widely used programming practice [2]. Turing advocated the use of assertions already in 1949 and wrote that “the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows” [3]. A lot of research works have addressed efficient techniques and tools for runtime assertion checking. Leucker and Schallhart provide a survey on *runtime*

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 167–182, 2013.

verification and conclude that "one of its main technical challenges is the synthesis of efficient monitors from logical specifications" [4]. An efficient memory monitoring for C programs is the purpose of the present work.

In this paper, we present the solution for memory monitoring of C programs we have developed for runtime assertion checking in Frama-C [5], a platform for analysis of C code. It includes an expressive executable specification language e-acsl and a translator, called e-acsl2c in this paper, that automatically translates an e-acsl specification into C code [6, 7]. In order to support memory-related annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.), we need to keep track of relevant memory operations previously executed by the program. Hence, we have developed a monitoring library for recording and retrieving validity and initialization information for the program's memory locations, as well as an automatic instrumentation of source code in e-acsl2c inserting necessary calls to the library during the translation of an e-acsl specification into C.

The proposed solution is designed both for *passive* and *active* monitoring, though this paper discusses only passive monitoring, that is the default one. Passive monitoring only aims at observing and reporting failures, while active monitoring introduces new actions e.g. for recovery from detected erroneous situations. Our solution implements a *non-invasive* source code instrumentation, that is, monitoring routines do not change the observed behavior of the program. In particular, it does not modify the memory layout and size of variables and memory blocks already present in the original program, and may only record additional monitoring data in a separate memory store.

The contributions of this paper include:

- a detailed description of our solution of memory monitoring for runtime assertion checking with Frama-C [5], allowing to automatically generate monitors from assertions and function contracts written in the e-acsl specification language [6];
- an efficient storage of memory related operations based on Patricia tries [8];
- optimized records and queries in the store for faster recording and retrieving information on memory blocks;
- an optimized instrumentation reducing the amount of memory monitoring for memory locations that are irrelevant with respect to the provided assertions;
- experiments illustrating the benefits of these optimizations and the capacity of error detection using e-acsl.

The paper is organized as follows. Sec. 2 presents the context of this work, including Frama-C and e-acsl. Sec. 3 gives a global overview of our solution for memory monitoring, in particular, the instrumentation realized by e-acsl2c and the basic primitives provided by our monitoring library. Optimized data storage and search operations are described respectively in Sec. 4 and 5. Sec. 6 presents the optimization reducing irrelevant memory monitoring. Our initial experiments are described in Sec. 7 and summarized at the end of Sec. 4, 5 and 6. Finally, Sec. 8 and 9 present respectively related work and the conclusion.

E-ACSL keyword	Its semantics	
`\base_addr``(p)`	the base address of the block containing pointer `p`	
`\block_length``(p)`	the size (in bytes) of the block containing pointer `p`	
`\offset``(p)`	the offset (in bytes) of `p` in its block (i.e., w.r.t. **`\base_addr`**`(p)`)	
`\valid_read``(p)`	is true iff reading `*p` is safe	here `p` must be a non-void pointer
`\valid``(p)`	is true iff reading and writing `*p` is safe	
`\initialized``(p)`	is true iff `*p` has been initialized	

Fig. 1. Memory-related E-ACSL constructs currently supported by E-ACSL2C

2 Executable Specifications Require Memory Monitoring

The executable specification language E-ACSL [6, 9] was designed to support runtime assertion checking in Frama-C. Frama-C [5] is a platform dedicated to analysis of C programs that includes various analyzers, such as abstract interpretation based value analysis (Value plug-in), dependency analysis, program slicing, Jessie and WP plug-ins for proof of programs, etc. ACSL [10] is a behavioral specification language shared by different Frama-C analyzers that takes the best of the specification languages of earlier tools Caveat [11] and Caduceus [12], themselves inspired by JML [13].

ACSL is expressive enough to express most functional properties of C programs and has already been used in many projects, including large-scale industrial ones [5]. It is based on a typed first-order logic in which terms may contain *pure* (*i.e.* side-effect free) C expressions and special keywords. An Eiffel-like contract [14] may be associated to each function in order to specify its pre- and postconditions. The contract can be split into several named guarded behaviors. Contracts may also be associated to statements, as well as assertions, loop invariants and loop variants. ACSL annotations also include definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

Designed as a large subset of ACSL, E-ACSL preserves ACSL semantics. Moreover, the E-ACSL language is *executable*: its annotations can be translated into C monitors by E-ACSL2C and executed at runtime. This makes it suitable for runtime assertion checking. Fig. 1 presents some memory-related E-ACSL annotations. We use the term *(memory) block* for any (statically, dynamically or automatically) allocated object. A block is characterized by its size and its *base address,* that is, the address of its first byte.

Fig. 2 shows a simple C function `findchr` with an ACSL contract (that is also an E-ACSL contract) enclosed into `@`-comments. Given a character `c` and a pointer `s` to an array of `n` characters, `findchr` returns a pointer to an occurrence of `c` in the array, and NULL otherwise. It is very similar to the C standard `memchr` function (basically, our contract does not require to find the *first* occurrence of `c`). The contract contains two behaviors (lines 2–6, 7–9) with a common precondition (line 1). The precondition states that `s` must refer to a valid readable location with at least `n` characters to the right of `s`. The first behavior `found` is defined by the **`assumes`** clause line 3. Whenever the **`assumes`** condition is satisfied, the behavior's postconditions (lines 4–6) must be ensured. They state that the returned

```
 1 /*@ requires \valid_read(s) && \offset(s)+n <= \block_length(s);
 2   @ behavior found:
 3   @   assumes \exists int i; 0 <= i < n && s[i] == c;
 4   @   ensures \base_addr(s) == \base_addr(\result);
 5   @   ensures \offset(s) <= \offset(\result) < \offset(s)+n;
 6   @   ensures * \result == c;
 7   @ behavior not_found:
 8   @   assumes \forall int i; 0 <= i < n ==> s[i] != c;
 9   @   ensures \result == \null;
10   @*/
11 char * findchr(char *s, char c, unsigned int n) {
12   unsigned int i;
13   for(i = 0; i < n; i++)
14     if(s[i] == c)
15       return s+i;  // found, returns the pointer
16   return (void*)0; // not found, returns NULL
17 }
```

Fig. 2. Function `findchr` specified with an E-ACSL contract

value (keyword `\result`) must refer to the same block as `s` (line 4), to one of the `n` characters starting from `*s` (line 5), and the referred character must be equal to `c` (line 6). Similarly, the second behavior states that the null pointer must be returned (line 9) whenever `c` in not present in the array (line 8).

Translation into C of basic E-ACSL features (including overflow-free arithmetic operations for integers, behaviors, quantifications over finite sets, some special keywords, values at the Pre, Post or any labeled state, etc.) was described in [6]. However, runtime assertion checking of E-ACSL specifications involving memory-related constructs of Fig. 1 is particularly complex. Languages with pointers, such as C or C++, do not allow the developer to easily check for pointer validity. The developer is supposed to know when a pointer is valid or not. For example, even when the size of an input array `a` is provided in a function signature `int f(int a[10])`, it is ignored according to the ISO C 99 norm [15, Sec. 6.7.5.3.7]. In other words, this declaration is equivalent to `int f(int *a)`, so the array size is lost. At runtime, `sizeof(a)` inside `f` returns the size of a pointer, and nothing guarantees that `a` really refers to an array with 10 elements. Runtime checking of memory-related E-ACSL annotations can be realized using systematic monitoring of memory operations as shown in the next section.

3 Memory Monitoring for E-ACSL: An Overview

Runtime assertion checking of E-ACSL specifications is based on a non-invasive source code instrumentation by E-ACSL2C. In order to evaluate memory-related E-ACSL annotations (Fig. 1), we record information on validity and initialization of memory locations during program execution in a dedicated data store, that we call below *the store*. We have developed a memory monitoring library that provides primitives for both evaluating memory-related E-ACSL annotations (by making queries to the store) and recording in the store all necessary data on allocation, deallocation and initialization of memory blocks. Thus E-ACSL2C inserts calls to library primitives for two purposes:

- to translate into C and evaluate memory-related E-ACSL annotations; and
- to record memory-related program operations in the store.

The following subsections present these two aspects in detail.

3.1 Translation and Evaluation of Memory-Related Annotations

When a specified property is violated at runtime, the instrumented code generated by E-ACSL2C calls a special function, that we denote here `e_ascl_fail`, whose default version reports the assertion failure and exits the execution.[1]

The instrumentation is different for an internal annotation in a function and for a function contract. An annotation inside a function `f` is directly translated by E-ACSL2C into C code that checks the annotation condition inside `f` (this case will be illustrated on Fig. 7). For a function `f` with a function contract, E-ACSL2C adds a new C function `__e_acsl_f` with the same signature as `f`, and replaces all initial calls to `f` by calls to `__e_acsl_f`. Basically, `__e_acsl_f` contains three parts: checking the precondition of `f`, a call to `f` and checking the postcondition of `f`. Thus a contract of `f` is systematically checked by `__e_acsl_f` in the instrumented code whenever `f` is called in the original code, even if the code of `f` is not provided during the instrumentation step.

The library provides primitives for the most frequently used memory-related E-ACSL annotations shown in Fig. 1. They have the same role and similar names (with "__" as prefix instead of "\"). For the last three of them, library functions expect an additional second argument indicating the size (in bytes) of the memory location `*p`.

For example, Fig. 3 presents (a simplified version of) function `__e_acsl_findchr` automatically generated by E-ACSL2C for the function `findchr` of Fig. 2. Lines 4–5 of Fig. 3 check the precondition (and report any violation). Lines 6–9 compute if the first behavior's **`assumes`** clause is satisfied, i.e. if this behavior is applicable for the current call of `findchr`. Since execution of an E-ACSL annotation must not introduce any risk of runtime error, an additional check of validity of reading `s[i]` is automatically added at line 7 before an access to `s[i]`. Similarly, lines 10–13 compute if the second behavior is activated by the current call of `findchr`. Then `findchr` is called line 14. Next, lines 15–22 check that if the first behavior's assumption is true, its postconditions are satisfied as well. Again, to avoid any risk of a runtime error inserted by E-ACSL2C, an additional validity check is added at line 20 before an access to `*__res`. Similarly, the second behavior is checked at lines 23–25.

3.2 Recording Validity and Initialization Data in the Store

In order to be able to provide requested information on memory locations, the code instrumented by E-ACSL2C records in the store for each block the *block*

[1] Actual instrumentation allows the user to customize this function by defining its own action according to several parameters [7].

```
char * __e_acsl_findchr(char *s, char c, unsigned int n) {
  char * __res; unsigned int i;
  int __e_acsl_exists = 0; int __e_acsl_forall = 1;
  if(!( __valid_read(s,1) && __offset(s)+n <= __block_length(s) ))
    e_acsl_fail("findchr","Pre","line␣3");
  for( i=0; i<n && __e_acsl_exists == 0 ; i++ ) {
    if(! __valid_read(s+i,1)) e_acsl_fail("findchr","mem_access:s[i]","line␣5");
    if( s[i] == c ) __e_acsl_exists = 1;
  }
  for( i=0; i<n && __e_acsl_forall == 1 ; i++ ) {
    if(! __valid_read(s+i,1)) e_acsl_fail("findchr","mem_access:s[i]","line␣10");
    if(!( s[i] != c )) __e_acsl_forall = 0;
  }
  __res = findchr(s, c, n);
  if( __e_acsl_exists ){
    if(!( __base_addr(s) == __base_addr(__res) ))
      e_acsl_fail("findchr","Post","line␣6");
    if(!( __offset(s) <= __offset(__res) && __offset(__res) < __offset(s)+n ))
      e_acsl_fail("findchr","Post","line␣7");
    if(! __valid_read(__res,1)) e_acsl_fail("Post","mem_access:*__res","line␣8");
    if(!( *__res == c )) e_acsl_fail("Post","findchr","line␣8");
  }
  if( __e_acsl_forall )
    if(!( __res == (void *)0 )) e_acsl_fail("Post","findchr","line␣11");
  return __res;
}
```

Fig. 3. Simplified version of function `__e_acsl_findchr` automatically generated by E-ACSL2C for runtime checking of the contract for the function `findchr` of Fig. 2

Function	Its meaning
`__store_block(p,len)`	records a block of size `len` and base address `p` in the store
`__delete_block(p)`	removes existing block with base address `p` from the store
`__malloc(len)`	allocates a block of size `len` and records it in the store
`__free(p)`	deallocates the block with base `p` and removes it from the store
`__initialize(p,len)`	marks `len` bytes starting from pointer `p` as initialized
`__full_init(p)`	marks the whole block with base address `p` as initialized

Fig. 4. Basic recording primitives provided by the memory monitoring library

metadata including its base address, size (in bytes), validity status (whether reading or writing the block is safe) and the initialization status for each byte of the block. Our memory monitoring library has been designed to be compatible with various implementations of the underlying store. This section presents the instrumentation scheme for recording operations using high-level library primitives, while the store implementation and optimizations are discussed later in Sec. 4, 5 and 6.

Fig. 4 presents some recording primitives provided by the library. They allow to register a new block in the store, to mark some particular bytes (or the whole block) as initialized and to remove the block from the store when it is not valid anymore. For convenience, instrumented versions of basic dynamic allocation functions (`malloc`, `calloc`, `realloc`, `free`) are provided as well. They directly add/remove from the store the allocated/deallocated block (and, in case of `realloc`, transfer recorded initialization information for the old block to the new one).

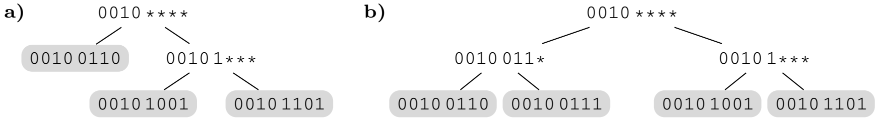


Fig. 5. Example of a Patricia trie **a)** before, and **b)** after inserting 0010 0111

Thanks to these primitives, the (unoptimized) instrumentation for recording in the store is mostly straightforward. To monitor the block of an argument or a local variable `v` of type `T` in function `f`, E-ACSL2C adds the calls `__store_block(&v,sizeof(T))` in the beginning, and `__delete_block(&v)` at the end of the scope of `v`. For a global variable, these calls are inserted in the beginning and at the end of the function `main`. In addition to them, for global variables (initialized by default to 0 in C) and function arguments (initialized by a function call), the `__store_block(&v,sizeof(T))` is followed by `__full_init(&v)` to mark the whole block as initialized. To monitor an assignment `v = exp;` to a variable (or a left value) `v`, a call to `__initialize(&v,sizeof(v));` is inserted. Literal strings and initializers are easily handled as well. Finally, dynamic allocation functions are simply replaced by their instrumented counterparts.

4 Optimized Storage for the Memory Monitoring Library

Efficient implementation of the store requires a data structure with a good time and space complexity, since the instrumented code may perform frequent modifications and lookups in the store. It is intuitively clear that the structure has to be sorted: treating E-ACSL constructs may require to access a block metadata directly by its base address as well as to find a block's predecessor or successor. For example, the query `__base_addr(p)` searches the store for the closest to `p` base address less than `p` (and checks the bounds afterwards). Thus, a hash table will not fit. Lists are not efficient enough due to a linear worst-case complexity. Unbalanced binary search trees have a linear worst-case complexity too when inserted base addresses are monotonically increasing, and this may be quite common. Finally, the cost of balancing (e.g. in a self-balancing binary search tree) would be amortized if the store modifications (that may lead to rebalancing) were less frequent than simple queries (that take advantage of a balanced structure). For tested examples of code instrumented by E-ACSL2C this is not necessarily true.

Our implementation of the store is based on a *Patricia trie* [8], also known as a *radix tree* or *compact prefix tree*, which is efficient even if the tree is unbalanced. Node keys are base addresses (e.g. 32-bit or 64-bit words) or address prefixes. Any leaf contains a block metadata with the block base address. Routing from the root to a block metadata is ensured by internal nodes, each of them contains the greatest common prefix of base addresses stored in its successors.

For instance, Fig. 5a shows a Patricia trie, for simplicity, over 8-bit addresses. It contains three blocks in its leaves (only block base addresses are shown here), and greatest common prefixes in internal nodes. A "*" denotes meaningless bits following the greatest common prefix. Fig. 5b presents another trie obtained from the first one by adding the base address `0010 0111`, that required to create a new internal node `0010 011*`. Conversely, removing `0010 0111` from the trie of Fig. 5b would give that of Fig. 5a.

Theoretical worst-case complexity of a lookup in a Patricia trie in our case is $\mathrm{O}(k)$ where k is the word length (e.g. 32-bit or 64-bit). In practice, since a program is allowed to allocate blocks in a limited memory space, the trie height remains far below this upper bound. In addition, unlike for arbitrary strings, comparisons for words can be very efficiently implemented by bit operations (see also Sec. 5).

Storage of a block metadata takes a few bytes, except for initialization information when the block itself is long. In this case, initialization of each byte is monitored separately (bit-level initialization through bit-fields is not yet supported). To reduce the memory space occupied by the store, recording block initialization information is optimized in two ways. First, since initialization of each byte in a block can be recorded in one bit, block initialization is recorded in a dynamically allocated array, whose size is therefore 8 times less than the block size. Second, when none or all of the bytes of a memory block have been initialized (that are very common cases), initialization array is freed. Instead, an integer field counting initialized bytes is used. Third, the `__full_init` primitive can be used to mark the whole block as initialized, avoiding multiple calls to `__initialize` for particular bytes.

Experiments. To choose which datastructure is most appropriate for implementing the store, we compared the implementation based on Patricia tries to three other implementations of the store: based on linked lists, on unbalanced binary search trees, and on Splay trees used in earlier memory safety related tools (see e.g. [16]). Our implementation appears to be in average more than 2500 times faster than linked lists, 200 times faster than unbalanced binary search trees and 27 times faster than Splay trees. For linked lists and search trees, it confirms the intuition given earlier in this section. The results for Splay trees are comparable to Patricia tries on most examples, maybe 2-3 times faster for some examples, and dramatically ($>$ 500 times) slower for examples (like multiplication of big matrices, matrix inversion etc.) where the program's consecutive accesses to memory are not at all performed to the same memory blocks. The reason is also intuitively clear: since Splay trees move recently accessed elements to the top of the tree, this takes time and brings no benefit when the following queries to the store are not related to the same memory blocks again. For instance, since matrix multiplication requires to take elements in different rows and columns each time, multiplication of big matrices, where all matrix elements do not fit to the same memory block, results in loss of time due to useless restructuring of the Splay tree. On the contrary, on programs with frequent consecutive accesses to the same block metadata in the store, Splay trees appear to be (up to three times in our examples) more efficient.

```
 1 typedef unsigned char byte;
 2 // index          0    1    2    3    4    5    6    7    8
 3 byte  masks[] = {0x00,0x80,0xC0,0xE0,0xF0,0xF8,0xFC,0xFE,0xFF};
 4 int longer [] = {   0,  -1,   3,  -3,   6,  -5,   7,   8,  -8};
 5 int shorter[] = {   0,   0,   1,  -2,   2,  -4,   5,  -6,  -7};
 6 byte gtCommonPrefixMask(byte a, byte b) {
 7   byte nxor = ~(a ^ b);   // a bit = 1 iff this bit is equal in a and b
 8   int i = 4;              // search starts in the middle of the word
 9   while(i > 0)            // if more comparisons needed
10     if (nxor >= masks[i]) i = longer[i]; // first i bits equal,try a longer prefix
11     else i = shorter[i];                 // otherwise, try a shorter prefix
12   return masks[-i];       // if i<=0, masks[-i] is the answer
13 }
```

Fig. 6. Search for greatest common prefix mask, illustrated here for bytes

5 Optimized Records and Queries in the Store

Queries for adding, removing, or searching a given base address A in the store based on a Patricia trie require comparisons of A with existing nodes and computations of the greatest common prefix for two elements (cf Fig.5). For Patricia tries storing addresses (strings of 0's and 1's of fixed length rather than strings of arbitrary length over a wider set of characters), these comparisons may be simplified due to the nature of elements. Let us call by the *greatest common prefix mask* M of A and B the mask containing 1's for the positions of common bits in the greatest common prefix P of A and B. So M starts with n 1's followed by 0's, where n is the number of common bits in P. For example, the greatest common prefix of bytes $A =$0110 0111 and $B =$0111 1111 is $P =$111⋆ ⋆⋆⋆⋆, while the greatest common prefix mask is $M =$1110 0000.

We carefully optimized all prefix computations and comparisons by intensive usage of efficient bit-to-bit operations. Interestingly, one optimization that we realized for computation of the greatest common prefix mask appeared particularly efficient. Fig. 6 illustrates the optimized version, for simplicity, over bytes instead of words. It is based on the classic dichotomic search of the index `i` such that the greatest common prefix mask starts with exactly `i` 1's. In addition to precomputed masks (line 3) and bit operations (line 7), our version uses precomputed indices (lines 4,5) for the next prefix length `i` to try, therefore it avoids the usual `mid=(high+low)/2` computation at each iteration, making frequent calls to the function much faster. A negative value `i<=0` indicates that `-i` is the final greatest common prefix length. The next value of `i` is simply extracted (lines 10,11) of the arrays depending if the next candidate prefix should be tried longer or shorter. For instance, for A and B above, `nxor` equals the byte 11100111, and the function will try `i=4`, then `i=shorter[4]=2`, then `i=longer[2]=3` and finally stop with `i=longer[3]=-3` and return the mask `masks[3]=0xE0` of length 3, that is in binary precisely 1110 0000.

Experiments. We compared our optimized implementation to a non-optimized version of the common prefix mask computation based on the usual comparison of strings commonly used for Patricia tries (with a linear run over the elements from left to right, that we have also optimized by bit operations). On the tested examples, our optimized version illustrated by Fig. 6 makes the execution of the

```
 1 #include<stdlib.h>
 2 int last;
 3 int* new_inversed(int len, int *v) {
 4   int i, *p;
 5   //@ assert \valid(v) && \offset(v)+len*sizeof(int) <= \block_length(v);
 6   p = malloc(sizeof(int)*len);   // allocate a new vector p
 7   for(i=0; i<len; i++)
 8     p[i] = v[len-i-1];           // write inversed vector v into p
 9   return p;
10 }
11 void main() {
12   int v1[3]={1,2,3}, *v2;
13   //@ assert \valid(&v1[2]);
14   last = v1[2];
15   v2 = new_inversed(3, v1);
16   last = v2[2];
17   //@ assert last == 1;
18   free(v2);
19 }
```

Fig. 7. File `vector.c` where the function `new_inversed` allocates and returns a new vector containing the inversed given vector `v` of `len` integers

instrumented code in average 2.7 times faster. This rate goes up to 4.7 times for examples with intensive usage of the memory monitoring library.

6 Optimized Instrumentation Using Static Analysis

The instrumentation presented in Sec. 3 is sound and complete: the code instrumented by E-ACSL2C reports an E-ACSL annotation failure at runtime if and only if this E-ACSL annotation is indeed violated. However it has the major drawback of being hugely verbose and time-consuming: for each variable, each (de)allocation and each assignment, one or even several new statements are generated. It is however sufficient to monitor the memory locations involved in memory-related constructs in the provided E-ACSL annotations.

To solve this drawback, we have designed an interprocedural backward dataflow analysis which computes an over-approximated set σ of memory locations that it is sufficient to monitor in order to preserve soundness and completeness of the instrumentation. Let us explain on the example of Fig. 7 how this analysis works (for lack of space, we do not give its formal presentation here). Without any analysis, we have to monitor every variable of the program and to record when it is allocated, initialized and deallocated by systematically adding calls to the recording primitives of Fig. 4 as explained in Sec. 3.2.

However, this monitoring is only required for memory blocks involved in memory-related E-ACSL constructs. In our example, they are **`\valid`**`(v)`, **`\offset`**`(v)` and **`\block_length`**`(v)` at line 5, and **`\valid`**`(&v1[2])` at line 13. So we need to monitor the formal parameter `v` of function `new_inversed` and the location `&v1[2]`. For the latter, we keep less precise information. Our current analysis is purely syntactical and does not perform any precise semantic aliasing analysis. To be sound, we perform an over-approximation and monitor any information about the whole local array `v1` of function `main`, including `*(v1+i)` for any offset `i`. Basically, from

```
int last;
int* new_inversed(int len, int *v) {
  int i, *p;
  __store_block(&v,sizeof(int*)); __full_init(&v);
  if(!( __valid(v,sizeof(int)) && __offset(v)+len*sizeof(int) <= __block_length(v) ))
    e_acsl_fail("new_inversed","assert","line␣4");
  p = __malloc(sizeof(int)*len);
  for(i=0; i<len; i++)
    p[i] = v[len-i-1];
  __delete_block(&v);
  return p;
}
int main() {
  int v1[3]={1,2,3}, *v2;
  __store_block(v1,3*sizeof(int)); __full_init(v1);
  if(! __valid(v1+2,sizeof(int)) ) e_acsl_fail("main","assert","line␣13");
  last = v1[2];
  v2 = new_inversed(3, v1);
  last = v2[2];
  if(!( last == 1 )) e_acsl_fail("main","assert","line␣17");
  __free(v2);
  __delete_block(v1); __clean();
}
```

Fig. 8. Simplified instrumentation of the file `vector.c` of Fig. 7 with E-ACSL2C

these E-ACSL annotations, the analysis goes backwards in the code in order to find where the monitored variables `v` and `v1` are assigned and where aliases are potentially created.

More precisely, the analysis starts from the end of the program with $\sigma = \emptyset$, and goes backwards up to the beginning, analyzing statements, annotations and called functions in order to collect memory locations to be monitored into σ. For the example of Fig. 7, it collects nothing until the assertion at line 5 in function `new_inversed` called from the line 15 (still in a context with $\sigma = \emptyset$). At this point, it remembers that `v` has to be monitored. Going back to the callsite (line 15), as the formal parameter `v` has to be monitored, the corresponding argument `v1` is also collected into σ. For the assertion of line 13, the analysis computes that `v1` has to be monitored (actually, it is already in σ, so nothing new is discovered). Finally the analysis concludes that `v` and `v1` have to be monitored, leading to the optimized instrumentation of Fig 8. We notice that variables `last`, `len`, `i`, `p` and `v2` are not monitored, unlike in the unoptimized instrumentation.

If `v1` was a pointer referring to another array `v3` (e.g. if the line 12 was `int v3[3]={1,2,3}, *v1=v3, *v2;`), the analysis would deduce from the assignment `v1=v3` that `v3` should be monitored as well.

Experiments. In the tested examples, the optimization based on dataflow analysis reduced the total execution time of instrumented code by 66% in average. It is due to a smaller number of monitored variables (decreasing by 78% in average) and hence a smaller number of records and queries to the store (number of calls to `gtCommonPrefixMask` throughout the execution decreased by 71% in average). The analysis was rather fast: no example has been slowed down by integrating this optimization.

Example	Orig.	Lists	BST	PT−anal.	PT−mask	**PT**	Splay	Valgrind
binarySearch	0.01	0.51	0.62	1.59	0.53	0.53	0.64	0.27
insertionSort	0.12	1.27	1.26	3.86	1.25	1.25	1.30	2.81
matrixMult	0.01	58.48	90.43	9.01	8.57	8.75	398.60	0.48
matrixInv	0.02	21.54	29.94	1.90	1.42	1.53	145.80	0.47
quickSort	0.01	11.15	2.67	0.48	0.36	0.13	0.02	0.27
bubbleSort	0.22	4.64	7.16	32.58	7.26	6.90	7.21	3.36
merge	0.01	101.33	94.80	0.29	0.47	0.14	0.05	0.45
RedBlackTree	0.01	101.69	145.20	0.30	0.39	0.27	19.59	0.51
mergeSort	0.01	>24h	>24h	513.85	25.02	7.63	2.50	0.27

Fig. 9. Detailed execution time (in sec.) for selected examples and techniques

7 Experimental Results

Performances. To evaluate our memory monitoring solution, we performed in total more than 300 executions for more than 30 programs obtained from about 10 examples with different levels of specifications and values of parameters. These initial experiments were conducted on small-size examples because they were mostly manually specified in E-ACSL. We measured the execution time of the original code and of the code instrumented by E-ACSL2C with various options in order to evaluate their performances (with and without optimizations, with four different implementations of the store, etc.). Such indicators as the number of monitored variables, memory allocations, records and queries in a Patricia trie were recorded as well.

Fig. 9 presents some of these examples and indicators in detail. Its columns give execution time of the original program, using store implementation based on lists, binary search trees, three versions of Patricia tries and Splay trees. Patricia tries based implementations were tested respectively without dataflow analysis of Sec. 6, without query optimization of Sec. 5 and with both optimizations. Time of analysis with Valgrind tool [17] is indicated in the last column.

Most experimental results have already been summarized at the end of Sec. 4, 5 and 6, where the average rates were computed for a complete list of examples (some of which are not presented in Fig. 9). We notice in addition that the execution time with Valgrind is not comparable with our solution and may depend on the number of memory-related annotations in the specification.

Error detection capacity. In addition to performance evaluation, we used mutational testing to evaluate the capacity of error detection using runtime assertion checking with Frama-C. We considered five annotated examples (see Fig. 10), generated their *mutants* (by performing a *mutation* in the source code) and applied assertion checking on them. Annotations included preconditions, postconditions, assertions, memory-related constructs etc., and were written in E-ACSL. Mutations included: numerical-arithmetic operator modifications, pointer-arithmetic operator modifications, comparison operator modifications and logic (*land* and *lor*) operator modifications. The PathCrawler test generation tool [18] has been used to produce test cases. Each mutant was

	alarms	mutants	equivalent	killed	% erroneous killed
fibonacci	19	27	2	25	100%
bubbleSort	15	44	2	42	100%
insertionSort	10	39	3	36	100%
binarySearch	7	38	1	37	100%
merge	5	92	5	87	100%

Fig. 10. Error detection for mutants

instrumented by E-ACSL2C and executed on each test case in order to check at runtime if the specification was satisfied. The original programs successfully passed all these checks. As usual, when a violation of an annotation was reported for at least one test case, the mutant was considered to be *killed*. Fig. 10 illustrates the results. Except for equivalent mutants (where the mutation produced by chance an equivalent program), all erroneous mutants were killed.

8 Related Work

Runtime Assertion Checking. The present work is part of an extension of FRAMA-C, an existing toolset for analysis of C code, for supporting runtime assertion checking. It is therefore related to a lot of works on runtime assertion checking [2] and, more generally, runtime verification [4]. More specifically, one of our main objectives is to support and execute annotations in E-ACSL, an expressive executable specification language shared by static and dynamic analysis tools. Hence, our work continues previous contributions to development of expressive specification languages such as Eiffel [14], JML [19] for Java and Alfa [20] for Ada.

Memory Safety. Since the main purpose of this paper is the support of memory-related E-ACSL annotations, our work is also related to previous efforts for ensuring memory safety of C programs at runtime. They include safe dialects of C, specific fail-safe C compilers and memory safety verification tools for C code. In particular, the idea to store object metadata on valid memory blocks in a separate database was previously exploited in [16, 21–24] and appeared well-adapted for most spatial errors (that is, accesses outside the bounds [25]). Advantages of this approach include relative efficiency (propagation of pointer metadata at each pointer assignment is not required) and compatibility (the memory layout of objects is preserved). However, this technique results in significant time overhead due to lookup operations in the database, and is not directly adapted to detect sub-object overflows inside nested objects (e.g. an array of structures) and temporal errors (that is, accesses to an object that has been deallocated [25]). An alternative approach is based on pointer metadata stored inside multi-word *fat pointers* extending the pointer representation with bounds information [26–28]. Hybrid techniques combining ideas of both approaches have been proposed as well [29, 25]. The technique of *shadow pages* [17, 30] makes it possible to immediately find stored validity information for a pointer without providing an easy way to find the base address of the block, block size and pointer offset required by memory-related E-ACSL clauses.

Our global objective is quite different from these efforts. Unlike these advanced works focused on detection of memory safety errors, we aim at supporting runtime checking for memory-related annotations of an expressive specification language E-ACSL. Even if we have already realized several optimizations, performances of our implementation remain below the most advanced proposals addressing memory safety [26–28, 30, 25]. It must be further studied if the efficient solutions they implement are compatible with our objective to support runtime assertion checking for such a rich specification language as E-ACSL. On the other hand, the ambitious objective to perform runtime assertion checking for C code completely specified in E-ACSL and directly compatible with integrated FRAMA-C tools for proof of programs (where manual analysis of proof failures can be even more costly) could justify a higher overhead.

Optimizations. Our proposal to record block metadata using Patricia tries is related to Jones's and Kelly's work [16] that proposed to use Splay trees for this purpose. Splay trees were also used in several recent tools related to memory safety [24, 29]. To the best of our knowledge, Patricia tries have never been used in this context. Static analysis based techniques to reduce memory monitoring have been used in earlier works, for instance, in [27, 28, 24]. Similarly, our dataflow analysis described in Sec. 6 performs an overapproximation of the necessary memory monitoring and successfully removes many irrelevant records and queries. We intend to further improve its precision in our future work.

9 Conclusion and Future Work

We have presented our solution of memory monitoring for runtime assertion checking with FRAMA-C. It can be applied on C code annotated in E-ACSL, an executable specification language offering among other features various memory-related constructs on validity and initialization of memory locations. The advantages of this solution include a very expressive specification formalism, a deep integration into the FRAMA-C platform and various possibilities of collaboration with other analyzers [7]. Thus, runtime assertion checking can benefit from annotations automatically generated by other plug-ins (e.g. VALUE or RTE), help to understand proof failures (e.g. during program proof with JESSIE or WP plug-ins), skip runtime checking of properties already established by other plug-ins and contribute to consolidated statuses of annotations in FRAMA-C [5].

We have described the global architecture, instrumentation with E-ACSL2C and particular aspects related to efficient storage of block metadata, efficient updates and lookups in the store and static analysis based optimization of monitored variables, as well as our initial experimental results. In particular, runtime assertion checking has indeed found errors in 100% of non-equivalent mutants for several simple C programs with complete E-ACSL contracts.

One future work direction is extending the support of E-ACSL language by E-ACSL2C, in particular, for temporal memory safety and advanced memory-related constructs like `\assigns`, `\freeable` or `\separated` [10]. Even if our main objective is different from many other works focused on memory safety, we would like to better evaluate our solution with respect to the state-of-the-art tools on commonly used benchmarks. Since we check only specified properties at runtime, that will require to write or automatically generate E-ACSL annotations related to memory safety. While runtime assertion checking for such a rich specification language as E-ACSL will likely have a greater overhead compared to these tools (that do not need to monitor function contracts and variable initialization, or treat specific memory-related E-ACSL constructs), some of the implementation solutions they used can still be applicable in our context. Future work also includes further optimizations to minimize the calls to the monitoring library (e.g. redundant checks or irrelevant monitoring).

References

1. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability: A study of field failures in operating systems. In: The 1991 International Symposium on Fault-Tolerant Computing (FTCS 1991), pp. 2–9. IEEE Computer Society (1991)
2. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes 31(3), 25–37 (2006)
3. Turing, A.: Checking a large routine. In: The Conference on High Speed Automatic Calculating Machines, pp. 67–69 (1949)
4. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A program analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
6. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: The 28th Annual ACM Symposium on Applied Computing (SAC 2013), pp. 1230–1235. ACM (2013)
7. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: Bensalem, S., Legay, A. (eds.) RV 2013. LNCS, vol. 8174, Springer, Heidelberg (2013)
8. Szpankowski, W.: Patricia tries again revisited. J. ACM 37(4), 691–711 (1990)
9. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language (January 2012), http://frama-c.com/download/e-acsl/e-acsl.pdf
10. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6 (April 2013), http://frama-c.com/acsl.html
11. Baudin, P., Pacalet, A., Raguideau, J., Schoen, D., Williams, N.: CAVEAT: a tool for software validation. In: The 2002 International Conference on Dependable Systems and Networks (DSN 2002), p. 537. IEEE Computer Society (2002)
12. Filliâtre, J.-C., Marché, C.: The why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)

13. Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language, Iowa State Univ. (2003), `http://cs.iastate.edu/~leavens/JML/Relatedpapers/index.html`
14. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc. (1988)
15. ISO/IEC 9899:1999: Programming languages – C
16. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in c programs. In: The Third International Workshop on Automatic Debugging (AADEBUG 1997), pp. 13–26 (1997)
17. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: The 3rd International Conference on Virtual Execution Environments (VEE 2007), pp. 65–74. ACM (2007)
18. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: The 4th Int. Workshop on Automation of Software Test (AST 2009), pp. 70–78. IEEE Computer Society (2009)
19. 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. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 262–284. Springer, Heidelberg (2003)
20. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: The Embedded Real-Time Software and Systems Congress, ERTS 2012 (2012)
21. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: The 11th Annual Network and Distributed System Security Symposium (NDSS 2004), pp. 159–169 (2004)
22. Xu, W., DuVarney, D.C., Sekar, R.: An efficient and backwards-compatible transformation to ensure memory safety of C programs. In: FSE 2004, pp. 117–126. ACM (2004)
23. Dhurjati, D., Adve, V.S.: Backwards-compatible array bounds checking for C with very low overhead. In: The 28th International Conference on Software Engineering (ICSE 2006), pp. 162–171 (2006)
24. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: The 18th USENIX Security Symposium (USENIX 2009), pp. 51–66. USENIX Association (2009)
25. Simpson, M.S., Barua, R.: MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Softw., Pract. Exper. 43(1), 93–128 (2013)
26. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: The ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI 1994), pp. 290–301. ACM (1994)
27. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. 27(3), 477–526 (2005)
28. Oiwa, Y.: Implementation of the memory-safe full ANSI-C compiler. In: The 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 259–269. ACM (2009)
29. Yuan, J., Johnson, R.: CAWDOR: compiler assisted worm defense. In: The 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), pp. 54–63. IEEE Computer Society (2012)
30. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: The 2012 USENIX Annual Technical Conference (USENIX ATC 2012), pp. 309–318. USENIX Association (2012)

Impartiality and Anticipation for Monitoring of Visibly Context-Free Properties

Normann Decker, Martin Leucker, and Daniel Thoma

Institute for Software Engineering and Programming Languages
Universität zu Lübeck, Germany
{decker,leucker,thoma}@isp.uni-luebeck.de

Abstract. We study monitoring of visibly context-free properties. These properties reflect the common concept of nesting which arises naturally in software systems. They can be expressed e.g. in the temporal logic CaRet which extends LTL by means of matching calls and returns. The future fragment of CaRet enables us to give a direct unfolding-based automaton construction, similar to LTL. We provide a four-valued, impartial semantics on finite words which is particularly suitable for monitoring. This allows us to synthesize monitors in terms of deterministic push-down Mealy machines. To go beyond impartiality, we develop a construction for anticipatory monitors from visibly push-down ω-automata by utilizing a decision procedure for emptiness.

1 Introduction

In Runtime Verification (RV) an actual execution of a system is checked with respect to a given correctness property [1]. Therefore, typically a so-called monitor is synthesized from the high-level specification of the correctness property, which yields an *assessment* or a *verdict*, denoting to which extent the property is satisfied by the current execution.

RV is a verification technique that is becoming more and more popular in recent years but is also a key ingredient in new programming paradigms such as *monitor-oriented programming* [2] or software architectures for reliable systems such as *runtime reflection* [1].

In runtime verification, one always faces a finite execution of a potentially infinite run of a system. Such an execution may be completed, and for example, completely stored in a log file and subsequently checked with respect to some property, or it may be checked on-line while it is continuously evolving. Depending on the application, different notions of correctness assessments are appropriate and monitors evaluating an execution and a property accordingly are needed.

As explained in [3] and [4], a two-valued assessment yielding *yes/true* or *no/false* seems appropriate when faced with completed executions as either a property is satisfied or not.

When checking an execution on-line, at least three different assessments (*true*/*false*/*inconclusive*) are needed to adhere to the maxim of *impartiality.*

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 183–200, 2013.

This states that a property should only be evaluated to true or false, if any continuation of the execution will yield the same verdict. This ensures that runtime verification is not stopped prematurely with the (misleading) understanding that a property is violated or fulfilled although subsequent observations may yield a different verdict.

The inconclusive verdict can be refined further to a verdict of *presumably true* and *presumably false*. *Presumably true* expresses the fact that no violation has been seen but one might still occur in the future as the observation might be extended. *Presumably false* describes that some obligation is not satisfied but might still be fulfilled in the future. These verdicts are of particular interest when a system terminates as they still allow for some assessment where an inconclusive verdict would have provided no information at all.

The maxim of impartiality can trivially be fulfilled with a monitor always yielding the verdict *inconclusive*. The maxim of *anticipation* on the other hand states that a verdict of true or false should be evaluated as soon as this is possible, meaning for example for a violated safety property that the violation should be reported by a monitor for the shortest execution of a run (i.e. the shorted prefix of the run) violating the property.

The methods for checking properties of executions can broadly be divided into rewriting-based and automata-based approaches. As described in [4], the latter can sometimes be seen as pre-computations of rewriting-based approaches, highlighting that rewriting can be understood as on-the-fly automata constructions. Thus, typically, rewriting-based approaches are easier to implement, may have a better memory performance but may have a worse runtime performance. Moreover, anticipatory approaches to runtime verification need a complex check in each verification step which can be done more efficiently using pre-computations with automata.

A prominent specification formalism for denoting properties to check is Linear-time Temporal Logic (LTL) [5], which allows to specify star-free regular properties. A bunch of different approaches for checking LTL properties at runtime have been proposed. These can be categorized into two-valued rewriting-based approaches [6,7,8], impartial three and four-valued rewriting and automata-based approaches [9,4], or impartial and anticipatory automata-based approaches [10,11]. The latter approach was then generalized to arbitrary linear-time temporal logics which come with an automaton-based abstraction for a satisfiability check in [12].

For practical applications, plain LTL specifications are typically not enough. Besides enrichments like dealing with data or real-time aspects, one of the important goals is to specify context-free properties, as, in software systems, nesting structures arise naturally, in particular in the context of recursive programs with calls and returns. State-full protocols impose nested structures on message sequences. For example, a transaction protocol requires (recursively) any sub-transactions of some transaction to finish before its completion. Similar properties arise in nested document formats such as XML or serialization of nested data structures.

This common concept of nesting is reflected in the class of visibly context-free languages. Alur et al. proposed in [13] visibly pushdown automata as an automaton characterization of visibly context-free languages. The nature of this automaton model is that the stack action is determined by the input symbol. This is analogous to calls and returns in recursive programs. In contrast, a push-down property that is not visibly, is the language a^nba^n where a stack is needed rather for counting than for recognizing a nesting structure.

In the context of temporal specifications, the logic CaRet is a natural extension of LTL with the ability to express nesting [14,15]. The concept of a direct temporal successor is extended to the concept of a so-called abstract successor. That is, the successor on the same level between a call and its matching return. CaRet, however does not cover the full class of visibly context-free properties. Logics with full expressiveness regarding visibly context-free languages are, for example VP-μTL [16] and MSO$_\mu$ [13].

Monitor synthesis for CaRet was first considered in [17]. More specifically, a monitoring approach for a version of CaRet is provided that allows for checking globally a past-time property, i.e. safety properties [18]. According to our taxonomy, the approach is rewriting-based. Due to the additional stack that has to be kept in this setting, a translation to an impartial automaton approach is not straightforward. Note that for CaRet the general scheme developed in [12] is not applicable.

In this paper, we study monitoring of visibly context-free properties. The future fragment of CaRet allows, similar to LTL a direct unfolding-based automaton construction. We provide a four-valued, impartial semantics on finite words in Section 3 which is particularly suitable for monitoring. It allows us to synthesize monitors in terms of deterministic push-down Mealy machines.

Additionally, we study an anticipatory approach to monitoring of visibly context-free properties in Section 4. We achieve to construct anticipatory monitors from visibly push-down ω-automata by utilizing a decision procedure for emptiness. Thus, this allows us to monitor properties expressed e.g. in full CaRet or VP-μTL. As such, we provide a complete picture of monitoring context-free properties in the taxonomy introduced in [4] and explained at the beginning of this paper.

2 Preliminaries

Alphabets and Words. Let AP be a finite set of atomic propositions and $\Sigma = 2^{\mathrm{AP}}$ a finite alphabet. We assume Σ to be the disjoint union of *call* symbols Σ_{c}, *return* symbols Σ_{r} and *internal* symbols Σ_{int}. Furthermore, call, int and ret denote propositional formulae characterizing exactly the call, internal and return symbols, respectively. A word over Σ is a possibly infinite sequence $w = w_0w_1w_2\ldots$ s.t. $w_i \in \Sigma$. We denote by $w^{(i)} = w_iw_{i+1}\ldots$ the suffix starting at position i and, if $w \in \Sigma^n$, by $|w| = n$ its length. Let Σ^* and Σ^ω denote the sets finite and infinite words over Σ, respectively, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$.

We denote $\mathbb{B}_4 = \{\top, \top^{\mathsf{p}}, \bot^{\mathsf{p}}, \bot\}$ the four-valued De-Morgan lattice with linear order, i.e. the lattice with $\top \sqsupseteq \top^{\mathsf{p}} \sqsupseteq \bot^{\mathsf{p}} \sqsupseteq \bot$, $\top = \neg\bot$ and $\top^{\mathsf{p}} = \neg\bot^{\mathsf{p}}$. Note, that we assume big operators to have lower precedence than small ones, thus $\bigsqcup a \sqcap b = \bigsqcup(a \sqcap b)$.

Visibly Push-Down Automata. A (non-deterministic) *push-down automaton* is a tuple $\mathcal{F} = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ where

- Σ, Γ are the finite *input* and *stack alphabet*, respectively, and $\Gamma_\# := \Gamma \dot{\cup} \{\#\}$ the stack alphabet enriched by a new bottom symbol $\# \notin \Gamma$,
- Q is a finite set of control *states*,
- $Q_0 \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *accepting states* and
- $\delta : Q \times \Gamma_\# \times \Sigma \to 2^{Q \times \Gamma_\#^{\leq 2}}$ is the non-deterministic, *transition function.*

A *configuration* of $\mathcal{F}$ is a tuple $(q, s) \in Q \times (\Gamma^* \{\#\})$ comprising the current control state and a stack assignment ending with $\#$. A *run* of $\mathcal{F}$ on a *finite* input word $w = w_0 w_1 \ldots w_n \in \Sigma^*$ is a sequence $c_0 c_1 \ldots c_{n+1}$ of configurations $c_i \in Q \times (\Gamma^* \{\#\})$ s.t.

- $c_0 \in Q_0 \times \{\#\}$ and
- if $c_{i+1} = (q', \gamma' \gamma'' s)$ then $c_i = (q, \gamma s)$ with $(q', \gamma' \gamma'') \in \delta(q, \gamma, w_i)$,

where $q \in Q$,$\gamma \in \Gamma_\#$. A run $(q_0, s_0)(q_1, s_1) \ldots (q_{n+1}, s_{n+1})$ is accepting if $q_{n+1} \in F$.

We call a push-down automaton $\mathcal{P}$ reading *infinite* words a *push-down ω-automaton.* A run of $\mathcal{P}$ on an infinite word $u = u_0 u_1 \ldots \in \Sigma^\omega$ is an infinite sequence of configurations $c_0 c_1 \ldots$ defined as above. A run $(q_0, s_0)(q_1, s_1) \ldots$ is accepting if the sequence of states $q_0 q_1 \ldots$ satisfies a Büchi condition, i.e. there is some $q \in F$ that occurs infinitely often in the sequence.

A push-down (ω-)automaton accepts a word $w \in \Sigma^\infty$ if there is an accepting run on w. By $\mathcal{L}(\mathcal{P}) \subseteq \Sigma^\omega$ and $\mathcal{L}(\mathcal{F}) \subseteq \Sigma^*$ we denote the set of words accepted by $\mathcal{P}$ and $\mathcal{F}$, respectively.

$\mathcal{F}$ and $\mathcal{P}$ are called a *visibly push-down automaton* (Vpa) and a *visibly push-down ω-automaton* (ω-Vpa), respectively, if the input alphabet Σ is the union of three disjoint alphabets Σ_c, Σ_r, Σ_int and for $(q', u) \in \delta(q, \gamma, a)$

- $u = \epsilon$ iff $a \in \Sigma_\mathsf{r}$ and $\gamma \neq \#$,
- $u = \#$ iff $a \in \Sigma_\mathsf{r}$ and $\gamma = \#$,
- $u = \gamma$ iff $a \in \Sigma_\mathsf{int}$ and
- $u \in (\Gamma\{\gamma\})$ iff $a \in \Sigma_\mathsf{c}$.

Emptiness of Push-Down Automata. Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta_\mathcal{P}, Q_0, F)$ be a push-down ω-automaton. Following [19], we can represent the set configurations of $\mathcal{P}$, from which all inputs are rejected (empty configurations), by means of a *multi automaton* $\mathcal{A} = (S \dot{\cup} Q, \Gamma, \delta, Q, A)$. $S \dot{\cup} Q$ is the *state space* where S is a finite set and disjoint from the states Q of $\mathcal{P}$, which are the *initial* states of $\mathcal{A}$. The stack alphabet Γ of $\mathcal{P}$ is the *input alphabet* of $\mathcal{A}$ and $A \subseteq S$ are the *accepting states.* $\delta : S \dot{\cup} Q \times \Gamma \to 2^{S \dot{\cup} Q}$ is the *transition function.* The multi-automaton $\mathcal{A}$ accepts configurations $(q, s\#)$ of $\mathcal{P}$ by behaving like a finite automaton with initial state q and reading the stack configuration $s \in \Gamma^*$ as input.

Note, that in [19] the state space of $\mathcal{A}$ and $\mathcal{P}$ are disjoint but each initial state s of $\mathcal{A}$ corresponds (bijectively) to some $q \in Q$. We therefore identify those.

Further, we can assume $\mathcal{A}$ to have a deterministic transition function δ since $\mathcal{A}$ is basically a compact representation of a set of finite automata.

3 Four-Valued Semantics of CaRet$^+$ on Finite Traces and Impartial Monitoring

In this section, we consider the logic CaRet as a specification formalism for nesting structures. For its future fragment CaRet$^+$, we provide a four-valued, impartial semantics on finite words. We show how to construct a push-down Mealy machine as monitor that incrementally reads input symbols and outputs the semantics of the observed trace. Our aim is to give an easily implementable monitor construction for properties expressing nesting structures.

3.1 Four-Valued CaRet$^+$

The syntax of CaRet$^+$ formulae is defined by the following grammar.

$$\begin{aligned}\varphi ::= \; & p \;\mid \varphi \wedge \varphi \mid \mathrm{X}\,\varphi \mid \mathrm{X}^{\mathrm{a}}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi \mid \varphi\,\mathrm{U}^{\mathrm{a}}\,\varphi \mid \\ & \neg p \mid \varphi \vee \varphi \mid \overline{\mathrm{X}}\,\varphi \mid \overline{\mathrm{X}^{\mathrm{a}}}\,\varphi \mid \varphi\,\mathrm{R}\,\varphi \mid \varphi\,\mathrm{R}^{\mathrm{a}}\,\varphi\end{aligned}$$

The idea of how CaRet extends the operators known form LTL is as follows: Consider for example a program procedure. While the direct successor of a line might be the first line in a called procedure, the abstract successor jumps directly to the next line in the current procedure and omits to enter any called procedures. Moreover, the last line in the procedure has a direct successor, namely the return position in its caller, but no abstract successor. CaRet uses the abstract next modality X^{a} to specify a property at the next abstract position. Further, in general, it contains the abstract past modality $\mathrm{X}^{\mathrm{a}-}$ for specifying a property at the call position of the current procedure. Intuitively, consecutive application of $\mathrm{X}^{\mathrm{a}-}$ walks up the call stack. Also, CaRet provides abstract versions of the common until and since operators. However, for sake of simplicity, we do not support the past operators in this section. It shall be noted, that in contrast to LTL, past modalities add expressiveness to the logic.

For a formula Φ, we denote $\mathsf{sub}(\Phi)$ the set of sub-formulae including unfoldings, e.g. $\psi \vee \varphi \wedge \mathrm{X}(\varphi\,\mathrm{U}\,\psi)$ for a sub-formula $\varphi\,\mathrm{U}\,\psi$ of Φ.

Semantics on Finite Traces. The semantics of CaRet is defined on infinite traces. Since monitoring inherently deals with finite traces we provide the impartial finitary semantics FCaRet$_4$. It is intended to intuitively resemble the infinite trace semantics, similar to finitary semantics for LTL formulae, e.g. FLTL and FLTL$_4$ [4].

As the latter, FCaRet$_4$ uses the four truth values *true* ($\top$), *false* ($\bot$), *presumably true* ($\top^{\mathsf{p}}$) and *presumably false* ($\bot^{\mathsf{p}}$), allowing for impartiality.

The distinguishing aspect between finitary and infinitary semantics is the need for handling the end of a trace which is reflected by discriminating weak and strong operators. A formula $\mathrm{X}\,\varphi$ describes what should happen at the next time step. If the trace ends here, it needs to be specified if that is desired or not. The temporal operators X, X^{a}, U and U^{a} are considered as strong operators having an existential character. They require the next position to exist and evaluate to $\bot^{\mathsf{p}}$ if not. Consequently, their duals $\overline{\mathrm{X}}$, $\overline{\mathrm{X}^{\mathrm{a}}}$, R and R^{a} have a weak, i.e. universal character; they impose restrictions only on actually existing positions. If there is no successive position, they evaluate to $\top^{\mathsf{p}}$. Note that the next operators and their (weak) duals therefore do not coincide on finite traces as they do on infinite ones.

Abstract Successors. For the semantics of the abstract temporal modalities, we use the notion of abstract steps in terms of the abstract successor function $\mathsf{succ}^{\mathsf{a}}$ as in the infinitary CaRet semantics. We define the partial function $\mathsf{succ}^{\mathsf{a}} : \Sigma^{\infty} \rightharpoonup \mathbb{N}_0$ mapping a word to the *abstract successor* of its first position. For any $w \in \Sigma^{\infty}$, $\mathsf{succ}^{\mathsf{a}}(w) = 1$ if $|w| \geq 2$, $w_0 \notin \Sigma_{\mathsf{c}}$ is not a call and $w_1 \notin \Sigma_{\mathsf{r}}$ is not a return. If w starts with a call, then the abstract successor is its matching return, if it exists: $\mathsf{succ}^{\mathsf{a}}(w) = i$ if $w_0 \in \Sigma_{\mathsf{c}}$ and $i \in \mathbb{N}_0$ is the smallest number s.t. $w_i \in \Sigma_{\mathsf{r}}$ and in $w_1 \dots w_i$ the number of positions j with $w_j \in \Sigma_{\mathsf{r}}$ is greater than the number of positions j' with $w_{j'} \in \Sigma_{\mathsf{c}}$. $\mathsf{succ}^{\mathsf{a}}(w)$ is undefined in all other cases. Further, we let $\mathsf{succ}^{\mathsf{a}*}(w)$ denote the set of positions $\{i_1, i_2, \dots\}$ on a word $w \in \Sigma^{\infty}$ s.t. $i_1 = 0$ and $i_{j+1} = \mathsf{succ}^{\mathsf{a}}(w^{(i_j)})$. Additionally, we define a predicate $\mathsf{complete}(w)$ that is true if $\mathsf{succ}^{\mathsf{a}*}(w)$ has a maximal element $i < |w| - 1$ and $w_{i+1} \in \Sigma_{\mathsf{r}}$ is a return, otherwise $\mathsf{complete}(w)$ is false. That is, $\mathsf{complete}(w)$ is true if a return in w is not matched and thus the abstract sequence formed by the positions in $\mathsf{succ}^{\mathsf{a}*}(w)$ terminates because of the first unmatched return.

We define the semantics in conformance with FLTL$_4$ as defined in [4]. The semantics of a formula is given in terms of a function that maps words $w \in \Sigma = 2^{\mathrm{AP}}$ to the $\mathbb{B}_4$-lattice. For propositions and boolean connectives, the two-valued semantics can be directly lifted to the $\mathbb{B}_4$-lattice.

$$[\![p]\!]_4(w) = \begin{cases} \top & \text{if } p \in w_0 \\ \bot & \text{if } p \notin w_0 \end{cases} \qquad [\![\neg p]\!]_4(w) = \begin{cases} \top & \text{if } p \notin w_0 \\ \bot & \text{if } p \in w_0 \end{cases}$$

$$[\![\varphi \wedge \psi]\!]_4(w) = [\![\varphi]\!]_4(w) \sqcap [\![\psi]\!]_4(w) \qquad [\![\varphi \vee \psi]\!]_4(w) = [\![\varphi]\!]_4(w) \sqcup [\![\psi]\!]_4(w)$$

For the (direct) strong and weak next operators the semantics is defined as discussed above.

$$[\![\mathrm{X}\,\varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(1)}) & \text{if } |w| > 1 \\ \bot^{\mathsf{p}} & \text{otherwise} \end{cases} \qquad [\![\overline{\mathrm{X}}\,\varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(1)}) & \text{if } |w| > 1 \\ \top^{\mathsf{p}} & \text{otherwise} \end{cases}$$

The standard semantics of the U and R operator is lifted to $\mathbb{B}_4$. The right parts of the definitions deal with the end of words.

$$[\![\varphi \,\mathrm{U}\, \psi]\!]_4(w) = \left(\bigsqcup_{i<|w|} [\![\psi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{j<i} [\![\varphi]\!]_4(w^{(j)}) \right) \sqcup \left(\bot^{\mathsf{p}} \sqcap \bigsqcap_{i<|w|} [\![\varphi]\!]_4(w^{(i)}) \right)$$

$$[\![\varphi \,\mathrm{R}\, \psi]\!]_4(w) = \left(\bigsqcup_{i<|w|} [\![\varphi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{j\leq i} [\![\psi]\!]_4(w^{(j)}) \right) \sqcup \left(\top^{\mathsf{p}} \sqcap \bigsqcap_{i<|w|} [\![\psi]\!]_4(w^{(i)}) \right)$$

The abstract next operator can be defined in a similar manner as their direct counter parts using the abstract successor $\mathsf{succ}^{\mathsf{a}}$ instead of the direct successor.

While the the two-valued semantics FLTL considers observations as terminated, the four-valued semantics FLTL$_4$ reflects the intuition, that a finite observation might still be continued and therefore next operators X and $\overline{\mathrm{X}}$ evaluate to $\bot^{\mathsf{p}}$ and $\top^{\mathsf{p}}$, respectively, at the end of a word. For the end of an abstract sequence, i.e. when there is no abstract successor, both cases are possible. When observing an (unmatched) return symbol as the direct successor, the current "procedure" definitely returns and there is no continuation. The abstract next operators shall then give a definite verdict, i.e. $\top$ or $\bot$. On the other hand, if the abstract sequence ends because the whole observation ends before the next abstract successor, there might be a continuation and hence the evaluation is preliminary, i.e. $\top^{\mathsf{p}}$ or $\bot^{\mathsf{p}}$.

$$[\![\mathrm{X}^{\mathrm{a}}\, \varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(n)}) & \text{if } \mathsf{succ}^{\mathsf{a}}(w) = n \in \mathbb{N} \\ \bot & \text{if } \mathsf{succ}^{\mathsf{a}}(w) \text{ is undef. } \wedge w_1 \in \Sigma_{\mathsf{r}} \\ \bot^{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\overline{\mathrm{X}^{\mathrm{a}}}\, \varphi]\!]_4(w) = \begin{cases} [\![\varphi]\!]_4(w^{(n)}) & \text{if } \mathsf{succ}^{\mathsf{a}}(w) = n \in \mathbb{N} \\ \top & \text{if } \mathsf{succ}^{\mathsf{a}}(w) \text{ is undef. } \wedge w_1 \in \Sigma_{\mathsf{r}} \\ \top^{\mathsf{p}} & \text{otherwise} \end{cases}$$

Note that the semantics of X^{a} is slightly different from the one in [14] to fit together with the U^{a} operator.

Based on the same idea, the abstract until and release operators are defined as follows.

$$[\![\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi]\!]_4(w) = \begin{cases} \left(\bigsqcup_{i\in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{\substack{j\in \mathsf{succ}^{\mathsf{a}*}(w) \\ j<i}} [\![\varphi]\!]_4(w^{(j)}) \right) & \text{if } \mathsf{complete}(w) \\ \left(\bigsqcup_{i\in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \sqcap \bigsqcap_{\substack{j\in \mathsf{succ}^{\mathsf{a}*}(w) \\ j<i}} [\![\varphi]\!]_4(w^{(j)}) \right) & \\ \quad \sqcup \left(\bot^{\mathsf{p}} \sqcap \bigsqcap_{i\in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \right) & \text{otherwise} \end{cases}$$

$$[\![\varphi \,\mathrm{R}^{\mathrm{a}}\, \psi]\!]_4(w) = \begin{cases} \left(\bigsqcup\limits_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \;\sqcap\; \bigsqcap\limits_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w)\\ j \le i}} [\![\psi]\!]_4(w^{(j)}) \right) & \text{if } \mathsf{complete}(w) \\[2ex] \left(\bigsqcup\limits_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\varphi]\!]_4(w^{(i)}) \;\sqcap\; \bigsqcap\limits_{\substack{j \in \mathsf{succ}^{\mathsf{a}*}(w)\\ j \le i}} [\![\psi]\!]_4(w^{(j)}) \right) & \text{otherwise} \\ \qquad \sqcup \left(\top^{\mathsf{p}} \sqcap \bigsqcap\limits_{i \in \mathsf{succ}^{\mathsf{a}*}(w)} [\![\psi]\!]_4(w^{(i)}) \right) & \end{cases}$$

In contrast to the until and release operators two cases have to be distinguished for their abstract counterparts. If the sequence of abstract successors for a word is complete, i.e. if the sequence terminates because of an umatched return, the operators cannot evaluate to $\bot^{\mathsf{p}}$ or $\top^{\mathsf{p}}$, respectively.

3.2 Visibly Push-down Mealy Machines (VPMM)

A typical approach to monitoring temporal properties is based on formula rewriting. When observing a symbol, the formula is evaluated and additionally rewritten to maintain the gained information. This requires equations for transforming any formula into a formula where each temporal operator is an X operator or is guarded by some X. Then, every sub-formula can explicitly be evaluated when reading only one new letter. For the U operator, the unfolding equation is standard and the abstract operator can be unfolded analogously:

$$\varphi \,\mathrm{U}\, \psi \equiv \psi \vee (\varphi \wedge \mathrm{X}(\varphi \,\mathrm{U}\, \psi))$$
$$\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi \equiv \psi \vee (\varphi \wedge \mathrm{X}^{\mathrm{a}}(\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi))$$

What remains is an unfolding of the abstract next operator X^{a}. According to the semantics, reading a return or an internal symbol it behaves like the classical X operator, accept that there must not follow a return symbol in the next step. Evaluating a formula $\mathrm{X}^{\mathrm{a}}\,\varphi$ for a call symbol, the evaluation of φ needs to be postponed until the matching return symbol is read. If a return follows immediately it is matching. Otherwise the matching return can be reached by following the abstract sequence of positions until the first unmatched return.

$$\begin{aligned} \mathrm{X}^{\mathrm{a}}(\varphi) \equiv\;& (\neg\mathsf{call} \wedge \mathrm{X}(\neg\mathsf{ret} \wedge \varphi)) \\ \vee\,& (\;\mathsf{call} \wedge \mathrm{X}(\;\mathsf{ret} \wedge \varphi)) \\ \vee\,& (\;\mathsf{call} \wedge \mathrm{X}(\neg\mathsf{ret} \wedge \mathsf{true}\,\mathrm{U}^{\mathrm{a}}(\neg\mathsf{call} \wedge \mathrm{X}(\mathsf{ret} \wedge \varphi)))) \end{aligned}$$

Using this equality for substituting X^{a} operators, we can equivalently transform any CaRet formula s.t. every temporal operator is guarded by X and hence do a step-wise evaluation. When only the classical operators are considered, the number of different formulae arising during evaluation is bounded (as long as the formulae are kept in e.g. disjunctive normal form). This is no longer the case for the abstract operators as during evaluation an unbounded number of inequivalent formulae may occur. This is expected, as the formula describes a pushdown-language and encodes a stack. In the following, we will use pushdown machines to handle the stack explicitly. This simplifies implementation as well as theoretical discussion.

The outline for the rest of this section is as follows. We introduce non-deterministic push-down Mealy machines (Pmm) and show how they can be determinized. Next, based on the FCaRet$_4$ semantics defined above, we give a procedure to construct a Pmm from a CaRet$^+$ formula, that reads symbols and outputs the FCaRet$_4$ semantics of the word read so far.

Mealy Machines. A non-deterministic Mealy machine can, in general reach multiple configurations at a time. Each such current configuration yields an output. To consistently define the overall output of the automaton, we need to be able to summarize the single outputs in each step. The existential character of a non-deterministic model is lifted to a supremum (join) operation on all possible outputs in each step. A configuration may have multiple successors, which have no order. We therefore need commutativity, associativity and idem-potency of the join operation on the outputs, that is, we require the output domain to be a semi-lattice.

Definition 1 (Push-down Mealy Machine). *A (non-deterministic)* push-down Mealy machine *(*Pmm*) is a tuple* $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{L})$ *where*

- Σ, Γ *are the finite* input *and* stack alphabet*, respectively, and* $\Gamma_\# := \Gamma \cup \{\#\}$ *the stack alphabet enriched by a new bottom symbol* $\# \notin \Gamma$*,*
- $\mathbb{L}$ *is the* output alphabet *with* $(\mathbb{L}, \sqcup)$ *forming a semi-lattice,*
- Q *is a finite set of control* states*,*
- $Q_0 \subseteq Q$ *is the set of* initial states *and*
- $\delta : Q \times \Gamma_\# \times \Sigma \to 2^{Q \times \Gamma_\#^{\leq 2} \times \mathbb{L}}$ *is the non-deterministic, labeled* transition function*.*

A *configuration* of $\mathcal{M}$ is a tuple $(q, s) \in Q \times (\Gamma^*\{\#\})$ comprising the current control state and a stack assignment ending with #. The *run* of $\mathcal{M}$ on a non-empty input word $w = w_0 w_1 \ldots w_n \in \Sigma^+$ is the alternating sequence $C_0 \xrightarrow{\ell_1} C_1 \ldots \xrightarrow{\ell_n} C_n$ of sets of configurations $C_i \subseteq Q \times (\Gamma^*\{\#\})$ and output symbols $\ell_i \in \mathbb{L}$ s.t.

- $C_0 = Q_0 \times \{\#\}$,
- $L_{i+1} \subseteq \mathbb{L}$ and C_{i+1} are the smallest sets such that, for $\gamma \in \Gamma$, $(q, \gamma s) \in C_i$ and $(q', \gamma'\gamma'', \ell) \in \delta(q, \gamma, a_{i+1})$ implies $(q', \gamma'\gamma'' s) \in C_{i+1}$ and $\ell \in L_{i+1}$, and
- $\ell_i = \bigsqcup L_{i+1}$.

The *output* of $\mathcal{M}$ on w is $\mathcal{M}(w) := \ell_n$.

$\mathcal{M}$ is called a *visibly* Pmm (Vpmm), if it satisfies the corresponding constraints defined above for Vpa.

3.3 Determinizing VPMM

In order to be able to actually implement a VPMM as monitor to evaluate observations it must be deterministic. We can lift the determinization construction for VPA [13] to determinize a VPMM $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{L})$ by adding treatment of output symols. We construct an equivalent deterministic VPMM $\mathcal{P}' = (Q', \Sigma, \Gamma', \delta', q_0', \mathbb{L})$ as follows.

In the finite control $Q' = 2^{Q \times Q} \times 2^Q$ we store, as in the standard subset construction for finite automata, a set of *current states* $R \subseteq Q$ and additionally an *effect relation* $S \subseteq Q \times Q$.

Inbetween a call action a_c and its corresponding return action a_r, S summarizes the transitions that were made on every state. That is, when $\mathcal{P}$ were in some state q just after reading a_c and from there possibly reached some state q' before reading a_r, S contains the tuple (q, q').

The stack of $\mathcal{P}'$, stores triples (S', R', a_c) from $\Gamma \subseteq Q' \times \Sigma_c$ where R', S' are the current states and the effect relation at the time the last open call a_c occurred. In the initial state $q_0' = \{(\mathsf{Id}_Q, Q_0)\}$, there is no recorded effect, i.e. each q points to itself, and the current states are the initial states of $\mathcal{P}$.

Internal. An internal action $a_{\mathsf{int}} \in \Sigma_{\mathsf{int}}$ simply updates the set of current states by applying δ element-wise. The effect relation is updated analogously. If $(q, q') \in S$ is a recorded effect on q and q' is mapped to q'' by δ on reading a_{int}, then in the next state of $\mathcal{P}'$ we record the tuple (q, q'') as effect on q.

We let $\delta'((S, R), a_{\mathsf{int}}, \gamma) = (S', R', \gamma, \ell)$ such that

$$\begin{aligned} S' &= \{(q, q') \mid \exists q'', \gamma', \ell' : (q, q'') \in S, (q', \gamma', \ell') \in \delta(q'', a_{\mathsf{int}}, \gamma')\} \\ R' &= \{q' \mid \exists q \in R, \gamma', \ell' : (q', \gamma', \ell') \in \delta(q, a_{\mathsf{int}}, \gamma')\} \\ \ell &= \bigsqcup \{\ell' \mid \exists q \in R, \gamma', q' : (q', \gamma', \ell') \in \delta(q, a_{\mathsf{int}}, \gamma')\} \end{aligned}$$

As opposed to the construction for VPA, we have also to compute the current output ℓ. It is obtained from all possible transitions from the current states $q \in R$ via reading a_{int}. Since δ is non-deterministic these are in general multiple values that are considered in disjunction. We therefore take the join, i.e. the supremum, of those.

Call. Upon reading a call symbol $a_{\mathsf{c}} \in \Sigma_{\mathsf{c}}$, the current states set R and the current effect S is stored by pushing them, together with a_{c}, onto the stack. While the set of current states is maintained by applying a_{c} via δ, the effect relation is reset s.t. every state q maps to itself. The output is obtained in the same way as by reading an internal action.

We let $\delta'((S, R), a_{\mathsf{c}}, \gamma) = (\mathsf{Id}_Q, R', (S, R, a_{\mathsf{c}})\gamma, \ell)$ such that

$$\begin{aligned} R' &= \{q' \mid \exists q \in R, \gamma', \gamma'', \ell' : (q', \gamma'', \ell') \in \delta(q, a_{\mathsf{c}}, \gamma')\} \\ \ell &= \bigsqcup \{\ell' \mid \exists q \in R, \gamma', \gamma'', q' : (q', \gamma'', \ell') \in \delta(q, a_{\mathsf{c}}, \gamma')\} \end{aligned}$$

Return. Having all the information from the stack when reading a return symbol $a_{\mathsf{r}} \in \Sigma_{\mathsf{r}}$, $\mathcal{P}'$ can simulate the transition relation δ on all current states. This, however is not done directly on R but on the current states at call-time R' by consecutively applying a_{c}, S and then a_{r} to obtain the new set of current states R'' and the new effect relation S''. We obtain the output from all possible transitions via a_{r}, after a_{c} and S have been applied.

We let $\delta'((S, R), a_{\mathsf{r}}, (S', R', a_{\mathsf{c}})) = (S'', R'', \epsilon, \ell)$ such that

$$\begin{aligned} U &= \{(q, q', \ell') \mid \exists q_1, q_2, \gamma', \gamma'', \ell'' : (q_1, \gamma''\gamma', \ell'') \in \delta(q, a_{\mathsf{c}}, \gamma'), \\ &\qquad\qquad (q_1, q_2) \in S, (q', \epsilon, \ell') \in \delta(q_2, a_{\mathsf{r}}, \gamma'')\} \\ S'' &= \{(q, q') \mid \exists q_3, \ell' : (q, q_3) \in S', (q_3, q', \ell') \in U\} \\ R'' &= \{q' \mid \exists q \in R, \ell' : (q, q', \ell') \in U\} \\ \ell &= \bigsqcup \{\ell' \mid \exists q \in R, q' : (q, q', \ell') \in U\} \end{aligned}$$

Note that the stack might have been necessary for computing the effect S but once it is known, the effect can be applied to a set of states without using the stack.

Finally, we need to specially treat the case of a return action $a_{\mathsf{r}} \in \Sigma_{\mathsf{r}}$ when reading the bottom symbol. Let $\delta'((S, R), a_{\mathsf{r}}, \#) = (S', R', \#, \ell)$ such that

$$\begin{aligned} S' &= \{(q, q') \mid \exists q'', \ell' : (q, q'') \in S, (q', \#, \ell') \in \delta(q'', a_{\mathsf{r}}, \#)\} \\ R' &= \{q' \mid \exists q \in R, \ell' : (q', \#, \ell') \in \delta(q, a_{\mathsf{r}}, \#)\} \\ \ell &= \bigsqcup \{\ell' \mid \exists q \in R, q' : (q', \#, \ell') \in \delta(q, a_{\mathsf{r}}, \#)\} \end{aligned}$$

3.4 Constructing a VPMM for CaRet$^+$

The idea of the construction is that the Mealy machine maintains the formulae that need to be proved. When reading an input symbol it verifies the propositional part and postpones the resulting future obligations. Standard LTL formulae are encoded into the finite control and evaluated on the next input. Abstract until and release operators are reduced to checking their unfolding.

When observing a call action, the abstract next modalities push their argument on the stack as future obligation. On return, this obligation is removed from the stack and evaluated together with the current obligation stored in the finite control.

The transition function maintains a disjunctive clause form of stored obligations and thereby removes alternation on the fly. The output in every step is the truth value from the evaluation of the current finite control state combined with the current evaluation of the stack. The stack evaluation describes the truth values of the suspended abstract operators from higher levels.

Given a CaRet$^+$ formula Φ we construct a VPMM $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, Q_0, \mathbb{B}_4)$ where the control states $Q = 2^{\mathsf{sub}(\Phi)}$ store a set of formulae to be evaluated upon the next input symbol and the stack alphabet $\Gamma = 2^{\mathsf{sub}(\Phi)} \times \mathbb{B}_4 \times \mathbb{B}_4$ stores the future obligations, the current and the previous stack evaluation.

For better readability, the transition function $\delta : Q \times \Gamma \times \Sigma \to 2^{Q \times \mathbb{B}_4 \times \Gamma^{\leq 2}}$, is specified in two parts, one handling the evaluation of the finite control separately and another part working on the stack. Let therefore be $\delta_{\mathsf{c}} : Q \times \Sigma \to 2^{Q \times \mathbb{B}_4}$ the control transition function.

Finite Control. The finite control evaluates propositional formulae directly according to the input symbol, also, next formulae just consume the input and delegate their argument to the next step. The semantics of a formula $\mathrm{X}^{\mathrm{a}}\,\varphi$ evaluates to $\bot$ if the next position is a return and the current position is not a call. However, δ_c will never be evaluated on X^{a} when reading a call symbol, as can be seen from the definition of the transition function δ below. Therefore we do not make a distinction for that case here. Until and release formulae are simply handled using their unfolding.

$$\begin{aligned}
\delta_{\mathsf{c}}(\{p\}, a) &= \begin{cases} \{(\emptyset, \top)\} & \text{if } p \in a \\ \emptyset & \text{if } p \notin a \end{cases} \\
\delta_{\mathsf{c}}(\{\neg p\}, a) &= \begin{cases} \emptyset & \text{if } p \in a \\ \{(\emptyset, \top)\} & \text{if } p \notin a \end{cases} \\
\delta_{\mathsf{c}}(\{\varphi \wedge \psi\}, a) &= \delta_{\mathsf{c}}(\{\varphi, \psi\}, a) \\
\delta_{\mathsf{c}}(\{\varphi \vee \psi\}, a) &= \delta_{\mathsf{c}}(\{\varphi\}, a) \cup \delta_{\mathsf{c}}(\{\psi\}, a) \\
\delta_{\mathsf{c}}(\{\mathrm{X}\,\varphi\}, a) &= \{(\{\varphi\}, \bot^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\overline{\mathrm{X}}\,\varphi\}, a) &= \{(\{\varphi\}, \top^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\mathrm{X}^{\mathrm{a}}\,\varphi\}, a) &= \{(\{\varphi, \neg\mathsf{ret}\}, \bot^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\overline{\mathrm{X}^{\mathrm{a}}}\,\varphi\}, a) &= \{(\{\varphi, \neg\mathsf{ret}\}, \top^{\mathsf{p}})\} \\
\delta_{\mathsf{c}}(\{\varphi \,\mathrm{U}\, \psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \vee (\varphi \wedge \mathrm{X}(\varphi \,\mathrm{U}\, \psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi \,\mathrm{R}\, \psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \wedge (\varphi \vee \overline{\mathrm{X}}(\varphi \,\mathrm{R}\, \psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \vee (\varphi \wedge \mathrm{X}^{\mathrm{a}}(\varphi \,\mathrm{U}^{\mathrm{a}}\, \psi))\}, a) \\
\delta_{\mathsf{c}}(\{\varphi \,\mathrm{R}^{\mathrm{a}}\, \psi\}, a) &= \delta_{\mathsf{c}}(\{\psi \wedge (\varphi \vee \overline{\mathrm{X}^{\mathrm{a}}}(\varphi \,\mathrm{R}^{\mathrm{a}}\, \psi))\}, a)
\end{aligned}$$

Sets of formulae (clauses) are interpreted as conjunctions. We can therefore remove alternation by directly evaluating the single formulae on the input symbol and combining the respective results:

$$\delta_{\mathsf{c}}(\{\varphi_1, \ldots, \varphi_n\}, a) = \delta_{\mathsf{c}}(\{\varphi_1\}, a) \mathbin{\dot{\sqcap}} \ldots \mathbin{\dot{\sqcap}} \delta_{\mathsf{c}}(\{\varphi_n\}, a)$$

The result of a single evaluation $\delta_{\mathsf{c}}(\varphi, a)$ are sets of tuples (K, b) of clauses and verdicts. In order to combine them the clauses need to be brought back to the disjunctive clause form which is realized by an operation $\dot{\sqcap}$: Let $\mathcal{K}_i = (K_i, b_i), \mathcal{H}_j = (H_j, c_j) \in Q \times \mathbb{B}_4$ be tuples of states (i.e. conjunctive clauses) and verdicts. Sets of such tuples are combined by means of a the meet-like operation $\dot{\sqcap}$: $2^{Q \times \mathbb{B}_4} \times 2^{Q \times \mathbb{B}_4} \to 2^{Q \times \mathbb{B}_4}$ as follows.

$$\{\mathcal{K}_1,\ldots,\mathcal{K}_n\} \mathbin{\dot{\sqcap}} \{\mathcal{H}_1,\ldots,\mathcal{H}_m\} = \bigcup_{\substack{i\in[1,n]\\ j\in[1,m]}} \{\mathcal{K}_i\} \mathbin{\dot{\sqcap}} \{\mathcal{H}_j\} \tag{1}$$

$$\{(K,b)\} \mathbin{\dot{\sqcap}} \{(H,c)\} = \{(K{\cup}H,\ b{\sqcap}c)\} \tag{2}$$

The operation maintains the disjunctive form of the clause structure (1). Single clauses are conjunctive and thus their meet is simply the union of the clauses. The truth values are combined in terms of the meet on $\mathbb{B}_4$ (2).

Stack Control. The actual transition function of $\mathcal{M}$ makes direct use of the finite control function δ_{c} on internal action $a_{\mathsf{int}} \in \varSigma_{\mathsf{int}}$ actions since they do not involve a stack operation. Only the stack evaluation is used in the output:

$$\delta(\{\varphi_1,\ldots,\varphi_n\},(K,b_1,b_2),a_{\mathsf{int}}) = (\delta_{\mathsf{c}}(\{\varphi_1,\ldots,\varphi_n\},a_{\mathsf{int}}) \mathbin{\dot{\sqcap}} \{(\emptyset,b_1)\}) \times \{(K,b_1,b_2)\}$$

On return operations $a_{\mathsf{r}} \in \varSigma_{\mathsf{r}}$, the top-most stack symbol is removed and combined to the current control state. That is, the obligation suspended to the stack earlier on the matching call is now evaluated. Note, that the preliminary verdict at call time (b_1) now is obsolete and the previous one (b_2) is evaluated.

$$\delta(\{\varphi_1,\ldots,\varphi_n\},(K,b_1,b_2),a_{\mathsf{r}}) = (\delta_{\mathsf{c}}(\{\varphi_1,\ldots,\varphi_n\},a_{\mathsf{r}}) \mathbin{\dot{\sqcap}} \{(K,b_2)\}) \times \{\epsilon\}$$

For a call $a \in \varSigma_{\mathsf{c}}$ we have

$$\begin{aligned}
\delta(\{\varphi_1,\ldots,\varphi_n\},\gamma,a) &= \delta(\{\varphi_1\},\gamma,a) \mathbin{\tilde{\sqcap}} \ldots \mathbin{\tilde{\sqcap}} \delta(\{\varphi_n\},\gamma,a)\\
\delta(\{\mathrm{X}^{\mathrm{a}}\,\varphi\},(K,b_1,b_2),a) &= \{(\emptyset,\ \bot^{\mathsf{p}}{\sqcap}b_1,\ (\{\varphi\},\bot^{\mathsf{p}}{\sqcap}b_1,b_1)(K,b_1,b_2))\}\\
\delta(\{\overline{\mathrm{X}^{\mathrm{a}}}\,\varphi\},(K,b_1,b_2),a) &= \{(\emptyset,\ \top^{\mathsf{p}}{\sqcap}b_1,\ (\{\varphi\},\top^{\mathsf{p}}{\sqcap}b_1,b_1)(K,b_1,b_2))\}\\
\delta(\{\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi\},\gamma,a) &= \delta(\{\psi \vee (\psi \wedge \mathrm{X}^{\mathrm{a}}(\varphi\,\mathrm{U}^{\mathrm{a}}\,\psi))\},\gamma,a)\\
\delta(\{\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi\},\gamma,a) &= \delta(\{\psi \wedge (\psi \vee \overline{\mathrm{X}^{\mathrm{a}}}(\varphi\,\mathrm{R}^{\mathrm{a}}\,\psi))\},\gamma,a)
\end{aligned}$$

and for $\varphi \neq \mathrm{X}^{\mathrm{a}}\,\varphi'$, $\varphi \neq \varphi'\,\mathrm{U}^{\mathrm{a}}\,\psi$, $\varphi \neq \overline{\mathrm{X}^{\mathrm{a}}}\,\varphi'$ and $\varphi \neq \varphi'\,\mathrm{R}^{\mathrm{a}}\,\psi$

$$\delta(\{\varphi\},(K,b_1,b_2),a) = (\delta_{\mathsf{c}}(\{\varphi\},a) \mathbin{\dot{\sqcap}} \{(\emptyset,b_1)\}) \times \{(\emptyset,b_1,b_1)(K,b_1,b_2)\}$$

Let $\mathcal{K}_i = (K_i,b_i,\alpha_i\gamma), \mathcal{H}_j = (H_j,c_j,\beta_j\gamma) \in Q \times \mathbb{B}_4 \times \varGamma^2$ be tuples of states (i.e. conjunctive clauses), verdicts and the top-most stack symbols. Note, that when ever a call occurs, the topmost stack symbol is not touched but a new symbol is pushed onto the stack. Therefore, the pushed symbols $\alpha_i = (A_i,u_i)$ and $\beta_i = (B_i,v_i)$ may differ whilst the symbol underneath is the same $\gamma = (G,g)$ for all tuples $\mathcal{K}_i$ and $\mathcal{H}_i$.

Sets of such tuples are combined by means of a the meet-like operation

$$\tilde{\sqcap} : 2^{Q\times\mathbb{B}_4\times\varGamma^2} \times 2^{Q\times\mathbb{B}_4\times\varGamma^2} \to 2^{Q\times\mathbb{B}_4\times\varGamma^2}$$

as follows:

$$\{K_1,\dots,K_n\} \,\tilde{\sqcap}\, \{H_1,\dots,H_m\} = \bigcup_{\substack{i\in[1,n]\\ j\in[1,m]}} \{K_i\} \,\tilde{\sqcap}\, \{H_j\}$$

$$\begin{aligned}\{(K,\ b,\ (A,u,g_1)(G,g_1,g_2))\} \,\tilde{\sqcap}\, \{(H,\ c,\ (B,v,g_1)(G,g_1,g_2))\}\\ = \{(K{\cup}H,\ b{\sqcap}c,\ (A{\cup}B, u{\sqcap}v{\sqcap}g_1, g_1)(G,g_1,g_2))\}\end{aligned}$$

Theorem 1. *Let Φ be a CaRet formula and $w \in \Sigma^*$. Then $\mathcal{M}_\Phi(w) = [\![\Phi]\!]_4(w)$.*

Corollary 1. *Given a CaRet formula φ, we can construct in 2-*ExpTime *a push-down Mealy machine $\mathcal{M}$ implementing the four-valued FCaRet$_4$ semantics of φ.*

4 Anticipatory Monitoring of Visibly Context-Free Properties

In this section we describe an anticipatory monitor construction for visibly context-free ω-languages. By basing the construction on properties given by ω-Vpa we provide support for complete CaRet including past operators and more expressive logics like VP-μTL and MSO$_\mu$ which are complete for the visibly context-free ω-languages. Furthermore, integrating an emptiness check into the monitor construction allows for the synthesis of *anticipatory* monitors, i.e. monitors that yield a definite verdict as early as possible.

Given a property $L \subseteq \Sigma^\omega$ we define a three-valued, anticipatory *monitor* function $\mathfrak{M}_3$ thereby lifting the concept of [10] from LTL to arbitrary ω-languages. $\mathfrak{M}_3 : 2^{\Sigma^\omega} \to (\Sigma^* \to \mathbb{B}_3)$ is given as

$$\mathfrak{M}_3(L)(w) = \begin{cases} \top & \text{if } \forall_{u\in\Sigma^\omega} : wu \in L \\ \bot & \text{if } \forall_{u\in\Sigma^\omega} : wu \notin L \\ ? & \text{otherwise.} \end{cases}$$

The monitor function yields $\top$ for a good prefix w i.e. if any continuation of w is in L, it yields $\bot$ for a bad prefix w i.e. if any continuation of w is not in L and it yields ? otherwise.

4.1 Emptiness Per Configuration

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, Q_0, F)$ be an ω-Vpa. In the following, we show how to construct a deterministic Vpa that accepts exactly the good and inconclusive prefixes of $\mathcal{L}(\mathcal{P})$, i.e. $\{w \in \Sigma^* \mid \exists_{u\in\Sigma^\omega} : wu \in \mathcal{L}(\mathcal{P})\}$.

As Bouajjani et al. describe in [19], we can, in polynomial time, construct a multi-automaton $\mathcal{A} = (S \cup Q, \Gamma, Q, \delta_\mathcal{A}, A)$ accepting exactly the set of configurations from which there is an accepting run of $\mathcal{P}$. That is, $\mathcal{P}$ can still accept at least one word in a configuration $(q, w\#)$ iff $w \in \Gamma^*$ accepted by $\mathcal{A}$ when starting in the state $q \in Q$.

We construct a VPA $\mathcal{F} = (Q, \Sigma, \Gamma \times S^Q, \delta_\mathcal{F}, Q_0, F_\mathcal{F})$ that behaves like $\mathcal{P}$ but simultaneously simulates $\mathcal{A}$. The initial configuration of $\mathcal{F}$ represents the initial configurations of $\mathcal{P}$ and $\mathcal{A}$. Reading inputs form Σ, $\mathcal{F}$ simulates the behavior of $\mathcal{P}$ and when $\mathcal{P}$ pushes a symbol γ onto the stack, $\mathcal{F}$ additionally simulates $\mathcal{A}$ reading γ and stores the new configuration of $\mathcal{A}$ on the stack. When $\mathcal{P}$ removes the top-most symbol γ from the stack, $\mathcal{F}$ also removes the top-most symbol including the current configuration of $\mathcal{A}$ and thereby restoring the configuration of $\mathcal{A}$ before having read γ.

A configuration of $\mathcal{A}$ is a state $s \in S \cup Q$ for each initial state $q \in Q$, meaning if $\mathcal{A}$ started in q it would currently be in state s. A configuration is therefore a mapping from Q to $S \cup Q$. Let $\hat{\delta}_\mathcal{A} : S^Q \times \Gamma \to S^Q$ be the transition function of $\mathcal{A}$ lifted to mappings $f \in S^Q$ s.t. $\hat{\delta}_\mathcal{A}(f, \gamma) : q \mapsto \delta_\mathcal{A}(f(q), \gamma)$. That is $\hat{\delta}_\mathcal{A}$ applies a γ transition "state-wise" to f. Following this idea, we define the transition function of $\mathcal{F}$ as follows. Note, ω-VPA can, in general, not be determinized and thus we construct a non-deterministic automaton $\mathcal{F}$. However since $\mathcal{F}$ is a VPA, it can be determinized afterwards [13].

$$(q', (\gamma, f)) \in \delta_\mathcal{F}(q, (\gamma, f), a) \Leftrightarrow (q', \gamma) \in \delta_\mathcal{P}(q, \gamma, a) \qquad (\text{for } a \in \Sigma_{int})$$

$$(q', \#_\mathcal{F}) \in \delta_\mathcal{F}(q, \#_\mathcal{F}, a) \Leftrightarrow (q', \#_\mathcal{P}) \in \delta_\mathcal{P}(q, \#_\mathcal{P}, a) \qquad (\text{for } a \in \Sigma_r)$$

$$(q', \epsilon) \in \delta_\mathcal{F}(q, (\gamma, f), a) \Leftrightarrow (q', \epsilon) \in \delta_\mathcal{P}(q, \gamma, a) \qquad \begin{pmatrix} \text{for } a \in \Sigma_r \\ \text{and } \gamma \neq \#_\mathcal{P} \end{pmatrix}$$

$$(q', (\gamma', f')(\gamma, f)) \in \delta_\mathcal{F}(q, (\gamma, f), a) \Leftrightarrow \begin{matrix} (q', \gamma'\gamma) \in \delta_\mathcal{P}(q, \gamma, a) \\ \text{and } f' = \hat{\delta}_\mathcal{A}(f, \gamma') \end{matrix} \qquad (\text{for } a \in \Sigma_c)$$

Here, $\#_\mathcal{F}$ and $\#_\mathcal{P}$ denote the bottom stack symbols of $\mathcal{F}$ and $\mathcal{P}$, respectively. To correctly treat the empty stack, we interpret the bottom symbol $\#_\mathcal{F}$ of $\mathcal{F}$ as $(\#_\mathcal{P}, \mathsf{id})$ since for each state q of $\mathcal{P}$, $\mathcal{A}$ is initially in the corresponding initial state, which is q itself.

In every state $q \in Q$, $\mathcal{P}$ is in a non-empty configuration, iff the multi-automaton $\mathcal{A}$ accepts the current stack for q. The current configuration f of $\mathcal{A}$ is stored in the top-most stack symbol of $\mathcal{F}$. So, when $f(q)$ is an accepting state of $\mathcal{A}$ and the current control state is q, $\mathcal{P}$ had a non-empty configuration and we hence let $\mathcal{F}$ accept exactly in the configurations $(q, (\gamma, f)s)$ s.t. $f(q) \in F_\mathcal{A}$. This condition can be realized technically by storing the top-most stack symbol in the finite control and define the set of accepting states of $\mathcal{F}$ accordingly.

From that construction we conclude that $\mathcal{F}$ accepts exactly the non-bad prefixes for the language accepted by $\mathcal{P}$.

Theorem 2. *For all $w \in \Sigma^*$, $w \in \mathcal{L}(\mathcal{F})$ if and only if $\mathfrak{M}_3(\mathcal{L}(\mathcal{P}))(w) \neq \bot$.*

4.2 Anticipatory Monitors for Visibly Context-Free Properties

Using the construction above we can now construct a Moore machine that computes the three-valued monitoring semantics $\mathfrak{M}_3(P)$ for any visibly context-free property $P \subseteq \Sigma^\omega$, assuming that P is presented as ω-VPA.

Definition 2 (Push-down Moore Machine). *A (deterministic)* push-down Moore machine *is a tuple* $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \Lambda, \lambda)$ *where*

- Q *is a finite set of* states *and* $q_0 \in Q$ *the* initial state*,*
- Σ*,* Γ*,* Λ *are the finite* input-*,* stack- *and* output alphabets*, respectively, and* $\Gamma_\# := \Gamma \cup \#$ *the stack alphabet enriched by a new* bottom symbol $\# \notin \Gamma$*,*
- $\delta : Q \times \Gamma_\# \times \Sigma \to Q \times \Gamma_\#^{\leq 2}$ *the deterministic* transition function *and*
- $\lambda : Q \to \Lambda$ *the* output function*.*

A *configuration* of $\mathcal{M}$ is a tuple $(q, s) \in Q \times (\Gamma^*\{\#\})$ comprising the current control state and a stack assignment ending with $\#$. The *run* of $\mathcal{M}$ on a word $w = a_1 \ldots a_n \in \Sigma^*$ is the sequence of configurations $c_0 c_1 \ldots c_{n+1}$ s.t. $c_0 = (q_0, \#)$ and for $\gamma \in \Gamma$, $c_i = (q, \gamma s)$ and $\delta(q, \gamma, a_{i+1}) = (q', \gamma'\gamma'')$ we have $c_{i+1} = (q', \gamma'\gamma'' s)$. The *output* of $\mathcal{M}$ on w is $\mathcal{M}(w) := \lambda(q_{\mathsf{last}})$ where $(q_{\mathsf{last}}, s_{\mathsf{last}}) = c_{n+1}$.

The Moore Machine for $\mathfrak{M}_3$. In the fashion of [10] we construct $\mathcal{F}_P$ and also $\mathcal{F}_{\neg P}$ accepting all non-bad prefixes for the complement of P and combine them to a Moore machine. We know that if some $w \in \Sigma^*$ is rejected by $\mathcal{F}_P$, then $\mathfrak{M}_3(P)(w) = \bot$ and consequently if w is rejected by $\mathcal{F}_{\neg P}$ then $\mathfrak{M}_3(P)(w) = \top$. These cases exclude each other and if both accept then $\mathfrak{M}_3(P)(w) = ?$.

Note, while it is always possible to complement an ω-Vpa for some property P and construct $\mathcal{F}_{\neg P}$ from it, it might be preferable to negate the property earlier. In particular, when using a logic that allows direct negation, it is advised to negate before constructing an automaton. Recall, we can assume $\mathcal{F}_P$ and $\mathcal{F}_{\neg P}$ determinized. We combine both and obtain a deterministic visibly push-down Moore machine $\mathcal{M}$, that outputs $\top$ for every good, $\bot$ for every bad and ? for every inconclusive prefix for P.

For $\mathcal{F}_P = (Q_P, \Sigma, \Gamma_P, \delta_P, I_P, F_P)$ and $\mathcal{F}_{\neg P} = (Q_{\neg P}, \Sigma, \Gamma_{\neg P}, \delta_{\neg P}, I_{\neg P}, F_{\neg P})$
we let $\mathcal{M} = (Q_P \times Q_{\neg P}, \Sigma, \Gamma_P \times \Gamma_{\neg P}, \delta, I_P \times I_{\neg P}, \mathbb{B}_3, \lambda)$
with $\delta((q_1, q_2), (\gamma_1, \gamma_2), a) := ((q_1', q_2'), (\gamma_1', \gamma_2')(\gamma_1'', \gamma_2''))$
where $(q_1', \gamma_1'\gamma_1'') = \delta_\varphi(q_1, \gamma_1, a)$ and $(q_2', \gamma_2'\gamma_2'') = \delta_{\neg\varphi}(q_2, \gamma_2, a)$.

The output of $\mathcal{M}$ is defined as

$$\lambda(q_1, q_2) = \begin{cases} \top & \text{if } q_2 \notin F_{\neg\varphi} \\ \bot & \text{if } q_1 \notin F_\varphi \\ ? & \text{otherwise.} \end{cases}$$

Note, that λ is well defined since P and $\neg P$ exclude each other.

Theorem 3. *Given an* ω-Vpa $\mathcal{P}$*, we can construct a deterministic push-down Moore Machine* $\mathcal{M}$ *implementing the three-valued monitoring function for* $\mathcal{L}(\mathcal{P})$*, i.e. for all* $w \in \Sigma^*$*,* $\mathcal{M}(w) = \mathfrak{M}_3(\mathcal{L}(\mathcal{P}))(w)$*.*

Corollary 2. *Given a CaRet formula* φ*, we can construct in 3-*ExpTime *a push-down Moore machine* $\mathcal{M}$ *implementing the three-valued semantics function for* φ*.*

5 Conclusion

In this paper, we investigated the problem of monitoring visibly context-free properties. In particular we proposed a four-valued semantics for the future fragment of the temporal logic CaRet on finite words, together with a monitor synthesis algorithm yielding deterministic push-down Mealy machines for properties with calls and returns.

For the full CaRet logic, or more generally, for any visibly context-free language, we provided a three-valued monitoring approach adhering both, to the maxims of impartiality and anticipation. It comprises a three-valued anticipatory semantics as well as corresponding synthesis algorithm yielding deterministic push-down Moore machine.

Together with [17] this gives a complete picture of two-valued, impartial and anticipatory semantics for runtime monitoring.

References

1. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
2. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., G.L.S. Jr. (eds.) OOPSLA, pp. 569–588. ACM (2007)
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
4. Leucker, M.: Teaching runtime verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 34–48. Springer, Heidelberg (2012)
5. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE Computer Society (1977)
6. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. 55(2), 181–199 (2001)
7. Havelund, K., Rosu, G.: Monitoring programs using rewriting. In: ASE, pp. 135–143. IEEE Computer Society (2001)
8. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
9. D'Amorim, M., Roşu, G.: Efficient monitoring of ω languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)
10. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)
11. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. 20(4), 14 (2011)
12. Dong, W., Leucker, M., Schallhart, C.: Impartial anticipation in runtime-verification. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 386–396. Springer, Heidelberg (2008)
13. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) STOC, pp. 202–211. ACM (2004)

14. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
15. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. Logical Methods in Computer Science 4(4) (2008)
16. Bozzelli, L.: Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 476–491. Springer, Heidelberg (2007)
17. Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008)
18. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Dwork, C. (ed.) PODC, pp. 377–410. ACM (1990)
19. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

Accelerating Data Race Detection Utilizing On-Chip Data-Parallel Cores

Vineeth Mekkat, Anup Holey, and Antonia Zhai

Department of Computer Science & Engineering,
University of Minnesota,
Minneapolis MN 55455, USA
{mekkat,aholey,zhai}@cs.umn.edu

Abstract. Programmers are taking advantage of the increasing availability of on-chip parallelism to meet the rising performance demands of diverse applications. Support of tools that can facilitate the detection of incorrect program execution when concurrent threads are involved is critical to this evolution. Many concurrency bugs manifest as some form of data race condition, and their runtime detection is inherently difficult due to the high overhead of the required memory trace comparisons. Various software and hardware tools have been proposed to detect concurrency bugs at runtime. However, software-based schemes lead to significant performance overhead, while, hardware-based schemes require significant hardware modifications. To enable cost-efficient design of data race detectors, it is desirable to utilize available on-chip resources. The recent integration of CPU cores with data-parallel accelerator cores, such as GPU, provides the opportunity to offload the task of data race detection to these accelerator cores. In this paper, we explore this opportunity by designing a GPU Accelerated Data Race Detector (GUARD) that utilizes GPU cores to process memory traces and detect data races in parallel applications executing on the CPU cores. GUARD further explores various optimization techniques for: (i) reducing the size of memory traces by employing signatures; and (ii) improving accuracy of signatures using coherence-based filtering. Overall, GUARD achieves the performance of hardware-based data race detection mechanisms with minimal hardware modifications.

1 Introduction

With the increased availability of on-chip parallelism in modern multicore processors, programmers are actively parallelizing applications from diverse domains to take advantage of the abundant computing power at their disposal. However, ensuring the correct execution of parallel applications is challenging due to the difficulty in tracking concurrency bugs [1, 2]. This characteristic has necessitated the development of efficient concurrency bug detection tools. Data race is one of the main classes of concurrency bugs; and being able to detect data races efficiently at runtime can facilitate the development of powerful tools that can enhance the reliability of parallel applications. However, existing proposals can be expensive in terms of performance overhead and/or implementation cost. Software-based data race detection tools [3, 4] often slow down the monitored application by orders of magnitude, thus, limiting their applicability. Although hardware-based data race detection tools [5, 6] inflict a near-zero performance

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 201–218, 2013.

impact on the monitored application, they often have a significant implementation overhead that limits their scalability. In this paper, we propose to utilize the computing power that is available on-chip emerging heterogeneous multicore processors to detect data races.

Integration of data-parallel accelerator cores with CPU cores on the same chip has emerged as a new trend to facilitate energy-efficient computing with diverse cores. The AMD Fusion APU [7], Intel Sandy Bridge [8], and Nvidia Project Denver [9] are becoming part of mainstream computing. The data-parallel cores in these designs can support a significant number of parallel threads providing computing power needed for executing data race detection algorithms efficiently. When these cores are not being employed for performance acceleration, we propose to utilize them for detecting data races in the application executing on the CPU cores. Without loss of generality, we consider the Graphics Processing Unit (GPU) as the data-parallel accelerator in our proposal. In the following sections, we refer to our design as GUARD which stands for GPU Accelerated Data Race Detector.

The volume of the memory access trace information generated makes it difficult for a data race detection mechanism to process the trace at runtime. GUARD handles this by encapsulating the trace generated by the CPU cores into signature. Compacting the memory access trace also helps in reducing the communication cost between the CPU and GPU cores. GUARD is also able to provide trade-offs among the two main characteristics of a data race detector: *performance* and *accuracy*. GUARD improves the performance of the data race detection algorithm by parallelizing the GPU kernel at different levels. By doing so, GUARD is able to perform data race detection at near-zero performance overhead. Additionally, to improve the accuracy of data race detection in the presence of signatures, we introduce a novel filtering mechanism that uses coherence state information in the cache line to filter out innocuous accesses. Overall, GUARD proves to be a highly customizable tool which can trade-off between speed and accuracy to achieve a particular performance-vs-precision goal.

2 Background and Related Work

When two threads access the same memory location without a separating synchronization, and at least one of the accesses is a write, there is a data race. A data race could lead to incorrect or unexpected program behavior, and is a potential security risk. An example of a data race is shown in Figure 1, where *thread0* and *thread1* access the same memory location *addr2*. This is a potential concurrency bug as thread0 could modify the value at address addr2 before its next intended use by thread1. Data races can be divided into three categories: (i) read-after-write (RAW); (ii) write-after-read (WAR); and (iii) write-after-write (WAW). A WAR data race condition is shown in Figure 1. Not all data races are hazardous or potential security risks; some of them could be benign. However, it is essential for data race detectors to identify and evaluate all potential data race conditions. Due to space limitations, we only discuss selected work that are closely related to GUARD in this section.

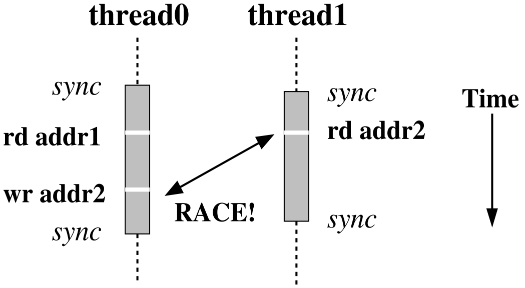


Fig. 1. An example of a data race, on address *addr2*, between *thread0* and *thread1*. Figure shows the synchronization and memory instructions in the two threads. Synchronization instructions define the beginning and end of *epochs*. Shaded boxes mark epochs in the instruction stream.

2.1 Data Race Detection Algorithms

There are two classes of data race detection algorithms: lockset-based [4,6] and happened-before-based [3, 5]. Lockset-based schemes track the set of locks held by threads while accessing a shared variable, and report a data race when the accesses are not protected by common locks. Happened-before (H-B) algorithm is based on Lamport's happened-before relation [10]. In this scheme, memory accesses between synchronizations are grouped into *epochs*. Epochs belonging to different threads are *concurrent* only if their execution times overlap. H-B schemes compare memory accesses in concurrent epochs and report a data race condition if they contain accesses to the same memory location with at least one of the access being a write. Lockset-based schemes do not track synchronizations other than locks, whereas, H-B scheme covers all types of synchronizations and hence can potentially detect more data races. For this reason, we use the H-B algorithm as the basis for our design. However, it is equally possible to use other data race detection algorithms, such as lockset, as the basis for GUARD.

2.2 Runtime Support for Data Race Detection

There exists a large body of work for identifying data races offline, either through static analysis [11] or by post-mortem analysis [12]. Static analysis based approaches analyze the source code to detect data race conditions and experience high false positive rates due to their conservative nature. Post-mortem methods that collect and analyze the execution trace of applications have significant storage overhead and cannot identify potential security risks in real-time. Due to these drawbacks, current data race detection techniques emphasize on runtime support.

Previous work has proposed utilizing hardware transactional memory (HTM) mechanism for data race detection. RaceTM [13] utilizes lightweight *debug transactions* to detect data races with the help of the conflict detection mechanism of the HTM. However, GUARD differs from such an HTM-based data race detection. RaceTM requires the underlying HTM support for operation, whereas, GUARD requires minimal hardware support for the extraction of memory access trace. The crux of GUARD's data race detection, the signature comparison, is performed by the general purpose GPU cores available on-chip. However, the memory access trace extraction mechanism can potentially be shared by several functionalities, including GUARD and HTM.

Concurrency bug detection tools for applications executing on GPU architectures have been proposed [14–16]. GUARD differs from these software-based mechanisms as it targets data race detection for CPU application, by utilizing on-chip GPU cores. A recently proposed work, KUDA [17], proposes to utilize GPU threads to improve the performance of data race detection on CPU threads. GUARD, however, differs from KUDA in several aspects. KUDA needs binary instrumentation and the help of additional CPU threads (worker threads) for the extraction of memory access trace. Additionally, the memory trace compression technique employed by GUARD helps in outperforming KUDA.

2.3 Instruction-Grain Program Monitoring

Instruction-grain program monitors have been proposed to efficiently extract runtime information from the CPU. These tools monitor programs at an instruction-level granularity and collect information such as program counter, instruction type, input/output operands, and access addresses. Such monitors have been used for specialized purposes such as memory checking, security tracking, and taint analysis [4,6,18]. Runtime data race detection requires extraction of memory access information from the CPU cores while the parallel applications are executing. General purpose instruction-grain program monitors such as Log-based Architecture [19] can efficiently extract runtime information from the CPU without significant hardware modifications. Previously, we have proposed utilizing hardware support for extracting runtime information for dynamic program execution monitoring [18]. GUARD utilizes a similar hardware *extraction logic* that tracks the program execution and extracts the execution trace of the CPU application being monitored. This extracted execution trace is then compressed into signatures and forms the input to the data race detection algorithm. GUARD's *Signature Generator*, described in the next section, is build on top of such previously proposed instruction-grain program monitors.

3 GPU Accelerated Data Race Detection

A snapshot of the basic GUARD mechanism is shown in Figure 2. The heterogeneous architecture we model consists of CPU and GPU cores connected to each other and their respective L2 caches through a common on-chip interconnection network (ICNT). Solid lines with double arrows indicate data communication paths between the cores and the caches through the interconnection network. Dotted lines indicate the flow of data race detection related information in GUARD. Features of the heterogeneous multicore processor modeled here are discussed in Section 4.

In GUARD, the memory access trace generation is orchestrated by a dedicated hardware component we refer to as the *Signature Generator* (SG). An extraction logic inside the SG extracts the memory access trace of the application executing on the CPU core. However, the volume of the trace generated at runtime makes it intractable to be processed in real-time, even with GPU. To reduce the storage, communication, and computation costs associated with managing these traces, they must be compressed before processing. SG utilizes Bloom Filter [20] to compress the extracted trace into signatures.

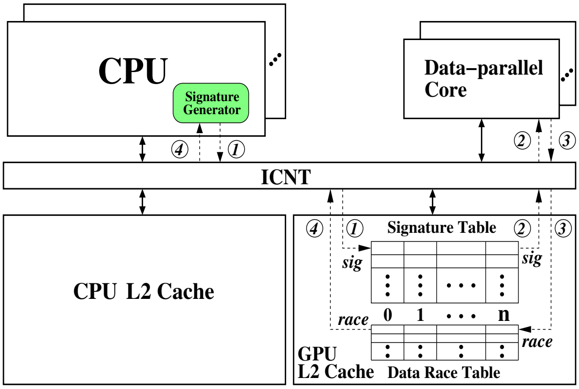


Fig. 2. GUARD mechanism is based on a heterogeneous multicore processor with CPU cores and Data-parallel accelerator (GPU) cores. *Signature Generator*, the only hardware modification to the baseline processor, is highlighted.

When GUARD is enabled for data race detection, a library function is invoked. It creates two data structures, the *signature table* and the *data race table*, in the GPU memory space. The SG is configured with the starting addresses of these tables. Henceforth, the SG is able to write generated signatures to the signature table and read flagged data race conditions from the data race table. It then launches the GPU kernel that performs the happened-before algorithm. In GUARD, the GPU cores work in tandem with the CPU cores to detect data races. We describe the CPU-side actions (① and ④ in Figure 2) and GPU-side actions (② and ③ in Figure 2) in detail in the next two sections.

3.1 CPU-Side Actions

Memory trace generated by each CPU core is partitioned into chunks called *epochs*. Synchronization instructions, such as lock/unlock, barriers, etc. define epochs. All the addresses belonging to an epoch are encapsulated into representative signatures using Bloom Filters and H3 hash functions [21]. For each epoch, the SG generates two signatures: a read (RD) and a write (WR). Once the signatures are generated, they are written to the signature table (action ① in Figure 2) stored in the GPU memory space. The signature table contains signatures from all CPUs, and forms the input to the H-B algorithm running on the GPU. It is a circular queue structure where the oldest *processed* entry for each processor is over-written by the latest entry. A flag is maintained for each signature entry indicating whether the entry has been processed by the GPU or not. The SG refers to this flag before the entry is over-written with a new signature, and resets the flag when a new entry is written.

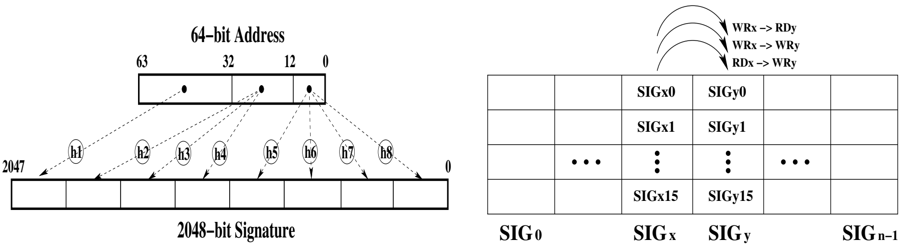
Once a data race is detected, the related information is written to the data race table by the GPU kernel and a notification is sent to the CPU in the form of an exception. An appropriate response such as rollback or replay is then initiated (action ④ in Figure 2). GUARD can utilize existing record/replay mechanisms [22] to perform this step. Efficient checkpointing systems such as Revive [23] can create checkpoints with low overhead. An appropriate checkpoint for rollback or replay is selected using information from the data race table. Further analysis could include detailed debugging to find out the exact memory location and instructions responsible for the data race. Information from the data race table and checkpoints could also be used to modify the thread scheduling to avoid the occurrence of data race conditions in re-execution.

Signature Generator. GUARD's only hardware addition, the SG, performs three key tasks: (i) extracting load/store information from committed instructions through an extraction logic; (ii) compressing the memory access traces into signatures using Bloom Filters; and (iii) forwarding signatures to the signature table. The extraction logic monitors the CPU application for load and store instructions and extracts the addresses accessed by these instructions. These load/store addresses are then compressed into respective RD/WR signatures using Bloom Filters.

The potential speed difference between the CPU application and the GUARD kernel means that the CPU could retire instructions faster than GUARD's ability to process them. This could lead to GUARD missing some instructions and consequently missing data race conditions. To avoid this, we design the SG on a feedback-based architecture where the CPU retire stage and the SG communicate through special registers. When GUARD is enabled, the CPU retire stage checks the SG state through the special registers and if SG is stalled, CPU pipeline is stalled to avoid missing any races. We evaluate the impact of this design on the performance of the CPU application in Section 5.1.

Signature Selection. Signatures are long bit vector structures used to encapsulate addresses in the memory access trace in a compressed form. Figure 3(a) shows the signature creation process used for a signature of size 2048-bits, divided into eight bins. The 64-bit address in the extracted memory access instruction is divided into three sections and these sections are passed through 8 different H3 hash functions, *h1* through *h8*, and a particular bit is set in each of the signature bins. Two signatures indicate a potential data race only when all the eight bins have at least one common bit set. Here, we analyze the effect of various signature parameters on its false positive rate. A *false positive* is defined as an incorrectly flagged data race condition due to two separate addresses mapping to the same signature bits. We observe that the false positive rates are 18.78%, 37.88%, and 89.86% for 2048-bit, 1024-bit, and 512-bit signatures respectively. Use of hardware signatures in data race detection has been explored by previous works [5, 6] and the false positive rates we observe are similar to the rates observed by them.

This false positive data is based on epochs that could contain up to 2000 individual instructions. Higher instruction count inside an epoch will lead to higher false positive rate for signatures of the same length. Ideally, an epoch is closed by a synchronization instruction. However, if there are no synchronization instructions within 2000 instructions, we forcibly close the epoch and write the signature to the signature table. This is a practical design choice as data race conditions between memory accesses that execute close to each other in time are the most critical, while those which occur far apart in time are potentially benign data races.



(a) Signature formation: A 2048-bit signature is divided into 8 bins and 8 different hash functions are applied on the 64-bit address to set the signature bits.

(b) Structure of the signature table for an n-core CPU. Curved arrows indicate the three required comparisons in H-B algorithm between every concurrent signatures of the CPUs X and Y.

Fig. 3. Signature formation process and the structure of the signature table

Signature Table Size: The difference in frequency of instructions in different cores could mean that an epoch in one core needs to be H-B compared with a significantly large number of concurrent epochs from another core. This, in turn, means that the H-B scheme would need a significantly large number of entry slots in the signature table to perform ideal data race detection without missing any concurrent epochs. However, in practical systems such as GUARD, we cannot afford such a large signature table. In addition to the larger signature table size, this will also lead to a greater performance penalty, as the data race detector will have to perform a significantly higher number of H-B signature comparisons. Limiting the number of entries in the signature table, on the other hand, inevitably leads to missing the comparison of some concurrent epochs; a parameter we refer to as *Missed Epoch Comparisons*.

In our experiments, we evaluate the missed epoch comparisons for a 16-entry and a 64-entry signature table compared with an ideal signature table with an infinite number of entries. We observe that the 16-entry signature table misses 3.16% of epoch comparisons, while the 64-entry signature table misses 0.12% of epoch comparisons, versus the ideal signature table. Since the 64-entry signature table incurs a significantly higher performance overhead compared to the 16-entry signature table, for a small improvement in missed epoch comparisons, we chose to evaluate GUARD with a 16-entry signature table.

Size of the signature table grows linearly with the number of CPU cores monitored and the number of signature entries. Even for a small core count and number of signature entries, this is high overhead to be constructed as a dedicated on-chip hardware structure. For example, a four-core CPU with 2048-bit signatures and 16 signature entries has a signature table size of 32 kilo bytes [5]. GUARD stores the signature table in the GPU last-level cache (LLC), without any additional hardware overhead. GUARD shares the LLC space with other GPU applications and hence the space is reusable. Designs that store the data race detection related information as an extension of the cache line, such as HARD [6], suffer from lost detection opportunities when the lines are evicted. GUARD does not suffer from this limitation as the signatures are not based on information in cache line extension.

Algorithm 1. The GPU kernel for the H-B comparison between CPU cores X and Y. SIGX and SIGY corresponds to signature from CPUX and CPUY. The algorithm is explained in Section 3.2. Custom kernel synchronization function __gpu_sync() and function *concurrent()* are discussed in Section 3.2.

```
Data: Memory access signatures (SIG).
Result: Flagged data race conditions.
while (1) do
    if isNew(SIGX) then
        for each current SIGY do
            if concurrent(SIGX,SIGY) && (SIGX ∩ SIGY ≠ NULL) then
                Flag data race condition;
            end
        end
        mark SIGX for graduation;
    end
    __gpu_sync();
end
```

3.2 GPU-Side Actions

The happened-before comparisons of the signatures generated by the SG is performed by a GPU kernel. In a nutshell, each signature from a particular CPU thread is compared with each concurrent signature from all other CPU threads. If the intersection of the concurrent signatures is not *NULL*, we have a potential data race condition. Figure 3(b) shows the signature table with entries for an *n*-core CPU. Entries for processors CPUX and CPUY are marked. Each entry in the signature table, say SIGX0, consists of a read (RDX) signature and a write (WRX) signature along with the epoch start (TS1) and end (TS2) timestamps. Since a read-after-read access is not potentially harmful, we have to compare only three signature combinations for CPUs X and Y: RDX-WRY, WRX-RDY, and WRX-WRY. These combinations are indicated in the figure. This signature comparison is extremely parallel and we utilize the data-parallel architecture of GPU to perform these comparisons.

Algorithm 1 shows the GPU kernel algorithm for GUARD. The GPU threads monitor the signature table for new incoming signatures. Once a new signature entry (SIGX) is identified, the GPU thread iterates through each of the current SIGY entries present in the signature table. Potential data race is identified if the *intersection* of concurrent signatures is not *NULL*. Timestamp information embedded in the signature is used to test the concurrency of these signatures using the function *concurrent()*. Bitwise AND operation is used to efficiently calculate the intersection of the signatures. When a data race condition is identified, information related to the race condition such as thread and epoch numbers is written to the data race table. Once the GPU thread has iterated through all the current SIGY signatures, it marks SIGX for graduation and moves on to the next SIGX. The signature marked for graduation can now be overwritten by CPUX with new signature.

GPU Kernel Parallelization. We map the H-B algorithm to the GPU in the following way: each GPU thread is assigned to perform signature comparison between two CPU threads X and Y. Each thread is also assigned a particular signature combination, among RDX-WRY, WRX-RDY, or WRX-WRY. To speedup the H-B data race detection algorithm, we parallelize the GPU kernel at different levels:

- Between two CPU threads X and Y, three different GPU threads are used to compare the three signature combinations (RDX-WRY, WRX-RDY, and WRX-WRY) in parallel.
- The current signature of CPUX could be compared with all 16 signatures of CPUY in parallel. We evaluate three levels of parallelization (*throttling*) for this: *full, half,* and *quart*. In *full* throttle, 16 different GPU threads are used to H-B compare the current signature of CPUX with the 16 signatures of CPUY in parallel. *Half* and *quart* throttle, on the other hand, use 8 and 4 GPU threads, respectively.
- We read the 2048-bit signatures in chunks of 64-bit UNSIGNED INTEGER data type for bitwise AND calculations for the intersection operation of the H-B algorithm. We further parallelize the GUARD's GPU kernel by utilizing different threads to perform the bitwise AND calculations on the different chunks of the same signature.

GPU Kernel Synchronization. The GPU kernel synchronizes all the threads after the comparison of the current SIGX with all the present SIGY entries, using a custom synchronization function __gpu_sync(). The current SIGX is then graduated before each thread moves to a new SIGX. This *lock-step* behavior ensures correctness of signature data accessed by GPU threads by avoiding untimely overwriting of SIGX by CPUX. Since GUARD's GPU Kernel could utilize several thread blocks, spread across multiple SMs, it is essential for __gpu_sync() to be able to synchronize across SMs. While the CUDA library function __syncthreads() [24] can only synchronize among threads in a block, __gpu_sync() utilizes a global mutex variable and atomic operations to synchronize among multiple SMs. __gpu_sync() is inspired by the GPU Lock-based Synchronization discussed by Xiao and Feng [25].

3.3 Coherence-Based Filtering

Use of signature to compress the memory access trace could lead to incorrect data race detection (false positive) as discussed in Section 3.1. GUARD compresses load (LD) and store (ST) addresses into separate read (RD) and write (WR) signatures of same size for comparison purposes. However, we observe that LD instructions generally outnumber ST instructions by ten to one. This means that LD instructions are the major source of false positive rate in GUARD. The false positive rate can be reduced by increasing signature sizes. However, this increases the signature table size and the signature comparison effort leading to significant performance penalty.

In this section, we discuss a novel coherence-based filtering mechanism that improves the accuracy of data race detection in GUARD. The filtering mechanism utilizes coherence state information to identify the LD instructions that access private and shared read-only addresses, and filters them out. This way, only LD instructions that access shared addresses modified by other threads are compressed into the RD signature.

These are the potential data race accesses and we call such addresses *shared-modified*. Since the impact of ST instructions on accuracy is very low, we do not apply any filtering on them. By filtering out innocuous LD instructions, we are able to bring down the false positive rate for GUARD without any negative impact on performance or data race detection capability.

We consider a memory hierarchy design with private L1 caches and a shared LLC which is common in current multicore processors. When the filtering mechanism is enabled, SG monitors the data response message from the LLC to check the *shared-modified* state. If the data was written to by another thread and is in modified state in the LLC, the shared-modified state is set by the LLC controller. When the state is set, SG concludes that this is a potential data race candidate and adds the address to the RD signature. Otherwise, the address is filtered out. The filtering mechanism considers the following three scenarios:

- L1 Hit: When a LD instruction hits the L1 data cache, the data is either private or shared read-only. Such an access will not cause a data race, and hence it is considered safe and the address is filtered out.
- L1 Miss & LLC Hit: When a LD instruction misses L1 data cache and hits the shared LLC, the LLC controller uses the coherence information to identify the state of the address. If the address was in a modified state prior to the load request, it was written to by another thread recently. Hence, this address is considered shared-modified and the corresponding bit is set in the response message.
- LLC Miss: If the access misses the shared LLC, it is potentially a cold miss or an access to the address after a long interval. Such accesses are considered safe as they will not cause a data race. Hence, the LLC controller resets the shared-modified bit and the address is filtered out.

These scenarios, however, could still experience a situation where the access could lead to a data race condition. In a potential write-after-read (WAR) race condition scenario, when the read instruction occurs at first, there is not enough information to make a decision on filtering. However, a future write to the same memory location by another thread in a concurrent epoch results in a potential data race. Hence, if this LD instruction was filtered out due to insufficient information, a potential WAR race condition could be missed.

This issue can be addressed by using temporary hardware signatures. For every thread, the filtered LD addresses from the current epoch are compressed and stored in temporary signatures. When a ST occurs (rather infrequent) in a thread, LLC controller sends invalidation messages to sharers and the cache line is set to *modified* state. The SG in these sharers compare the address in the invalidation message with the addresses in their temporary RD signature, and if there is a match, the address is added back to the thread's RD signature. However, only the addresses from the current epoch could be saved as the previous epochs would have already been dispatched to the GPU for data race detection. Also, limited capacity of the LLC or time gap between the two instructions could lead to the related cache line being evicted from the shared LLC. The scheme will then filter out the LD instruction, due to lack of information in the LLC. However, it should be emphasized here that the most crucial data race accesses are the ones that occur in close proximity, and those are unlikely to be filtered out due to this limitation.

SG picks up the shared-modified state of the address from the cache response message from the LLC controller. Since the SG is private to a core, it need only monitor coherence messages destined to the local first-level cache. Only a single additional *shared-modified* bit is required to pass this information. The filtering mechanism does not alter the cache coherence scheme in any way, which is desirable as they are highly optimized designs. The temporary signatures will be the size of a RD signature per core, which is 256 bytes for a 2048-bit signature. Prior work [26] has proposed an algorithm that uses coherence state information to detect data races. Also, software-based data race detection mechanisms [27] have employed techniques to filter stack and duplicate addresses to improve performance. However, to the best of our knowledge, this is the first work to utilize a coherence-based filtering technique to improve the accuracy of a data race detection tool that already works at near-hardware speed.

4 Evaluation Infrastructure

In spite of the recent heterogeneous designs [7–9], some of which are already in the market, the optimal design of a multicore CPU with on-chip data-parallel cores is still unclear. The memory hierarchy design and shared memory consistency models are ambiguous and the programming model is still in its nascent state. Nevertheless, such designs provide a suitable infrastructure to off-load the task of CPU data race detection to on-chip accelerator cores. In this work, we describe a generic execution model and propose a data race detector inspired by these designs.

4.1 Heterogeneous Execution Environment

We utilize a heterogeneous multicore processor, consisting of CPU and GPU cores on the same die, as shown in Figure 2. The cores and their respective LLCs are connected through a common on-chip interconnection network. Communicating through the shared on-chip interconnection network improves the efficiency of GUARD. These cores work on different address spaces and hence we do not consider the complexities of coherence between CPU and GPU cores in our design. We base our evaluation on a GPU SM, with 8 SPs, that can each support up to 1024 threads. This is modeled on Nvidia Geforce® 8600 GTS. Various parameters of the CPU and GPU cores simulated are given in Table 1.

To simulate multicore CPU in detail, we use Simics [28] combined with GEMS [29]. The GPU cores are simulated using GPGPU-sim [30]. The on-chip interconnection network is simulated using Garnet [31]. GUARD GPU Kernel is compiled using CUDA 2.3 [24]. We evaluate GUARD with applications from two widely used benchmark suites: PARSEC [32] and SPLASH-2 [33]. Our evaluation reports data from 15 programs in total: seven PARSEC and eight SPLASH-2 programs as indicated in Table 2.

Using Simics and GEMS, we simulate a many-core system with Sun Microsystem's UltraSPARC® III processor running Solaris® 8 operating system. All the benchmark programs are written in C/C++ and parallelized using either OpenMP or Pthreads. They are compiled using GCC 4.5.2 at -O3 optimization level. The reported results are based on running the selected benchmarks for 1 billion instructions in total from the

Table 1. System configuration parameters for the heterogeneous CPU-GPU evaluation infrastructure

CPU		**GPU**	
Cores	4 / 8 / 16 / 32	Warp Size	32
Frequency	2600MHz	Frequency	1300MHz
Pipeline Width	4	SIMD Pipeline Width	8
L1 Cache (Size/Assoc/Line)	32KB / 2 / 64B	L1 Cache (Size/Assoc/Line)	32KB / 2 / 64B
L2 Cache (Size/Assoc/Line)	2MB / 4 / 64B	L2 Cache (Size/Assoc/Line)	512KB / 4 / 64B
RoB / IW Size	64 / 32	Shared Memory per Core	16KB
MSHR / TLB Entries	256 / 64	Threads / Registers per core	1024 / 16384
L2 / DRAM Access Latency	6 / 200 Cycles	Memory Channels	8

start of their respective parallel sections, also known as the *region of interest*. Full system simulation is extremely time consuming, and therefore it is practical to simulate 1 billion instructions in the region of interest. We observe that GUARD's ability to detect data race conditions and its performance characteristics are comprehensively evaluated by simulating 1 billion instructions in the region of interest.

Cycle accurate simulators are utilized to evaluate the performance impact of GUARD on the CPU application being monitored. GUARD GPU kernel invocations, data transfer operations and signature comparison operations are simulated in a cycle accurate manner. We enable L1 data cache in the GPU to improve the performance of GUARD kernel. Potentially, we could also make use of the GPU shared memory to store the signature table. However, we utilize the L1 data caches as the access times are similar. Shared memory in the GPU is explicitly managed by the programmer and when the signature table is updated at regular interval by the CPU, copy of the signature table in shared memory will also need to be manually updated. This will prove to be an additional overhead when using GPU shared memory.

5 Evaluation

This section performs a detailed evaluation of the effectiveness of GUARD. First of all, we look at the effectiveness of our scheme in detecting data races. Then, we move on to the performance characteristics of GUARD. We also discuss the performance-accuracy trade-off achievable, given the limited on-chip resources available.

Table 2 shows the number of data races GUARD detects. GUARD is based on the happened-before principle that has been used by prior work such as SigRace [5], and thus is expected to capture the same set of data races. It is worth pointing out that similar to SigRace, GUARD does not capture all potential data races. The set of data races captured are only those that lead to violation of happen-before principal at runtime. GUARD works at address level granularity and hence each data race reported corresponds to a unique address. The ability to detect actual data races proves the effectiveness of GUARD. Some of the data race conditions reported here are benign, harmless, or intended race conditions. However, it is essential for a concurrency bug detection tool to report all potential bugs and let the programmer make a decision on its severity.

Table 2. Number of data race conditions detected by GUARD

Parsec	Races	Splash − 2	Races
blackscholes	1	barnes	2
bodytrack	0	cholesky	2
canneal	1	fft	4
fluidanimate	4	lu	2
freqmine	0	ocean	0
streamcluster	7	radiosity	2
swaptions	1	raytrace	1
		waterNS	0

5.1 Performance-Accuracy Trade-Offs

Although massively parallel, signature comparison based data race detection involves significant amount of computational work. If not properly managed, it could slow down the data race detection process and, in turn, stall the CPU application. Here, we analyze the performance cost of GUARD and the performance-accuracy trade-offs we could make. In particular, we look at two main parameters of GUARD, *signature size* and *throttling*:

- We consider three signature sizes in our experiments: 2048-bits, 1024-bits, and 512-bits. The maximum size of an epoch is limited to 2000 instructions. The false positive rate increases with decreasing signature size as discussed in Section 3.1.
- We consider three levels of parallelization (*throttling*) as discussed in Section 3.2: *full, half,* and *quart*. For an n-core CPU, the number of GPU threads required for GUARD throttling at T grows at the rate of $\mathcal{O}(n^2 * \mathrm{T})$.

Figure 4 presents the performance-accuracy trade-off characteristics of GUARD for a 4-core CPU. The performance overhead (in bars) is evaluated as the slowdown (% increase in cycles per instruction) of the CPU application being monitored with GUARD, over its native execution. The values shown are average (geometric mean) of all the 15 benchmarks we evaluated. The accuracy (in lines) is evaluated as the false positive rate (% of data races reported that are false) for the signature size used in GUARD. In this section, we consider false positive rate without any filtering mechanisms (w/o Filter). We discuss the filtering mechanism (w/ Filter) later in this section.

We observe that the difference in throttle level is well pronounced in the results. For any particular signature size, *full* throttle performs better than *half* throttle which in turn performs better than *quart* throttle. This is expected as the data race detection algorithm is extremely parallel and with more GPU threads assigned, better performance is obtained. Similarly, for any particular throttling, the performance of GUARD improves with decreasing signature size as GPU kernel has less signature comparisons to perform. However, this performance improvement is accompanied by increase in the false positive rate. We observe that at *full* throttle, we are able to achieve near-zero performance overhead for data race detection on a 4-core CPU. Furthermore, by scaling the number of GPU cores employed for data race detection, GUARD is able to perform data race

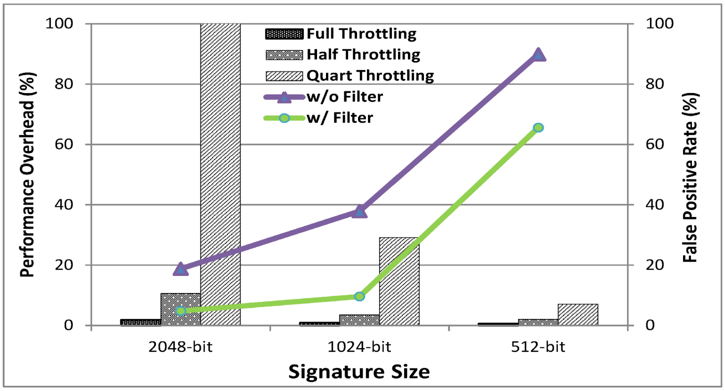


Fig. 4. Performance and accuracy characteristics of GUARD. The graph shows (in bars) the slowdown (%) of application being monitored for different throttling levels and signature sizes. The graph also shows (in lines) the false positive rate (%) for different signature sizes used in GUARD.

detection for 8-core, 16-core, and 32-core CPUs with near-zero (less than 2%) performance overhead at *full* throttling. Table 3 shows the amount of GPU resources required to perform data race detection, for different CPU configurations, at different throttling.

Table 3. Number of GPU SMs required for data race detection, for different CPU core count, at different throttling. The GPU SM architecture is described in Section 4

CPU Core Count	quart throttling	half throttling	full throttling
4	1	1	1
8	1	2	3
16	3	6	12
32	12	24	48

On detailed analysis of the performance of the GPU kernel, we observe that the performance overhead of GUARD is mainly due to two reasons: (i) data accesses related to the long signatures; and (ii) synchronization of the hundreds of threads used for H-B comparisons. GUARD's GPU kernel stalls only for about 1.54% of its execution cycles due to unavailability of data in any threads (memory related stalls). We see that the signature table size is small enough to fit inside the GPU L2 cache. For a reasonable GPU L1 data cache size, as in Table 1, the L1 data cache hit rate is more than 99%. We also observe that GPU does a good job of *coalescing* memory accesses and limiting the impact of data access latency on the performance of GUARD. Thread synchronizations, on the other hand, are necessary for the correctness of H-B algorithm when mapped to a highly parallel architecture like GPU.

Kernel Parallelizations. We observe that not all parallelization opportunities discussed in Section 3.2 work equally well. In addition to throttling, we also discussed utilizing multiple threads to compare the chunks inside each signature. While we observe that throttling has a significant impact on the performance of GUARD, the signature chunk-level parallelism does not improve the performance significantly. When utilizing chunk-level parallelism, each GPU thread performs a very short computation (comparing two 64-bit unsigned integers) which does not yield significant benefits. Additionally, the overhead of managing a high number of GPU threads is not recovered by the short 64-bit comparison. This indicates that the H-B algorithm used in GUARD benefits more from coarse-grained parallelism than from fine-grained parallelism.

Customizable Design. The high performance of *full* throttle mode is obtained at the cost of utilizing larger amount of on-chip GPU resources as shown in Table 3. If on-chip resources are constrained, we could also select a smaller signature size and still achieve better performance for the same level of throttling as shown in Figure 4. However, this will be achieved at the cost of higher false positive rate. GUARD allows customizing either of these parameters, signature size, or throttling, to achieve the performance goal we set for a particular accuracy constraint. This level of performance-accuracy customizability is hard to achieve in hardware-based data race detection mechanisms.

Coherence Filtering. In Section 3.3 we introduced a novel coherence-based filtering mechanism to reduce the false positive rate of data race detection using signatures. Here, we evaluate the impact of the coherence-based filtering on GUARD. The coherence-based mechanism filters 93.6% of all LD instructions, which results in filtering out accesses to 96.56% of unique addresses. With filtering, the false positive rate drops significantly as shown (w/ Filter) in Figure 4:

- from 18.8% to 4.8% for 2048-bit signatures
- from 37.9% to 9.6% for 1024-bit signatures
- from 89.9% to 65.6% for 512-bit signatures

Additionally, the filtering mechanism achieves this improvement without missing any data race conditions in our experiments. Thus, coherence-based filtering proves to be very efficient in improving the accuracy of GUARD. Our evaluations are based on MOSI coherence protocol. However, the filtering mechanism can easily be adapted to other coherence protocols. With filtering, false positive rate for 1024-bit signature is now under 10%. Hence, *half* throttling with 1024-bit signatures can be utilized to run GUARD with negligible performance overhead, reasonable accuracy, and low GPU utilization. This is particularly attractive for CPUs with higher number of cores as the GPU resources required to perform data race detection at *full* throttling can become quite large as shown in Table 3.

Bandwidth Utilization. Signature transfer between CPU and GPU consumes on-chip bandwidth. For a 2048-bit signature, we observe that GUARD utilizes less than 15% of the on-chip bandwidth provided by current designs [7] to transfer signatures. This bandwidth utilization can further be reduced by using additional hardware to compress the signatures [5] before transferring through the on-chip interconnection network.

Effect on GPU Applications. GUARD shares the GPU computational power with other GPU applications. Hence, while GUARD is enabled, other applications will have less GPU resources available and their performance could suffer. GUARD, however, is envisioned as a runtime tool that is exclusively used for debugging purposes and not for continuous usage while other applications are utilizing GPU resources. Hence, the impact of GUARD on the performance of other GPU applications is minimal.

Supporting Thread Migration and Simultaneous Multithreading. Thread migration in a multicore processor enables application threads to migrate from one core to another. GUARD can support thread migration as the signature table entries correspond to a thread, and are not tied to any particular core. When a thread migrates from a core, the current signature is forcibly closed and transferred to the signature table for data race detection. Additionally, GUARD can handle parallel applications utilizing more number of threads than the number of cores present in the processor. Since the signature table is stored in memory, instead of dedicated hardware, GUARD is able to adapt to the number of threads utilized by the application. This capability also lets GUARD support simultaneous multithreading.

Hardware Support. In baseline GUARD, the only additional hardware support required is the SG. We build SG on top of well studied generic instruction-grain program execution monitors [18, 19] that is used for efficient extraction of execution trace. Bloom filter hardware is used to compress the extracted traces into signatures. Hardware buffers are used to temporarily store the signature while an epoch is being created. For a 2048-bit signature, combined RD/WR signature size will be 512 bytes per core.

6 Conclusions

As the integration of data-parallel accelerator cores onto the modern multicore processor becomes common, it is desirable to be able to utilize this computing power for enhancing non-performance aspects of parallel execution. Concurrency bug detection, particularly data race detection, assumes increased importance in the current landscape of parallel computing. In this paper, we design, implement, and evaluate a GPU Accelerated Data Race Detector (GUARD). GUARD utilizes GPU cores available on-chip to perform data race detection for the multithreaded applications running on the CPU cores. The GPU cores are employed for data race detection when they are not being utilized for performance acceleration of applications. This paper proposes several optimizations each allowing a different trade-off between performance and accuracy of data race detection: (i) accelerating CPU data race detection utilizing available on-chip data-parallel cores; (ii) compressing generated memory traces, using Bloom filters, to drastically reduce the computational requirement; and (iii) filtering out innocuous memory accesses, using coherence state information, to improve the accuracy of signatures.

Using a single GPU core (SM architecture described in Section 4), GUARD performs data race detection on a 4-core CPU with 1.8% performance overhead and 18.8% false positive rate. Coherence-based filtering mechanism reduces the false positive rate by nearly 75%, without missing any data race conditions. Furthermore, by scaling the number of GPU cores employed for data race detection, GUARD is able to perform data race detection for 8-core, 16-core, and 32-core CPUs with near-zero performance overhead. With minimal hardware support, GUARD can be invoked for data race detection with negligible performance impact. Overall, GUARD proves to be a powerful tool in the parallel programming environment, necessitated by the emergence of many-core processors, and facilitated by the development of heterogeneous architectures with on-chip data-parallel cores.

Acknowledgments. This work is supported in part by National Science Foundation grants CCF-0916583 and CPS-0931931. We would like to thank all anonymous reviewers for their constructive comments that helped to improve the quality of this paper. We would also like to thank Jieming Yin and Ragavendra Natarajan for their suggestions to improve the paper.

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (2008)
2. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (2010)
3. Intel Corporation: Intel Thread Checker, `http://www.intel.com`
4. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (1997)
5. Muzahid, A., Suárez, D., Qi, S., Torrellas, J.: Sigrace: signature-based data race detection. In: International Symposium on Computer Architecture, ISCA (2009)
6. Zhou, P., Teodorescu, R., Zhou, Y.: Hard: Hardware-assisted lockset-based race detection. In: International Symposium on High Performance Computer Architecture, HPCA (2007)
7. Brookwood, N.: AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. Advanced Micro Devices(AMD) White Paper (2010)
8. Intel Corporation: Intel Sandy Bridge Microarchitecture, `http://www.intel.com`
9. NVIDIA Corporation: NVIDIA Project Denver, `http://goo.gl/i2Q3Z`
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of ACM (1978)
11. Engler, D., Ashcraft, K.: Racerx: effective, static detection of race conditions and deadlocks. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (2003)
12. Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B.: Detecting data races on weak memory systems. In: Proceedings of the 18th Annual International Symposium on Computer Architecture (1991)
13. Gupta, S., Sultan, F., Cadambi, S., Ivancic, F., Rotteler, M.: Using hardware transactional memory for data race detection. In: IEEE International Symposium on Parallel Distributed Processing, IPDPS (2009)
14. Boyer, M., Skadron, K., Weimer, W.: Automated dynamic analysis of cuda programs. In: Third Workshop on Software Tools for MultiCore Systems (2008)
15. Hou, Q., Zhou, K., Guo, B.: Debugging gpu stream programs through automatic dataflow recording and visualization. ACM Transactions on Graphics, TOG (2009)

16. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Grace: a low-overhead mechanism for detecting data races in gpu programs. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (2011)
17. Bekar, U.C., Elmas, T., Okur, S., Tasiran, S.: KUDA: GPU accelerated split race checker. In: Workshop on Determinism and Correctness in Parallel Programming, WoDet (2012)
18. He, G., Zhai, A.: Improving the performance of program monitors with compiler support in multi-core environment. In: IEEE International Symposium on Parallel Distributed Processing, IPDPS (2010)
19. Chen, S., Falsafi, B., Gibbons, P.B., Kozuch, M., Mowry, T.C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G.R., Lin, B., Schlosser, S.W.: Log-based architectures for general-purpose monitoring of deployed code. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (2006)
20. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of ACM (1970)
21. Carter, J.L., Wegman, M.N.: Universal Classes of Hash Functions. In: ACM Symposium on Theory of Computing (1977)
22. Xu, M., Bodik, R., Hill, M.: A ”flight data recorder” for enabling full-system multiprocessor deterministic replay. In: International Symposium on Computer Architecture, ISCA (2003)
23. Prvulovic, M., Zhang, Z., Torrellas, J.: Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: International Symposium on Computer Architecture, ISCA (2002)
24. NVIDIA Corporation: NVIDIA CUDA C Programming Guide, `http://www.nvidia.com`
25. Xiao, S., Feng, W.C.: Inter-block gpu communication via fast barrier synchronization. In: 2010 IEEE International Symposium on Parallel Distributed Processing, IPDPS (2010)
26. Gonzalez-Alberquilla, R., Strauss, K., Ceze, L., Piñuel, L.: Accelerating data race detection with minimal hardware support. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 27–38. Springer, Heidelberg (2011)
27. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (2006)
28. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer (2002)
29. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset. SIGARCH Computer Architecture News (2005)
30. Bakhoda, A., Yuan, G., Fung, W., Wong, H., Aamodt, T.: Analyzing cuda workloads using a detailed gpu simulator. In: International Symposium on Performance Analysis of Systems and Software, ISPASS (2009)
31. Agarwal, N., Krishna, T., Peh, L.S., Jha, N.: GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator. In: ISPASS (2009)
32. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: International Conference on Parallel Architectures and Compilation Techniques, PACT (2008)
33. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: characterization and methodological considerations. In: International Symposium on Computer Architecture, ISCA (1995)

Efficient Model to Query and Visualize the System States Extracted from Trace Data

Alexandre Montplaisir, Naser Ezzati-Jivan,
Florian Wininger, and Michel Dagenais

Ecole Polytechnique de Montreal
{firstname.lastname}@polymtl.ca

Abstract. Any application, database server, telephony server, or operating system maintains different states for their internal elements and resources. When tracing is enabled on such systems, the corresponding events in the trace logs can be used to extract and model the different state values of the traced modules to analyze their runtime behavior. In this paper, a generic method and corresponding data structures are proposed to model and manage the system state values, allowing efficient storage and access. The proposed state organization mechanism generates state intervals from trace events and stores them in a tree-based state history database. The state history database can then be used to extract the state of any system resources (i. e. cpu, process, memory, file, etc.) at any timestamp. The extracted state values can be used to track system problems (e. g. performance degradation). The proposed system is usable in both the offline tracing mode, when there is a set of trace files, and online tracing mode, when there is a stream of trace events. The proposed system has been implemented and used to display and analyze interactively various information extracted from very large traces in the magnitude order of 1 TB.

1 Introduction

Tracing of computer systems allows programmers and administrators to extract useful data about the runtime behavior of their systems or applications. From a high-level point of view, the concept of tracing relies on inserting trace points, or probes, at specific places in a program's code. Whenever execution reaches those points, an event about reaching this location is sent to the tracer. The LTTng tracer (Linux Tracing Toolkit Next Generation) [DD08, DD06] was developed by the DORSAL lab in the Ecole Polytechnique de Montreal university. This tracer integrates with the Linux kernel, and allows kernel and user-space tracing.

Although the trace data provides valuable information from system runtime execution, the event generation rate can be quite large, especially in a multi-core system and for a detailed kernel execution trace (system calls, interrupts, etc.). Extracting, saving and analyzing this information without impacting the running system is a serious challenge.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 219–234, 2013.

There are several tools available to visualize and analyze such traces. Viewers like LTTV (Linux Tracing Toolkit Viewer)[1], TMF (Tracing and Monitoring Framework, an Eclipse plugin part of Linux Tools)[2], Jumpshot [ZLGS99] or Triva [SHN09] give a graphical representation of different runtime aspects (cpu usage, memory consumption, file accesses, critical path analysis, etc.) of the system under study using the collected trace logs.

While it is being traced, a process, application or individual component may have different execution states. For instance, the state of a process may be changed subsequently between new, ready, waiting, running or dead. Efficiently managing the different state values for the different system attributes (the term "attribute" is used here to describe the system resources and modules) and making them quickly available to the administrators and monitoring tools at any requested time, can be used to better comprehend the execution or track system problems, as used in [CZG$^+$05, CGK$^+$04]. For example, suppose a problem or attack is detected, or a performance degradation is reported. In these cases, having the states of the important system resources (e. g. what are the running processes, what files are opened, which CPUs are scheduled, how many bytes are read or written through network devices) at the reported time can help administrators to understand better the problems and possibly to find their root cause.

Re-reading and re-running the trace events or checkpoint method (Figure 1) can be used to manage the different state values of the system, as used in TMF and LTTV viewers [Mon11]. However, by having a large number of system attributes and a large trace duration (traces up to several hours and in terabyte range), these solutions may not be efficient or scalable to extract the values of any given attribute at any given time. As an example, assume a trace viewer aimed to display a histogram of some metrics such as the number of interrupts (as a defined system attribute) within a 1 TB trace. To do so, the viewer may extract the values at 100 different points (corresponding to the number of available pixels of the graphical view), and for each point reading of a 10 GB (= 1 TB / 100 points) length of the trace is required. However, rereading such a large section of the trace would clearly be unacceptable for an interactive browsing. This is where an efficient state history database may greatly help.

The main contribution of this work consists of a generic, scalable and efficient tree-based state history data structure, and corresponding algorithms to store, manage and retrieve the different state values for an arbitrary trace size, for any number of system resources. The method works by building incrementally a data structure to store the state history as it sequentially reads the trace events. The state values, extracted from the trace events, are stored as intervals in this state history database. The genericity comes from the fact that the method does not hard-code the state definitions, neither in the viewer modules nor in the tracer. This makes it possible to support different trace formats, and allows defining different attributes, state variables and values. For scalability purposes,

[1] `http://lttng.org/LTTV`

[2] `http://www.eclipse.org/linuxtools/projectPages/lttng/`

the method uses a disk-based data structure and does not impose limitations on the input trace size. Using the tree-based structures, the history database also offers a fast query response time (O(log n)).

The remainder of the paper is organized as follows: after investigating the background and related work, we present the architecture of the method, and the details of its modules. Then, we discuss the implementation, visualization and also evaluation and experimental results of the method. Finally, we conclude and outline the possible future work.

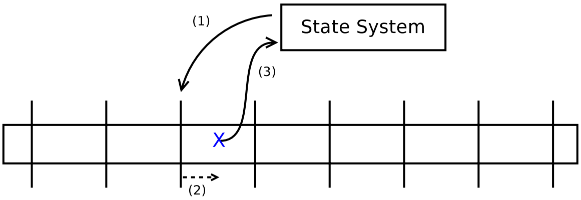


Fig. 1. The process of resolving a query with the checkpoint method. (1) loading the nearest previous checkpoint value, (2) replaying the trace events and updating the state value, (3) returning the current state in response to the query.

2 Background

Modeling the system attributes and corresponding state values allows quickly responding to queries about the system runtime behavior. In this research, we use events and state values extracted from LTTng kernel tracer [DD08]. Colleagues have shown [GDG$^+$11, EJD12] examples of information and state values that can be extracted from LTTng trace events. It would be technically possible for LTTng to generate the state model at the same time as tracing, and save the resulting information in the same trace events. However, this method may cause problem: LTTng is presented as a high-performance tracer [DD08], with very low impact on system behavior when tracing is enabled. The computation of the state while tracing would increase the overhead.

To manage the state values, an inefficient method could be rereading and replaying the trace, from the beginning to the requested query time, and updating and extracting the required state values. Although this method can be used for small traces and for specific usages (e. g. no space on disk for indexes), it is slow and the delay increases with the position within the trace.

To improve the overall access time, some tools and viewers (e. g. LTTV and TMF) use checkpoint method to store and manage the state vales, which consists in saving in memory or disk at regular intervals (e. g., every 50,000 events or every 10 minutes of tracing), a complete snapshot of all system state values. When users request the state value of a specific resource at any arbitrary time t of the trace, the method loads the snapshot from the nearest previous checkpoint (t_0) and starts rereading and replaying the trace events from that point and updates the state values for events whose time value is greater than or equal to t_0 but lower than t. Then, the resulting updated state is returned as response to the user. Figure 1 represents a trace and its various snapshots at checkpoints. It also shows the process of resolving a user query for the time indicated by X.

The checkpoint method avoids having to reread the trace from the beginning every time. Using such a method works well enough for small traces (tens or hundreds of MB). However, it brings some storage scalability issues, since the snapshot storage space grows with both the number of state elements and the trace duration. There is some wasteful redundancy arising from the fact that some information may not change from one checkpoint to another and yet is repeated in each snapshot. In addition, the size of each snapshot increases on a larger system, if there are more processes running and more activity at the same time. If these checkpoints are stored in RAM, there is a hard limit on the size of the trace that can be analyzed. It could, up to a certain point, reduce the number of checkpoints, but that would degrade the query performance of the state system.

A dynamic checkpoint method is proposed in [EJD13], which uses dynamic intervals, instead of a fixed number of events, for storing the snapshots. Although it utilizes the memory better than the previous solution, but still encompasses the other potential problems of the checkpoint method.

Another possible alternative is to store state information directly in the trace. This does not replace the need for state storage, but avoids the viewer having to define the state changes associated with events. This approach is used by [CGL08]. Their traces contain both specific events and states with associated time interval. They display the events and states with the Jumpshot tool [ZLGS99] which clearly shows the communication between MPI processes. Our work focuses on the management of state information at the viewer level, not in the tracer. The generation and processing of this information, during tracing, however, may be explored in a future work.

The proposed state history database stores different intervals representing the state value changes. Several data structures have been presented in the literature to store and manage the records of intervals [GG98] like segment-tree, interval-tree, Hb-tree, R-tree (and its many variants). These solutions are mostly designed to work inside the main memory, rather than the disk, which threaten the scalability and can not be used well for very large traces (in the range of hundreds of gigabytes). Even by forcing and changing the above methods to store the data in disk, a problem still exists. These data structures work properly for static data sets, but not for the incrementally built intervals. For instance, the R-Tree has to be constantly rebalanced to cope with the incrementally received intervals which requires a large number of nodes splitting and merging (re-balancing) operations. This resulted in very long history construction times, which makes them inappropriate to use in trace viewers where the data is received incrementally.

Another disk based interval management method, SLOG File Format, is proposed by Chan et al. [CGL08], which is a hierarchical r-tree based file format to store drawable trace objects (events, states, message arrows and so on). It stores the objects belong to the same time buckets in the same blocks. This solution works well for the traces with uniform event distribution. However, in cases where there exists unbalanced traces, in which some parts have more operations and events than other parts, the tree will be unbalanced and some blocks will be larger than others, which results in having different response times for different trace areas. However, our solution does not fix the size of the time buckets and creates more blocks for the busy areas which makes the response time almost the same for different parts of the trace.

3 Modeled State System Architecture

The general idea of the solution is to extract and record incrementally the intervals of different state values for system resources (processes, CPUs, disks, etc.) from trace events. Tree-based organization is used to store the state values. Figure 2 depicts the architecture of the system which shows the different components and their interactions during the construction of the data structure. From now on we use term "modeled state system" to refer to the architecture shown in Figure 2.

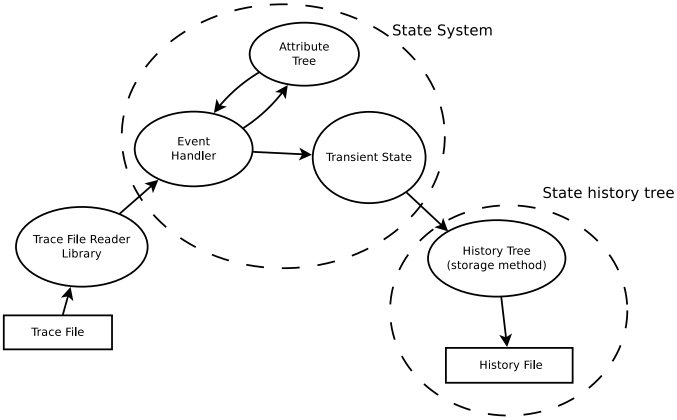


Fig. 2. Modeled state system architecture

The modeled state system contains two important parts: the current state and the state history tree. The current state manages the ongoing state values for the current time of the trace, while the state history tree encompasses the past state values of the system attributes. The following sections describe the different components of the system.

3.1 State Provider

The *state provider* computes the effects of an event on the current state and converts the trace events into *state changes*. It observes the type and contents of all events in the trace one by one (using a `switch/case` for example), and updates the modeled state accordingly.

It is possible for events to cause several state changes, or no change at all (n-to-m relationship). For instance, a file_close event changes the state of the corresponding file to "closed", and linux_sched_switch (p1, p2) event changes the states of two processes, one to "waiting" and the other to "running", respectively. The state provider module uses a mapping table which contains different mapping rules, from a simple "if x then y" rule to complex state transition patterns, depending on the type of event and its effects on the status of the system resources.

Therefore, the state provider module needs to know, for example, the types of events that will be presented in the trace, their names and contents. This is the only part of the modeled state that is platform-specific, and should be customized for any new trace format. A possible solution to make this part also generic is to inject the mapping table dynamically, e. g. though an XML file. This will be investigated as a future work.

The state provider should also be aware of the various outputs of other analysis modules. For instance, higher level synthetic events, which are typically created using different trace abstraction techniques [EJD12], may in turn be used to further modify the system state. For example, several low level events may be used to create a "brute-force attack" abstract event, which could change the state of the system to "compromised".

The state provider is a critical part of the system. It is called for each event in the trace and updates the state database for each change. It is therefore important that it is optimized. One method is to avoid recording redundant or unnecessary state changes. Another method, the partial history update, is to update the state database for a group of state changes (each 10 state changes), instead of at each state change [EJD13]. This significantly reduces the amount of information to be recorded. The impact of the state provider on system performance, using both the complete and partial history updates, will be shown in the experimental results section.

3.2 Attribute Tree

We define a general term, "attribute", as any basic unit of the modeled state system, which can contain only one state value at any given time during the execution. An attribute can be basically anything, as it is defined by applications. For example, "name or ID of the process scheduled on CPU 0", "the name of an open file" or "the number of active processes" could be attributes in the model. With this definition, each attribute can be used to describe a system resource, but a resource may also have several attributes to describe its different aspects, e. g. parent process id of a process, exec_name of a process, fd (file descriptor) of a file, address of a socket, and so on.

To manage a large number of different but somehow related attributes, we can organize them in a tree, called the *attribute tree*. The attribute tree is an in-memory organization of the attributes of the system resources. One could keep everything in one level for very simple systems. For complex systems, as the number of attributes increases, placing them in a tree provides a better structured organization and, depending on the type of queries, better performance. An example of an attribute tree is shown in Figure 3.

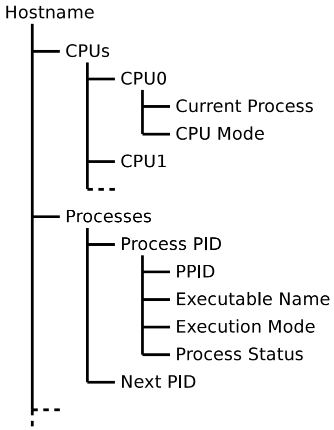


Fig. 3. An example of an attribute tree

In the attribute tree, there is a unique access path for each attribute from the root node. We can compare this arrangement to the files and directories in a filesystem. Directories correspond to attribute access paths, and filenames correspond to the names of attributes. The content of the file represents the different state values of that attribute. Unlike a regular filesystem where you have either directories or files, here attributes are both and can simultaneously link to children and contain a value. This behavior is similar to that of the ZFS file-system[3].

Attributes are considered to be static and created on-the-fly. A new CPU could be hot-plugged, which would add an entry in the tree, but those entries cannot be removed (if a CPU is "hot-unplugged", the entry will simply not be used anymore). However, their values may be changing often as the system runs. It is important to note that no state value (attribute value modification) is stored in the attribute tree. The purpose of the attribute tree is to give a structure to the resources and attributes. Each attribute in this tree has a pointer to the state values in another database in which all current and previous values are stored. For example, the attribute "current process scheduled on CPU 0" may have different values during the system execution, which are stored in the state database and not in the attribute tree.

[3] http://en.wikipedia.org/wiki/ZFS

3.3 Transient State and History State

When a state change is reported by the state provider, a transient state record is created for the requested attribute and stored in memory, in the (current) state system. Each transient record contains an attribute name, a time and a state value. If an entry currently exists for this attribute (i. e. there was already a state value for this particular attribute) then the current value is *replaced* by the new value. However, the previous value is kept and stored in the state history, to be available in subsequent enquiries. To do so, we first create one *interval record* from the available information containing and attribute (from the attribute tree), the old status value, the old value of time (interval start time) and *new* time value (interval end time). Start and end times represent the bounds of the period for which the state value is valid for the corresponding attribute.

The completed interval can now be inserted into the state history to become part of the history. This process is continuing until the end of trace, during which we obtain a set of completed intervals (i. e. state history values) for each attribute.

As mentioned earlier, the history state values are stored in a tree-based structure named "state history tree". This set of data will be used later to navigate through the state history for different time values and to reason about the run-time behavior of the system resources.

3.4 History Tree

Finally, the biggest piece of the puzzle, the State History Tree is a data structure to store intervals, optimized to be stored on a rotational disk. It is by no means balanced, so there is no concept of re-balancing the tree. Its two main components are intervals, as mentioned in the previous section, and nodes. The nodes are the direct containers for intervals. There is a configurable size (e. g. 32KB, 64KB, 1MB,...) for the nodes. Each node also keeps track of all its children and parent.

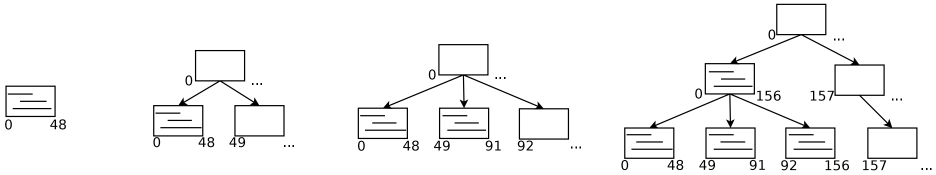


Fig. 4. The process of incrementally building the history tree

The State History Tree is based on the fact that insertions will be done sequentially, with intervals being inserted to be sorted by ascending end times as much as possible. The tree supports inserting elements farther in the past, but doing it too often can lead to some imbalance (higher level nodes being filled faster than leaves), and at worst the tree would degenerate in a simple list of nodes. When a node is full (that is, there is no more room in the block on disk), it gets "closed off", and the latest end time found in the intervals it contains becomes the node's end time. A new sibling node is then added to its right. If the maximum number of nodes is reached in the parent, a new sibling is added to the parent, and so on up to the root node. Figure 4 shows the steps of a tree that gets built from scratch (from left to right). The numbers represent hypothetic start and end times of each node.

4 State System Querying and Visualization

This section investigates the querying and visualization of the values stored in the state system in both the offline and online tracing modes.

4.1 Querying in Offline Mode

The transient state is only required during the construction of the history. It means that when the trace reading is completed, no more events will be sent to the handler. Therefore, no new state changes will be created on the system. Consequently, at trace end, all transient intervals are closed and added to the state history. The state history is then complete and ready for future queries.

Once the history is completed, the state system is ready to receive queries. The user application can access the attribute tree and resources hierarchy, and use it to extract and browse the different state values of the attributes.

A typical query starts by providing a timestamp and a resource (attribute). Since the attribute tree always resides in main memory, we can use a multi-level hashmap of the attribute's path elements, which will give us the key of the given resource or attribute in the attribute tree. Having a timestamp and attribute key, a search in the history tree will be started from the root node downwards, exploring only the branch that can possibly contain the target timestamp. Within each node, it iterates through all possible intervals and return only those intersecting the target time. (Since intervals are sorted in the node by their end times, we can cut in half, on average, the number of intervals to iterate over.) All returned key/value pairs are returned and the external application is then free to look at the contents for whatever information it needs. Figure 5 shows an example of a complete and closed-off tree, on which we would run a query for timestamp t = 280.

4.2 Live Mode

Note that if we are querying a "live" state system, it is possible for some information to still be "current" and not yet part of the state history. Indeed, the start timestamp may be earlier than the queried time but still ongoing. Thus, the query system first looks at the current state and, if the value is not found, then searches the state history.

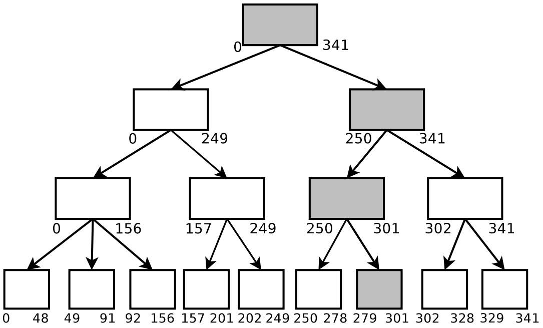


Fig. 5. Searching through the tree to get all intervals intersecting t = 280

Although the traces may be arbitrarily long, the current state normally does not grow with the trace size. The state history, however, grows linearly with the trace size. In other words, the current state contains the state values for a given time, like a snapshot, while the history encompasses all of the previous snapshots. Keeping separate the *current state* and the *history* makes the proposed method usable for live trace mode, where the method just keeps track the current state system and may not keep any history, or possibly a fraction of that (e. g. the last 10 seconds).

4.3 Visualization

The proposed modeled state system is contributed to TMF (Tracing and Monitoring Framework), the Eclipse plugin of Linux Tools)[4] and is accessible under the terms of Eclispe Public License [5].

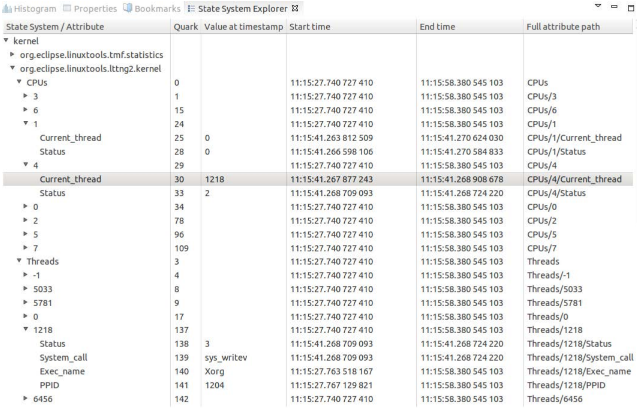
The implementation has actually two parts: a plugin that implements the library to handle the state provider, attribute tree, history tree and other structures, and a visualization part named the State System Explorer, which enables browsing and exploring the state system data directly (Figure 6). The viewer firstly displays the hierarchical organization of the resources and their attributes, extracted from the attribute tree, and then shows the state values of any selected attribute, at any given time. Using this view, users can directly browse the state values of the resources/attributes, which can help to study its overall behavior. This is interesting because users may not need to look at the trace events anymore and can directly analyze the state values gathered previously from the trace. For example, to study the behavior of a particular CPU or a process during the system execution.

Furthermore, there are other graphical views in TMF that allow analyzing a system from different aspects using the state values stored in the modeled state system. For instance, as shown in Figure 7, the Control Flow and Resources views

[4] `http://www.eclipse.org/linuxtools/projectPages/lttng/`

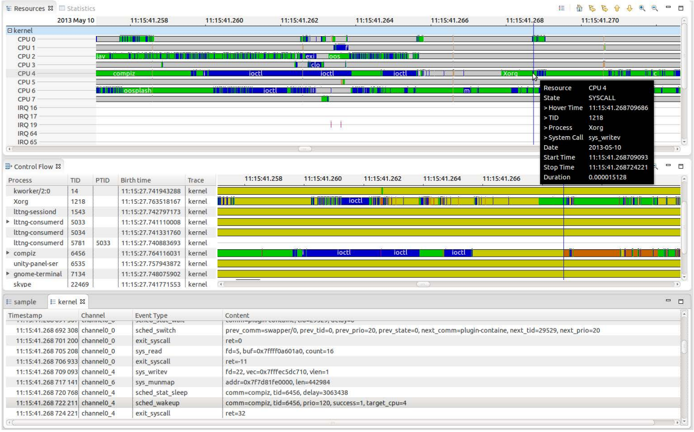
[5] `http://git.eclipse.org/c/linuxtools/org.eclipse.linuxtools.git/`

show the different states of various processes, CPUs, and other resources along the time axis using the stored state values. In the Resource view, for example, green states denote the times that CPU used in userspace mode in which a process utilizes the CPU for its internal instructions, while blue states show the times that CPU is executing system calls. Using the Resource the Control Flow views simultaneously, we can observe that the Xorg process is doing a system call on CPU4. In parallel, the *Compiz* process requests access to the same processor CPU4 with an event *sched_wakeup*. As we can see, this process changes its state from WAIT (yellow) to WAIT_FOR_CPU (orange).



State System / Attribute	Quark	Value at timestamp	Start time	End time	Full attribute path
kernel					
org.eclipse.linuxtools.tmf.statistics					
org.eclipse.linuxtools.lttng2.kernel					
CPUs	0		11:15:27.740 727 410	11:15:58.380 545 103	CPUs
3	1		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/3
6	15		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/6
1	24		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/1
Current_thread	25	0	11:15:41.263 812 509	11:15:41.270 624 030	CPUs/1/Current_thread
Status	28	0	11:15:41.266 598 106	11:15:41.270 584 833	CPUs/1/Status
4	29		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/4
Current_thread	30	1218	11:15:41.267 877 243	11:15:41.268 908 678	CPUs/4/Current_thread
Status	33	2	11:15:41.268 709 093	11:15:41.268 724 220	CPUs/4/Status
0	34		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/0
2	78		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/2
5	96		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/5
7	109		11:15:27.740 727 410	11:15:58.380 545 103	CPUs/7
Threads	3		11:15:27.740 727 410	11:15:58.380 545 103	Threads
-1	4		11:15:27.740 727 410	11:15:58.380 545 103	Threads/-1
5033	8		11:15:27.740 727 410	11:15:58.380 545 103	Threads/5033
5781	9		11:15:27.740 727 410	11:15:58.380 545 103	Threads/5781
0	17		11:15:27.740 727 410	11:15:58.380 545 103	Threads/0
1218	137		11:15:27.740 727 410	11:15:58.380 545 103	Threads/1218
Status	138	3	11:15:41.268 709 093	11:15:41.268 724 220	Threads/1218/Status
System_call	139	sys_writev	11:15:41.268 709 093	11:15:41.268 724 220	Threads/1218/System_call
Exec_name	140	Xorg	11:15:27.763 518 167	11:15:58.380 545 103	Threads/1218/Exec_name
PPID	141	1204	11:15:27.767 129 821	11:15:58.380 545 103	Threads/1218/PPID
6456	142		11:15:27.740 727 410	11:15:58.380 545 103	Threads/6456

Fig. 6. A detailed view of state values in TMF State System Explorer



Resource	CPU 4
State	SYSCALL
> Hover Time	11:15:41.268709686
> TID	1218
> Process	Xorg
> System Call	sys_writev
Date	2013-05-10
Start Time	11:15:41.268709093
Stop Time	11:15:41.268724221
Duration	0.000015128

Process	TID	PTID	Birth time	Trace
kworker/2:0	14		11:15:27.741943288	kernel
Xorg	1218		11:15:27.763518167	kernel
lttng-sessiond	1543		11:15:27.742797173	kernel
lttng-consumerd	5033		11:15:27.741110008	kernel
lttng-consumerd	5034		11:15:27.741331760	kernel
lttng-consumerd	5781	5033	11:15:27.740883693	kernel
compiz	6456		11:15:27.764116031	kernel
unity-panel-ser	6535		11:15:27.757943872	kernel
gnome-terminal	7134		11:15:27.748075902	kernel
skype	22469		11:15:27.741771553	kernel

Timestamp	Channel	Event Type	Content
11:15:41.268 692 308	channel0_0	sched_switch	prev_comm=swapper/0, prev_tid=0, prev_prio=20, prev_state=0, next_comm=plugin-containe, next_tid=29529, next_prio=20
11:15:41.268 701 200	channel0_0	exit_syscall	ret=0
11:15:41.268 705 208	channel0_0	sys_read	fd=5, buf=0x7ffff0a601a0, count=16
11:15:41.268 706 933	channel0_0	exit_syscall	ret=-11
11:15:41.268 709 093	channel0_4	sys_writev	fd=22, vec=0x7fffec5dc710, vlen=1
11:15:41.268 717 141	channel0_6	sys_munmap	addr=0x7f7d81fe0000, len=442984
11:15:41.268 720 768	channel0_4	sched_stat_sleep	comm=compiz, tid=6456, delay=3063438
11:15:41.268 722 211	channel0_4	sched_wakeup	comm=compiz, tid=6456, prio=120, success=1, target_cpu=4
11:15:41.268 724 221	channel0_4	exit_syscall	ret=32

Fig. 7. Summary view of the states in TMF State System Explorer

5 Experimental Results

All experiments were conducted on a Intel Core i7 920 @ 2.67GHz machine with 6 GB RAM, using Eclipse version 3.7 and OpenJDK version 7. The trace files were generated using LTTng version 0.232. The trace files contain the detailed execution trace at kernel level, including all the system calls, scheduling events and interrupts.

5.1 Construction Time

History Tree Construction: Figure 8 shows the time required to read a trace, update the attribute tree and transient (current) state, and build the history tree. The first case is to measure the time taken by the trace reader to only read the trace events, without passing them to the state provider module. This gives an idea of the proportion of time taken by each step. As Figure 8 shows, the time required to write the intervals to the disk-based history tree is an important factor. A solution to reduce the construction time is using the partial history update, instead of complete update. In this case, as mentioned previously, the history database is updated for a group of state changes, instead of each single state change. The effect of using a partial database update (partial history tree) on the construction time, storage space and query time is studied in the following sections.

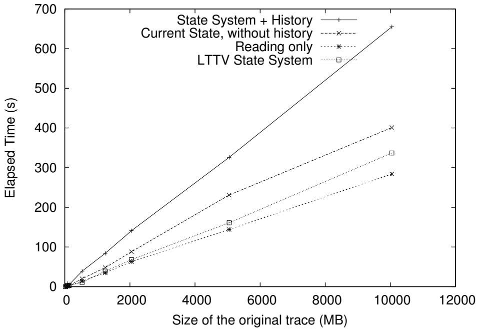


Fig. 8. Time for reading the trace and constructing the attribute tree

Partial Tree Update: Figure 9 shows the time required to store the values in the history tree in different cases: the case that updates the database for each single state changes and the case that updates the database partially for each 20,000, 50,000, 100,000 events, respectively.

With a partial history tree, the number of updates to database will be decreased during the construction phase, therefore less time is required. Please note that in all cases, it still requires to inspect each event. For that reason, the time required for the partial storage cases is almost the same.

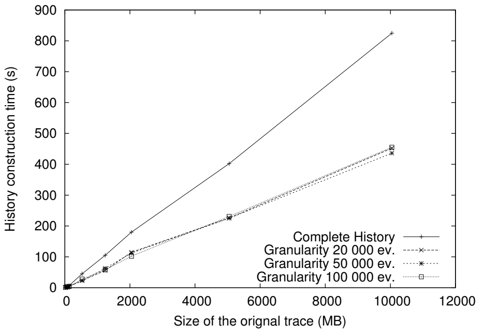


Fig. 9. History tree construction time

Comparison with R-Tree: To compare the construction time of the proposed method against a R-Tree, we use a main memory implementation of a R-Tree. Figure 10 shows the construction time comparison of two methods. As mentioned earlier, the R-Tree is very slow because it requires lots of re-balancing. The comparison proves the efficiency of our method for the incrementally arriving data.

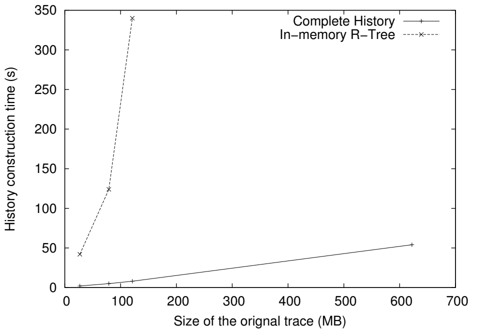


Fig. 10. Comparing the construction time with the R-Tree method

5.2 Storage Space

Figure 11 compares the storage required in 4 cases: the case that updates the database completely for each single state change, and the cases that update the database partially, for each 20,000, 50,000, 100,000 events, respectively. As mentioned earlier, updating the history database for every single state change is a costly operation and requires more storage space.

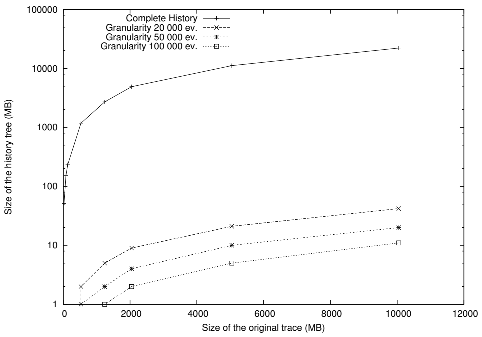


Fig. 11. Comparison of history database sizes for different cases

5.3 Query Time

For the different database update strategies of the previous experiment, we measure the query time, which is shown in Figure 12. The best case is using the complete history, where all data is accessible through a direct query of the history tree. For the other cases, although the size of the database is smaller, it first needs to fetch the states stored in the previous snapshot, and then reread the trace to extract the current state values up to the query time.

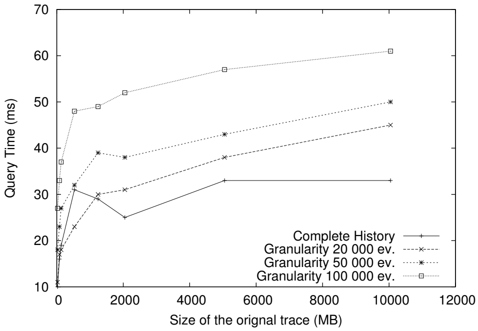


Fig. 12. Comparison of query times for different cases

6 Conclusion

In trace analysis, it can be sufficient to process the trace once, update the current state all along, and displaying it live. In *a posteriori* analysis however, one wants to be able to navigate through the trace back and forth and be able to look at the system state for any point in the trace.

To solve this problem, we proposed efficient data structures and algorithms to store and manage the system state values gathered from trace logs. In the proposed solution, we model the state values as intervals and store them in an interval tree, called the history tree, which has special support for typical interval queries. To fill the tree, every time a state change is reported, a state interval is created and sent to state history database. However, by maintaining a distinction between the creation of intervals and storage it is possible to easily change the storage method, and even use different methods or chain several of them. Comparisons with R-Trees and other experiments are undertaken, and the results are provided.

This solution is designed to work with any trace format. We have tested our solution with LTTng trace format in Linux and is accessible through Eclipse TMF framework. However, adding the capability to work with other trace formats (e. g. Dtrace, Event Tracing for Windows (ETW), etc) will be done as a future work. Reading the event-state mapping from XML files and doing it dynamically is another possible future work.

Acknowledgement. The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research, and Defence Research and Development Canada (DRDC) is gratefully acknowledged.

References

[CGK+04] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., Chase, J.S.: Correlating instrumentation data to system states: A building block for automated diagnosis and control. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design Implementation, vol. 6, p. 16. USENIX Association, Berkeley (2004)

[CGL08] Chan, A., Gropp, W., Lusk, E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. Scientific Programming 16(2), 155–165 (2008)

[CZG+05] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, indexing, clustering, and retrieving system history. SIGOPS Operating Systems Review 39(5), 105–118 (2005)

[DD06] Desnoyers, M., Dagenais, M.R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: Proceedings of the Ottawa Linux Symposium (2006)

[DD08] Desnoyers, M., Dagenais, M.: LTTng: Tracing across execution layers, from the hypervisor to user-space. In: Proceedings of the Ottawa Linux Symposium (2008)

[EJD12] Ezzati-Jivan, N., Dagenais, M.R.: A stateful approach to generate synthetic events from kernel traces. In: Advances in Software Engineering (April 2012)

[EJD13] Ezzati-Jivan, N., Dagenais, M.R.: A framework to compute statistics of system parameters from very large trace files. SIGOPS Oper. Syst. Rev. 47(1), 43–54 (2013)

[GDG+11] Giraldeau, F., Desfossez, J., Goulet, D., Dagenais, M., Desnoyers, M.: Recovering system metrics from kernel trace. In: Linux Symposium, p. 109 (June 2011)

[GG98] Gaede, V., Gunther, O.: Multidimensional access methods. ACM Computing Surveys 30(2), 170–231 (1998)

[Mon11] Montplaisir, A.: Stockage sur disque pour acceés rapide d' attributs avec intervalles de temps. Master's thesis, Ecole polytechnique de Montreal (2011)

[SHN09] Schnorr, L.M., Huard, G., Navaux, P.O.A.: Towards visualization scalability through time intervals and hierarchical organization of monitoring data. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2009, pp. 428–435. IEEE Computer Society, Washington, DC (2009)

[ZLGS99] Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with jumpshot. International Journal of High Performance Computing Applications 13(3), 277–288 (1999)

Repair Abstractions for More Efficient Data Structure Repair

Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, and Sarfraz Khurshid

The University of Texas at Austin, Austin TX 78712, USA
{rnokhbehzaeem,mzmalik,khurshid}@ece.utexas.edu

Abstract. Despite the substantial advances in techniques for finding and removing bugs, code is often deployed with (unknown or known) bugs, which pose a fundamental problem for software reliability. A promising approach to address this problem is *data structure repair*—a runtime approach designed to perform *repair actions*, i.e., mutations of erroneous data structures to repair (certain) errors in program state, to allow the program to *recover* from those errors and continue to execute. While data structure repair holds much promise, current techniques for repair do not scale to real applications.

This paper introduces *repair abstractions* for more efficient data structure repair. Our key insight is that if an error in the program state is due to a fault in software or hardware, a similar error may occur again, say when the same buggy code segment is executed again or when the same faulty memory location is accessed again. Conceptually, repair abstractions capture how erroneous program executions are repaired using concrete mutations to enable faster repair of similar errors in future. Experimental results using a suite of complex data structures show how repair abstractions allow more efficient repair than previous techniques.

Keywords: Data structure repair, Error recovery, Runtime analysis.

1 Introduction

Despite substantial advances in finding and removing bugs in code, software systems are often deployed with unknown or known bugs. Bugs in deployed code can be costly – not only in terms of the cost of failures they can cause but also in terms of the cost of fixing them. *Specification-based data structure repair* [6, 8, 17] is a promising approach for handling and recovering from errors in deployed systems. The key idea is to use specifications of crucial properties, e.g., data structure invariants, at runtime for error recovery. Thus, the specification use is not just for monitoring executions as in traditional runtime checking, say using assertions [4], but additionally for repairing erroneous executions by mutating erroneous states to conform to the specifications. Given an erroneous state and the specification that it violates, data structure repair techniques utilize the specific properties that are violated to perform a sequence of *repair actions*, which update erroneous field values to new values that conform to the expected properties.

While data structure repair provides a powerful mechanism for enforcing conformance of actual behavior to expected behavior as specified, existing techniques that

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 235–250, 2013.

embody this approach remain computationally expensive and the promise of the approach remains largely unfulfilled for real applications. The key issue is that finding a sequence of repair actions, which produces a desired state necessitates trasmuting the specification into a partial implementation, say using a backtracking search over a large state space – an inherently complex operation.

This paper introduces *repair abstraction*, a novel mechanism for more efficient data structure repair. Our key insight is that specific repair actions that are performed to recover from an error may be required again in the future when the same error occurs again, e.g., when a particular faulty line of code is re-executed to create a new state with an error similar to the erroneous state created when that line of code was executed in the past. Conceptually, repair abstraction provide a form of memoization, which captures the essence of how erroneous program executions in specific error scenarios are repaired using concrete repair actions and allow replaying similar actions in future, thereby enabling substantially faster repair of errors that recur.

We make the following contributions:

- **Repair abstraction.** We introduce a novel abstraction mechanism – repair abstraction – for runtime error recovery using data structure repair;
- **Abstract repair actions.** We present a representation for abstract repair actions, which provides the foundations of our work;
- **Framework.** We present our framework DREAM (Data structure Repair using Efficient Abstraction Methods) for repair abstraction. DREAM provides a generic framework that can be embodied by different data structure repair techniques.
- **Embodiment.** We present two techniques that embody DREAM. Our first technique utilizes specifications in Alloy [14], a first-order, relational language, and its accompanying SAT-based tool-set. Our second technique utilizes specifications in Java and an algorithm for solving constraints using Java predicates [3].
- **Evaluation.** We present an experimental evaluation using a suite of small programs that perform intricate operations on the program heap to evaluate the efficacy of repair abstraction in the context of complex data structure properties. Experimental results show that the use of repair abstraction enables significantly faster repair than previous techniques.

2 Running Example: Faulty Singly Linked List

In this section, we provide a motivating example. The example shows how DREAM efficiently finds and fixes a subtle error and helps the program recover from an otherwise fatal state. Listing 1.1 shows an implementation of the `addLast` method for a Singly Linked List data structure in Java. This method, which is supposed to add a node to the end of a list while maintaining acyclicity, works well when it receives a newly generated node that has `null` as its `next` pointer. However, it produces a wrong (i.e., cyclic) list if provided with an input node that already has a self loop. While the logic of the implementation is correct, missing to check the input causes an incorrect output. Such a wrong output list is shown in Fig. 1 (a).

Data structure repair can be utilized to fix this wrong output list and similar erroneous states. The basic idea is to augment the program with specifications, and use

```
public class Example{
   public static void main(String args[]){
      LinkedList l = new LinkedList();
      l.header = new Node();
      Node n = new Node();
      n.next = n;
      l.addLast(n);}}
class LinkedList{
   Node header;
   void addLast(Node n){
      Node newN = n.clone();
      if (header == null){
         header = newN;
      }else{
         Node pointer = header.next;
         Node prevPointer = header;
         while(pointer != null){
            pointer = pointer.next;
            prevPointer = prevPointer.next;}
         prevPointer.next = newN;}}}
class Node{
   Node next;
   protected Node clone(){
      ...}}
```

Listing 1.1. Motivating example: erroneous Singly Linked List `addLast` method

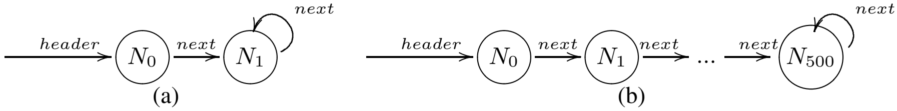


Fig. 1. The output of the method of Listing 1.1 on an initial list of (a) one and (b) 500 node(s)

those specifications to check and repair data structures. In addition to data structure invariant specifications supported by most repair frameworks [8, 13, 17, 23, 24, 27], some data structure repair frameworks [23, 24, 27] support pre- and post-conditions of program methods too. As an example of specifications, Listing 1.2 displays the invariant (commonly called repOK [18]) of Singly Linked List and a partial post-condition of the `addLast` method in the Alloy [14] specification language[1]. When repair is triggered on the faulty data structures of Fig. 1, it enforces this specification by breaking the cycle and setting the `next` pointer of the last node to `null`.

Most data structure repair frameworks [8, 13, 17, 23, 24, 27] instantiate a search in the space of valid data structures to find a close and correct data structure to replace the faulty one. This search poses a major challenge to scalability of data structure repair, as the size of the state space increases exponentially with the size of the data structure. For example, our previous work Cobbler [23] uses a combination of SAT solvers and heuristics for data structure repair. While Cobbler can break the cycle and repair the faulty list in Fig. 1 (a) in few hundred milliseconds, it runs out of heap space and a time out of 500 seconds when there are 500 nodes in the list (Fig. 1 (b)). Yet, the conceptual

[1] We use the syntactic sugar of adding back-tick (‘ ` ’) to distinguish post-state from pre-state in this Alloy specification. More details on Alloy specifications can be found elsewhere [14, 23].

```
sig LinkedList{
   header: lone Node,
   header': lone Node}
sig Node{
   next: lone Node,
   next': lone Node}
pred repOk(l: LinkedList){ // class invariant of LinkedList
   all n: l.header.*next | n !in n.^next} // acyclicity
pred add_postcondition(This: LinkedList, n: Node){
   repOk[This]
   This.header.*next + n = This.header'.*next'}
```

Listing 1.2. Alloy specification for the `addLast` method

action required to break the cycle is the same, no matter how many nodes are present in the list. Indeed, it is enough to set the `next` pointer of the last node to `null`.

Our key insight is to abstract out repair actions and use them as possible repair action candidates in the future, before opting into searching the state space. The idea is that if an error in the data structure is due to a fault in software or hardware, a similar error may occur again, for example when the same buggy code segment is executed again or when the same faulty memory location is accessed again. Repair abstractions capture the essence of how certain data structure corruptions are repaired by specific actions of a data structure repair routine, such as Cobbler [23], Juzi [8], PBnJ [27] or any other repair framework. Conceptually, a repair abstraction is a tuple *(condition; action)* where action is an abstract repair action performed when condition is met on a program state that needs repair.

Consider the example of repairing the faulty output of `addLast` shown in Fig 1 (a). The concrete repair action suggested by any repair framework should include the assignment N_1`.next = null`. We abstract out this concrete repair action to the abstract action `[LAST_NODE](in post-state).next = null`. Suppose that a similar error occurs again, now on a list of 500 nodes as shown in Fig 1 (b). Before starting to search the state space of correct data structures, we first try the previous abstraction in the hope of finding a quick fix. Concretizing the abstract repair action on the current data structures gives N_{500}`.next = null` which is a correct fix.

Repair abstractions offer two key advantages. One, they allow summarizing concrete repair actions into intuitive descriptions of how certain errors in data structures were fixed, which helps users understand and debug faulty program behaviors (when the errors in state were due to bugs in code). Two, they allow a direct reuse of repair actions without the need for a systematic exploration of a large number of data structures when the same error appears in a future program execution. The cost of repair, in cases that we do perform a search, will now be amortized over many repairs.

3 DREAM Framework

In this section, we explain the fundamentals of DREAM in abstracting and reusing repair actions. DREAM sits on top of an external data structure repair framework (Fig. 2). When a data structure repair framework is in place, specifications are periodically checked to make sure that data structure invariants and/or method post-conditions hold. Once a check fails, the repair routine is triggered. Our repair algorithm (shown as a Java like pseudo code in Listing 1.3) has three major phases:

1. DREAM tries previously abstracted repair actions to see if a fix can be found without calling the repair routine of the underlying repair framework.
2. If the previous phase does not generate an acceptable fix, the repair routine of the underlying repair framework is called.
3. The concrete repair actions taken by the underlying repair framework are abstracted out and saved as possible repair candidates for future.

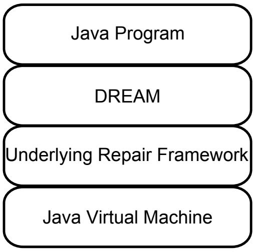


Fig. 2. The relationship between DREAM, the underlying repair framework, the Java Virtual Machine, and the program

To illustrate, consider the execution of Listing 1.1. The very first time this execution causes a failure, no previous repair abstraction is available (in Listing 1.3, `abstractRepairCandidateSets` is empty). Therefore, the first phase (Lines 3 to 19 of Listing 1.3) is skipped. Line 20 performs the second phase and calls the underlying repair framework, which repairs the data structure by setting N_1`.next = null`. This concrete action is abstracted in the third phase by Lines 21 to 23 to be saved as `[LAST_NODE](in post-state).next = null`. More details about the abstraction process will follow in Section 3.1.

The next time an error is observed in the data structure, DREAM attempts to reuse previous repair actions to avoid the prohibitively costly repair routine of the underlying repair framework. Let us say that we have Fig. 1 (b) this time. Lines 3 to 19 implement the first phase of DREAM. They examine candidate sets of abstract repair actions. Firstly, DREAM concretizes each abstract action on the current data structure. An abstract action (like `[LAST_NODE](in post-state).next = null`) contains a left hand side dereferencing list (`[LAST_NODE]` in this example), a field on which the assignment should be applied (here `next`), and a right hand side dereferencing list (here `null`). Such dereferencing lists are abstracted forms of actual dereferencing lists that were used in concrete repair actions and may contain abstract fields (e.g., `[LAST_NODE]`) as well as concrete fields (e.g., `next`). In the concretization process, which is the reverse process of abstraction (Section 3.1), abstract fields are translated back into sequences of concrete data structure fields. (Here the left hand side dereferencing list would become $header\underbrace{.next\ldots.next}_{\text{500 times}}$.)

Secondly, the concretized lists are then applied on the input or the faulty output to identify the target object on which the assignment should take place, the target field, and the target value (e.g., $header\underbrace{.next\ldots.next}_{\text{500 times}}$ gives N_{500}). DREAM can utilize either the input or the faulty output for concretizing abstract actions and identifying target objects by using `baseObject` and flags like `derefLeftInOutput` (see Section 3.1 for more details).

```
 1 Object dreamRepair(Object input, Object faultyOutput){
 2    Object repairedOutput;
 3    for(Set<AbstractRA> abstractCand : abstractRepairCandidateSets){
 4       Set<ConcreteRA> concreteCand = new HashSet<ConcreteRA>();
 5       for(AbstractRA action : abstractCand){
 6          List<Field.ConcreteField> left = new LinkedList<Field.ConcreteField>();
 7          Object baseObject = action.derefLeftInOutput ? faultyOutput : input;
 8          for(Field.AbstractField f : action.derefLeft)
 9             left.addAll(concretizeOnObject(f, baseObject));
10          Object leftHandSide = getObject(left, baseObject);
11          ...//Similarly for the right hand side and field
12          concreteCand.add(new ConcreteRA(leftHandSide, concreteField,
                rightHandSide));
13       }
14       repairedOutput = apply(faultyOutput, concreteCand);
15       if(check(input, repairedOutput)){
16          increaseScore(abstractCand);
17          return repairedOutput;
18       }
19    }
20    repairedOutput = repair(input, faultyOutput);
21    Set<ConcreteRA> newConcreteCand = getConcreteRA(faultyOutput, repairedOutput)
          ;
22    Set<AbstractRA> newAbstractCand = abstractOut(newConcreteCand, input,
          faultyOutput);
23    abstractRepairCandidateSets.add(newAbstractCand);
24 }
```

Listing 1.3. DREAM main algorithm

Thirdly, on Line 14, the set of concrete actions is applied on the faulty output. Finally, the next line checks if the result is indeed a correct output with respect to the specification. If so, DREAM ascends the abstract set that created this fix in the ordered list of candidates `abstractRepairCandidateSets` and returns the repaired output without continuing to phases two and three.

3.1 Abstraction and Concretization

DREAM uses a pre-defined yet generic and extensible repository of meaningful abstractions. We define the following abstractions as the basis of our approach:

- *First*: the first object of a type reachable from the given root pointer (e.g., the root of a tree or the first node in a list);
- *Self*: the object itself;
- *Last* or *Leaf*: the furthest object(s) of a type reachable from the given root pointer (e.g., leaves of a tree or the last node in a list);
- *Neighbor*: a neighboring object, where two object O_1 and O_2 are neighbors if a field of O_1 points to O_2 (e.g., the parent of a node in a tree);
- *A value with an offset*: the numeric value of a node plus/minus an offset (e.g., the size of a binary heap plus one);
- *A value with a coefficient*: the numeric value of a node multiplied/divided by a coefficient (e.g., twice the value of a key in a Red Black Tree);
- *Null*: the `null` value;

The abstraction process has two steps:

1. A breath first search of the data structure (for both the input and the faulty output) is performed along all concrete fields. This search assigns a concrete dereferencing list that can be used to reach any object. For example, N_0 in Fig. 1 (a) is reached via `header` and N_1 is reached via `header.next`.
2. Using the above repository of abstractions, all possible abstractions that are equal to a concrete dereferencing list are built. For instance `header.next` is considered equal to `FIRST_NODE.next`, `FIRST_NODE.NEIGHBOR`, and `LAST_NODE`. The abstractions Self, Null, Offset, and Coefficient are only helpful as the right hand side of a repair action.

The concretization process is an exact reverse of the abstraction. First, DREAM transforms the abstract fields of a dereferencing list to their concrete forms which only use the data structure fields. Then, it traverses the data structure along those fields to get to the desired objects. When multiple abstractions/concretizations are valid, all of them are used as possible candidates.

Both abstraction and concretization can be performed on the input data structure as well as the faulty output of a method. This flexibility enhances DREAM's ability to access objects that get lost because of broken pointers. `derefLeftInOutput` and similar boolean flags are put in place to distinguish between the cases that the faulty output and the input are used to access an object.

4 DREAM with Different Back-Ends

The idea of abstracting concrete repair actions is orthogonal to the underlying repair framework used. Therefore, we can plug in our repair framework of interest to DREAM and utilize its abstraction power. Some repair frameworks [23, 24, 27] use a SAT solver to search the space of correct data structures while others [8, 13, 17] leverage dedicated solvers. To showcase how DREAM can be used regardless of the underlying repair method, we integrate it with Cobbler [23], our previous SAT based repair framework, and Juzi [8, 9], which uses a dedicated Java based solver.

4.1 DREAM with Alloy Back-End

Connecting DREAM with an Alloy based repair framework (like PBnJ [27], Cobbler [23] or Tarmeem [24, 25]) is quite straightforward. The underlying repair framework performs regular checks and provides concrete repair actions in case the abstractions do not work.

The only limitation is that even *checking* the correctness of a data structure using SAT might be rather time consuming. To tackle this limitation, we used the Minshar [1] technique. Minshar is a tool that translates Alloy specification checks to Java assertions by viewing Alloy as a set based language. Minshar starts by parsing the Alloy specification into Alloy Abstract Syntax Tree (AST), which indicates how Alloy expressions are recursively built from subexpressions. It then traverses this AST and translates it to Java in the following manner: Minshar views Java objects as one-element sets and navigates the data structure fields (obtained via reflection) to construct the sets that correspond to subexpressions used in the AST, such as subexpressions that have reflective closure. Once the sets are generated, Minshar asserts the Alloy checks and returns false if they do not hold.

4.2 DREAM with Juzi Back-End

Juzi [8, 9] is an automated data structure repair framework. Given a corrupt data structure, as well as a repOK method that describes the invariants of the data structure, Juzi monitors the execution of repOK which checks the invariants. Upon a failure, Juzi systematically mutates the fields of the corrupt data structure, starting from the ones that were accessed last in repOK and trying different values, to make them satisfy the given specifications. In addition to repairing the data structure, Juzi reports the mutations it performed in a log file that holds a sequence of tuples $< O_1, f, O_2 >$, i.e., an assignment to field f of object O_1 to the value O_2 – each tuple represents a repair action.

Juzi has two properties that make it a good fit for DREAM. First, it uses checks written in Java (i.e., repOK methods) instead of Alloy or other specification languages. Second, once it reports a faulty data structure, it also reports the field that is responsible for the failure of the repOK method on the current data structure. DREAM uses these properties of Juzi to facilitate finding effective repair abstractions. The first property provides a fast way of checking the correctness without a need to translate specifications to Java (as it was the case with Alloy for which we used Minshar). The second property pinpoints to the exact object and field that need mutation in the current faulty data structure, taking care of the left hand side and the filed of the repair action.

As with any other repair framework, every time Juzi finds a correct fix for a specification violation, DREAM computes an abstraction for the repair. For example, if Juzi repairs the data structure by assigning a field f of an object O to *null*, then DREAM records *O.f* = *null*, meaning that if f needs to be mutated, it should be set to *null*. Thereby, DREAM prioritizes *null* when a future execution encounters the same error even if the underlying repair routine would have first tried a non-null value according to its default search.

Juzi has one drawback compared to Alloy based repair routines: It does not support pre- and post-conditions of methods, only invariants. Therefore, it fails to find cases that a method is not conforming to its specification in changing a data structure.

5 Evaluation

We present the experimental evaluation of DREAM combined with Cobbler [23] in Section 5.1, and combined with Juzi [8, 9] in Section 5.2. All the experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running 64 bit Windows 7 and Sun's Java SDK 1.7.0 JVM. Cobbler used MiniSat and MiniSatProver SAT solvers.

Table 1. Description of the Singly Linked List errors used for experimental evaluation

	Err. #	*Description*
Singly Linked List	1	Sets the header to null
	2	Fails to update the size
	3	Deletes a node with a non-matching element
	4	Introduces a cycle after performing correct remove
	5	Breaks the list to retain only the first two nodes
	6	Deletes the matching element but adds it again
	7	Fails to remove the element and updates the size incorrectly
	8	Fails to remove the element
	9	Fails to update the size because of a missing left hand side in an assignment

5.1 Evaluation: DREAM with Alloy Back-End

We used our tool Cobbler, a data structure repair framework that utilizes SAT and heuristics, for the first set of experiments with DREAM. In order to improve the performance of SAT solving for data structure repair, Cobbler iteratively calls SAT and gradually increases the size of the search space. To guesstimate the size of the search space (i.e., to find out which fields it should include for a possible change when calling SAT), Cobbler uses program execution history, obtained via reads and writes performed to the heap, as a source of identifying corrupt fields of data structures and fixes for them. Furthermore, it uses unsatisfiable cores that SAT solvers provide after a failing call, to further prune the search space.

The first data structure we looked at was a basic Singly Linked List that also keeps its size. To minimize threats to validity, we used independently written errors we used to evaluate Cobbler in our previous work [23]. In that work, we included seven erroneous remove methods for Singly Linked List. We used the same seven errors plus two new ones here. Table 1 shows the errors and a brief description of each of them. Some of the errors violate the invariants of Singly Linked List (e.g., Error 4), some violate the postcondition of the remove method (e.g., Error 1), and some violate both (e.g., Error 7).

We started by repairing a Singly Linked List of ten nodes. Upon the very first error, no repair abstraction is available, so DREAM has to use the underlying repair routine which is Cobbler here. Then DREAM abstracts out the set of concrete repair actions taken by Cobbler and memorizes them for future use. In the next experiment, we used a Singly Linked List of 500 nodes with each error. DREAM applies the abstract repair actions which fix the problem without calling Cobbler in 8 out of 9 errors. Table 2 shows the abstractions that DREAM extracted for each error. Some abstractions, e.g., the first and second abstraction for Error 9, are unnecessary but harmless since they change now unreachable nodes. These unnecessary actions exist because SAT suggested them as concrete repair actions.

Table 3 displays the time performance of Cobbler repairing lists of size 10 and 500, as well as DREAM repairing the same lists. For the case of calling Cobbler, an initial call is made to SAT to discover the problem and trigger repair (the check column in Table 3). Hence, the total time for repair with Cobbler includes the initial check time plus the

Table 2. Abstract repair actions suggested by DREAM for Singly Linked List

	Err. #	*Abstract Repair Action(s)*
Singly Linked List	1	[] (in post-state).header = [FIRST_NODE] (in pre-state)
	2	DREAM Not Working: Call SAT
	3	[FIRST_NODE] (in post-state).next = [header.next.next] (in pre-state)
	4	[LAST_NODE] (in post-state).next = [null]
	5	[LAST_NODE] (in post-state).next = [header.next.next.next] (in pre-state)
	6	[FIRST_NODE] (in post-state).elt = [header.elt] (in pre-state)
	7	[FIRST_NODE] (in post-state).elt = [header.elt] (in pre-state)
		[](in post-state).size = [size] - 1 (in post-state)
	8	[FIRST_NODE] (in post-state).next = [null]
		[] (in post-state).header = [header.next] (in post-state)
		[header.next] (in post-state).elt = [header.elt] (in post-state)
		[FIRST_NODE] (in post-state).elt = [null]
	9	[header.next] (in pre-state).next = [null]
		[header.next] (in pre-state).elt = [null]
		[] (in post-state).size = [size] - 1 (in post-state)

Table 3. Time taken to repair erroneous Singly Linked Lists (ms). OH means Out of Heap.

Error #	*Cobbler/DREAM (Size = 10)*			*Cobbler (Size = 500)*			*DREAM (Size = 500)*						*Improvement*
									Repair				
	Check	*Repair*	*Total*	*Check*	*Repair*	*Total*	*Abs.*	*Check*	*Con.*	*App.*	*Check*	*Total*	
1	320	799	1119	287034	125638	412672	24	0	0	1	77	78	5291x
2	915	8846	9761	241701	446	OH	Not Working: Call SAT						N/A
3	166	417	583	127434	240674	368108	14	38	0	39	37	114	3229x
4	128	381	509	81428	52	OH	6	0	0	1	41	42	N/A
5	130	292	422	52621	61751	114372	6	0	0	6	40	46	2486x
6	145	410	555	55691	142061	197752	7	42	0	44	32	118	1676x
7	126	319	445	52356	133512	185868	6	19	0	21	32	72	2582x
8	131	259	390	51766	234913	286679	6	16	1	17	32	66	4344x
9	228	797	1025	92219	298215	390434	8	64	1	69	69	203	1923x

repair time. For DREAM, first the repair actions are abstracted (column Abs.) using concrete repair actions taken by Cobbler on the data structure of ten nodes. Then, using the data structure of 500 nodes, a Java check is performed to find that the specification is violated. This Java check is a manual translation of the specification from Alloy to Java using the Minshar technique which can be automated using the Minshar tool. When this initial check fails, DREAM repair performs concretization (the Con. column) followed by the application of concretized actions (the App. column). Unlike Cobbler which only suggests correct fixes, the result of applying a set of abstract repair actions by DREAM should be checked to see if the abstractions can indeed resolve the problem. Therefore,

Table 4. Description of the Red Black Tree errors used for experimental evaluation

	Err. #	*Description*
Red Black Tree	1	Sets the root's parent to itself
	2	Changes the color of the root's grand child
	3	Assigns an incorrect key to the root's child
	4	Makes a part of the tree unaccessible

Table 5. Abstract repair actions suggested by DREAM for Red Black Tree

	Err. #	*Abstract Repair Action(s)*
Red Black Tree	1	[root] (in post-state).parent = [null] (in post-state)
		[root] (in post-state).color = [root.right.right.right.left.key] (in post-state)
	2	[root.left.left] (in post-state).color = [root.color] (in post-state)
		[root] (in post-state).color = [root.right.right.right.left.color] (in post-state)
	3	[root.right] (in post-state).key = [root.right.key] (in pre-state)
	4	[root] (in post-state).right = [root.right] (in pre-state)
		[root] (in post-state).color = [root.right.right.right.right.color] (in pre-state)

there is another Java based check after DREAM repair. Note that abstracting repair actions is a one time procedure whose results are reused multiple times, therefore it is not included in the total time for repairing with DREAM.

DREAM abstractions do not work for Error 2, mainly because the fix suggested by Cobbler is too tailored to the specific data structure of 10 nodes and cannot be generalized. However, even Cobbler cannot fix a data structure of 500 nodes for Error 2 because it runs out of the heap space. Cobbler also fails to fix Error 4 on 500 nodes while DREAM solves this error in a total of 42 ms. As Table 3 shows DREAM is substantially (about 3000 times on average) faster than Cobbler and it fixes 8 out of 9 errors in less than a quarter of a second.

The second data structure we considered was the Red Black Tree implementation in the open source Kodkod model finder [29]. We used the `insert` method in the `Kodkod.util.ints.IntTree` class. This class has 570 lines of code and 21 methods and serves as one of the most important data structures used by Kodkod. We injected errors that violate data structure invariants (acyclicity, color constraints, key constraints, and parent constraints) as well as method post-conditions. Table 4 shows a brief description of each error.

Similar to the Singly Linked List experiment, we repaired Red Black Trees of 10 and 500 nodes. Table 5 shows the abstract repair actions suggested by DREAM.

Table 6 shows the performance measurements. For a Red Black Tree of 500 nodes, Cobbler always times out where the time out value is 500,000 ms. DREAM repairs all the errors in less than one minute.

Table 6. Time taken to repair erroneous Red Black Trees (ms). TO represents a time out of 500,000 ms.

Error #	*Cobbler/DREAM* (Size = 10)			*Cobbler* (Size = 500)			*DREAM* (Size = 500)						*Improvement*
									Repair				
	Check	*Repair*	*Total*	*Check*	*Repair*	*Total*	*Abs.*	*Check*	*Con.*	*App.*	*Check*	*Total*	
1	282	582	864	TO	TO	TO	7	11	0	14	54642	54667	N/A
2	249	521	770	TO	TO	TO	5	37	0	40	56042	56119	N/A
3	331	772	1103	TO	TO	TO	6	9	0	11	53954	53974	N/A
4	251	494	745	TO	TO	TO	6	1045	0	1048	53508	55601	N/A

5.2 Evaluation: DREAM with Juzi Back-End

In this section, we compare DREAM with the original Juzi [8, 9] repair algorithm for efficiency improvement. Juzi uses imperative descriptions of data structure invariants called `repOk` methods [26]. Given a predicate `repOK` that represents desired structural integrity constraints and a structure S_1 that violates them, the Juzi algorithm performs repair actions that mutate the given structure to generate a new structure S_2 that satisfies the constraints. Each repair action assigns some value to a field of an object in the structure. This assignment is made based on the exploration of the possible set of field assignments to reference variables. The fundamental problem that Juzi addresses is the enormous number of combinations of field assignments that make it impossible to enumerate all possible assignments (even for small structures) and check whether any assignment represents a repaired structure. DREAM drastically reduces this search space for error patterns that are repeated. The basic Juzi algorithm assumes that each structural error is purely random and requires re-execution of the search for repair every time. When the errors are repeated, Juzi performance does not improve but DREAM is able to use repair abstractions to quickly fix the error.

Both Juzi and imperative implementation of DREAM have to check the structure for validity after every mutation. Therefore, the number of calls made to `repOK` (the routine checking of structural validity) is a good measure of the size of the exploration each algorithm had to perform before fixing the structure. In our experiments, we use the number of calls made to `repOK` to compare the efficiency of the two algorithms.

In our experiments we used the following structures:

- A Linked List based implementation of Circular List. The errors injected in this structure violated the circularity constraint.
- Doubly Linked List with bad previous field assignment. The violated structural constraint is that `next` is the transpose of `previous`.
- Binary Tree with acyclicity constraint violation.
- Binary Tree with Parent Pointer having a bad parent field assignment. The violated structural constraint is that `child` is the transpose of `parent`.

Table 7 summarizes the results of our experiments with the subject structures. For these experiments, each structure had only one injected error in it, also the error was a

Table 7. The number of `repOK` calls made by DREAM vs Juzi for fixing the errors

Structure	*size = 10*		*size = 500*	
	Juzi	*DREAM*	*Juzi*	*DREAM*
Circular List	8	2	477	2
Doubly Linked List	6	2	251	2
Binary Tree	2	2	2	2
Binary Tree with Parent Pointers	8	2	263	2

repeating error and not the first time error. As expected, Juzi's time to repair is a linear function for single errors while DREAM was able to fix the error in constant time. The abstraction rules (Section 3.1) used by DREAM for each of the error are as follows.

- In the case of circularity violation in the Circular List, DREAM used *First* and *Last* abstractions to point the `next` of the last node to `header`.
- For Doubly Linked List the *Neighbor* rule was used by DREAM.
- Binary Tree with acyclicity constraint violation is an interesting case because both Juzi and DREAM performed equally well. The rule used by DREAM was *Null*, and it is interesting to note that Juzi also attempts `null` as the first value mutation to a field.
- Binary Tree with Parent Pointer having a bad parent field assignment was also able to exploit the *Neighbor* rule.

Juzi bounds the space of possible mutations to a structure and then performs systematic exploration of this space. The implementation of the algorithm employs various heuristics to make its search efficient but the basic algorithm is memoryless and does not benefit from observed repairs. DREAM improves over Juzi by remembering the fixes used earlier and reusing them. A real challenge for DREAM was to recall the prior fixes when the underlying structure and object references have changed. The abstract representation and code instrumentation make it possible for DREAM to learn from Juzi repairs and subsequently try them before exhaustive exploration is attempted.

6 Related Work

Dynamic repair techniques which fix the faults at runtime and keep the state consistent have been in existence for a long time. Examples of such techniques are file system utilities such as fsck [10] and chkdsk [21], database check-pointing, and rollback techniques.

As opposed to generic data structure repair, some systems support dedicated routines for monitoring and repairing data structures. The idea of dedicated repair routines has been applied in some commercial systems such as the IBM MVS operating system [22] and the Lucent 5ESS telephone switch [12]. The most important drawback of such repair routines is the lack of generality and extensibility.

The first work on generic constraint based data structure repair belongs to Demsky and Rinard [5,6]. Their method supports constraints in a declarative language similar to Alloy. Also similar to SAT, their method translates constraints to disjunctive normal form and solves them using an ad hoc search.

The Juzi tool [7,17] implements assertion based data structure repair. The user writes data structure invariants in repOK methods. Juzi monitors the execution of repOK as it checks the invariants. If a repOK check returns false, Juzi keeps systematically mutating fields and running repOK until repOK returns true. Juzi mutates fields starting from the ones accessed later in repOK, hypothesizing that those are more likely to be responsible for the false return value. As an improvement, symbolic execution of repOK is added to make the repair process even faster. Juzi only supports invariants that are checked at one given point during the execution, hence it misses failures that correspond to the relationship between pre- and post-conditions of methods.

Dynamic Symbolic Data Structure Repair [13] (DSDR) extends Juzi's technique by producing a symbolic representation of fields and objects along the path executed in repOK. DSDR builds the path constraint required to take the current path in repOK. When repOK returns false, DSDR uses the conjunction of the negation of the path constraint with the other path conditions and solves them, directly generating a fix irrespective of the exact location of the corrupted object references or fields in the repOK method.

The Plan B approach and its tool PBnJ [27] support data structure invariants as well as method pre- and post-conditions in a declarative first order relational logic extension to Java that is similar to Alloy. Once a failure is observed, PBnJ falls back on *executing* the specifications: i.e., it ignores the Java implementation and uses a SAT solver to come up with a data structure that satisfies both invariants and method post-conditions. Similarly to DREAM, PBnJ translates specifications to Java predicates which it uses for fast checking. However, PBnJ suffers from a low repair performance, as it completely ignores the Java code, the execution history of the program, the previous repair actions, and the current faulty data structure.

Cobbler [23] aims to improve the scalability of SAT based data structure repair by iteratively calling SAT and pruning the state space. To do so, Cobbler takes advantage of program execution history: It considers the dynamic trace of field write and reads that happened during the execution to guide repair actions. In addition, it uses unsatisfiable cores provided by SAT solvers to limit the search space.

We would like to emphasize that DREAM's technique is independent of the underlying repair framework and will enhance the performance of any repair routine. DREAM can be combined with any of the above repair tools, or new data structure repair framework tools and techniques. Furthermore, in the event that the failure is due to a bug in code, DREAM can serve as a debugging aid, by providing an intuitive description of the repair actions performed so that the user can incorporate a bug fix and eliminate the need to repair altogether.

Even though our technique differs from automated debugging and program repair techniques [2,11,15,16,20,28,30,31], which mainly try to debug programs before deployment, as we previously suggested [19] dynamic data structure repair can translate into program statements that patch programs. Data structure repair actions can act as an input to program repair frameworks such as the AUTO E-FIX tool [30], providing useful information regarding the differences between faulty and correct concrete program states. DREAM abstractions directly aid in program repair by summarizing repair actions and helping users comprehend concrete repair actions better.

7 Conclusion

Data structure repair, a runtime approach designed to keep program state (i.e., data structures) consistent in the event of software or hardware errors, has seen improvements in recent years. However, it still suffers from low performance and lack of scalability. We introduced repair abstractions to enhance the efficiency and scalability of data structure repair. Our insight is that if an error is due to a fault in software or hardware, it is likely to recur. Therefore, we can abstract the concrete repair actions taken to fix a particular state and reuse them when a similar error is detected in future.

We implemented the idea of repair abstractions in the DREAM (Data structure Repair using Efficient Abstraction Methods) tool. DREAM piggybacks on other repair frameworks and records concrete repair actions they take to fix a particular erroneous state. DREAM abstracts the concrete actions and attempts to reuse them when a similar error is detected, eliminating the need to go to the underlying repair framework again. Hence, DREAM amortizes the repair cost from the cases it has to invoke the underlying repair framework among many repairs.

We combined DREAM with two data structure repair frameworks: Cobbler that uses a combination of SAT solvers and heuristics, and Juzi that implements a dedicated search engine for repair. The experimental evaluation of the use of DREAM in accordance with these two frameworks on basic and complex data structures showed that DREAM offers significant performance improvement. We envision that repair abstractions can be a valuable addition to data structure repair frameworks. DREAM's ability to integrate with different repair frameworks provides a promising step towards making repair scale to real applications.

Acknowledgments. This work is partially supported by the National Science Foundation under Grant No. CCF-0845628.

References

1. Al-Naffouri, B.Y.: MintEra: A testing environment for Java programs. Master's thesis, MIT (2004)
2. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Directed test generation for effective fault localization. In: ISSTA (2010)
3. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA (2002)
4. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. SIGSOFT Software Engineering Notes, 31(3) (2006)
5. Demsky, B.: Data Structure Repair Using Goal-Directed Reasoning. PhD thesis. MIT (2006)
6. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: OOPSLA (2003)

7. Elkarablieh, B.: Assertion-based Repair of Complex Data Structures. PhD thesis, UT Austin (2009)
8. Elkarablieh, B., Garcia, I., Suen, Y.L., Khurshid, S.: Assertion-based repair of complex data structures. In: ASE (2007)
9. Elkarablieh, B., Khurshid, S.: Juzi: A tool for repairing complex data structures. In: ICSE (2008)
10. Ext2 fsck. manual page, `http://e2fsprogs.sourceforge.net`
11. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to C. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 358–371. Springer, Heidelberg (2006)
12. Haugk, G., Lax, F., Royer, R., Williams, J.: The 5ESS(TM) switching system: Maintenance capabilities. AT&T Technical Journal 64(6 pt. 2), 1385–1416 (1985)
13. Hussain, I., Csallner, C.: Dynamic symbolic data structure repair. In: ICSE (2010)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
15. Jeffrey, D., Feng, M., Gupta, N., Gupta, R.: BugFix: a learning-based tool to assist developers in fixing bugs. In: ICPC (2009)
16. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
17. Khurshid, S., García, I., Suen, Y.L.: Repairing structurally complex data. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 123–138. Springer, Heidelberg (2005)
18. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley (2000)
19. Malik, M.Z., Ghori, K., Elkarablieh, B., Khurshid, S.: A case for automated debugging using data structure repair. In: ASE (2009)
20. Mayer, W., Stumptner, M.: Evaluating models for Model-Based debugging. In: ASE (2008)
21. Microsoft. chkdsk manual page, `http://support.microsoft.com/kb/315265`
22. Mourad, S., Andrews, D.: On the reliability of the IBM MVS/XA operating system. IEEE Transactions on Software Engineering 13(10), 1135–1139 (1987)
23. Nokhbeh Zaeem, R., Gopinath, D., Khurshid, S., McKinley, K.S.: History-aware data structure repair using SAT. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 2–17. Springer, Heidelberg (2012)
24. Nokhbeh Zaeem, R., Khurshid, S.: Contract-Based Data Structure Repair Using Alloy. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 577–598. Springer, Heidelberg (2010)
25. Nokhbeh Zaeem, R., Khurshid, S.: Introducing Specification-Based Data Structure Repair Using Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 398–399. Springer, Heidelberg (2010)
26. Rosenblum, D.S.: Towards a method of programming with assertions. In: ICSE (1992)
27. Samimi, H., Aung, E.D., Millstein, T.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
28. Staber, S., Jobstmann, B., Bloem, R.: Finding and fixing faults. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 35–49. Springer, Heidelberg (2005)
29. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
30. Wei, Y., et al.: Automated fixing of programs with contracts. In: ISSTA (2010)
31. Weimer, W.: Patches as better bug reports. In: GPCE (2006)

To Run What No One Has Run Before: Executing an Intermediate Verification Language*

Nadia Polikarpova, Carlo A. Furia, and Scott West

Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. When program verification fails, it is often hard to understand what went wrong in the absence of concrete executions that expose parts of the implementation or specification responsible for the failure. Automatic generation of such tests would require "executing" the complex specifications typically used for verification (with unbounded quantification and other expressive constructs), something beyond the capabilities of standard testing tools.

This paper presents a technique to automatically generate executions of programs annotated with complex specifications, and its implementation for the Boogie intermediate verification language. Our approach combines symbolic execution and SMT constraint solving to generate small tests that are easy to read and understand. The evaluation on several program verification examples demonstrates that our test case generation technique can help understand failed verification attempts in conditions where traditional testing is not applicable, thus making formal verification techniques easier to use in practice.

1 Help Needed to Understand Verification

Static program verification has made tremendous progress, and is now being applied to real programs [16,11] well beyond the scale of "toy" examples. These achievements are impressive, but still require massive efforts and highly-trained experts. One of the biggest remaining obstacles is understanding failed verification attempts [19]. Most difficulties in this area stem from inherent limits of static verification, and hence could benefit from complementary *dynamic* techniques.

Static program proving techniques—implemented in tools such as Boogie [17], Dafny [18], and VeriFast [8]—are necessarily incomplete, since they target undecidable problems. Incompleteness implies that program verifiers are "best effort": when they fail, it is no conclusive evidence of error. It may as well be that the specification is sound but insufficient to prove the implementation correct; for example, a loop invariant may be too weak to establish the postcondition. Even leaving the issue of incomplete specifications aside, the feedback provided by failed verification attempts is often of little use to understand the ultimate source of failure. A typical error message states that some executions might violate a certain assertion but, without concrete input values that trigger the violation, it is difficult to understand which parts of the programs

* Work partially supported by SNF grants 200021-137931 (FullContracts), 200020-134974 (LSAT), and 200021-134976 (ASII); and by ERC grant 291389 (CME).

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 251–268, 2013.

should be adjusted. And even when verification is successful, it would still be useful to have “sanity checks” in the form of concrete executions, to increase confidence that the written specification is not only consistent but sufficiently detailed to capture the intended program behavior.

Dynamic verification techniques are natural candidates to address these shortcomings of static program proving, since they can provide concrete executions that conclusively show errors and help narrow down probable causes. Traditional dynamic techniques based on *testing* are, however, poor matches to the capabilities of static provers. Testing typically targets simple properties, such as out-of-bound and null dereferencing errors, or, only in a minority of cases, lightweight executable specifications (e.g., contracts). Program provers, in contrast, work with very expressive specification and implementation languages supporting features such as nondeterminism, unbounded quantification, infinitary structures (sets, sequences, etc.), and complex first- or even higher-order axioms; none of these is executable in the traditional sense. As we argue in Sec. 2, however, even relatively simple programs may require such complex specifications. Program provers also support modular verification, where sufficiently detailed specifications of modules or routines are used in lieu of missing or incomplete implementations; this is another scenario where runtime techniques fall short because they require complete implementations.

In this paper, we propose a technique to generate executions of programs annotated with complex specifications using features commonly supported by program provers (nondeterminism, unbounded quantification, partial implementations, etc.). The technique combines symbolic execution with SMT constraint solving to generate small and readable test cases that expose errors (failing executions) or validate specifications (passing executions).

The proposed approach supports executing both imperative and declarative program elements, which accommodates the *implementation* semantics of loops and procedure calls, defined by their bodies, as well as their *specification* semantics, used in modular verification, where the effect of a procedure call is defined solely the procedure’s pre- and postcondition and the effect of a loop by its invariant. The implementation semantics is useful to discriminate between inconsistent and incomplete specifications; while the specification semantics makes it possible to generate executions in the presence of partial implementations, as well as to expose spurious executions permitted by incomplete specifications.

Our technique simplifies the constraints passed to the SMT solver, only targeting the values required for a particular symbolic execution. This avoids the solver getting bogged down when reasoning about complex specifications—a problem often arising with program provers—without need for additional guidance in the form of quantifier instantiation heuristics. The simplification also improves the predictability of test case generation. Combined with model minimization techniques, it produces short—often minimal-length—executions that are quite easy to read. While constraint simplification might also produce false positives (infeasible executions), the evaluation of Sec. 5 shows that this rarely happens in practice: the small risk amply pays off by producing easy-to-understand executions, symptomatic of the rough patches in the implementation

```
1  procedure Max(N: int, a: [int] int) returns (max: int)
2    ensures (∀ j: int • 0 ≤ j ∧ j < N ⟹ a[j] ≤ max);
3    ensures (∃ j: int • 0 ≤ j ∧ j < N ∧ a[j] = max);
4  {
5    var i: int;
6    i := 0;
7    max := 0;
8    while (i < N) {
9      if (a[i] > max) { max := a[i]; }
10     i := i + 1;
11   }
12 }
```

Fig. 1. Boogie procedure `Max` that finds the maximum element in an array. Both the specification and the implementation contain errors, and no loop invariant is provided.

or specification that require further attention. We also identify a subset of the annotation language for which no infeasible executions are generated.

We implemented our technique for the Boogie intermediate verification language, used as back-end of numerous program verifiers [18,4,26]. Working atop an intermediate language opens up the possibility of reusing the tool with multiple high-level languages and verifiers that already translate to Boogie. It also ensures that our technique is sufficiently general: Boogie is a small yet very expressive language (including both specification and imperative constructs), designed to support translations of disparate notations with their own supporting methodologies. Our implementation is available as a tool called Boogaloo, distributed as free software: `https://bitbucket.org/nadiapolikarpova/boogaloo/` and accessible through the web: `http://cloudstudio.ethz.ch/comcom/#Boogaloo`. For simplicity, in the paper we will use "Boogaloo" to denote the execution generation technique as well as its implementation, and will employ the self-explanatory Boogie syntax in the examples.

2 Illustrative Example

We give a concise overview of the capabilities of Boogaloo using a simple verification example: finding the maximum element in an integer array.[1] Fig. 1 shows a straightforward Boogie implementation as procedure `Max`, which inputs an integer `N`, denoting the array size, and a map[2] `a` that represents the array elements `a[0]`, . . ., `a[N-1]`; it returns an integer `max` for `a`'s maximum. The Boogie procedure includes specification in the form of two postconditions (**`ensures`**), formalizing the definition of maximum: `max` should be no smaller than any element of `a` (line 2); and it should be an element of `a` (line 3).

What happens if we try to verify procedure `Max`, as shown in Fig. 1, using Boogie? Verification fails with a vague error message ("`Postconditions on lines 2 and 3`

[1] The tool output messages in this section are abridged without sacrificing the gist of the original.

[2] In general, maps have an infinite domain in Boogie.

`might not hold.`") which is inconclusive and of little help to understand the source of failure. Rather, running Boogaloo on the same input generates concrete inputs that make the program fail; we get the message "`Postcondition on line 3 violated with N -> 0, a -> [], max -> 0`", which clearly singles out a problem with `Max`: the maximum of an empty array is undefined.

We can formalize the expectation that `Max` ought to be a partial function (undefined for empty arrays) as the precondition **`requires`** `N > 0`. Boogie's output does not, however, change if we add such a precondition: it still cannot establish either postcondition since it would need a loop invariant to reason about loops—no matter how simple they are. Instead, running Boogaloo on `Max` annotated with the precondition shows another input that triggers a failure: "`Postcondition on line 3 violated with N -> 1, a -> [0 -> -1], max -> 0`". This time the problem is with the implementation rather than with the specification: when `a` contains a single negative value, initializing `max` to `0` (line 7) does not work. With this concise concrete counterexample, it is easy to understand that the same problem occurs with any array containing only negative elements. Designing a correction is also routine: we change the initialization on line 7 to `max := a[0]`, which is well-defined thanks to the precondition `N > 0`.

We can see that the modified program—including precondition and new initialization of `max`—is finally correct. However, Boogie's behavior on it does not change at all: without a loop invariant, it still fails to prove either postcondition. Boogaloo, in contrast, can generate a number of test cases and successfully check all of them against the specification. While this still falls short of a formal correctness proof, it provides evidence that the program is indeed correct, and that all we have to do is strengthen the specification by adding a suitable loop invariant.

While we selected a simple example which can be briefly presented, we were able to demonstrate, in a nutshell, several fundamental issues of working with static program verifiers such as Boogie, and how Boogaloo can complement their weaknesses. Specifically, Boogaloo's capabilities to provide concrete inputs that show errors or amass evidence for correctness; and to work with the same programs used for verification including elements such as first-order quantification (lines 2 and 3), but without requiring specifications at all costs (a loop invariant). Another distinguishing, and practically crucially important, feature of Boogaloo is that it produces small (often minimal) tests: in the example, the smallest arrays and the smallest integer values exposing faults and discrepancies.

Comparison with Other Approaches. To further demonstrate the unique features of Boogaloo, let us consider the behavior of other approaches to complementing static program verification on the same example of procedure `Max`.

Assuming `Max` were a Boogie encoding produced from some high-level programming language, we could use standard testing tools on the source program to generate concrete inputs and discover failures. One problem is that first-order quantifications (and other features used by Boogie) are inexpressible using the simple Boolean expressions of standard programming languages. While the quantifications used in `Max` are bounded, and hence expressible using executable constructs such as finite iterations over arrays or list comprehensions, getting rid of quantifiers and other non-executable constructs is neither possible nor desirable in general. As soon as we look at examples more complex than Max, we need to express abstract properties potentially involving infinitely many elements, such as for framing and for reasoning about unbounded sequences of pointers to heap-allocated data. Even in an example as simple as sorting, if a sorting procedure takes a function pointer as argument to denote the comparison function, we need to express that it encodes a total order—something involving quantification over a potentially unbounded domain. More generally, we designed Boogaloo to work with the same proof-oriented annotated programs used by static verifiers, which involve features difficult to execute and normally not found in high-level programming languages.

Another option to debug Max is using the Boogie Verification Debugger (BVD [14]), which extracts concrete counterexamples from failed verification attempts. The relevance of such counterexamples is, however, limited in the presence of loops and procedure calls with incomplete specifications. On Max as in Fig. 1, BVD returns the assignment "N = 1, a = [], max = -900"; after adding N > 0 as precondition, it returns "N = 797, a = [], max = -900"; after fixing the implementation, it returns "N = 797, a = [0 -> -901], max = -901". These examples fail to point out the two errors in Max, because according to modular reasoning [17] and in the absence of an invariant, any loop is equivalent to assigning arbitrary values to program variables. While BVD's modular semantics helps debug incompleteness in specifications, it also enforces an "all-or-nothing" development style, where developers first have to get right the most complicated part (the invariants), before they can proceed with debugging the rest of the program. This lack of incrementality is what makes modular verification so hard in the first place.

It is possible to make Boogie use loop and procedure bodies instead of their specification by *unrolling* loops and *inlining* procedures U times, for a given $U \geq 0$. With unrolling, BVD finds counterexamples for executions where N $\leq U$, and in particular the same counterexamples constructed by Boogaloo. The approach, however, has its limitations. First, unrolling and inlining require users to guess a suitable U; since all longer executions are ignored, verification vacuously succeeds when the shortest counterexample requires $> U$ iterations or nested calls, without providing any concrete feedback. Second, unrolling of complex loops and inlining of recursive procedures scale poorly, as they consist of literally rewriting the code U times; Boogaloo, in contrast, uses symbolic execution techniques, which are less likely to incur blow up. Building a debugger on top of the Boogie verifier also means that it cannot generate passing executions (Boogie does not produce a model in case verification succeeds) and cannot help when the theorem prover gets bogged down. In contrast, Boogaloo uses simpler verification conditions, designed for predictable generation and readability of counter examples as opposed to sound proofs.

3 A Runtime Semantics of Boogie Programs

This section describes the syntax of Boogaloo programs (Sec. 3.1) and their operational semantics (Sec. 3.2). We use the following notation: $\mathbb{Z}$ is the set of mathematical integers; and $\mathbb{B}$ is the set $\{\top, \bot\}$ of Boolean values. A *map* m is a mathematical function from a domain $D_1 \times \cdots \times D_n$, for $n \geq 0$, to a codomain D_0;

$$
\begin{array}{rcl}
P & ::= & D^* \\
D & ::= & \texttt{type}\ tid \mid \texttt{var}\ V : T \\
 & \mid & \texttt{procedure}\ pid\ (\ \langle V : T\rangle^*\ )\ \texttt{returns}\ (\ \langle V : T\rangle^*\ )\ \texttt{modifies}\ (V^*)\ \langle \texttt{\{}\ \langle V : T\rangle^*\ \langle lid : S\rangle^*\ \texttt{\}}\rangle^? \\
S & ::= & S\texttt{;}S \mid \texttt{havoc}\ V^+ \mid V^+ := E^+ \mid \texttt{call}\ V^* := pid\ (E^+) \mid \texttt{assume}\ E \mid \texttt{goto}\ lid^+ \mid \texttt{return} \mid R \\
R & ::= & \texttt{halt} \mid \texttt{abort} \mid \texttt{pick} \\
T & ::= & \texttt{bool} \mid \texttt{int} \mid [\,T^+\,]\,T \mid tid \\
E & ::= & C \mid V \mid E\,[\,E^+\,] \mid E\,[\,E^+ := E\,] \mid \texttt{old}\ E \\
 & \mid & UOp\ E \mid E\ BOp\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \mid QOp\ \langle V : T\rangle^+\ \bullet\ E \\
V & ::= & vid \qquad\qquad C\ ::=\ \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots \\
UOp & ::= & - \mid \neg \qquad BOp\ ::=\ + \mid - \mid \cdots \mid < \mid \leq \mid = \mid \cdots \mid \wedge \mid \vee \mid \cdots \qquad QOp\ ::=\ \exists \mid \forall \mid \lambda
\end{array}
$$

Fig. 2. Desugared language supported by Boogaloo, consisting of programs P, declarations D, statements S, types T, and expressions E. Angular brackets $\langle\ \rangle$ are part of the grammar metalanguage, used to mark optional (?) or repeated ($*$, $+$) expressions.

square brackets denote map applications. Whenever convenient, we see m as a set of $(n+1)$-tuples: $m \subset D_1 \times \cdots \times D_n \times D_0$ such that $(d_1, \ldots, d_n, d_0) \in m$ iff $m[d_1, \ldots, d_n] = d_0$. $\mathrm{dom}(m)$ and $\mathrm{rng}(m)$ denote the *domain* and *range* of m; m is *total* if $\mathrm{dom}(m) = D_1 \times \cdots \times D_n$, and *finite* if $|\mathrm{dom}(m)| \in \mathbb{Z}$; $m[d_1, \ldots, d_n \mapsto d]$ denotes a map m' identical to m except that $m'[d_1, \ldots, d_n] = d$. We overload this notation to denote variable substitution: if $e, y_1, \ldots, y_n$ are expressions, and $x_1, \ldots, x_n$, are variable names, $e[x_1, \ldots, x_n \mapsto y_1, \ldots y_n]$ denotes e with all occurrences of x_k replaced by y_k, for $k = 1, \ldots, n$.

3.1 Input Language

Boogaloo desugars generic Boogie programs [17] into the simpler language described in Fig. 2. Programs P are lists of declarations D, whose order is immaterial. Declarations include uninterpreted **`type`**s, global **`var`**iables, and **`procedure`**s with input parameters, output parameters (**`returns`**), global variables the procedure may modify (**`modifies`** clause), and body (between braces). Procedure bodies consist of local variable declarations followed by a list of labeled statements S. The latter include sequential composition, regular and nondeterministic assignment ($:=$ and **`havoc`**, possibly in parallel to multiple variables), procedure **`call`**, **`assume`**, nondeterministic **`goto`** a set of label identifiers, and abrupt **`return`** to the caller procedure, as well as *directives* R described shortly. Expressions must be properly typed as **`bool`**eans, **`int`**egers, maps $[T_1, \ldots, T_n]T_0$ from arbitrary domain $(T_1, \ldots, T_n)$ and codomain T_0 types, and user-defined uninterpreted types[3]. Expressions E include literal constants C, variables V, map applications $m[t_1, \ldots, t_n]$, map updates $m[t_1, \ldots, t_n := t]$, **`old`** expressions which refer to the value of an expression when the procedure was entered, plus the usual applications of unary operators UOp, binary operators BOp, a ternary **`if/then/else`** operator, and quantifications and lambda expressions QOp.

The *directives* **`halt`**, **`abort`**, and **`pick`** are Boogaloo-specific and characterize symbolic executions: **`halt`** terminates the current execution with success (marking *pass-*

[3] While Boogaloo supports Boogie's type constructors with arguments, as well type parameters in procedures and maps, we do not include them in the discussion for simplicity.

ing executions); `abort` also terminates the current execution but with error (marking *failing* executions); `pick` forces the interpreter to resolve nondeterminism by trying to build a concrete state out of the current symbolic constraints. Boogaloo automatically inserts a `halt` at the end of every control path in the input program; and uses `abort` to desugar **`assert`** statements as follows. A Boogie statement **`assert`** `B`, where `B` is a Boolean expression, indicates that `B` must hold in every non-failing execution reaching the statement; **`assume`** `B`, on the other hand, indicates that only executions where `B` holds upon reaching the **`assume`** are feasible. Therefore, Boogaloo expresses the semantics of **`assert`** `B` using **`assume`**, `abort`, and nondeterministic choice as: **`goto`** `T, F; F:` **`assume`** `¬ B; abort; T:` **`assume`** `B`. Boogaloo also injects a `pick` statement right before every `halt` and `abort`, so that every terminating execution gets a concrete state. Boogaloo automatically instruments programs with the directives R, based on different strategies (see Section 5) so that one can use Boogie programs without additional annotations.

The rest of the desugaring of Boogie into the language of Fig. 2 is fairly standard. We rewrite function declarations **`function`** `f`$(T_1, \ldots, T_n)$ **`returns`**(T_0) into constants **`const`** `f`: $[T_1, \ldots, T_n]\, T_0$ of map type, and express the corresponding function definitions as axioms. In turn, we express axioms and other specification constructs—**`where`** clauses (used to constrain the values of uninitialized variables), pre- and postconditions, and loop invariants—using **`assume`** and **`assert`** reflecting the standard semantics. We replace constants with variables. Finally, we transform procedure bodies into sets of *basic blocks* (labeled sequential blocks of code that end with a **`return`** or **`goto`**) using standard techniques [17].

3.2 Runtime Operational Semantics

We now describe an operational semantics for the language in Fig. 2. The presentation focuses on the most interesting aspects while omitting standard details.

Let us start with an informal overview of the basic concepts. The operational semantics describes the effect, on the symbolic state, of executing statements. The symbolic state associates symbolic values to program variables in scope. Executing some statements may involve enforcing constraints between symbolic values; the most obvious example is that of **`assume`** `P`: the symbolic values associated to variables mentioned in `P` must satisfy `P` in every computation that continues after the statement. Therefore, the symbolic state includes *constraints* which are updated as execution progresses. Finally, `pick` directives select *concrete* values that satisfy the current constraints; executions continue after `pick` with the selected concrete state components replacing the corresponding symbolic state components (but subsequent statements will be executed symbolically until the next `pick`). In this sense, symbolic executions are *speculative*, in that the constraints may not have a solution (infeasible executions), and *nondeterministic*, in that the constraints may have more than one solution; `pick` forces the interpreter to resolve nondeterministic choice before continuing. Another source of nondeterminism comes from executing **`goto`**s with multiple labels; such choices are resolved immediately, resulting in explicit path enumeration. Since Boogaloo injects `pick` statements at every terminating location, it can provide concrete input and output values for every

$$\text{LOG-IN}\ \dfrac{\ell \in \mathrm{dom}(\lambda)}{\llbracket \ell \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ \lambda[\ell]} \qquad \text{LOG-OUT}\ \dfrac{\ell \notin \mathrm{dom}(\lambda)}{\llbracket \ell \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ \ell} \qquad \text{QUANT-F}\ \dfrac{\llbracket \widetilde{Q}_1 x_1 \cdots \widetilde{Q}_n x_n \bullet\ \neg q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \top}{\llbracket Q_1 x_1 \cdots Q_n x_n \bullet q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \bot}$$

$$\text{VAR-IN}\ \dfrac{v \in \mathrm{dom}(\sigma) \qquad \llbracket \sigma[v] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ e}{\llbracket v \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ e} \qquad \text{VAR-OUT}\ \dfrac{v \notin \mathrm{dom}(\sigma) \qquad \ell \text{ is fresh} \qquad \sigma' = \sigma[v \mapsto \ell]}{\llbracket v \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell}$$

$$\text{SEL-IN}\ \dfrac{\llbracket (m, \boldsymbol{a}) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ (\ell_m, \boldsymbol{a}') \qquad \boldsymbol{a}' \in \mathrm{dom}(\mu'[\ell_m]) \qquad \llbracket \mu'[\ell_m][\boldsymbol{a}'] \rrbracket\ {}^{\mathcal{E}'}{=}^{\mathcal{E}'}\ e}{\llbracket m[\boldsymbol{a}] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ e}$$

$$\text{SEL-OUT}\ \dfrac{\ell \text{ is fresh} \qquad \llbracket (m, \boldsymbol{a}) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ (\ell_m, \boldsymbol{a}_1) \qquad \boldsymbol{a}_1 \notin \mathrm{dom}(\mu_1[\ell_m]) \qquad m' = [\mu_1[\ell_m][\boldsymbol{a}_1] \mapsto \ell]}{\llbracket m[\boldsymbol{a}] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell_m \mapsto m'], \upsilon_1 \rangle}$$

$$\text{UPD}\ \dfrac{\ell \text{ is fresh} \qquad \llbracket (m, \boldsymbol{a}, e) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ (\ell_m, \boldsymbol{a}_1, e_1) \qquad m' = [\mu_1[\ell_m][\boldsymbol{a}_1] \mapsto e_1]}{\llbracket m[\boldsymbol{a} := e] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell \mapsto m'], \upsilon_1 \cup \{\forall \boldsymbol{x} \bullet \boldsymbol{x} \neq \boldsymbol{a}_1 \Rightarrow \ell[\boldsymbol{x}] = \ell_m[\boldsymbol{x}]\} \rangle}$$

$$\text{LAMBDA}\ \dfrac{\ell \text{ is fresh} \qquad \llbracket e \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ e_1 \qquad \sigma_1(\boldsymbol{x}) = \boldsymbol{\ell}_1}{\llbracket \lambda \boldsymbol{x} \bullet e \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1, \upsilon_1 \cup \{\forall \boldsymbol{x} \bullet e_1[\boldsymbol{\ell}_1 \mapsto \boldsymbol{x}]\} \rangle}$$

$$\text{QUANT-T}\ \dfrac{\mathsf{Skolem}[Q_1 x_1 \cdots Q_n x_n \bullet q]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1}\ \forall \boldsymbol{y} \bullet q_1 \qquad \llbracket q_1 \rrbracket\ {}^{\mathcal{E}_1}{=}^{\mathcal{E}_2}\ q_2 \qquad \sigma_2(\boldsymbol{y}) = \boldsymbol{\ell}}{\llbracket Q_1 x_1 \cdots Q_n x_n \bullet q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ \top \qquad \mathcal{E}' = \langle \sigma_2, \mu_2, \upsilon_2 \cup \{\forall \boldsymbol{y} \bullet q_2[\boldsymbol{\ell} \mapsto \boldsymbol{y}]\} \rangle}$$

Fig. 3. Symbolic evaluation (significant cases)

feasible execution, while still availing of symbolic representation to limit the combinatorial explosion introduced by the inherently nondeterministic nature of specifications.

The main source of complexity in executing Boogie programs lies in solving constraints, in particular when they involve universal quantifiers and uninterpreted maps with infinite domains. Even though state-of-the-art SMT solvers can decide satisfiability of quantified formulas in many practical cases, they can hardly generate readable "natural" infinite models. In light of these difficulties, we drop Boogie's standard interpretation—where all maps are total—and replace it with a *finitary* interpretation where maps have finite domains. Finite, small instances are sufficient to expose errors and inconsistencies in most programs; Alloy's techniques are based on a similar "small scope" hypothesis [7]. We also treat universally quantified constraints in a special way: the `pick` directive *finitizes* them, that is turns them into simpler quantifier-free constraints. Finitization is in general unsound, but Sec. 5 demonstrates that the precision loss is normally acceptable, especially if the goal is finding inconsistencies and errors.

Concrete Values. Each Boogie type corresponds to a set of concrete values: **`bool`** corresponds to $\mathbb{B}$, **`int`** corresponds to $\mathbb{Z}$; each user-defined **`type`** `U` corresponds to a countable uninterpreted set U; each map type $[\mathsf{T}_1, \ldots, \mathsf{T}_n]\,\mathsf{T}_0$ corresponds to the set of all *finite* maps from $T_1 \times \cdots \times T_n$ to T_0, where T_k is the set of concrete values of type T_k, for $k = 0, \ldots, n$. K denotes the union of all concrete value sets.

Symbolic Values correspond to the set Σ defined as:

$$\Sigma \ ::= \ K \mid L \mid \mathit{UOp}\ \Sigma \mid \Sigma\ \mathit{BOp}\ \Sigma \mid \textbf{if}\ \Sigma\ \textbf{then}\ \Sigma\ \textbf{else}\ \Sigma\,,$$

where K is the set of concrete values defined above; unary *UOp* and binary *BOp* operators are in Fig. 2, and L denotes a set of *logical variables* of the same types as the concrete values. A logical variable ℓ of type T corresponds to a symbolic placeholder (a "promise") for a concrete value of type T. To represent quantifiers in constraints, we also introduce a set of *universal symbolic values* $\Sigma_\forall ::= \forall \langle V : T \rangle^+ \bullet \Sigma_V$, where Σ_V is a symbolic expression, which can also include bound variables V. Given a set X of expressions, $\mathrm{LV}(X)$ is the set of all logical variables appearing in X.

Symbolic States. A symbolic state (environment) is a tuple $\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle$, where the store $\sigma : V \rightarrow \Sigma$ maps variables to symbolic values; the logical store $\lambda : L \rightarrow \Sigma$ maps scalar logical variables to symbolic values; the map store $\mu : L \rightarrow (\Sigma^* \rightarrow \Sigma)$ maps map logical variables to symbolic maps; $\kappa \subset \Sigma$ is a finite set of *simple state constraints*; $\upsilon \subset \Sigma_\forall$ is a finite set of *universal state constraints*; and τ is one of ✧, ✓, ✗, denoting an intermediate state (✧), or the final state of a passing (✓) or failing (✗) execution. The map store associates logical variables of map type to *symbolic maps*: finite maps whose domain and range are in Σ; symbolic maps extend their finite domains as execution progresses; `pick` concretizes their domain and range, turning symbolic maps into concrete ones.

Expression Evaluation. Let E denote the set of all expressions defined by E in Fig. 2 but whose atoms range over $C \cup V \cup L$ (i.e., including logical variables L). The *evaluation* of an expression $e \in \mathsf{E}$ in an environment $\mathcal{E}$ is a symbolic value in Σ. We use the notation: $[\![e]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} e'$ to denote that $e \in \mathsf{E}$ evaluates in $\mathcal{E}$ to $e' \in \Sigma$. As we detail shortly, evaluating an expression may change the environment; correspondingly, $\mathcal{E}'$ denotes the updated environment, whose components are written $\langle \sigma', \lambda', \mu', \kappa', \upsilon', \tau' \rangle$. When convenient, we extend this notation to *sequences* $\boldsymbol{e} = e_1, \ldots, e_n$ of expressions, evaluated one after another. Fig. 3 shows the evaluation rules for the most interesting expression kinds. Since evaluation does not change the λ, κ, and τ environment components, Fig. 3 omits them. Also notice that evaluating a symbolic value never changes the environment, and every concrete value evaluates to itself.

Rules LOG-IN and LOG-OUT describe the simple cases of evaluating a logical variable ℓ: if $\lambda[\ell]$ is defined, it yields ℓ's evaluation; otherwise, ℓ evaluates to itself.

Rules VAR-IN and VAR-OUT describe the evaluation of a (program) variable v. If it has already been initialized, the evaluation of $\sigma[v]$ gives its symbolic value. Otherwise (VAR-OUT), such as when v enters the scope or after executing **havoc** v, $\sigma[v]$ gets initialized to a fresh logical variable ℓ.

The rules for map selection are similar to those for variables but target the map store μ: if a map selection has already been evaluated, its symbolic value is returned (SEL-IN); otherwise, a fresh logical variable is generated and stored in μ (SEL-OUT).

Rules UPD and LAMBDA deal with evaluating expressions of map type for updates and lambda abstractions. Both rules introduce a fresh map logical variable and add to υ a universally quantified constraint that defines the map. Thus, map expressions (variables, updates, and lambdas) always evaluate to a logical variable; this justifies using the evaluation ℓ_m of m as an index in μ in the premises of SEL-IN, SEL-OUT, and UPD.

The rules for quantified expressions are non-deterministic. Consider an expression $\mathcal{Q} = Q_1 x_1 \cdots Q_n x_n \bullet q$ in prenex normal form, where $n > 0$, Q_k is one of $\forall$ and $\exists$ for all k's, and q is quantifier-free. Rule QUANT-T evaluates $\mathcal{Q}$ to true and adds it to the universal constraints υ after the following transformation. First, $\mathcal{Q}$ is Skolemized as $\forall \boldsymbol{y} \bullet q_1$, where $\boldsymbol{y}$ is the subsequence of $x_1, \ldots, x_n$ including only those x_k's for which Q_k is $\forall$; $\mathcal{E}_1$ is the environment after Skolemization, which contains fresh logical variables for the Skolem functions introduced by the process. Evaluating q_1 in $\mathcal{E}_1$ yields some q_2 where the bound variables $\boldsymbol{y}$ map to fresh logical variables $\boldsymbol{\ell}$; after performing the substitution $q' = q_2[\boldsymbol{\ell} \mapsto \boldsymbol{y}]$, $\forall \boldsymbol{y} \bullet q'$ is added to υ. Rule QUANT-F, which evaluates $\mathcal{Q}$ to false, follows by duality ($\widetilde{\forall}$ denotes $\exists$, and $\widetilde{\exists}$ denotes $\forall$).

Procedure Call Semantics. The precise semantics of procedure calls involves several details to support recursion—mainly, maintaining a stack of environments and correspondingly keeping track of scope. We overlook these tedious aspects and focus on the gist of the semantics of a call to a generic procedure P (before desugaring):

$$\textbf{procedure}\ \mathtt{P}\ (\boldsymbol{a})\ \textbf{returns}\ (\boldsymbol{o})\ \textbf{requires}\ p\ \textbf{ensures}\ q\ \textbf{modifies}(\boldsymbol{m})\ \ \langle\{B\}\rangle^{?}$$

with formal input $\boldsymbol{a}$ and output $\boldsymbol{o}$ parameters, modified global variables $\boldsymbol{m}$, body B, and pre- and postcondition p and q. The desugaring of Sec. 3.1 turns pre- and postcondition into checks at the call site:

$$\textbf{assert}\ p[\boldsymbol{a} \mapsto \boldsymbol{u}];\ \textbf{call}\ \boldsymbol{v} := \mathtt{P}(\boldsymbol{u});\ \textbf{assume}\ q[\boldsymbol{a}, \boldsymbol{o} \mapsto \boldsymbol{u}, \boldsymbol{v}];$$

(For brevity, we do not discuss the handling of **old** expressions in postconditions.) It also generates a modified procedure body B' to reflect the implementation or specification semantics, according to whether P has a body or not: if B is defined, B' adds an **assert** q before each **return** statement in B; if B is not defined, B' consists of the single statement **havoc** $\boldsymbol{o}, \boldsymbol{m}$. The effect of the call statement is then given by $B'[\boldsymbol{a} \mapsto \boldsymbol{u}]$@entry where @entry denotes the basic block of statements at procedure P's entry. Even though Boogaloo defaults to implementation semantics whenever a body is available, one can always switch to the specification semantics for a particular procedure by commenting out its body.

Operational Semantics. Fig. 4 describes the operational semantics of statements other than procedure calls, using the notation $\mathcal{E}\ {-}\mathtt{S}{\rightsquigarrow}\ \mathcal{E}'$ to denote that executing statement S changes the environment $\mathcal{E}$ into $\mathcal{E}'$. Rules are applicable only if $\tau =$ ✧, that is if the computation has not terminated yet; after rules HALT or ABORT have changed τ to passing ✓ or failing ✗ no rule is applicable and hence the computation terminates.

Rules SEQ for sequential composition, GOTO for branch, and RETURN for abrupt termination are standard, using the notation @caller to denote the basic block beginning after the current call in the caller procedure. Rule GOTO is clearly nondeterministic.

Rule ASSUME adds the assumed Boolean formula to the set κ of state constraints. Rule HAVOC "forgets" the symbolic value of the variable v, as if uninitialized. Rule ASS updates the symbolic value in σ associated to the assigned variable v.

The most interesting rule is PICK, which details how **pick** concretizes symbolic states. It extends κ into κ', adding map instance constraints $\kappa_\mu = \{m[\boldsymbol{x}] = y \mid$

$$\text{SEQ}\;\frac{\tau = \text{✧} \quad \mathcal{E} \;{-}\mathtt{I}{\rightsquigarrow}\; \mathcal{E}_1 \quad \mathcal{E}_1 \;{-}\mathtt{J}{\rightsquigarrow}\; \mathcal{E}'}{\mathcal{E} \;{-}\mathtt{I;J}{\rightsquigarrow}\; \mathcal{E}'} \qquad \text{GOTO}\;\frac{\tau = \text{✧} \quad k \in \{1, \ldots, n\} \quad \mathcal{E} \;{-}@\mathsf{x}_k{\rightsquigarrow}\; \mathcal{E}'}{\mathcal{E} \;{-}\mathbf{goto}\ \mathsf{x}_1, \ldots, \mathsf{x}_n{\rightsquigarrow}\; \mathcal{E}'}$$

$$\text{RETURN}\;\frac{\tau = \text{✧} \quad \mathcal{E} \;{-}@\mathsf{caller}{\rightsquigarrow}\; \mathcal{E}'}{\mathcal{E} \;{-}\mathbf{return}{\rightsquigarrow}\; \mathcal{E}'} \qquad \text{ASSUME}\;\frac{\tau = \text{✧} \quad \llbracket \mathsf{P} \rrbracket \;{}^{\mathcal{E}}{=}^{\mathcal{E}'}\, p}{\mathcal{E} \;{-}\mathbf{assume}\ \mathsf{P}{\rightsquigarrow}\; \langle \sigma', \lambda', \mu', \kappa' \cup \{p\}, \upsilon', \tau' \rangle}$$

$$\text{HAVOC}\;\frac{\tau = \text{✧}}{\mathcal{E} \;{-}\mathbf{havoc}\ \mathsf{v}{\rightsquigarrow}\; \langle \sigma \setminus \{(\mathsf{v}, \sigma[\mathsf{v}])\}, \lambda, \mu, \kappa, \upsilon, \tau \rangle} \qquad \text{ASS}\;\frac{\tau = \text{✧} \quad \llbracket \mathsf{e} \rrbracket \;{}^{\mathcal{E}}{=}^{\mathcal{E}'}\, e'}{\mathcal{E} \;{-}\mathsf{v} := \mathsf{e}{\rightsquigarrow}\; \langle \sigma[\mathsf{v} \mapsto e'], \lambda', \mu', \kappa', \upsilon', \tau' \rangle}$$

$$\text{HALT}\;\frac{\tau = \text{✧}}{\mathcal{E} \;{-}\mathtt{halt}{\rightsquigarrow}\; \langle \sigma, \lambda, \mu, \kappa, \upsilon, \checkmark \rangle} \qquad \text{ABORT}\;\frac{\tau = \text{✧}}{\mathcal{E} \;{-}\mathtt{abort}{\rightsquigarrow}\; \langle \sigma, \lambda, \mu, \kappa, \upsilon, \text{✗} \rangle}$$

$$\text{PICK}\;\frac{\tau = \text{✧} \quad \kappa' = \kappa \cup \kappa_\mu \cup \varPhi(\upsilon) \quad \mathrm{dom}(\varLambda) = \{\ell \in \mathrm{LV}(\kappa') \mid \ell \text{ scalar}\} \quad \mathcal{E}' = \langle \sigma, \lambda \cup \varLambda, \mu, \emptyset, \upsilon, \tau \rangle \quad \llbracket \textstyle\bigwedge \kappa' \rrbracket \;{}^{\mathcal{E}'}{=}^{\mathcal{E}'}\, \top}{\mathcal{E} \;{-}\mathtt{pick}{\rightsquigarrow}\; \mathcal{E}'}$$

Fig. 4. Symbolic execution: operational semantics. All rules describe transformations of a generic symbolic state $\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle$.

$(m, \boldsymbol{x}, y) \in \mu\}$, which express the information in μ about symbolic maps; as well as *finitized* universal constraints $\varPhi(\upsilon)$. It then picks a *solution* $\varLambda : L \to K$ of κ': an assignment of concrete values to scalar logical variables for which the conjunction of constraints in κ' evaluates to true. It finally adds the picked solution to λ and drops the solved constraints. The rule is nondeterministic, as κ' might have multiple solutions. When κ' has no solutions, the rule cannot apply and executions gets stuck at `pick`: we call such executions *infeasible*. The rule is also agnostic with respect to the exact method of solving simple constraints, as well as to the finitization mapping $\varPhi$. The only requirement on $\varPhi : \varSigma_\forall{}^* \to \varSigma$ is that it is an over-approximation: any valid solution of υ is also a solution of $\varPhi(\upsilon)$. In practice, $\varPhi$ performs quantifier instantiation: it replaces a quantified formula $\forall \boldsymbol{x} \bullet q$ with a finite set of quantifier-free formulas $\{q[\boldsymbol{x} \mapsto \boldsymbol{e}] \mid e \in R\}$, for some finite set R of "relevant" symbolic values. The challenge is to choose an R that is large enough to describe all relevant values in the current environment, yet small enough to produce constraints that can be solved efficiently. Sec. 4 gives more details about Boogaloo's finitization procedure.

Boogaloo vs. Boogie Semantics. How does the operational semantics discussed in this section compare with the original Boogie semantics? For this discussion, a *semantics* of a program P is a set of sequences of concrete states, corresponding to its feasible terminating executions; a (concrete) state $\mathcal{C} = \langle \sigma, \tau \rangle$ consists of a store σ (involving finitely many variables) and a termination flag $\tau \in \{\text{✧}, \checkmark, \text{✗}\}$. A state $\mathcal{C}$ is *finitary* if it involves only finite maps: for all $m \in \mathrm{dom}(\sigma)$, $|\mathrm{dom}(\sigma[m])|$ is finite; otherwise, it is *infinitary*. A state $\mathcal{C}_F$ *finitizes* another state $\mathcal{C}$ (written $\mathcal{C}_F \sqsubseteq_{\mathcal{F}} \mathcal{C}$) iff $\mathcal{C}_F$ is finitary, $\tau_F = \tau$, $\mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma)$ and, for all map variables $m \in \mathrm{dom}(\sigma)$, $\sigma_F[m] \subseteq \sigma[m]$. A sequence e of states is finitary (infinitary) if all its elements are finitary (infinitary); e finitizes another sequence e' if every state of e finitizes the corresponding state of e'.

For a program P, $\mathcal{I}[P]$ denotes the Boogie semantics defined in [17], which is infinitary since all maps are total; and $\mathcal{F}[P]$ denotes the finitary semantics of this paper, where all maps have finite domains. Assuming perfect constraint-solving capabilities, the only aspect where $\mathcal{F}$ may drop information w.r.t. $\mathcal{I}$ is in the rule PICK, and more precisely in the finitization mapping $\varPhi$. The requirement that $\varPhi$ be an over-approximation implies that every Boogie execution is finitized by *some* Boogaloo execution. The converse does not hold in general: in particular, for some programs $S, \mathcal{I}[S] = \emptyset$ but $\mathcal{F}[S] \neq \emptyset$ contains executions (which we regard as spurious). For example, the following program:

```
var a: [int] int;
assume (∀ i, j: int • i < j ⟹ a[i] < a[j]);
assume a[0] = 0 ∧ a[1000] = 1;
```

has no feasible executions in $\mathcal{I}$, while the current implementation of Boogaloo produces a passing execution where the quantified constraint is only instantiated for $\texttt{i} = 0$ and $\texttt{j} = 1000$. Sec. 5 demonstrates that such unsound executions are infrequent in practice, and, even when they occur, workarounds are possible, for example forcing the evaluation on more points by accessing them in a loop. Also, Boogaloo's implementation of Φ does not produces spurious executions for programs where all quantified constraints are derived from terminating recursive function definition (see Sec. 4).

There is an additional source of discrepancies between $\mathcal{I}$ and $\mathcal{F}$, due to the fact that Boogie always uses the specification semantics for loops and procedure calls, while Boogaloo defaults to the implementation semantics, which might contain fewer executions. This discrepancy between the two semantics is a useful feature, which makes it possible to debug programs in presence of incomplete specifications. The specification semantics is still available on demand in Boogaloo: it is sufficient to replace an imperative construct with its specification.

4 Boogaloo: Implementation Details

This section presents some details of the Boogaloo tool—our prototype implementation of the approach described in the previous sections. The tool takes as input a Boogie source file and a procedure name as entry point, and produces a set of feasible executions, characterized by their concrete initial and final states. Boogaloo is implemented in Haskell, and uses the SMT solver Z3 [5] in the back-end.

Finitizing Universal Constraints. The choice of the finitization mapping Φ plays an important role. Our experiments suggest that the powerful quantifier instantiation strategies available in SMT solvers such as Z3 have some downsides when applied to solve constraints generated by executing Boogie programs, as their performance is unpredictable unless additional user input (in the form of "triggers") is provided. Instead, Boogaloo preprocesses constraints using a simple strategy, based on the observation that universally quantified formulas are typically used to axiomatize uninterpreted maps; since we are only interested in finitely many points (those stored in μ), we just instantiate the bound variables at those points. Quantified constraints that do not contain map applications are simply ignored; the examples in Sec. 5 suggests that this finitization strategy is not too restrictive on typical verification examples.

This is how Boogaloo implements Φ for a formula $\forall \boldsymbol{x} \bullet P(\boldsymbol{x})$. For each term $m[\boldsymbol{y}]$ in P such that $\boldsymbol{y}$ includes some bound variable (i.e., $\boldsymbol{y} \cap \boldsymbol{x} \neq \emptyset$), Boogaloo extracts a parametrized map constraint of the form $(m, Q(\boldsymbol{y}, m[\boldsymbol{y}]))$, where Q is a subformula of P including the term, and $\boldsymbol{y}$ are the parameters free in Q; if $\boldsymbol{x} \not\subseteq \boldsymbol{y}$, then Q is itself quantified. For example, $\forall i \bullet a[i] > i \wedge b[i, 0] = 1$ determines two parametrized constraints: $(a, a[i] > i)$ and $(b, j = 0 \implies b[i,j] = 1)$; whereas $\forall i, j \bullet i < j \implies c[i] < c[j]$ determines: $(a, \forall j \bullet i < j \implies c[i] < c[j])$.

Boogaloo evaluates parametrized constraints for a given map store μ iteratively: pick an element $p = (m, \boldsymbol{e}, s)$ of μ, instantiate all parametrized constraints for m with $\boldsymbol{e}$ and evaluate them, and mark p; repeat until all elements of μ are marked. If a Q in a parametrized constraint (m, Q) contains quantifiers, instantiating m triggers the generation of new parametrized constraints from Q.

Since evaluating a parametrized constraint may add new points to μ, termination of the evaluation procedure is not guaranteed in the presence of recursive formulas, which determine constraints $(m, Q(\boldsymbol{y}, m[\boldsymbol{y}]))$ where Q also contains applications of m to elements other than $\boldsymbol{y}$. For example, an axiomatization of the factorial f as $f[0] = 1$ and $\forall i \bullet i > 0 \implies f[i] = i \cdot f[i-1]$ determines the constraint $q = (f, i > 0 \implies f[i] = i \cdot f[i-1])$. If $\mu[f] = (\ell_0, \ell_1)$, evaluating q for $i = \ell_0$ introduces a new map application at $\ell_0 - 1$, which then introduces an application at $\ell_0 - 2$, and so on. Boogaloo evaluates such recursive constraints using fair unrolling similarly to [24], based on the notion of guard: a parametrized constraint is *guarded* if has the form $(m, G(\boldsymbol{y}) \implies B(\boldsymbol{y}))$. When Boogaloo's iterative evaluation picks an element $p = (m, \boldsymbol{e}, s)$, it nondeterministically chooses a subset D of all guarded constraints for m and "disables" them in the evaluation determined by p: for a parametrized constraint $q = (m, G \implies B)$, it evaluates the constraint $\neg G$ if $q \in D$, and $G \wedge B$ otherwise. For the "right" selection of D, recursive definitions are disabled, so that they do not add new points to μ and evaluation terminates. In the factorial example, there are two choices for $f[\ell_0]$: disabling or enabling the guarded constraint. Disabling it terminates the finitization process, producing an execution with $\ell_0 = 0$; enabling the guarded constraints produces one iteration (for $f[\ell_0 - 1]$), which in turn recursively leads to the same two choices, and so on. Unlike [24], which works only with function definitions and thus assumes that guards are mutually exclusive and cover all cases, Boogaloo's fair enumeration applies to guards of any form and constraints other than equality; it also provides an option to limit the number of unrollings, because recursive constraints may be not well-founded (a sufficient condition for termination).

Nondeterminism. There are four sources of nondeterminism in Boogaloo semantics: evaluation of quantified expressions, **goto**s, and pick—involving the disabling of guarded constraints in Φ and constraint solving to select a solution Λ. Boogaloo enumerates nondeterministic choices using backtracking monads (e.g. [9]). The command-line interface currently offers depth-first and fair exploration strategies, but the implementation easily accommodate others parametrically.

When executing **goto** statements, the order in which labels are tried may affect progress: if the first chosen label leads back to the same statement, execution gets stuck in a loop. To avoid this situation, Boogaloo keeps track of how often each label was taken along the current execution path, and always tries labels in ascending order of their frequencies (least frequent first). This strategy also has the nice effect of enumerating shorter executions before longer ones in the long run. A similar strategy applies to disabling parametrized constraints.

Since symbolic computation is speculative, it introduces the risk that long computations are constructed only to realize, when solving the symbolic constraints, that they are infeasible. This risk is mitigated by the enumeration technique, which produces short execution first. Moreover, whenever a constraint evaluates to the concrete value $\perp$, the current execution path is immediately aborted. This mitigates the overhead incurred by nondeterministic evaluation of quantified expressions: such expressions are likely to occur inside **assume** statements, thus branches where they evaluate to false are immediately abandoned. Additionally, Boogaloo transparently tests the satisfiability of the current constraints κ at various points during an execution, and proceeds only if the constraints are satisfiable; unlike **pick** which may enumerate multiple solutions, a satisfiability check does not cause additional nondeterminism. One can still explore the trade-off between few expensive symbolic executions and many cheap concrete executions by adding **pick** directives at arbitrary points.

Minimization. In addition to producing short executions first, Boogaloo also uses a minimization technique based on binary search (similar to the one in [12]) in order to favor *small* integers for concrete values. In our experience, this significantly improves readability: for example, a constraint "x is positive and divisible by 5" with minimization produces the most natural solution $x = 5$ as first witness.

5 Experimental Evaluation

We evaluated Boogaloo on a choice of 15 examples from various sources[4]. Tab. 1 lists the programs and some data about them. The bulk of the evaluation targets the *verification* of algorithms of various kinds, listed in the top part of the table. For each of these problems, we constructed a correct version equipped with consistent but generally incomplete specifications, and a buggy one, obtained by injecting implementation or specification errors. We ran Boogaloo on both versions, with the goal of generating executions: passing executions for the correct programs, and failing executions exposing the bug for the buggy programs. The rest of the programs, in the bottom part of Tab. 1, are examples of *declarative* programming, which exercise Boogaloo's constraint solving capabilities to generate outputs satisfying given properties, in the absence of imperative implementations. We now briefly mention the most interesting features of our examples, and summarize the experimental results.

Verification. The majority of the programs in the top part of Tab. 1 are slightly adapted examples from the Boogie project repository[5], verification competitions [10], or previous work [6,14]; they contain features that exercise various aspects of the test-case generation process. Strong preconditions (such as an array being sorted in BinarySearch or being a permutation in Invert) make generating valid executions challenging using standard testing enumeration techniques. Inlining (available in Boogie) scales poorly with the recursive procedure calls of Fibonacci and QuickSort. The specifications of BinarySearch, BubbleSort, QuickSort, and Invert use nested universal quantifiers with bound variables mentioned in different predicates. QuickSort PI (partial implementation) is a

[4] Examples are available online at
`http://se.inf.ethz.ch/people/polikarpova/boogaloo/`

[5] `http://boogie.codeplex.com/`

Table 1. Programs tested with Boogaloo: name, main features, lines of code, number of specification functions, annotations (`axiom`s A, `assert`s S, `assume`s U, `requires` R, `ensures` E, loop `invariant`s I), time to generate passing executions, time to generate failing executions for the buggy version. Times in seconds, rounded to the nearest integer: for a given input size N, time t_Σ with fully symbolic execution and t_C with concretization. ∞ denotes a timeout of 180 seconds.

PROGRAM	FEATURES	LOC	FUN	ANNOTATIONS						TIME			BUG		
				A	S	U	R	E	I	N	t_Σ	t_C	N	t_Σ	t_C
ArrayMax	see Sec. 2	33	0	0	0	1	1	2	1	26	4	0	0	0	0
ArraySum	recursive definition	34	1	0	0	1	1	1	2	26	75	2	1	0	0
BinarySearch	complex precondition	49	1	0	0	2	2	3	2	26	2	0	0	0	0
BubbleSort	complex postcond. and inv.	74	1	0	0	2	1	4	5	6	80	21	2	0	0
DutchFlag	user-defined types [6]	96	3	0	0	2	2	8	6	7	132	43	1	0	0
Fibonacci	recursive procedure	40	1	3	1	0	2	0	0	11	88	0	0	0	0
Invert	complex pre- and postcond.	37	0	0	0	3	3	2	1	9	19	118	2	0	2
LinkedListTraversal	heap model	49	3	2	0	0	1	1	1	8	50	54	2	0	0
ListInsert	see [14]	52	1	0	0	2	1	1	0	6	78	5	1	0	0
QuickSort	helper and recursive proc.	89	3	0	0	2	1	6	0	3	4	1	2	∞	0
QuickSort PI	partial implementation	79	3	0	0	2	2	9	0	4	∞	64	2	1	0
TuringFactorial	unstructured control flow	37	1	2	5	0	1	1	0	11	1	0	3	0	0
Split	linear arithmetic [13]	22	0	0	0	0	1	3	0	–	0	0			
SendMoreMoney	fixed-size array constr. [12]	36	1	0	0	15	0	0	0	–	2	2			
Primes	recursive definition [12]	31	2	0	0	0	0	2	0	8	7	4			
NQueens	variable-size array constraints	37	2	1	0	3	0	0	0	11	45	1			

variant of QuickSort whose partitioning procedure has a complete pre- and postcondition but no implementation. This may represent an intermediate development step where we want to validate the overall logic of QuickSort before proceeding with implementing the partitioning procedure. Boogaloo simulates array partitioning based only on its specification—something unachievable with traditional testing techniques. The injected bugs are mostly off-by-one errors and missing preconditions, both of which frequently occur in practice; the bugs in BinarySearch are among those found in textbooks [22].

Declarative Programming. The other four examples come from previous work on constraint programming and code synthesis [13,12], and involve linear arithmetic, recursively defined functions, and quantification over variable-sized arrays. Constraints are declared using `assume` statements or procedures without implementation; Boogaloo generates program outputs satisfying the constraints.

Experimental Results. All problems in Tab. 1 but two include a parameter N that defines the input size (the input array or list for most problems). Column TIME displays the value of N used in the experiments; and the time required to generate a passing execution with different concretization strategies: t_Σ corresponds to fully symbolic executions where the state is concretized only once after terminating; t_C, instead, corresponds to executions where the state is concretized before every jump statement. Column BUGGY displays the same time measures for the buggy programs; for these, the value of N corresponds to the input size exposing the bug found by Boogaloo (which is the smallest possible for all programs). In all experiments we imposed a timeout of 180 seconds, to reflect the expectation to use Boogaloo with good responsiveness.

Concretizing before jumping generally leads to faster executions, even with an order-of-magnitude difference. This strategy may lead to heavy backtracking when the constraints on a given logical variable are imposed incrementally, with one or more concretization points in between, producing potentially lengthy combinatorial enumerations.

In most examples this does not happen—constraints are "local"—and hence concretizing does not degrade performance. The performance difference between fully symbolic and concretized executions for the same example is particularly conspicuous in the current prototype implementation, which uses Z3 through a pure API and hence cannot make use of incremental constraint solving (which would be useful to avoid solving the same constraints multiple times during a single symbolic execution). We speculate that incremental solving would greatly reduce the performance difference between the two concretization strategies. Even though the current implementation has a big potential for improving performance, the experimental results are encouraging: in particular, exposing bugs—the primary purpose of Boogaloo—is fast, even in the presence of partial implementations. Understanding the unexpected behavior with QuickSort, where fully symbolic execution times out, requires further investigation.

6 Related Work

Debugging Failed Verification Attempts. While still an incipient research area, a few techniques have recently been proposed to help understand and debug failed attempts of program verifiers. Sec. 2 already mentioned the Boogie Verification Debugger (BVD, [14]); the Spec# debugger [21] implements similar functionalities which construct concrete counterexamples from failed Boogie runs. Two-step verification [27] compares verification with different semantics (based on unrolling and inlining) to attribute verification failures to either inconsistent or incomplete specifications.

The fact that all these approaches are built around the output provided by a program verifier determines their main limitations compared to Boogaloo. As we demonstrated in Sec. 2, when verification fails because of insufficient specification, the counterexamples generated by BVD or similar tools are typically uninformative or even misleading, because they ignore the implementation even when it is correct (e.g., a loop), unless it is comes with an accurate specification (e.g., a loop invariant). Boogaloo supports a more incremental approach, where users can concentrate on fixing major bugs first. Sec. 2 also discussed how inlining and unrolling (available in Boogie and automatically used in two-step verification) ameliorate these problems, but they are also not directly comparable to Boogaloo, since they scale poorly and require to know explicit unrolling bounds. Of course, the finitary semantics implemented by Boogaloo comes with its own shortcomings: if the shortest counterexamples are very long, it may be infeasible to generate them by enumeration, whereas a static verifier's modular reasoning is insensitive to the length of concrete execution paths since it is entirely symbolic; tools such as BVD can directly work on any failed verifier attempts.

Another approach to produce readable counterexamples is restricting the input language (e.g., [24]), trading off expressiveness for decidability. Bounded model-checking techniques (e.g., [3]) also target standard programming languages and the verification of properties that do not include features such as infinite mappings and unbounded quantification. Boogaloo follows a different course: it supports the entire Boogie language as used in practice, which does not restrict expressiveness a priori, but may produce spurious counterexamples.

Testing is the process of executing programs to make them fail. Since it is based on execution, it is typically limited to violations of simple properties that can be efficiently evaluated at runtime and are implicit in the programming language semantics (e.g., null dereferencing). Languages such as Eiffel [23], JML [15], and Jahob [28] incorporate a richer language for annotations that is still executable, so as to extend the applicability of standard testing techniques. Another line of research in testing is the combination with static techniques, with the goal of complementing each other's strengths to search the input state space more efficiently. In [25], we combined testing with program proving at a high level. A different array of techniques combines testing with symbolic execution; see the recent survey [2]. Boogaloo is also based on symbolic execution, but with a different overall goal; as future work, we will leverage other techniques from symbolic execution to improve the enumeration of executions.

Constraint Programming supports program definitions based on declarative constraints, describing properties of the solution, rather than on traditional imperative constructs. Logic programming extends functional programming languages [1]; more recent approaches combine declarative constraints with imperative languages [20,12]. All these approaches restrict the expressiveness of the constraint language to have predictable performance and some guarantees about soundness, completeness, or both. As briefly demonstrated in Sec. 5, Boogaloo can also be used as a Boogie-based constraint programming language. Unless we also restrict the language of assertions, we cannot offer strong guarantees about properties of the executions generated by Boogaloo (see the end of Sec. 3.2 for a discussion). However, the usage as a constraint programming language brings much flexibility to Boogaloo as a testing environment for Boogie programs, since users can achieve different trade-offs between modularity and scalability opting for the implementation or the specification semantics.

7 Conclusions and Future Work

We presented a technique and a prototype implementation to execute programs with complex specifications and nondeterministic constructs, written in the Boogie intermediate verification language.

Among the various directions for future work, let us mention: integrating domain-specific decision procedure to reduce spurious counterexamples; improving the performance by solving constraints incrementally and by pruning infeasible branches; more experiments with automatically generated Boogie translations; and a user study to assess the practical usability alongside Boogie.

References

1. Antoy, S., Hanus, M.: Functional logic programming. Commun. ACM 53(4), 74–85 (2010)
2. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013)
3. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
4. 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)

5. 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)
6. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: Analysis, classification, and examples (2012), `http://arxiv.org/abs/1211.4470`
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
8. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (August 2008)
9. Kiselyov, O., Shan, C.-C., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers (functional pearl). In: ICFP, pp. 192–203. ACM (2005)
10. Klebanov, V., et al.: The 1st verified software competition: Experience report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP, pp. 207–220. ACM (2009)
12. Köksal, A., Kuncak, V., Suter, P.: Constraints as control. In: POPL, pp. 151–164 (2012)
13. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI, pp. 316–329. ACM (2010)
14. Le Goues, C., Leino, K.R.M., Moskal, M.: The boogie verification debugger (Tool paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 407–414. Springer, Heidelberg (2011)
15. 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)
16. Leinenbach, D., Santen, T.: Verifying the microsoft hyper-V hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
17. Leino, K.R.M.: This is Boogie 2 (2008), `http://goo.gl/QsH6g`
18. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
19. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), `http://fm.csl.sri.com/UV10/`
20. Milicevic, A., Rayside, D., Yessenov, K., Jackson, D.: Unifying execution of imperative and declarative code. In: ICSE, pp. 511–520. ACM (2011)
21. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)
22. Pattis, R.E.: Textbook errors in binary searching. In: SIGCSE, pp. 190–194. ACM (1988)
23. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: ICSE, pp. 257–266. ACM (2013)
24. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011)
25. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Usable verification of object-oriented programs by combining static and dynamic techniques. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 382–398. Springer, Heidelberg (2011)
26. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Verifying Eiffel programs with Boogie. In: BOOGIE Workshop (2011), `http://arxiv.org/abs/1106.4700`
27. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Program checking with less hassle. In: VSTTE (to appear, 2013)
28. Zee, K., Kuncak, V., Taylor, M., Rinard, M.C.: Runtime checking for program verification. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 202–213. Springer, Heidelberg (2007)

Optimizing Nop-shadows Typestate Analysis by Filtering Interferential Configurations

Chengsong Wang[1], Zhenbang Chen[1,2,⋆], and Xiaoguang Mao[1]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] National Laboratory for Parallel and Distributed Processing, Changsha, China
jameschen186@gmail.com, {zbchen,xgmao}@nudt.edu.cn

Abstract. Nop-shadows Analysis (NSA) is an efficient static typestate analysis, which can be used to eliminate unnecessary monitoring instrumentations for runtime monitors. In this paper, we propose two optimizations to improve the precision of NSA. Both of the optimizations filter interferential configurations when determining whether a monitoring instrumentation is necessary. We have implemented our optimization methods in Clara and conducted extensive experiments on the DaCapo benchmark. The experimental results indicate that the optimized NSA can further remove unnecessary instrumentations after the original NSA in more than half of the cases, without a significant overhead. In addition, for two cases, all the instrumentations are removed, which implies the program is proved to satisfy the typestate property.

Keywords: Typestate Analysis, Runtime Monitoring, Static Analysis, Nop-shadows Analysis.

1 Introduction

A typestate property [23] describes the acceptable operations on a single object or a group of inter-related objects, according to the current state (i.e., the typestate) of the object or the group [7,10]. For example, usually, programmers cannot call the method *write* until the method *open* is called on a same File object. Lots of large-scale software system errors are caused by the violations of typestate properties. What is worse, it is very difficult and time-consuming to find out and fix these errors [6,22]. The static analysis of a program with respect to a typestate property is generally undecidable. The existing static typestate checking tools [3,19] suffer from the scalability and the false-alarm problems. Dynamic typestate checking methods complement the static methods with runtime monitoring to improve the scalability and the accuracy of the analysis, but sacrifice the completeness.

Usually, dynamic typestate analysis approaches, such as runtime verification [5,11,15,16], automatically convert typestate properties into runtime monitors that can detect the property violations at runtime. Implementing runtime monitors needs to instrument the monitored programs. The instrumentation can

[⋆] Corresponding author.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 269–284, 2013.

be done manually or automatically based on existing techniques, such as AOP [14]. However, the programs instrumented with runtime monitors usually contain many redundant instrumentations, which result in a significant monitoring overhead. Therefore, some approaches [6,12] exploit static analysis information to remove provable unnecessary instrumentations for reducing the overhead of runtime monitoring. These methods are often called hybrid typestate analysis.

Theoretically, hybrid typestate analysis is equivalent to the static analysis of typestate properties. If all the instrumentations of a runtime monitor can be removed, the program is proved to satisfy the typestate property. Nop-shadows analysis (NSA) [6,7] is one of the existing hybrid analysis methods. NSA is implemented in Clara [1] to optimize the runtime monitoring of large-scale Java programs. NSA uses intra-procedural flow-sensitive and partially context-sensitive data-flow analysis to identify the redundant instrumentations generated for monitors.

Although NSA is effective [6,7], there are some cases in which unnecessary instrumentations still remain after NSA. One of the main reasons is that NSA is only an intra-procedural flow-sensitive static analysis. The overly conservative approximations of inter-procedural cases in NSA reduce the accuracy of the analysis. In this paper, we propose two optimizations to improve the precision of NSA. Both of the optimizations can filter interferential configurations when determining whether a monitoring instrumentation is necessary. An interferential configuration refers to the configuration that lowers the precision of identifying "nop shadows" in NSA. One optimization identifies changeless configurations produced by the backward data-flow analysis of NSA; the other one utilizes local object information to refine the iterations of data-flow analysis. Using the two optimizations, more unnecessary instrumentations can be removed.

To evaluate our optimizations, we have integrated our optimizations into Clara, and applied them to the DaCapo benchmark suite [4]. In more than half of the cases, the optimized NSA can further remove unnecessary instrumentations after the original NSA. In two cases, we get a perfect result, *i.e.*, all the monitoring instrumentations are removed, entirely obviating the need for monitoring at runtime.

To summarize, our paper has the following contributions:

- Propose two optimizations for NSA to improve the precision of the analysis. Both of the optimizations filter interferential configurations by identifying changeless configurations and exploiting local object information.
- Propose and implement an approximate, but sound, intra-procedural flow-sensitive algorithm to determine whether a variable points to a local object.
- Implement the two optimizations and integrate them into Clara.
- Conduct extensive experiments on the DaCapo benchmark suite to show the effectiveness of our optimizations.

The remainder of this paper is organized as follows. We begin with an overview of NSA in Section 2. In Section 3, we give two motivating examples to illustrate the two different optimization methods, respectively. Section 4 formulates the details of our proposed optimizations. Our experiments, described in Section 5, justify that our optimizations are effective in the majority of cases. Section 6 describes the related work and the paper is concluded in Section 7.

2 Nop-shadows Analysis

As in the literature [6,7,17], we also use the term "shadow" to represent an instrumentation point created for runtime monitoring. NSA is a static typestate analysis method proposed and implemented in the Clara framework [6], which extends tracematch [2] with static analysis to remove "nop shadows". Here a "nop shadow" means that the shadow does not influence the results of runtime monitoring, *i.e.*, it can neither trigger nor suppress a property violation [6,7].

Clara consists of three static analysis stages, in which NSA is the most expensive and precise one. Given a typestate property (usually a finite-state machine (FSM)) and an instrumented Java program, NSA uses an intra-procedural data-flow analysis to check whether a shadow in a method of the program can be removed. The basic idea of NSA is to compute the reachable states of each statement in a program according to the semantics of the program and the monitored typestate property. Given an FSM typestate property M and its state set S, for each statement *st*, there are two types of reachable states: $source(st)$ and $futures(st)$, which are calculated respectively by a *forward* data-flow analysis (forward analysis) and a *backward* data-flow analysis (backward analysis). The source set $source(st) \subseteq S$ contains all the states that can be reached before executing *st* from the beginning of the program; $futures(st) \subseteq \mathbb{P}(S)$ is the *future* set, and each element of $futures(st)$ contains the states from which the remainder program execution after *st* can reach a final state (usually the error state) of M. Therefore, for a given shadow s, which is usually a method call statement in the program, NSA identifies s as a "nop shadow" if the execution of the shadow has no impact on the monitoring result, which can be formalized by following two conditions:

- $target(s) \cap F = \emptyset$, where $target(s) = \{q_2 \mid \exists q_1 \in sources(s) \bullet q_2 = \delta(q_1, s)\}$ is the resulting state set after executing s, $\delta(q_1, s)$ is the resulting state after executing s from the state q_1 according to the FSM property M, and F is the final state set of M. This condition means the execution of s does not directly lead to an error state.
- $\forall q_1 \in source(s), \forall Q \in futures(s) \bullet q_1 \in Q \Leftrightarrow \delta(q_1, s) \in Q$. It means the execution of s does not influence whether or not a final state will be reached.

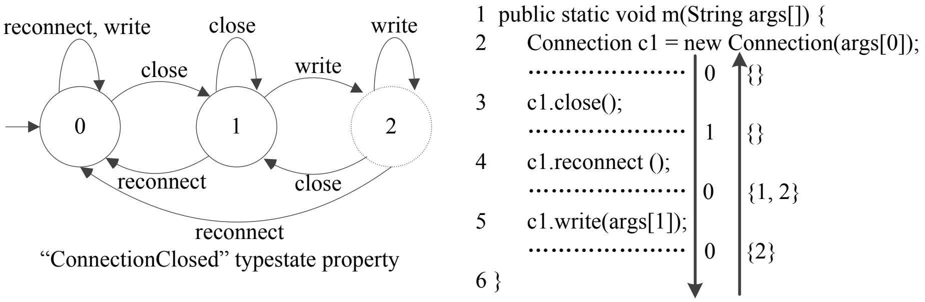
The shadow s can be removed if both conditions are valid.

Figure 1 gives an example for NSA. The left part is an FSM for "ConnectionClosed" [7] typestate property, which requires the "write" operation should not be called after a connection is closed. The right part displays a program annotated with the state information of each statement. The elements in the *source* set and the $futures$ set of each statement are next to the downward and upward arrows, respectively. For instance, for the shadow s_3 at line 3, we have:

$$source(s_3) = \{0\}$$

$$target(s_3) = \{1\}$$
$$futures(s_3) = \emptyset$$

$futures(s_3) = \emptyset$ means that there is no state from which the property state machine can reach the final state via the execution after line 3. According to the preceding two conditions, s_3 is a "nop shadow" that can be removed.



```
1 public static void m(String args[]) {
2     Connection c1 = new Connection(args[0]);
      ....................  0   {}
3     c1.close();
      ....................  1   {}
4     c1.reconnect ();
      ....................  0   {1, 2}
5     c1.write(args[1]);
      ....................  0   {2}
6 }
```

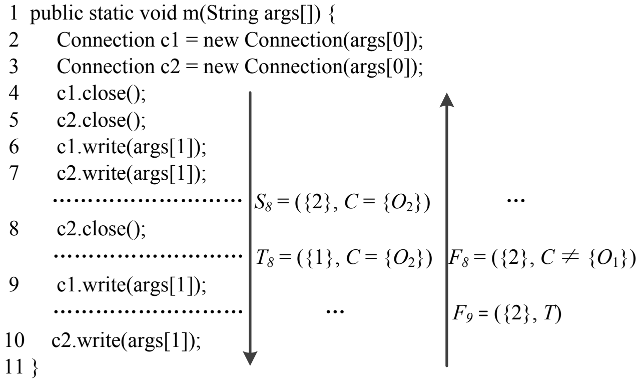
Fig. 1. An example for Nop-shadows Analysis

After removing a "nop shadow", the $source(st)$ and $futures(st)$ of each statement will be calculated again, until no "nop shadow" exists. If there is no shadow after NSA, the program is proved to satisfy the typestate property. For example, all the shadows of the program in Figure 1 will be removed finally. For the inter-procedural cases, the method calls are soundly approximated by using the transitive closure of the shadows in the called methods.

3 Motivating Examples

We motivate our optimizations of NSA through two examples. We also use the "ConnectionClosed" property in Figure 1 as the typestate property. Figure 2 shows an example that invokes the "close" and "write" methods of the class Connection. The shadows at line 7 and 10 violate the typestate property, because they can both drive the state machine into the final state. The "close" operation at line 8 is between these two violating shadows. Hence, from the semantics of the program and the property FSM (*c.f.* Figure 1), the runtime monitor does not need to monitor the shadow at line 8. Whereas, the original NSA cannot identify the shadow at line 8 as a "nop shadow" at compile-time. The reason is explained as follows.

For the sake of brevity, Figure 2 only shows partial critical state information calculated by the forward and backward analysis. In order to distinguish the typestates of multiple different objects or groups of related objects, the dataflow analysis of NSA propagates "configurations" instead of only state sets [7]. A configuration specifies the state information of some specific objects. A configuration $C = (Q, b)$ is composed by a state set Q and a variable binding b.



```
1 public static void m(String args[]) {
2     Connection c1 = new Connection(args[0]);
3     Connection c2 = new Connection(args[0]);
4     c1.close();
5     c2.close();
6     c1.write(args[1]);
7     c2.write(args[1]);
      .........................
8     c2.close();
      .........................
9     c1.write(args[1]);
      .........................
10    c2.write(args[1]);
11 }
```

$S_8 = (\{2\}, C = \{O_2\})$
$T_8 = (\{1\}, C = \{O_2\})$
...
...
$F_8 = (\{2\}, C \neq \{O_1\})$
$F_9 = (\{2\}, T)$

Fig. 2. The example for motivating the first optimization

The variable binding specifies the static objects [9] which represent the concrete runtime objects. Actually, for a shadow s, it also has a variable binding [8] specifying the objects whose typestates can be changed by s. Two variable bindings are *compatible* if they can be bound to a same static object or a same group of related static objects. A configuration and a shadow are *compatible* if their variable bindings are compatible. For a statement st associated with a configuration (Q, b), in forward analysis, the elements in set Q represent all the possible states which the static objects specified by the variable binding b can reach just before st; in backward analysis, they are the states from which the static objects specified by b can reach a final state via the execution after st. For example, the configuration S_8 in Figure 2 represents that the static object O_2 can reach state 2 before executing line 8. The configuration F_{12} in Figure 3 represents that the static object O can reach the final state from state 0 or 1 via the execution after line 3.

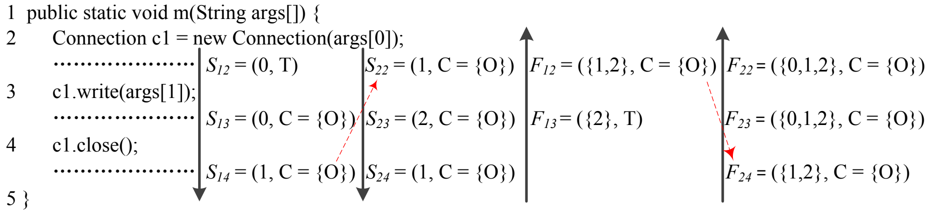
Providing that the program creates the compile-time static objects O_1 and O_2 at line 2 and line 3, respectively. The variable binding of the shadow at line 8 is $C = \{O_2\}$, and the shadow at line 8 changes the configuration from $S_8 = (\{2\}, C = \{O_2\})$ to $T_8 = (\{1\}, C = \{O_2\})$ in the forward analysis, with respect to the typestate property. $F_8 = (\{2\}, C \neq \{O_1\})$ associated to the shadow at line 8 is one of the resulting configurations produced by the backward analysis starting at line 9, which means the typestate of the object does not change if the object is not O_1. The variable bindings of S_8 and F_8 are both compatible to that of the shadow at line 8. According to the "nop shadow" conditions, because the states of the state transition caused by the shadow at line 8, *i.e.*, state 2 in S_8 and state 1 in T_8, are not both contained in the state set $\{2\}$ of F_8, NSA fails to identify this shadow as a "nop shadow".

Actually, the configuration F_8 is induced by the "final shadow"[1] at line 9, in which the variable c_1 is totally unrelated to the variable c_2 at line 8, *i.e.*,

[1] "final shadow", which can drive the FSM of the property into a final state [6].

they must not alias. Hence, in principle, we should filter this type of interferential configurations generated from backward analysis when checking whether a shadow can be removed. Based on this insight, our optimized NSA can successfully identify the shadows, similar to the shadow at line 8, as "nop shadows".

Figure 3 shows another example to motivate the second optimization approach. Different from the former one, the typestate property "Connection-Closed" is not violated by the method m. Therefore, all the shadows in method m can be safely removed. However, by using the original NSA, all the shadows will remain.



```
1 public static void m(String args[]) {
2     Connection c1 = new Connection(args[0]);
      ....................
3     c1.write(args[1]);
      ....................
4     c1.close();
      ....................
5 }
```

Fig. 3. The example for motivating the second optimization

The problem is mainly resulted by the approximated inter-procedure analysis. Figure 3 shows partial forward and backward analysis results that are next to the two downward arrows and two upward arrows, respectively. Because there may be several consecutive method calls to a method in a program, for ensuring the soundness, the forward analysis needs to propagate the configuration at the end of a method to the entry of the method until a fixed-point is reached. For example, S_{14} is propagated to the entry configuration S_{22} of the next iteration (indicated by the red dotted line). The propagation also happens in backward analysis. After reaching the fixed-point, the shadow at line 3 can produce the configuration S_{23}, which contains an error state. Thus, this shadow cannot be removed. In addition, the shadow at line 4 changes the configuration S_{13} to S_{14}, but state 0 in S_{13} and state 1 in S_{14} are not both contained in the state set $\{1, 2\}$ of the configuration F_{24}. Therefore, the shadow at line 4 cannot be removed either.

After carefully analyzing the example program, we find the reason is that the configuration propagation disregards the *local object* information. In this paper, we call a static object, which is created by a "*new*" statement within the method currently being analyzed, a *local object*. For example, the static object O created by the statement at line 2 is a local object. Obviously, at runtime, each local object will be assigned with a different runtime object each time when the method is invoked and the "*new*" statement is executed. Therefore, for the example in Figure 3, the configuration S_{22} should not have a same variable binding as S_{14}. If we have the local object information of the example program, *i.e.*, no need to do the second forward iteration and the second backward iteration, then both of the "nop shadows" at line 3 and line 4 can be removed. Based on the observations motivated by this example, we optimize NSA by exploiting local object information.

For simplicity, the motivating examples do not contain complex programming language features, such as recursion, exception handling and aliasing. In Section 4, we will give the details of our optimization methods that can be applied in general.

4 Optimization Methods

This section presents the details of our optimization methods for filtering interferential configurations when checking whether a shadow is a “nop shadow”. The first subsection explains how to identify changeless configurations generated from backward analysis. The second subsection proposes an algorithm for determining whether a static object is a local object, and describes how to propagate configurations along the inter-procedural control-flow of a analyzed method. The two optimizations are complementary to each other. They separately address different issues that can potentially lead NSA to lose precision. Therefore, these two optimizations can be combined together to further improve the accuracy of NSA.

4.1 Identifying Changeless Configurations

How can we identify changeless configurations, like F_8 in Figure 2, from the results produced by backward analysis? Basically, if the states of a configuration have never been through a state transition during backward analysis, then we consider the configuration as a *changeless* configuration. For a changeless configuration $C_i = (Q_i, b_i)$ that is induced by a “final shadow” s_f and associated to a shadow s_i, even if C_i is compatible with s_i, there is no need to consider C_i when checking whether the shadow s_i is a “nop shadow”. The reason is: the states in set Q_i of the objects specified by the variable bindings b_i will definitely not change anymore before program execution passes the “final shadow” s_f, and the execution of the “final shadow” s_f would not trigger an error because of the incompatibility of s_f and C_i.

Hence, we extend the original configuration tuple from (Q, b) to (Q, b, T), where Q is the state set, b is the variable bindings and T indicates whether the states of this configuration have ever been through a state transition before. Therefore, we need to record the information of T during the configuration transitions in backward analysis. The new configuration transition algorithm is displayed in Algorithm 1.

The algorithm is basically the same as that in [6]. The different parts are enclosed in boxes. Line 4 computes the state set of the successor configuration. If the shadow s can drive the state set Q_c to Q_t (*c.f.* line 4), and the shadow is compatible to the configuration (determined by $\beta^+ \neq \perp$ at line 7), the value of T in the successor configurations is assigned with *true*, indicated by Lines 7-9; otherwise, the value of T remains the same during the configuration transition

Algorithm 1. $transition((Q_c, b_c, T_c), s, \delta)$

```
1: $cs := \emptyset$; // initialize result set
2: $l := label(s), \beta_s := shadowBinding(s)$; //extract label and bindings from $s$
3: //compute target states
4: $Q_t := \delta(Q_c, l)$;
5: //compute configurations for objects moving to $Q_t$;
6: $\beta^+ := \underline{and}(b_c, \beta_s)$;
7: if $\beta^+ \neq \perp$ then
8:     $\boxed{cs := cs \cup (Q_t, \beta^+, true);}$
9: end if
10: //compute configurations for objects staying in $Q_c$;
11: $B^- := \bigcup_{v \in dom(\beta_s)} andNot(b_c, \beta_s, v) \setminus \{\perp\}$;
12: $\boxed{cs := cs \cup \{(Q_c, \beta^-, T_c) \mid \beta^- \in B^-\};}$
13: return $cs$;
```

(Lines 11-12). Moreover, for the backward analysis, when we create an initial configuration, the value of T in the initial configuration is set to *false*.

Based on the extended configuration definition and the transition algorithm, we can identify changeless configurations produced by backward analysis. For a given shadow s and a configuration (Q, b, T) in $futures(s)$, if T is *false*, the configuration will be considered to be interferential, and should be filtered when checking whether the shadow s is a "nop shadow".

4.2 Exploiting Local Object Information

First, we present how to determine whether a static object is a *local object*. We have the following two observations: first, for a given static object inside in a method, if it is created by a "*new*" statement within the method, the object must be a *local object* to this method; second, for any two *strong must-alias* [9] static objects O_1 and O_2 inside a method, these two objects always refer to a same heap object, which implies that they always point to a same local object or a same non-local object. Based on these two insights, Algorithm 2 is designed to identify local objects.

For a given method m and a static object O, the algorithm returns *true* if O is a local object in m; otherwise returns *false*. Algorithm 2 first declares a set *newObjects*, and then adds all the local objects created by the "*new*" statements in m to *newObjects* (Lines 1-7). Then, the algorithm checks whether there exists an element in *newObjects* that is *strong must-alias* to O (Lines 8-12). Currently, the must-alias analysis is only intra-procedural flow-sensitive, and makes a conservative assumption that any two static objects coming from different methods may alias. Therefore, Algorithm 2 is an approximated, but sound, evaluation. In order to gain more efficiency, we extract Lines 2-7 from Algorithm 2 and compute the *newObjects* set before the optimized NSA.

Algorithm 2. isLocalObject(m, O)

```
Set⟨staticObject⟩ $newObjects$;
for all $stmt \in m$ do
    if $stmt$ is a $new$ statement then
        create a new static object $O_i$;
        $newObjects := newObjects \cup \{O_i\}$;
    end if
end for
for all $O_j \in newObjects$ do
    if $O_j$ must-alias $O$ then
        return $true$;
    end if
end for
return $false$;
```

Besides identifying *local objects*, for a configuration, we also need to know whether it is gotten by statically modeling the multiple consecutive invocations of the analyzed method in forward or backward analysis. Same as the first optimization, we also extend the original configuration tuple to a triple (Q, b, R), where R is *true* if the current configuration is indeed gotten by statically modeling the multiple consecutive invocations of the analyzed method.

For a given method m, the intra-procedural control-flows cannot lead to the multiple consecutive invocations of m. Hence, the shadows in methods cannot change the value of R. Figure 4 visualizes four types of possible inter-procedural control-flows (*solid* arrows) of m [7]. Solid arrows (1) and (2) are used to model the transitively recursive method calls to m. We cannot determine that a method call must be transitively recursive at compile-time. Hence, both of the arrows (3a) and (3b) are used to model the non-recursive method calls within m. Additionally, method m can re-executes again after its returning. Arrows (4) is used to model this case. Obviously, there are only three types of inter-procedural control flows (*solid* arrows (1), (2) and (4)) in Figure 4, which can lead to the multiple consecutive invocations of the method m. Therefore, in forward analysis, for all the configurations that reach the entry statements of m or the recursive call sites within m along these inter-procedural control flows, we should assign *true* to R in these configurations. Based on the same argument, in backward analysis, the value of R in configurations, which reach the exit statements of m or the recursive call sites within m along these reverse inter-procedural control flows, should be assigned *true*. Furthermore, for each initial configuration, the value of R is set to *false* in both forward and backward analysis.

Based on Algorithm 2 and the extended configuration, we can filter interferential configurations as follows: for a given shadow s, which is usually a method call statement, inside a method, if there exists a variable v in the variable bindings of s pointing to a *local object*, a configuration (Q, b, R) in $source(s)$ or $futures(s)$ can be safely eliminated if R is *true*. The reason is: even if the shadow s and the configuration have a same static variable binding with respect to the variable

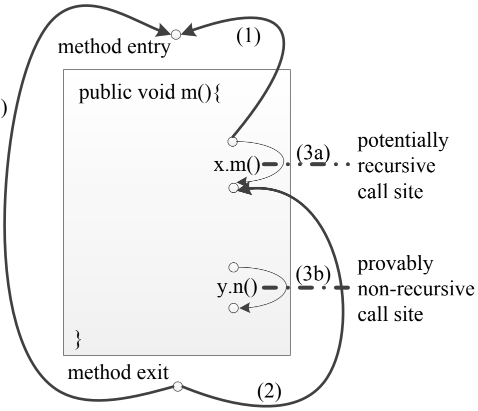


Fig. 4. Possible inter-procedural control-flows of a method [7]

v, v will definitely point to a different object during each method invocation at runtime, which means that the shadow s and the configuration are actually not compatible at runtime. After eliminating all the interferential configurations from $sources(s)$ and $futures(s)$, we use the remaining configurations to determine whether the shadow s is a "nop shadow" according to the conditions in Section 2.

5 Experiments and Discussion

We have implemented our optimizations on the Clara framework [6] and conducted experiments on the DaCapo benchmark suite [4]. NSA cannot support multi-threaded programs. Hence, we ignore the multi-threaded programs *hsqldb*, *lusearch* and *xalan* in the benchmark. Our experiments are based on the experiments of the original NSA in [7]. We are only interested in 23 property/program combinations for each of which the original NSA cannot remove all the shadows. These 23 combinations involve 8 typestate properties and 7 programs. Table 1 lists the typestate properties used in the experiments.

In order to make our experimental results more convincible, we also limit the maximum number of configurations to be 15000, which is the same as that of the original NSA in [7]. Once the number of configurations computed by our optimized analysis is above the threshold, it will abort the analysis of the current method and process the next one.

We evaluate our optimizations as follows: for each optimization, we carry out original NSA first, then use the optimized NSA to further identify "nop shadows". This way of evaluation is *different* from using an optimized NSA directly. Actually, according to our experimental results, under the same configuration limit, using an optimized NSA after original NSA will have better results than that of using the optimized NSA directly. The reason is that using each optimized analysis directly generates more configurations than the original NSA.

Table 1. Typestates properties used in our experiments

Property Name	Description
FailSafeEnum	do not update a vector while iterating over it
FailSafeEnumHT	do not update a hash table while iterating over its elements or keys
FailSafeIter	do not update a collection while iterating over it
FailSafeIterMap	do not update a map while iterating over its keys or values
HasNextElem	always call hasMoreElements before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
Reader	do not use a Reader after its InputStream is closed
Writer	do not use a Writer after its OutputStream is closed

We conducted all the experiments on a Server with 256GB memory and four 2.13GHz XEON CPUs.

5.1 Experiment Results

To justify the effectiveness of our optimizations, we use original NSA as the baseline for our experiments. Table 2 shows the results of our optimizations, and the cases on which optimizations have no effect are not listed. The forth column (**Opt1**) shows the number of remained shadows after using the first optimization, *i.e.*, identifying changeless configurations generated from backward analysis. For 4 out of 23 combinations (17.4%), our optimized analysis can further identify removable shadows after the original NSA. In one case (**FailSafeIterMap + bloat**), the shadows removed by the optimized analysis are twice more than the shadows removed by the original NSA.

The fifth column (**Opt2**) of Tables 2 shows the number of the shadows that remain after using the optimization based on local object information. For 10 out of these 23 combinations (43.5%), the optimized NSA can further remove shadows after the original NSA. In two cases (**FailSafeEnum + fop** and **FailSafeIter + luindex**), the optimization can remove all the shadows that remain after the original NSA. Hence, the optimized NSA by local object information can give the static guarantee that the program satisfies the typestate property in each of these two cases. Furthermore, for 5 out of these 10 cases (50%), the optimized analysis can further remove more irrelevant shadows than the original one. Especially, in two cases (**FailSafeEnum + fop** and **FailSafeEnumHT + jython**) out of these 5, the original NSA cannot remove any shadow.

The last column of Tables 2 shows the results of the combination of two optimizations, *i.e.*, optimizing by local object information first and then by removing changeless configurations. For 13 out of these 23 combinations (56.5%), the combined optimized analysis can further identify "nop shadows" after the original NSA. In one case (**FailSafeIter + bloat**), the two optimizations both have positive effects and identify different "nop shadows" respectively. Interestingly, compared to perform these two optimizations after the original NSA

Table 2. Results of the optimized NSA

Property + Program	BN	AN	Opt1	Opt2	Both
FailSafeEnum + fop	5	5	5	0	0
FailSafeEnum + jython	47	44	44	36	36
FailSafeEnumHT + jython	76	76	76	72	72
FailSafeIter + bloat	1010	916	911	905	899
FailSafeIter + chart	158	150	150	120	120
FailSafeIter + jython	119	115	115	105	105
FailSafeIter + luindex	30	15	15	0	0
FailSafeIter + pmd	305	290	290	262	262
FailSafeIterMap + bloat	481	479	476	479	476
FailSafeIterMap + jython	153	133	133	119	119
FailSafeIterMap + pmd	372	262	262	260	260
Writer + antlr	44	35	34	35	34
Writer + bloat	19	11	9	11	9

BN: The number of shadows that remain before the original NSA. **AN**: The number of shadows that remain after the original NSA. **Opt1**: The number of shadows that remain after the first optimization. **Opt2**: The number of shadows that remain after the second optimization. **Both**: The number of shadows that remain after the combination of two optimizations.

individually, the combination can identify one more “nop shadow” in this case. The reason is: after the original NSA, the second optimization firstly removes some shadows from the instrumented program, so the first optimization generates less configurations and can identify one more “nop shadow” under the same configuration limit. In addition, there are three cases where local object optimization cannot remove any “nop shadow” but the other one can, which also justifies that the two optimizations complement each other.

5.2 Analysis Time

The analysis time of NSA is mainly dominated by the prior supporting analyses, such as constructing call graphs and computing points-to information. Because we evaluate each optimization by using NSA first and then the optimized NSA, the analysis time for evaluating each optimization is definitely longer than that of the original NSA. Table 3 displays the results of the analysis time of the cases on which our optimizations have effects.

From the experimental results, it can be justified that the time for NSA is just a small part of the total compilation time. In all cases but two, the total compilation time including our optimizations is under 10 minutes. The average analysis time of the original NSA is under 1 minute, though in some cases it needs a few more minutes, such as **FailSafeIter + bloat**. In some cases, the analysis time for the optimized analysis is less than that of the original NSA. One of the key reasons is that the optimized NSA analyzes less shadows. For

Table 3. The results of analysis time (*in seconds*)

Property + Program	**The 1st optimization**			**The 2nd optimization**		
	NSA	**Opt**	**Total**	**NSA**	**Opt**	**Total**
FailSafeEnum + fop	0.98	0.1	251.14	1.21	0.39	276.87
FailSafeEnum + jython	8.06	0.2	243.55	8.23	1.87	269.35
FailSafeEnumHT + jython	10.30	0.69	240.91	10.41	8.19	271.05
FailSafeIter + bloat	298.05	133.94	806.02	288.50	516.82	1214.12
FailSafeIter + chart	25.35	1.64	305.92	24.15	66.08	393.04
FailSafeIter + jython	16.9	0.74	270.23	17.69	14.87	303.80
FailSafeIter + luindex	2.21	0.07	99.1	2.53	0.47	110.64
FailSafeIter + pmd	46.01	2.51	352.67	46.97	112.01	490.54
FailSafeIterMap + bloat	58.78	30.17	433.4	65.92	85.19	521.37
FailSafeIterMap + jython	49.67	17.61	276.74	58.38	56	337.73
FailSafeIterMap + pmd	77.67	4.25	420.05	77.72	96.84	541.66
Writer + antlr	12.95	0.83	223.76	13.7	9.06	253.48
Writer + bloat	1.56	0.24	128.66	1.68	0.34	140.08

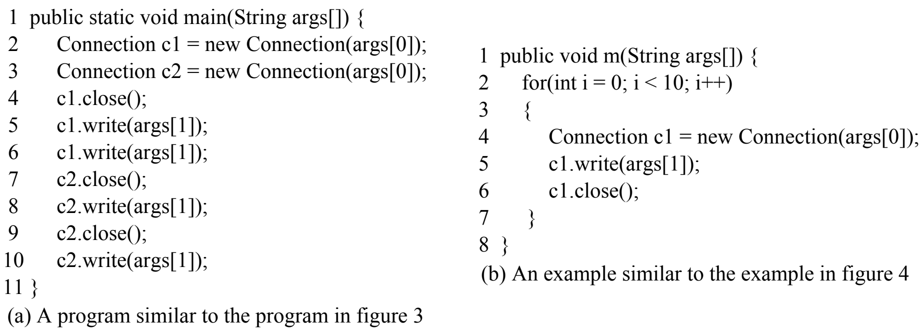
NSA: The analysis time that original NSA consumes. **Opt**: The analysis time of the optimized NSA after the original NSA. **Total**: the total compilation time of the case.

the optimized NSA with the combination of two optimizations, the analysis time introduced by optimization is under 2 minutes in the majority of cases. Overall, our optimization methods do not cause a significant overhead on the weaving process in experiments. Considering the total compilation time, the overhead incurred by our optimizations is acceptable.

5.3 Discussions

According to the experimental results, the first optimization only has effects in 17.4% cases, which is not very impressive. The reason is that the optimization works well on the methods containing several *interleaved* relevant method invocations on different objects. For example, for the program in Figure 5(a), which is slightly different from that in Figure 2 (the method calls on c_1 and c_2 are not interleaved), the original NSA can identify the shadow at line 9 as a "nop shadow". Hence, the capability of the optimized NSA is the same as that of the original one in this case.

The optimization based on local objects also has limitations. For example, it has no effect on the local objects created within loop statements. In Figure 5(b), we show a method m that extends the method in Figure 3 by adding a loop. Obviously, the method satisfies the typestate property in Figure 1, but the optimized NSA by using local object information cannot remove the shadows at lines 5 and 6. The reason is the forward analysis will propagate the configurations at the end of a "*for*" statement to the entry of the "*for*" statement, and the backward analysis will propagate configurations in the inverse direction too.



```
public static void main(String args[]) {
    Connection c1 = new Connection(args[0]);
    Connection c2 = new Connection(args[0]);
    c1.close();
    c1.write(args[1]);
    c1.write(args[1]);
    c2.close();
    c2.write(args[1]);
    c2.close();
    c2.write(args[1]);
}
```

(a) A program similar to the program in figure 3

```
public void m(String args[]) {
    for(int i = 0; i < 10; i++)
    {
        Connection c1 = new Connection(args[0]);
        c1.write(args[1]);
        c1.close();
    }
}
```

(b) An example similar to the example in figure 4

Fig. 5. Examples on which optimizations have no effect

Therefore, we can further optimize NSA based on the local objects created in loop statements, which will be the future work.

Finally, we should note that even if the original NSA is inter-procedural flow-sensitive, it could not remove the shadows in Figure 2 and Figure 3 either. Hence, the main ideas of our optimizations can also be used in the inter-procedural flow-sensitive static analysis.

6 Related Work

Recently, typestate analysis of large-scale programs attracts much attention, and several static and dynamic typestate analysis methods are proposed and implemented. In [13], Fink *et al.* propose a context-sensitive, flow-sensitive and integrated static typestate verifier. The verifier utilizes a combined abstract domain of typestate and pointer abstractions to improve the precision of alias analysis. Their static analysis framework is designed to be a staged system to improve the scalability and efficiency. However, their approach cannot verify the typestate specifications of multiple objects. In [18], a hybrid typestate analysis is proposed and implemented to be context-sensitive and inter-procedural flow-sensitive. The static analysis in [18] is based on a lattice-based operational semantics, which supports to track individual objects along control-flow paths and compute typestate information and points-to information simultaneously. However, their approach suffers from unsoundness problem [7]. Besides those work, Rahul Purandare presents in [20] a cost model for runtime monitoring. The model explains key factors of monitoring overhead and the relationship among them. The cost model guides the optimization of runtime monitoring. Different from the hybrid method in this paper, the approach in [20] also tries to remove instrumentations at runtime [21]. Furthermore, their optimization can reduce the runtime overhead by reclaiming unnecessary monitors. Whereas, their hybrid approach may easily lead to unacceptable overhead at runtime, especially for the typestate properties involving multiple interacting objects. In addition, when unchecked exceptions happen, the method may produce unsound results.

7 Conclusion

In this paper, we present two optimization approaches for NSA to improve the precision. One optimization identifies changeless configurations during the backward analysis; the other one use local object information to refine the forward analysis and backward analysis of NSA. According to the experiments on the DaCapo benchmark suite, in more than half of the studied cases, the optimized NSA can further remove unnecessary instrumentations, without a significant overhead. Additionally, we dissect the experimental results and the situations in which our optimizations have no effect. Furthermore, the main ideas of our optimizations can also be used in inter-procedural flow-sensitive static analysis.

Acknowledgement. This research is supported in part by grants from the National 973 project 2011CB302603, the National NSFC projects (Nos. 91118007 and 61103013), the National 863 projects (Nos. 2011AA010106 and 2012AA011201), the Specialized Research Fund for the Doctoral Program of Higher Education 20114307120015, and the Program for New Century Excellent Talents in University.

References

1. Clara, `http://www.bodden.de/clara/`
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. ACM SIGPLAN Notices 40(10), 345–364 (2005)
3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA), pp. 301–320. ACM, New York (2007)
4. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. In: 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 169–190. ACM, New York (2006)
5. Bodden, E.: J-lo-a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university (2005)
6. Bodden, E.: Verifying Finite-state Properties of Large-scale Programs. Ph.D. thesis, McGill University (2009)
7. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: 32nd International Conference on Software Engineerin (ICSE), pp. 5–14. IEEE, ACM, New York (2010)
8. Bodden, E., Lam, P., Hendren, L.: Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 36–47. ACM, New York (2008)

9. Bodden, E., Lam, P., Hendren, L.: Object representatives: A uniform abstraction for pointer information. In: International Conference on Visions of Computer Science: BCS International Academic Conference (VoCS), pp. 391–405. British Computer Society, Swinton (2008)
10. Bodden, E., Lam, P., Hendren, L.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 183–197. Springer, Heidelberg (2010)
11. Chen, F., Roşu, G.: Mop: An efficient and generic runtime verification framework. In: 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA), pp. 569–588. ACM, New York (2007)
12. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 124–133. ACM, New York (2007)
13. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: 15th International Symposium on Software Testing and Analysis (ISSTA), pp. 133–144. ACM, New York (2006)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. Springer (1997)
15. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with m2aspects. In: International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM), pp. 51–58. ACM, New York (2006)
16. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling lscs into aspectj. In: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE), pp. 219–230. ACM, New York (2006)
17. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 46–60. Springer, Heidelberg (2003)
18. Naeem, N.A., Lhotak, O.: Typestate-like analysis of multiple interacting objects. In: 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA), pp. 347–366. ACM, New York (2008)
19. Pradel, M., Jaspan, C., Aldrich, J., Gross, T.R.: Statically checking api protocol conformance with mined multi-object specifications. In: 34th International Conference on Software Engineering (ICSE). IEEE Press, Piscataway (2012)
20. Purandare, R.: Exploiting Program and Property Structure for Efficient Runtime Monitoring. Ph.D. thesis, University of Nebraska-Lincoln (2011)
21. Purandare, R., Dwyer, M.B., Elbaum, S.: Monitor optimization via stutter-equivalent loop transformation. In: 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 270–285. ACM, New York (2010)
22. Slaughter, S.A., Harter, D.E., Krishnan, M.S.: Evaluating the cost of software quality. Communications of the ACM 41(8), 67–73 (1998)
23. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering 12(1), 157–171 (1986)

A Causality Analysis Framework for Component-Based Real-Time Systems*

Shaohui Wang[1], Anaheed Ayoub[1], BaekGyu Kim[1], Gregor Gössler[2], Oleg Sokolsky[1], and Insup Lee[1]

[1] Department of Computer and Information Science
University of Pennsylvania
{shaohui,anaheed,baekgyu}@seas.upenn.edu, {sokolsky,lee}@cis.upenn.edu
[2] INRIA Grenoble – Rhône-Alpes, France
gregor.goessler@inria.fr

Abstract. We propose an approach to enhance the fault diagnosis in black-box component-based systems, in which only events on component interfaces are observable, and assume that causal dependencies between component interface events within components are not known. For such systems, we describe a causality analysis framework that helps us establish the causal relationship between component failures and system failures, given an observed system execution trace. The analysis is based on a formalization of counterfactual reasoning, and applicable to real-time systems. We illustrate the analysis with a case study from the medical device domain.

1 Introduction

Component-based design in systems engineering enables independent development of system components as well as their incremental construction and modification. The complexity of systems that are built with component-based design renders it difficult to determine the culprit components of the system that are responsible for the discovered system failure on a given system execution. We in this paper aim to present a formal framework for the analysis of the causal relation between the faulty components and an observed system failure on a given system execution.

While this problem is common to all safety-critical domains, our immediate motivation comes from the domain of medical devices. In the United States, the Food and Drug Administration (FDA) is responsible for assessing safety of medical devices and regulating their use in health care. When a system failure that harms a patient, known as an *adverse event* occurs, the hospital is required to report it to the FDA-maintained database [9]. Diagnosis of the root cause is crucial for the subsequent recovery and follow-up prevention measures. Such diagnosis requires recording of system executions leading to the failure, as well as methods for the efficient analysis of the recorded system trace.

* Research is supported in part by the National Science Foundation grants CNS-0930647 and CNS-1035715, and NSF/FDA SiR grant CNS-1042829.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 285–303, 2013.

Existing work in fault diagnosis (e.g., [6,21,8,5,23,17] to name only a few) aims to study (1) the discovery of existence of faults in the system, and (2) the identification of the types and locations of the faults. A main assumption implicitly used in the work of fault diagnosis is that, the computed fault propagation chain is the actual cause-effect chain [17].

We in this work consider systems whose components are black-boxes, where only events on component interfaces are observable, and assume that causal dependencies between component interface events within components are not known. The presence of uncertainty in computing fault propagation chain inside components leads to an over-approximation of the fault propagation chain. We have shown in our preliminary study [26] that, the precision of this over-approximation can be improved by *causality analysis*, i.e., reasoning about whether a fault inside a component is the cause for system failure.

Causality is commonly defined by the use of counterfactual reasoning [13,16,19]. Some recent work in the engineering domain has discussed several versions of causality definitions for finite state automata [11] and temporal logics [4,14,15]. In this work, we extend our previous result in [26] to consider the case of real-time systems where a system execution trace is a sequence of timestamped events, and the system/component specifications are based on the timing of events.

Contributions. We present a framework for the causality analysis for component-based systems. We identify the steps of the analysis and the input and output for each step. We show with a case study from the medical device domain how to use the proposed framework to establish the causal relationship between component failures and the system failure. In particular, we extend our approach presented in [26] to handle the causality analysis for real-time systems.

Paper Organization. We first use a simple example as an illustration to define the causality analysis problem in Section 2. We then present a proposed causality analysis framework for component-based systems in Section 3. In Section 4, we present the main technique used for causality analysis. We show how to apply the causality analysis to our case study in Section 5. We discuss some of the assumptions of our approach in Section 6 and related work in Section 7, and conclude in Section 8.

2 Motivating Example and Problem Statement

2.1 The Generic Patient-Controlled Analgesia Pump Case Study

The Generic Patient Controlled Analgesic (GPCA) infusion pump project [10] aims at developing a reference software model for PCA infusion pump systems with which formal techniques can be performed to ensure the GPCA safety requirements [12]. We focus on the core safety requirements in this case study to demonstrate our causality analysis framework:

> *A bolus dose shall be given when requested by the patient, and when the drug reservoir is empty and an infusion session is in progress, an alarm shall be issued and the pump motor should be stopped.*

An *infusion session* is defined as the interval from the start of the pump motor till its stop.

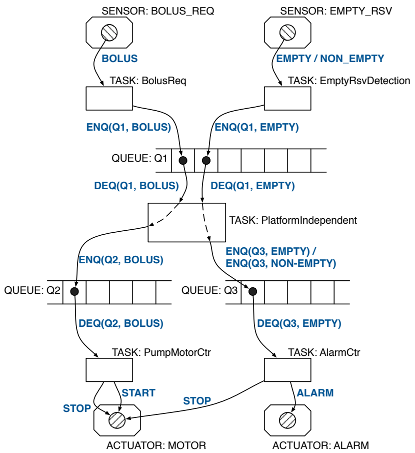


Fig. 1. Data flow of the GPCA

We implemented software that captures the requirements on an Atmel SAM7X-EK development board [2], running the FreeRTOS real-time operating system [3]. The development board is interfaced to the sensors and actuators of the Baxter PCA infusion pump hardware. Sensor signals from the bolus request button and the empty reservoir detection switch are captured through periodic sampling. Instructions for the pump motor to start and stop are delivered via pulse width modulation. Alarms are signaled with flashing LEDs in our experiments.

The FreeRTOS in our case study runs a priority-based, preemptive scheduler. Five tasks, each implemented with an independent C function, communicate with each other by sending and receiving messages in three message queues, Q1, Q2, and Q3. Some tasks have individual local variables, which we deem as unknown to the analysis due to our black-box assumption, but they do not share global variables in our implementation. In this case study, we view each task as a component; the terms *task* and *component* are used interchangeably.

The five tasks are summarized in Table 1. The Priority column indicates the task priorities in addition to the system idle task's priority of FreeRTOS. The left (right, resp.) arrow means that the corresponding task reads (sends, resp.) messages from (to, resp.) the queue. An *event* in the system is a single action performed by a task. We attach each event with a timestamp, denoted as a pair (e, t), where e is the event and $t \in \mathbb{R}_{\geq 0}$ is the timestamp. For example, $(\textsc{alarm}, 9760)$ represents an event where the AC task has raised an alarm, at time 9760 ms since the system starts execution. The events we recorded for this case study is summarized in Table 2. We assume it is known which task produces which event. In particular, we assume we know whether an instance of the STOP event is produced by AC or PM.

Table 1. Tasks in FreeRTOS Implementation for GPCA

Task Name	Abbreviation	Period	Priority	Queues Accessed
PumpMotorCtr	PM	300 ms	+6	$\leftarrow Q2$
PlatformIndependent	PI	Aperiodic	+5	$\leftarrow Q1, \rightarrow Q2, Q3$
BolusReq	BR	500 ms	+4	$\rightarrow Q1$
AlarmCtr	AC	500 ms	+3	$\leftarrow Q3$
EmptyRsvDetection	ER	1000 ms	+2	$\leftarrow Q3$

Table 2. Events Recorded in GPCA Controller

Event	Task	Description
START	*PM*	The pump motor has been started.
STOP	*PM* or *AC*	The pump motor has been stopped.
ALARM	*AC*	The alarm has been fired.
ENQ(Q, M)	*BR*, *ER*, or *PI*	MessageM has been put to queue Q.
DEQ(Q, M)	*PI*, *PM*, or *AC*	MessageM has been retrieved from queue Q.

The data flow of the events in the system is depicted in Figure 1. The BolusReq (*BR*) and EmptyRsvDetection (*ER*) tasks periodically check if there are patient bolus request or empty/non-empty reservoir signals from sensors, respectively; if there are, they put the messages to Q1. We do not consider faults in these two tasks in our analysis. The aperiodic PlatformIndependent (*PI*) task is triggered whenever there is a message sent to Q1. It moves the bolus request and empty/non-empty reservoir messages to Q2 and Q3, respectively. The PumpMotorCtr (*PM*) task periodically checks if there are bolus request messages in Q2; if there are, it will start the infusion session by keeping the pump motor running for 30 periods (i.e., 9 seconds for each patient bolus request). The AlarmCtr (*AC*) task periodically checks if there are empty reservoir messages in Q3; if there are, it will raise an alarm and stop the pump motor. Each task has a response time of 10 ms after a message is received. We assume here that the queues in the system are reliable, i.e., no messages are lost/duplicated/altered in a queue.

The task behaviors described above reflect our black-box assumption: the two data flow paths shown in Figure 1 both pass through queue Q1 and the task *PI*, yet we do not know whether there is fault propagation from the EMPTY_RSV sensor to the *PM* task, due to the assumption that *PI* is a black-box to the analysis. (The dashed links inside *PI* in Figure 1 indicate unknown data flows.) Essentially, this is what we intend to infer from the causality analysis.

With the recorded events, we express the GPCA safety requirement as the following Metric Interval Temporal Logic (MITL) [1] property:

$$\begin{aligned}\varphi_S := \Box_{(0,\infty)}&[\mathrm{ENQ}(\mathrm{Q1},\ \mathrm{BOLUS}) \rightarrow \Diamond_{(0,650)}[\mathrm{START} \wedge \\ &[\Box_{(0,9000)}\neg\mathrm{ENQ}(\mathrm{Q1},\ \mathrm{EMPTY}) \wedge \Diamond_{(8990,9010)}\mathrm{STOP}] \vee \\ &[\Box_{(0,9000)}[\mathrm{ENQ}(\mathrm{Q1},\mathrm{EMPTY}) \rightarrow [\Diamond_{(0,1050)}\mathrm{ALARM} \wedge \Diamond_{(0,1050)}\mathrm{STOP}]]]]].\end{aligned} \tag{1}$$

The values in the formula are obtained from the system implementation. For example, the value 650, indicating the maximal allowed delay (in milliseconds) from the instance when a BOLUS is put to Q1 to the instance when the START message is delivered by *PM*, is due to that (a) the aperiodic task *PI* has a worst case delay of 10 ms to retrieve the message BOLUS from Q1, plus a possible 10 ms delay due to preemption by *PM*; (b) similarly a 20 ms worst case delay for *PI* to move the BOLUS message to Q2; (c) since *PM* has a period of 300 ms, in the worst case, it takes up to two periods of *PM* to read the message once it is enqueued in Q2 (see Figure 3 in Subsection 5.2 for details), and (d) the worst case delay of the PM task is 10 ms. The rest of the time periods are analogously specified. It is required that the behaviors of the tasks constitute a subset of the behaviors specified by the system constraint φ_S, which is formally stated later in Hypothesis 1 in Subsection 2.2.

A system execution is captured by collecting the events with their timestamps by instrumenting the GPCA implementation. We assume in this paper that recording is perfect, i.e., no events in the system are missing on a trace, and each event on a trace actually happened at its recorded timestamp.

A trace is a set of timestamped events. For example, $\{(\textsc{enq}(\text{Q1}, \textsc{bolus}), 8500), (\textsc{deq}(\text{Q1}, \textsc{bolus}), 8502), (\textsc{enq}(\text{Q3}, \textsc{empty}), 8503), (\textsc{deq}(\text{Q3}, \textsc{empty}), 8701), (\textsc{alarm}, 9760), (\textsc{stop}, 9760)\}$ is a trace with six events observed. The events are naturally ordered by their corresponding timestamps. On this trace, PI mistakenly put the bolus request message to the wrong queue with the wrong message. AC reads the empty reservoir message but fails to ALARM and STOP within its deadline. The system property φ_S is violated since there is no bolus dose delivered to the patient after the bolus request event $\textsc{enq}(\text{Q1}, \textsc{bolus})$ (i.e., Equation (1)). So two faulty tasks, PM and AC, may have caused the system property violation.

In the causality analysis problem, we would like to investigate which subset of the faulty tasks, $\{PI\}$, $\{AC\}$, or $\{PI, AC\}$, caused the system property violation. We leave the details of the analysis to Section 5 but only show the result here: both $\{PI\}$ and $\{PI, AC\}$ satisfy the counterfactual test for causality, so we report the minimal subset $\{PI\}$ as the cause for the system property violation.

2.2 The Causality Analysis Problem Definition

In this subsection, we abstract the problem illustrated by the example in Subsection 2.1 and provide the formal definition of the *causality analysis problem.*

Definition 1 (Trace)**.** *A* trace *of length n is a set of n timestamped events, denoted $\{(e_1, t_1), \ldots, (e_n, t_n)\}$, such that $t_1 \leq \cdots \leq t_n$.*

A trace only contains a finite number of events. For time beyond t_n, no events happen in the system. We use logical formula to express component/system behaviors. It is assumed that given a trace Tr, the semantics of the chosen logic is two-valued: for any formula ϕ, either $Tr \models \phi$ or $Tr \not\models \phi$. In this paper, MITL and first order logic (FOL) are used for component/system specifications.

Definition 2 (Constraint)**.** *Given a set E of events, a* constraint *is a logical formula defined on E. In details, for MITL, E is the set of atomic propositions; for an event $e \in E$, $(Tr, t) \models e$ if and only if $(e, t) \in Tr$. For FOL, E is the set of logical constants.*

Definition 3 (Component)**.** *A* component *$C = \langle I_C, O_C, \varphi_C \rangle$ is a tuple where the I_C and O_C are its set of input and output, respectively, such that $I_C \cap O_C = \emptyset$, and φ_C is a constraint defined on $I_C \cup O_C$.*

The notion of the component input/output is general. In the GPCA case study, the input and output for each component are the events it could receive and send through the queues, respectively; in [26], the input and output are values passing through component data ports.

Definition 4 (System Definition). *A* system definition $S = \langle C_1, \ldots, C_J \rangle$ *consists of a set of components.*

The set of all events in the system is defined by $E_S = \bigcup_{j=1}^{J} I_{C_j} \cup O_{C_j}$, where J is the number of components in the system.

Definition 5 (System Property). *A* system property φ_S *for system definition* S *is a constraint defined on the set* E_S *of system events.*

Hypothesis 1. *There must be at least one component violation for a system property violation, or equivalently,* $\bigwedge_{j=1}^{J} \varphi_{C_j} \rightarrow \varphi_S$.

Hypothesis 1 is the basis for the causality analysis. A violation to Hypothesis 1 implies a flawed system design, which is out of the scope of this paper.

Definition 6 (Violation). *We say that a property* φ *is violated on trace* Tr *if and only if* $Tr \not\models \varphi$. *A system property violation is called a system failure. A component property violation is called a component failure; in such cases, the component is called faulty.*

Definition 7 (Faulty Components). *Given an observed trace* Tr *and a system definition* S *on which a system property* φ_S *is violated, we define*

$$\mathcal{F} = \{C \mid C \text{ is a component in } S \text{ and } Tr \not\models \varphi_C\} \tag{2}$$

to be the set of faulty components for the violation of φ_S *on* Tr.

Consider a *suspected subset* $\mathcal{C} \subseteq \mathcal{F}$ of faulty components. Replacing every component in $\mathcal{C}$ with a correct one would result in an alternative system S'. Let

$$\begin{aligned} TR_{\mathcal{C}} = \{tr \mid\ & tr \text{ is a trace for } S', \text{ and} \\ & tr \text{ has the same system input as observed on } Tr\} \end{aligned} \tag{3}$$

be the set of possible system traces for S' when rerunning the system S' with the same system input as observed on Tr. The formal characterization of $TR_{\mathcal{C}}$ is a case-by-case analysis, for which we show with the GPCA case study in Section 5. Based on $TR_{\mathcal{C}}$, several notions of causes can be defined.

Definition 8 (Contributory Cause [22]). *A (non-empty) suspected subset* $\mathcal{C} \subseteq \mathcal{F}$ *of faulty components is a* contributory cause *for the violation of a system property* φ_S *on an observed trace* Tr *if and only if* $\exists tr \in TR_{\mathcal{C}}.tr \models \varphi_S$.

Definition 9 (Main Contributory Cause/Necessary Cause [26,11]). *A (non-empty) suspected subset* $\mathcal{C} \subseteq \mathcal{F}$ *of faulty components is a* main contributory cause *for the violation of a system property* φ_S *on an observed trace* Tr *if and only if* $\forall tr \in TR_{\mathcal{C}}.tr \models \varphi_S$.

Definitions 8 and 9 bound the two extremes of defining necessary cause. Definition 8 requires there exists at least one alternative system execution trace on which the system failure disappears while Definition 9 requires so on all alternative system execution traces. In this work, we do not fix a causality definition, but take it as a parameter of the causality analysis problem.

Definition 10 (Causality Analysis Problem Definition). *Given a system definition* S, *a system property* φ_S, *and a trace* Tr *such that* $Tr \not\models \varphi_S$, *let* $\mathcal{F}$ *be* *as defined in Equation* (2)*. The* causality analysis problem *with respect to a causality definition CD, is to identify the set*

$$Culprit = \{\mathcal{C} \in 2^{\mathcal{F}} \mid \mathcal{C} \textit{ is a cause according to causality definition } CD, \\ \textit{and no proper subset of } \mathcal{C} \textit{ satisfies } CD\}. \tag{4}$$

We call the tuple $\langle S, \varphi_S, Tr, CD\rangle$ an instance of the causality analysis problem. It can be seen from the causality definitions that, the reconstruction of the set $TR_{\mathcal{C}}$ of alternative system execution traces is at the heart of the causality analysis. In [26] we have proposed an approach based on the transformation of a causality analysis problem instance into an unsatisfiability checking problem instance. In this paper we extend the technique to handle real-time systems where a system execution trace is a set of events ordered by their timestamps, and the system/component specifications are based on both the occurrences and timestamps of events. In the following, we first show an overview of the causality analysis framework in Section 3, and detail the techniques for causality analysis in Section 4 with a case study in Section 5.

3 The Causality Analysis Framework

In a bird's-eye view, the causality analysis process is conceptually divided into four steps, as shown in Figure 2. The shaded ovals System Definition S, System Property φ_S, observed Trace Tr with system failure, and Causality Definition CD are the input to the analysis; the output is a set *Culprit* of Minimal Culprits for the violation of φ_S on trace Tr with respect to the causality definition CD. The intermediate artifacts, shown as unshaded ovals, and the four steps of the analysis, shown as solid boxes, are discussed below.

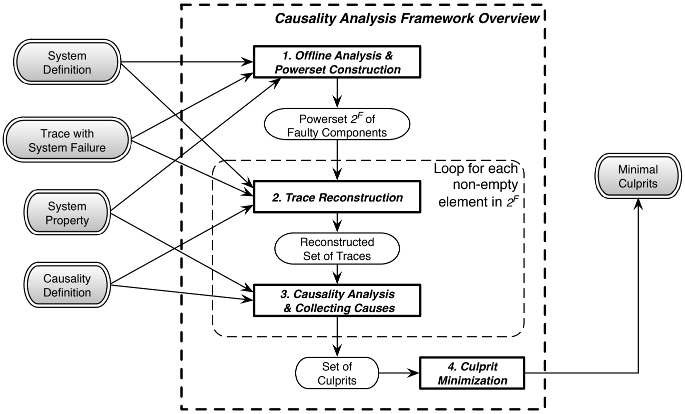

Causality Analysis Framework Overview
System Definition
Trace with System Failure
System Property
Causality Definition
1. Offline Analysis & Powerset Construction
Powerset $2^{\mathcal{F}}$ of Faulty Components
Loop for each non-empty element in $2^{\mathcal{F}}$
2. Trace Reconstruction
Reconstructed Set of Traces
3. Causality Analysis & Collecting Causes
Set of Culprits
4. Culprit Minimization
Minimal Culprits

Fig. 2. The Causality Analysis Framework Overview

Step 1. *Offline Analysis & Powerset Construction.* In this step, a sanity check of whether a system property violation occurs is first performed. If not, then there is no need for the causality analysis. Otherwise, we check Hypothesis 1 defined in Subsection 2.2. When Hypothesis 1 holds, we gather the set $\mathcal{F}$ of faulty components for the violation of φ_S on trace Tr, and construct the powerset $2^{\mathcal{F}}$ of $\mathcal{F}$.

Step 2. *Trace Reconstruction.* The trace reconstruction for the causality analysis is based on the system specification, and parametric to the suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components and the causality definition. This step is at the core of the causality analysis, and we will discuss it in Section 4.

Step 3. *Causality Analysis & Collecting Causes.* For each suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components, the causality analysis checks whether $\mathcal{C}$ is a culprit according to the chosen causality definition CD. If yes, it is collected for the subsequent culprit minimization; otherwise $\mathcal{C}$ is not a cause for the violation of system property φ_S according to CD.

Step 4. *Culprit Minimization.* The last step of causality analysis is to check the minimality of each collected culprit, according to Definition 4. Non-minimal culprits are pruned for precise results of causality analysis.

4 Trace Reconstruction and Causality Analysis

The trace reconstruction step in the causality analysis is to identify the set $TR_{\mathcal{C}}$ of traces when the suspected subset $\mathcal{C}$ of faulty tasks in system S are replaced with correct ones and the system is rerun with the same input as observed on trace Tr. The main idea in obtaining $TR_{\mathcal{C}}$ is to specify the logical constraint ψ that exactly the traces in $TR_{\mathcal{C}}$ satisfy. The constraint ψ is composed based on (1) task constraints for correct tasks, (2) tasks constraints for faulty tasks, (3) constraints on values observed on trace Tr, and (4) *trace reconstruction rules.* With the constructed logical constraint ψ, the problem of checking of causality based on Definition 8 (Definition 9, resp.) can be transformed into the problem of satisfiability (unsatisfiability, resp.) checking, for which state-of-the-art solvers exist [26].

In this section, we show the extension of the work presented in [26] to the case where real-time systems are considered, i.e., traces are sequences of timestamped events as in Definition 1, and system/task specifications are given as logical constraints in either temporal logics or first order logic on events.

Given a causality analysis problem instance $\langle S, \varphi_S, Tr, CD \rangle$, Step 1 of the causality analysis framework is to identify the set $\mathcal{F}$ of faulty tasks according to Definition 2. In Step 2, for each non-empty suspected subset $\mathcal{C} \subseteq \mathcal{F}$, a set $TR_{\mathcal{C}}$ of system traces is reconstructed, given that the faulty tasks in $\mathcal{C}$ are replaced with good ones and the system S is rerun with the same input events. Each task's behavior in the system is determined by the trace reconstruction rules, which indicate what constraint must be put on each task C_j in S, according to whether the task is (1) non-faulty, (2) faulty but not suspected, and (3) faulty and suspected. Informally, the three rules are summarized as follows.

(R1) If $C_j \notin \mathcal{F}$, then it is deemed as a good task. In the trace reconstruction, C_j's behavior is constrained by φ_{C_j}, i.e., a correct task's constraint.

(R2) If $C_j \in \mathcal{F} \setminus \mathcal{C}$, i.e., C_j is faulty but not in the consideration of being suspected, then all output events produced by C_j on trace Tr are preserved on any reconstructed traces.

(R3) If $C_j \in \mathcal{C} \subseteq \mathcal{F}$, then C_j is a faulty task that is replaced by a good one. In this case, the trace reconstruction "removes" the events that should not have occurred on the trace Tr, and "adds" those which must be produced by C_j.

The logical constraint to express that an event e is observed at time t on the trace Tr is expressed as

$$on_{Tr}(e,t) := \exists (e',t') \in Tr.e' = e \wedge t' = t. \tag{5}$$

The constraint that all events task C_j produced on Tr are preserved on reconstructed traces is specified with

$$\kappa_{C_j} := \forall e \in O_{C_j}.\forall t \in \mathbb{R}_{\geq 0}.[on_{Tr}(e,t) \rightarrow \exists (e',t').e' = e \wedge t' = t]. \tag{6}$$

The constraint κ_{C_j} means that, any execution trace that satisfies κ_{C_j} must have an event e' which is the same as the e delivered at time $t' = t$, for any timestamped event (e,t) on Tr.

The task constraint of "removing" events from a trace in the trace reconstruction is done by adding more constraints to rule out traces where the events that have to be removed occur. An event e must be removed in the trace reconstruction if (1) e is produced by a suspected faulty task C_j, and (2) there is no other event on the trace that triggers the event e. The task constraint of "adding" events that a faulty task C_j must have produced is specified by augmenting the task constraint φ_{C_j} to specify the allowed time ranges for output events from C_j. The definitions of the "removing" and "adding" constraints are application dependent. We defer the details to Subsection 5.3, and use ρ_{C_j} and α_{C_j} for now to represent the two constraints for "removing" and "adding", respectively.

The conditions for the rules (R1)–(R3) are defined as

$$\xi_{C_j,1} := \neg in(C_j, \mathcal{F}). \tag{7}$$
$$\xi_{C_j,2} := in(C_j, \mathcal{F}) \wedge \neg in(C_j, \mathcal{C}). \tag{8}$$
$$\xi_{C_j,3} := \neg in(C_j, \mathcal{C}). \tag{9}$$

Here the in is the set membership relation defined as $in(C_j, \mathcal{F}) := \bigvee_{C \in \mathcal{F}} C = C_j$. The task constraint for C_j in the trace reconstruction is then specified as

$$\psi_{C_j} := \begin{cases} \varphi_{C_j}, & \text{if } \xi_{C_j,1}, \\ \kappa_{C_j}, & \text{if } \xi_{C_j,2}, \\ \alpha_{C_j} \wedge \rho_{C_j}, & \text{if } \xi_{C_j,3}. \end{cases} \tag{10}$$

Finally, it is required that exactly the set of observed system input events on Tr occur in reconstructed traces. The set I of possible system input events is application dependent. For example, for the GPCA case study, $I = \{\textsc{enq}(\text{Q1}, \textsc{bolus}), \textsc{enq}(\text{Q1}, \textsc{empty}), \textsc{enq}(\text{Q1}, \textsc{non-empty})\}$. This constraint is defined as

$$\iota := \forall e \in I.\forall t \in \mathbb{R}_{\geq 0}.[on_{Tr}(e,t) \leftrightarrow \exists!(e',t').e' = e \land t' = t]. \tag{11}$$

Here, the ∃! quantifier means "there exists one and only one." The behavior of the reconstructed system is then specified with the formula

$$\psi := \iota \land \psi_{C_1} \land \dots \psi_{C_J}. \tag{12}$$

Proposition 1. *The formula ψ defined in Equation* (12) *defines the set $TR_{\mathcal{C}}$ of the possible system behaviors with the same input as observed on Tr, after suspected tasks in $\mathcal{C}$ are replaced with correct ones. That is, $TR_{\mathcal{C}} = \{tr \mid tr \models \psi\}$.*

The construction in this section is a combination of Steps 2 and 3 in the causality analysis framework (cf. Figure 2) for a given suspected faulty subset $\mathcal{C}$. The formula ψ in Equation (12) characterizes the set of reconstructed traces, whereas the satisfiability (unsatisfiability, resp.) result corresponds to whether the subset $\mathcal{C}$ is a cause with respect to Definition 8 (Definition 9, resp.). Due to Proposition 1, to check that the subset $\mathcal{C}$ is a cause according to Definition 8, it suffices to check that $\psi \land \varphi_S$ is satisfiable. To check that the subset $\mathcal{C}$ is a cause according to Definition 9, it suffices to check that $\psi \land \neg\varphi_S$ is unsatisfiable. State-of-the-art SAT/SMT solvers, e.g., Z3 [7], can be leveraged in solving the causality analysis problem, as shown in our previous work [26].

5 The GPCA Case Study

In this section, we use the GPCA case study to illustrate how the causality analysis problem is solved. We first show a few informal examples, then the formal definitions of the GPCA system, and finally the analysis using the causality analysis framework and trace reconstruction techniques from Sections 3 and 4.

A sample trace we will analyze is shown in Table 3. The ID column is added for the convenience of reference. The Task column indicates which task has produced the corresponding event. The Time column is the timestamp for the corresponding event. On this trace, a bolus request is detected at 8500 ms, and an infusion session starts at 8702 ms. An empty reservoir is detected at 17000 ms, and an alarm is raised at 17008 ms, together with a STOP event from *AC* which ends the infusion session. The STOP event from *PM* does not affect the pump operation in this case.

Table 3. A Sample Trace for GPCA

ID	Task	Event	Time (ms)
1	*BR*	ENQ(Q1, BOLUS)	8500
2	*PI*	DEQ(Q1, BOLUS)	8502
3	*PI*	ENQ(Q2, BOLUS)	8503
4	*PM*	DEQ(Q2, BOLUS)	8701
5	*PM*	START	8702
6	*ER*	ENQ(Q1, EMPTY)	17000
7	*PI*	DEQ(Q1, EMPTY)	17004
8	*PI*	ENQ(Q3, EMPTY)	17005
9	*AC*	DEQ(Q3, EMPTY)	17007
10	*AC*	ALARM	17008
11	*AC*	STOP	17008
12	*PM*	STOP	17701

5.1 Informal Causality Analysis Examples

It can be easily verified that the trace shown in Table 3 satisfies the GPCA safety property in Equation (1). Now we show the causality analyses via a series of examples, based on variants of the trace observed in Table 3.

Example 1 (Faulty tasks, no system failure). Given a trace as observed in Table 3, with Event 12 missing. In this case, PM is faulty by not sending the STOP event. However the system property is not violated since the AC task detects an empty reservoir message and alarms and stops the pump motor. □

According to Step 1 of the causality analysis framework, there is no need for subsequent causality analysis.

Example 2 (Single faulty task caused system failure). Consider a trace where only Event 1 and Event 2 in Table 3 are observed. In this case, the PI task fails to move the bolus request event from Q1 to Q2, read by the PM task. Subsequently the PM task does not perform any actions, since it does not know there is a bolus request. In this case the PI task is faulty while PM is not. □

Example 3 (Multiple faulty tasks jointly caused system failure). Consider the trace in Table 3 with Events 3–5 and Events 10–12 missing. In this case, the PI task is faulty by not moving the bolus request message to Q2. The AC task is faulty by not delivering the ALARM and STOP events. However, replacing neither the PI nor the AC task individually could make the system failure disappear. Both PI and AC must be replaced with good ones for the system failure to disappear. □

Example 4 (Multiple faulty tasks, but only one caused system failure). In the example in Subsection 2.1, a trace with six events {(ENQ(Q1, BOLUS), 8500), (DEQ(Q1, BOLUS), 8502), (ENQ(Q3, EMPTY), 8503), (DEQ(Q3, EMPTY), 8701), (ALARM, 9760), (STOP, 9760)} is shown. In this example, both PI and AC are faulty. However AC's faulty behavior would not have been triggered in the first place if PI were not faulty, and thus should not be considered as a cause for system failure. In the meanwhile, although it is the PM task's job to send the START event, it should not be the cause of system failure in this case either since it is not a faulty task: it does not receive the bolus request message in the first place, due to PI's fault. □

Example 5 (Multiple faulty tasks, but only one caused system property violation). Consider the trace in Table 3 with only Events 1–9 observed. In this case, both the AC and the PM tasks are faulty by not delivering the corresponding events. In this case, if the AC task were not faulty, the system failure would disappear. □

Examples 4 and 5 show the improvement in precision that we have achieved using causality analysis: not all of the identified faulty tasks are the culprits for the system failure. By ruling out the tasks which are not culprits, the subsequent analysis for the system failure can be focused on the identified minimal culprits.

5.2 Formal Definitions for GPCA System

We first define constraints φ_{AC}, φ_{PI}, and φ_{PM} for the three tasks that can fail. We do not consider faults in BR and ER in this paper. The constraint for each task consists of two parts:

(1) what would a task do when it reads a message from a queue, and
(2) when would a task read a message if there is one in the corresponding queue.

For Part (1), the AC task's constraint is specified with

$$\begin{aligned}\tau_{AC} := \forall(e,t).[e = \textsc{deq}(\text{Q3, }\textsc{empty}) \rightarrow & \exists!(e',t').[e' = \textsc{stop} \wedge t' \leq t+10] \wedge \\ & \exists!(e'',t'').[e'' = \textsc{alarm} \wedge t'' \leq t+10]].\end{aligned} \tag{13}$$

It is interpreted as, as long as there is a $\textsc{deq}(\text{Q3, }\textsc{empty})$ event on the trace, there must be a STOP event within 10 ms and an ALARM event within 10 ms. Similarly, the PI task's constraint is specified with

$$\begin{aligned}\tau_{PI} := \forall(e,t).[&[e = \textsc{deq}(\text{Q1,}\textsc{bolus}) \rightarrow \exists!(e',t').[e' = \textsc{enq}(\text{Q2, }\textsc{bolus}) \wedge t' \leq t+10]] \wedge \\ &[e = \textsc{deq}(\text{Q1, }\textsc{empty}) \rightarrow \exists!(e',t').[e' = \textsc{enq}(\text{Q3, }\textsc{empty}) \wedge t' \leq t+10]] \wedge \\ &[e = \textsc{deq}(\text{Q1, }\textsc{non-empty}) \rightarrow \exists!(e',t').[e' = \textsc{enq}(\text{Q3, }\textsc{non-empty}) \wedge t' \leq t+10]]].\end{aligned} \tag{14}$$

The PM task's constraint is specified with

$$\begin{aligned}\tau_{PM} := \forall(e,t).[e = \textsc{deq}(\text{Q2,}\textsc{bolus}) \rightarrow & \exists!(e',t').[e' = \textsc{start} \wedge t' \leq t+10 \wedge \\ & \exists!(e'',t'').[e'' = \textsc{stop} \wedge 8990 \leq t''-t' \leq 9010]]].\end{aligned} \tag{15}$$

For Part (2), the aperiodic task PI and the two periodic tasks PM and AC have different behaviors when a message the task should read is present in the corresponding queue. In the GPCA implementation, PI is set to preempt lower priority tasks to process the message and put it to the corresponding queues. This behavior is specified with the constraint γ_{PI} in Equation (16). The value 20 ms is due to the possible preemption of PI by the even higher priority PM.

$$\begin{aligned}\gamma_{PI} := \forall(e,t).[&[e = \textsc{enq}(\text{Q1,}\textsc{bolus}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q1, }\textsc{bolus}) \wedge t' \leq t+20]] \wedge \\ &[e = \textsc{enq}(\text{Q1, }\textsc{empty}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q1, }\textsc{empty}) \wedge t' \leq t+20]] \wedge \\ &[e = \textsc{enq}(\text{Q1, }\textsc{non-empty}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q1, }\textsc{non-empty}) \wedge t' \leq t+20]]].\end{aligned} \tag{16}$$

For periodic tasks AC and PM, when a message is present in a queue, it may be processed nearly two periods later as illustrated in Figure 3 (for AC). AC is dispatched at the beginning of each period, but can only execute in boxed durations where the FreeRTOS schedules AC by switching it IN and OUT. Suppose there is a check function in the AC's implementation to check if there is an EMPTY message in Q3. It may happen that the check function is executed at the very beginning of one period, slightly before the message EMPTY is put to Q3 (in which case AC does not know there is a message EMPTY during the rest of the period); in the following period, the check function may be executed at the very end of the period.

This scenario is the worst case for a periodic task to detect a message in a queue. Therefore, the constraints for AC and PM are respectively:

$$\begin{aligned}\gamma_{AC} := \forall(e,t).[&[e = \textsc{enq}(\text{Q3, }\textsc{empty}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q3, }\textsc{empty}) \wedge t' < t+1000]] \wedge \\ &[e = \textsc{enq}(\text{Q3, }\textsc{non-empty}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q3, }\textsc{non-empty}) \wedge t' < t+1000]]].\end{aligned} \tag{17}$$

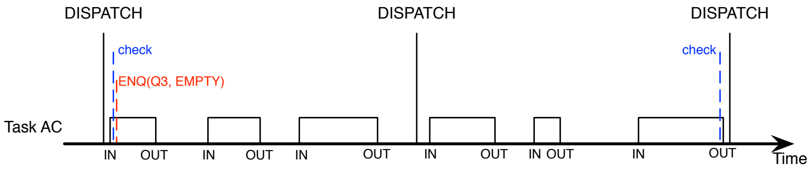


Fig. 3. Illustration of Message Processing for Periodic Tasks in FreeRTOS

$$\gamma_{PM} := \forall(e,t).[e = \textsc{enq}(\text{Q2},\ \textsc{bolus}) \rightarrow \exists!(e',t').[e' = \textsc{deq}(\text{Q2},\ \textsc{bolus}) \wedge t' < t + 600]]. \tag{18}$$

Notice that we have asymmetric treatments on the response time requirements for Part (1) and Part (2) of a task's constraint. For Part (1), a hard response time of 10 ms is imposed; for Part (2), the task scheduling in FreeRTOS is considered. This is due to the view that for Part (1), a task knows that a message has arrived and is thus required to deliver the corresponding event within the imposed response time; Part (2) of the constraint reflects the fact that a task's worst case delay to detect a message in a queue.

With Equations (13)–(18), the complete task constraints are defined as:

$$\varphi_{AC} := \tau_{AC} \wedge \gamma_{AC}, \tag{19}$$

$$\varphi_{PI} := \tau_{PI} \wedge \gamma_{PI}, \tag{20}$$

$$\varphi_{PM} := \tau_{PM} \wedge \gamma_{PM}. \tag{21}$$

The GPCA system is then defined as $S = \langle BR, ER, AC, PI, PM \rangle$. For each task C, its sets I_C and O_C of input/output events, as well as its constraint φ_C, are shown in Table 4.

Table 4. Tasks in GPCA Case Study

Task C	Input I_C	Output O_C	φ_C
BR		$\{\textsc{enq}(\text{Q1},\ \textsc{bolus})\}$	True
ER		$\{\textsc{enq}(\text{Q1},\ \textsc{empty}), \textsc{enq}(\text{Q1},\ \textsc{non-empty})\}$	True
AC	$\{\textsc{deq}(\text{Q3},\ \textsc{empty}), \textsc{deq}(\text{Q3},\ \textsc{non-empty})\}$	$\{\textsc{alarm}, \textsc{stop}\}$	φ_{AC} (19)
PI	$\{\textsc{deq}(\text{Q1},\ *)\}$	$\{\textsc{enq}(\text{Q2},\ \textsc{bolus}),$ $\textsc{enq}(\text{Q3},\ \textsc{empty}), \textsc{enq}(\text{Q3},\ \textsc{non-empty})\}$	φ_{PI} (20)
PM	$\{\textsc{deq}(\text{Q2},\ \textsc{bolus})\}$	$\{\textsc{start}, \textsc{stop}\}$	φ_{PM} (21)

5.3 Formal Causality Analysis

An instance of the causality analysis problem is defined by a tuple $\langle S, \varphi_S, Tr, CD \rangle$. Now we show the application of the causality analysis framework and trace reconstruction technique to solve the causality analysis problem in Example 4.

The system property is defined as in Equation (1). The trace is $Tr = \{(\textsc{enq}(\text{Q1},\ \textsc{bolus}), 8500), (\textsc{deq}(\text{Q1},\ \textsc{bolus}), 8502), (\textsc{enq}(\text{Q3},\ \textsc{empty}), 8503), (\textsc{deq}(\text{Q3},\ \textsc{empty}), 8701), (\textsc{alarm}, 9760), (\textsc{stop}, 9760)\}$. The causality definition CD is the main contributory cause in Definition 9. In Step 1 of the causality analysis frame work, the set $\mathcal{F} = \{PI, AC\}$ of faulty tasks is identified.

In Step 2, for each non-empty subset $\mathcal{C} \subseteq \mathcal{F}$, the formula ψ in Equation (12) is constructed according to the Equations (5) through (12) and information from S, Tr, $\mathcal{C}$ and $\mathcal{F}$. We now discuss the construction for the two application dependent constraints ρ_{C_j} and α_{C_j} not discussed in Section 4.

Defining "Removing" and "Adding" Constraints. As discussed in Section 4, one condition for an event to be removed from a trace is that it is not triggered by any other events. The trigger relation between timestamped input and output for a task is derived from the task's constraint. For example, the constraint for AC specifies that the STOP event and ALARM event must be delivered within 10 ms once the ENQ(Q3, EMPTY) event is read from queue Q3, as defined in Equation (13). In this case, the trigger relation is expressed as

$$\begin{aligned} \mathit{trig}_{AC} := &\{((\textsc{deq}(\text{Q3, }\textsc{empty}), t), (\textsc{stop}, t')) \mid \text{ for all } t' \leq t + 10\} \\ &\cup \{((\textsc{deq}(\text{Q3, }\textsc{empty}), t), (\textsc{alarm}, t')) \mid \text{ for all } t' \leq t + 10\}. \end{aligned} \tag{22}$$

Similarly, the trigger relations for PI and PM are defined as follows.

$$\begin{aligned} \mathit{trig}_{PI} := &\{((\textsc{deq}(\text{Q1, }\textsc{bolus}), t), (\textsc{enq}(\text{Q2, }\textsc{bolus}), t')) \mid \forall t' \leq t + 10\} \\ &\cup \{((\textsc{deq}(\text{Q1, }\textsc{empty}), t), (\textsc{enq}(\text{Q3, }\textsc{empty}), t')) \mid \forall t' \leq t + 10\} \\ &\cup \{((\textsc{deq}(\text{Q1, }\textsc{non-empty}), t), (\textsc{enq}(\text{Q3, }\textsc{non-empty}), t')) \mid \forall t' \leq t + 10\}. \end{aligned} \tag{23}$$

$$\mathit{trig}_{PM} := \{((\textsc{deq}(\text{Q3, }\textsc{empty}), t), (\textsc{start}, t')) \mid \forall t' \leq t + 10\}. \tag{24}$$

The constraint for traces where the events produced by a faulty task C_j are removed is specified with

$$\begin{aligned} \rho_{C_j} := \forall e \in O_{C_j}.\forall t \in \mathbb{R}_{\geq 0}.[[\neg\exists(e', t').((e', t'), (e, t)) \in \mathit{trig}_{C_j}] \rightarrow \\ \neg\exists(e'', t'').e'' = t \wedge t'' = t]. \end{aligned} \tag{25}$$

Informally, this constraint means that if there is no trigger for event e at time t, then it should not occur on any reconstructed traces.

For "adding" events to a trace, a task must only deliver output events when it is activated by the FreeRTOS scheduler. This piece of information is unavailable to offline analysis. We assume that the FreeRTOS scheduler would schedule each task the same as on the observed trace Tr. The instance at which an event can be produced on a reconstructed trace is then limited both by the task response time and its activation time. For example, if AC reads the EMPTY message at time 8701 ms, and it is observed on Tr that AC is active in time ranges [8700 ms, 8703 ms], [8709 ms, 8712 ms], etc., then in addition to the 10 ms deadline to deliver the events ALARM and STOP in the range [8701 ms, 8711 ms], AC can only produce the events during the ranges of [8701 ms, 8703 ms] or [8709 ms, 8711 ms]. The constraint for this requirement is obtained by augmenting the task constraint with the time information. For example, for AC, the specification is

$$\begin{aligned} &\alpha_{AC} := \ \gamma_{AC} \wedge \forall(e, t).[e = \textsc{deq}(\text{Q3, }\textsc{empty}) \rightarrow \exists(e_i^1, t_i^1), (e_o^1, t_o^1), (e_i^2, t_i^2), (e_o^2, t_o^2) \in Tr. \\ &[e_i^1 = e_i^2 = \textsc{in}(AC) \wedge e_o^1 = e_o^2 = \textsc{out}(AC) \wedge t \leq t_o^1 \wedge t \leq t_o^2 \wedge \\ &\neg\exists(e_i', t_i'), (e_o', t_o') \in Tr.[[e_i' = \textsc{in}(AC) \wedge t_i^1 < t_i' \leq t_o^1] \ \vee [e_o' = \textsc{out}(AC) \wedge t_i^1 \leq t_o' < t_o^1]] \wedge \\ &\neg\exists(e_i', t_i'), (e_o', t_o') \in Tr.[[e_i' = \textsc{in}(AC) \wedge t_i^2 < t_i' \leq t_o^2] \ \vee [e_o' = \textsc{out}(AC) \wedge t_i^2 \leq t_o' < t_o^2]]] \rightarrow \\ &\exists!(e^1, t^1).[e^1 = \textsc{stop} \wedge [\max(t, t_i^1) \leq t^1 \leq \min(t + 10, t_o^1)]] \wedge \\ &\exists!(e^2, t^2).[e^1 = \textsc{alarm} \wedge [\max(t, t_i^2) \leq t^2 \leq \min(t + 10, t_o^2)]]]. \end{aligned} \tag{26}$$

The third and fourth lines of Equation (26) constraint the pairs (e_i^1, e_o^1) and (e_i^2, e_o^2) to bound single time chunks of execution for task AC (i.e., a single box in Figure 3). The constraint α_{AC} means, if an event $e = \textsc{deq}(\mathrm{Q3}, \textsc{empty})$ at time t is on the reconstructed trace, then its corresponding events (STOP and ALARM) must be produced by AC within the 10 ms deadline, as well as when AC is active. α_{PM} and α_{PI} can be similarly defined.

Causality Analysis Result. For the constructed constraint ψ for each case, we have manually proved that $\psi \wedge \neg\varphi_S$ is unsatisfiable for the cases when $\mathcal{C} = \{PI\}$ or $\mathcal{C} = \{PI, AC\}$, while it is satisfiable for $\mathcal{C} = \{AC\}$. This result shows that both $\{PI\}$ and $\{PI, AC\}$ are culprits, according to the main contributory cause definition (Definition 9). These two subsets are collected as the set of culprits in Step 3 of the causality analysis framework. In Step 4, the two culprits $\{PI\}$ and $\{PI, AC\}$ are minimized to be $\{PI\}$ only. This result is consistent with our intuition in that it is the PI task's fault in the first place to put a bogus empty reservoir message to Q3, which triggers AC's fault.

6 Discussion

FreeRTOS Scheduling in Trace Reconstruction. When defining the "adding" constraint, we have assumed that the FreeRTOS scheduler would schedule all the tasks the same as on the observed trace during trace reconstruction. This assumption must be made due to the unavailability of FreeRTOS scheduling information should the system be rerun. Without this assumption, the analysis would have to include the FreeRTOS scheduler as part of the system and model it (or even the entire FreeRTOS operating system) as a component too. This is by itself a challenging task and is beyond the scope of this paper.

Causality Analysis vs. Fault Diagnosis. Unlike many approaches to fault diagnosis, we address the case of black-box components [25], in which internal flows of information between component input and output are unknown. In this case, techniques based on computing fault propagation paths lead to an over-approximation of cause-effect chains. The causality analysis we proposed in the paper improves the precision of this over-approximation.

Alternative Ways to Trace Reconstruction. Our causality analysis is based on counterfactual reasoning [16], where the system behavior is reevaluated on the possible alternative traces. A commonly used criterion for constructing alternative traces is to measure the *similarity* between the reconstructed traces and the actual observed one. Causality analysis is only performed on alternative traces which are *similar* to the observed trace. However, the notion of *similarity* is subjective, reflected by the rules used for the trace reconstruction.

In our approach, the trace reconstruction rules (R1)–(R3) represent a view at the *task level*: a faulty task is replaced with a good one, and all its events, except for system inputs, are reconstructed via the "removing" and "adding" operations. In contrast, one could perform trace reconstruction at the finer grained *event level*: the trace under analysis is scanned through until the first occurrence e_f of an event that leads to task failure is found, and trace reconstruction is started only from that particular occurrence; every event that happens before e_f is kept the same as observed.

Compared to the set $TR^t_{\mathcal{C}}$ of reconstructed traces produced by task-level trace reconstruction, the event-level trace reconstruction could produce a smaller set $TR^e_{\mathcal{C}} \subseteq TR^t_{\mathcal{C}}$ of traces *more similar* to the observed trace. Using the finer-grained event-level trace reconstruction, it is comparatively more likely to establish a necessary cause (Definition 9), since less traces have to be examined for the "for all" quantification to be satisfied. On the other hand, it is comparatively less likely to establish a contributory cause (Definition 8), since less possible alternative traces can be used to satisfy the "exists" quantification.

Full Observability. *Full observability* involves two assumptions: (1) we are able to put probes at the interfaces of components so that each event is observable, and (2) the recording facility is capable of capturing all events at component interfaces. The first assumption is by our consideration of black-box components, where internal events within a component is not observable, but the events at its interface are observable. Violations to the second assumption may lead to undetected faulty components, yielding a smaller set $\mathcal{F}$ of faulty components. This may possibly lead to spuriously identified culprits.

Causality Definitions. Several causality definitions have been discussed in previous work [13,24,11,15,4,26], all based on the notion of counterfactual reasoning [16]. We in this work used the main contributory cause (Definition 9), but showed that the causality analysis framework is parametric to the causality definition of choice. The capability of using different causality definitions in the analysis increases the flexibility for the investigator to make reasonable arguments.

The definitions of contributory and main contributory causes express different levels of necessity needed to judge for the cause. If the sufficiency of causality definition is of concern, one could use alternative trace reconstruction rules and causality definitions.

Scalability. While we are working on larger case studies to gain empirical results on the scalability of our approach, we foresee two limitations. First, for a given subset $\mathcal{C}$ of suspected faulty components, the complexity of computing whether $\mathcal{C}$ is a necessary cause is coNP-complete for propositional logic [26] and undecidable in general for first order logic [18]. This limits the scalability of our approach to the capability of state-of-the-art SAT/SMT solvers, such as Z3 [7]. Second, we have shown in the paper the direct computation of the minimal culprit, which requires the explicit generation of the powerset of $\mathcal{F}$, limiting the possible number of faulty components that can be analyzed practically. Further studies on algorithms exploiting the underlying structure of the sets of reconstructed traces could potentially speed up the explicit computation.

7 Related Work

Halpern and Pearl [13] were among the first to introduce the counterfactual reasoning for causes into the engineering domain. Some later development [15,4] is based on the notions in [13]. In this work, we formally characterized the set of reconstructed traces, and showed that causality can be defined based on the set of reconstructed traces. One advantage of our work is the explicit treatment of real-time systems, which is not presented in previous work on causality analysis. Timestamps are considered as variables so that constraints on timestamped events symbolically characterize sets of traces that satisfy the constraints.

The treatment of trace reconstruction is another difference between our work and previous ones [15,11]. In [15], each occurrence of an event on a trace is represented by a boolean variable e, indicating whether the event is present on the trace (e is true) or not (e is false). The underlying component behaviors are not considered in [15]. Similarly, in [11], the trace reconstruction rules place a more rigid requirement than in this paper, which may occasionally lead to undesired analysis result, as we have discussed in [26]. On the other hand, the work in [11] in addition defines *horizontal* causality between one component's failure and another's, which is not discussed in any other work in causality analysis. Also, our Hypothesis 1 is due to [11].

The result of causality analysis naturally provides an *explanation* to the system failure: which components' faulty behaviors are the causes to the system property violation. The work in [4] provides an application in explaining counterexamples from formal verification of system properties specified in linear temporal logics (LTL) [20]. We believe the approach in [4] can be extended to the setting in this paper.

8 Conclusion

We proposed the causality analysis problem for black-box component-based systems. By using causality analysis we are able to establish causal relationship between component failures and system failure. We provided a formal analysis framework to solve the causality analysis problem, and detailed the trace reconstruction rules for the analysis for real-time systems. We illustrated our approach with the GPCA case study. In the future, we are planning to enhance the application of the analysis by providing tool support for safety-critical systems in the medical device domain.

Acknowledgement. We would like to thank FDA researchers Paul L. Jones and Yi Zhang for their motivating discussions on the causality analysis problem and help in explaining infusion pumps and the GPCA safety requirement document [12].

References

1. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM 43(1), 116–146 (1996)
2. Atmel Corporation. AT91SAM7S-EK Evaluation Board User Guide (2007), http://www.atmel.com/Images/doc6112.pdf
3. Barry, R.: FreeRTOS User Manual, http://www.freertos.org
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009)
5. Bhattacharyya, S., Huang, Z., Chandra, V., Kumar, R.: A discrete event systems approach to network fault management: detection and diagnosis of faults. In: American Control Conference, vol. 6, pp. 5108–5113 (2004)
6. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence 32(1), 97–130 (1987)
7. 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)
8. Dubey, A., Karsai, G., Kereskenyi, R., Mahadevan, N.: Towards a real-time component framework for software health management. Technical Report ISIS-09-111, Vanderbilt University (2009)
9. FDA. FDA MAUDE Database, http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfmaude/search.cfm.
10. Generic PCA Infusion Pump Reference Implementation, http://rtg.cis.upenn.edu/medical/gpca/gpca.html
11. Gössler, G., Le Métayer, D., Raclet, J.-B.: Causality analysis in contract violation. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 270–284. Springer, Heidelberg (2010)
12. Safety Requirements for the Generic PCA Pump, http://rtg.cis.upenn.edu/gip-docs/Safety_Requirements_GPCA.doc
13. Halpern, J.Y., Pearl, J.: Causes and Explanations: A Structural-Model Approach. Part I: Causes. The British Journal for the Philosophy of Science 56(4), 843–887 (2005)
14. Kuntz, M., Leitner-Fischer, F., Leue, S.: From probabilistic counterexamples via causality to fault trees. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 71–84. Springer, Heidelberg (2011)
15. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. Technical Report soft-12-02, University of Konstanz (2012)
16. Lewis, D.: Counterfactuals, 2nd edn. Wiley-Blackwell (2001)
17. Mahadevan, N., Abdelwahed, S., Dubey, A., Karsai, G.: Distributed diagnosis of complex systems using timed failure propagation graph models. In: The IEEE Systems Readiness Technology Conference, pp. 1–6 (2010)
18. Mendelson, E.: Introduction to Mathematical Logic, 4th edn. Chapman and Hall/CRC (1997)
19. Pearl, J.: Causality: Models, Reasoning, and Inference. Cambridge University Press (2009)
20. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS 1977, pp. 46–57 (1977)
21. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)

22. Riegelman, R., et al.: Contributory cause: unnecessary and insufficient. Postgrad. Med. 66(2), 177 (1979)
23. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Failure diagnosis using discrete-event models. IEEE Transactions on Control Systems Technology 4(2), 105–124 (1996)
24. Tian, J., Pearl, J.: Probabilities of causation: Bounds and identification. Annals of Mathematics and Artificial Intelligence 28, 287–313 (2000)
25. Tripakis, S.: A combined on-line/off-line framework for black-box fault diagnosis. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 152–167. Springer, Heidelberg (2009)
26. Wang, S., Ayoub, A., Ivanov, R., Sokolsky, O., Lee, I.: Contract-based blame assignment by trace analysis. In: HiCoNS, pp. 117–125 (2013)

Reducing Monitoring Overhead by Integrating Event- and Time-Triggered Techniques

Chun Wah Wallace Wu[1], Deepak Kumar[1],
Borzoo Bonakdarpour[2], and Sebastian Fischmeister[1]

[1] Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
{cwwwu,d6kumar,sfischme}@uwaterloo.ca
[2] School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
borzoo@cs.uwaterloo.ca

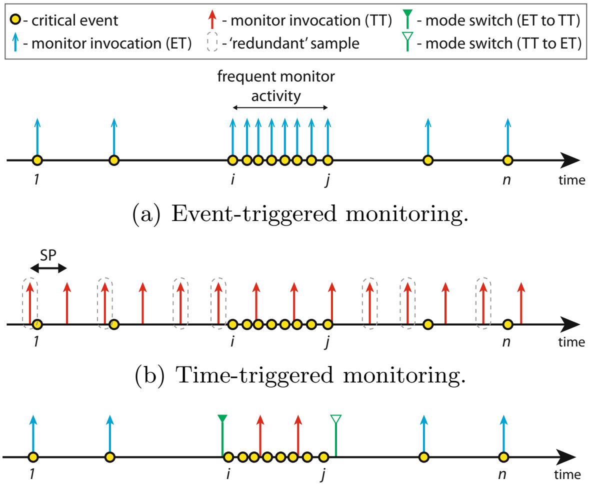
Abstract. *Runtime verification* is a formal technique used to check whether a program under inspection satisfies its specification by using a runtime monitor. Existing monitoring approaches use one of two ways for evaluating a set of logical properties: (1) *event-triggered*, where the program invokes the monitor when the state of the program changes, and (2) *time-triggered*, where the monitor periodically preempts the program and reads its state. Realizing the former is straightforward, but the runtime behaviour of event-triggered monitors are difficult to predict. Time-triggered monitoring (designed for real-time embedded systems), on the other hand, provides predictable monitoring behavior and overhead bounds at run time. Our previous work shows that time-triggered monitoring can potentially reduce the runtime overhead provided that the monitor samples the program state at a low frequency.

In this paper, we propose a *hybrid* method that leverages the benefits of both event- and time-triggered methods to reduce the overall monitoring overhead. We formulate an optimization problem, whose solution is a set of instrumentation instructions that switches between event-triggered and time-triggered modes of monitoring at run time; the solution may indicate the use of exactly one mode or a combination of the two modes. We fully implemented this method to produce instrumentation schemes for C programs that run on an ARM Cortex-M3 processor, and experimental results validate the effectiveness of this approach.

1 Introduction

Runtime verification [5, 19, 26] is a technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. The main challenge in augmenting a system with runtime

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 304–321, 2013.



(a) Event-triggered monitoring.

(b) Time-triggered monitoring.

(c) Hybrid monitoring.

Fig. 1. Comparing different methods of monitoring

verification is to contain its runtime *overhead*. Most monitoring approaches in the literature are *event-triggered* (ET), where the occurrence of a new (critical) event (e.g., change of value of a variable) triggers the monitor to verify a set of logical properties. For example, in the timing diagrams in Figure 1(a), the dots 1 through n along the timeline represent the critical events that occur for an execution trace of the program under scrutiny at run time. The calls to the monitor are added as instrumentation instructions in the program. As shown in the figure, there is a burst of events in this execution trace from event i to event j. The frequent monitor invocations that occur from i to j leads to a burst of monitoring, which causes high execution overhead and unpredictability of program behavior.

In [2,3], we introduced a *time-triggered* (TT) method that makes the runtime overhead controllable and predictable, and makes monitoring tasks schedulable. In this method, a monitor samples the state of the program in periodic time intervals. The period, known as the *sampling period* (SP) is such that the monitor misses no critical events. Time-triggered monitoring is especially desirable for designing real-time embedded systems, where time predictability plays a central role. Figure 1(b) shows the interactions that occur between the program and a TT monitor. To decrease the sampling frequency and thus decrease the overhead, we introduced a technique, where the program stores critical events in a history buffer and the monitor reads this buffer to evaluate properties with respect to all state changes stored in the history [2, 3]. From the figure, it is evident that the monitoring activity between events i and j is significantly less than what an event-triggered monitor would require. However, for the sampling period adopted in this example, there are some 'redundant' samples that the monitor takes; a *'redundant' sample* is an invocation of the monitor, where there are no events to process in the buffer. The dashed ovals in Figure 1(b) mark the redundant samples in this example. Although our goal in [2, 3] was tackling the unpredictability of runtime overhead, we observed that time-triggered runtime verification (TTRV) may also reduce the cumulative runtime overhead effectively.

From Figures 1(a) and 1(b), it is evident that both event- and time-triggered monitoring techniques have some advantages and disadvantages with respect to the monitor's execution overhead. Event-triggered monitoring tends to be advantageous in situations, where critical events occur sparsely since the monitor is active only when the program encounters a critical event; time-triggered monitoring tends to be better when many critical events to process within a short time frame.

With this motivation, in this paper, we propose a novel technique based on static analysis that exploits the benefits of both ETRV and TTRV to reduce the runtime overhead, which we call *hybrid* runtime verification (HyRV). Our goal is to supply a program under scrutiny with a monitor that supports both ET and TT modes of operation. The program switches from one mode to another at run time depending upon the current execution path. HyRV automatically obtains the locations to *switch modes* in the program by solving an optimization problem; this method accounts for all monitoring and switching costs in terms of execution time overhead. The main challenge in formulating the optimization problem is threefold:

1. determining the precise timing behaviour of the program under inspection,
2. identifying the overhead of all required activities for implementing an ET or TT monitor (e.g., cost of monitoring mode switching, sampling, monitor invocation),
3. identifying the execution subpaths that are likely to be suitable for ET and TT monitoring modes.

The solution to the problem is an instrumentation scheme for a program that may switch monitoring modes at runtime. For instance, in Figure 1(c), the reduction in monitoring activity will likely reduce the overall monitoring execution overhead. Obviously, using hybrid monitoring will incur overhead costs in performing mode switches. In this example, a mode switch occurs right before i and right after j to switch from ET to TT and TT to ET monitoring modes, respectively.

We implemented this technique in a toolchain that leverages static analysis techniques and integer linear programming (ILP) to solve the optimization problem. The input to our toolchain is a C program and a set of variables to monitor. The toolchain outputs the program source code augmented with the instrumentation scheme that may toggle the monitoring mode at runtime to reduce the monitoring overhead. Currently, our toolchain does not include static analysis of library calls. The results of our experiments on a benchmark suite for real-time embedded programs strongly validate the effectiveness of our technique.

Organization. The rest of the paper is organized as follows. Section 2 describes the concepts of ETRV and TTRV. Section 3 introduces the HyRV optimization problem. We analyze the results of our experiments in Section 4. Section 5 discusses the related work. Finally, in Section 6, we make concluding remarks and discuss future work.

2 Background

Let P be a program under inspection and Π be a logical property (e.g., in LTL), where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. In *event-triggered runtime verification* (ETRV), the instrumented version of P invokes the monitor to evaluate Π whenever the value of some variable in $\mathcal{V}_\Pi$ changes.

In *time-triggered runtime verification* (TTRV) [2, 3], a monitor *samples* the value of variables in $\mathcal{V}_\Pi$ periodically and evaluates Π. Accurate reconstruction of states of P between two consecutive samples is the main challenge in using this mechanism; e.g., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two samples, then the monitor may fail to detect violations of Π. TTRV usually leverages control-flow analysis to reconstruct the states of P.

To ensure that the behaviour of a time-triggered monitor is correct, the monitor must sample at a 'safe' rate determined by statically analyzing P's control-flow graph:

Definition 1. *The* control-flow graph *(CFG) of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w, v^f \rangle$, where:*

- *V: is a set of* vertices*, each representing a basic block of P. Each basic block consists of a sequence of instructions in P.*
- *v^0: is the* initial vertex *with in-degree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u immediately leads to the execution of basic block v.*
- *w: is a function $w : A \to \mathbb{N}$, which defines a* weight *for each arc in A. The weight of an arc is the* best-case execution time *(BCET) of the source basic block.*
- *v^f: is a dummy vertex which acts as final vertex. It has incoming arcs from all actual final vertices. This helps in simplifying analysis by allowing us to easily consider weight of final vertices.*

For example, consider the C program in Figure 2 [2]. Figure 3(a) shows the resulting CFG assuming that the BCET of each line of code is one time unit. Vertices of the graph in Figure 3 list the corresponding line numbers of the C program in Figure 2.

To identify the *sampling period* that a monitor can accurately reconstruct program states between two samples, we modify CFG_P as follows:

```
 1  scanf(”%d”, &a);
 2  if ( a % 2 == 0 ) {
 3      printf(”%d is even”, a);
 4  } else {
 5      b = a / 2;
 6      c = a / 2 + 1;
 7      printf(”%d is odd”, a);
 8  }
 9  d = b + c;
10  end program
```

Fig. 2. A simple C program

Step 1: Identify Critical Vertices

We ensure that each *critical instruction* (i.e., an instruction that modifies a variable in $\mathcal{V}_\Pi$) is in a basic block that contains no other critical instructions. We refer to such a basic block as a *critical basic block* or *critical vertex*. For example, in Figure 2, if variables `b`, `c`, and `d` are in $\mathcal{V}_\Pi$, then lines 5, 6, and 9 are critical instructions. Since instructions in lines 5 and 6 are critical and they both reside in basic block c, we split c into c_1 and c_2 as shown in Figure 3(b); the highlighted vertices in the figure denote the critical basic blocks.

Step 2: Calculate the Longest Sampling Period

As mentioned earlier, the main challenge in using TTRV is accurate program state reconstruction. To preserve all critical program state changes, the monitor must sample at a rate that can capture all possible critical state changes of P at run time. The corresponding sampling period is called the *longest sampling period* (LSP). Definition 2 formally defines LSP.

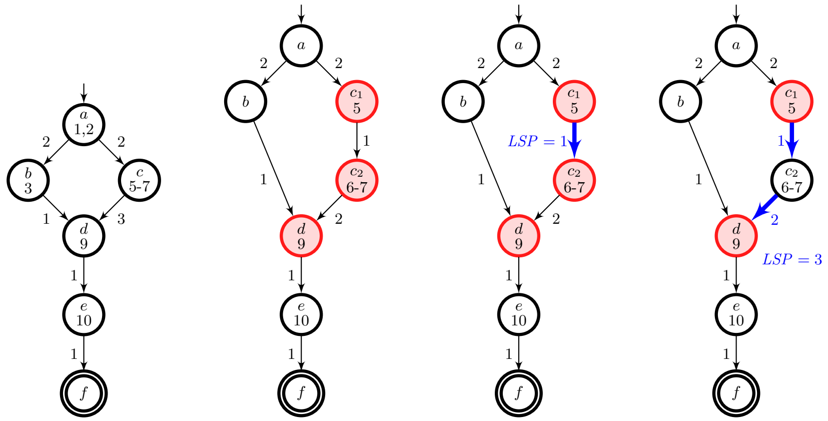
Definition 2. *Let CFG* $= \langle V, v^0, A, w\rangle$ *be a control-flow graph;* $V_c \subseteq V$ *be the set of vertices that correspond to critical basic blocks of CFG; and* Π_c *be the set of paths* $\langle v_h \rightarrow v_{h+1} \rightarrow \cdots \rightarrow v_{k-1} \rightarrow v_k\rangle$ *in CFG such that* $v_h, v_k \in V_c$ *and* $v_{h+1}, \ldots, v_{k-1} \in V \setminus V_c$*. The* longest sampling period (LSP) *for CFG is*

$$LSP_{CFG} = \min_{\pi \in \Pi_c} \left\{ \sum_{\substack{(v_i, v_j) \in A \\ v_i, v_j \in \pi}} w(v_i, v_j) \right\}$$

Intuitively, the LSP is the minimum timespan between two successive changes of any two variables in $\mathcal{V}_\Pi$. This means that the minimum distance between all pairs of critical vertices in *CFG* is the LSP. For example, the LSP of the CFG shown in Figure 3(c) is $LSP = 1$, as indicated in the figure. All property violations can be detected if the monitor samples with a period of LSP [2].

Step 3: Increase the Sampling Period Using Auxiliary Memory

To increase the longest sampling period (and, hence, decrease the involvement of the monitor), we use auxiliary memory to buffer critical state changes between two consecutive samples. Precisely, let v be a critical vertex in a control-flow



(a) CFG (b) Step 1 (c) Step 2 (d) Step 3

Fig. 3. Steps for obtaining optimized instrumentation and sampling period

graph, *CFG*, where critical instruction *inst* in v changes the value of a variable $a \in \mathcal{V}_\Pi$. We insert an instruction $inst' : a' \leftarrow a$ immediately following *inst*, where a' is an auxiliary memory location, to the sequence of instructions corresponding to vertex v. After instrumenting (i.e., adding $inst'$) v, v is no longer a critical basic block (i.e., $v \in V \setminus V_c$) because the added instruction guarantees that the monitor will observe this change when it processes the history stored in auxiliary memory. For example, instrumenting vertex c_2 in Figure 3(c) by adding an instruction of the form '`ch = c`' directly after line 6 of the program results in the CFG shown in Figure 3(d). Instrumenting the critical instruction in c_2 effectively increases the LSP to 3 because of the buffered event. The maximum *violation detection latency* (i.e., the time elapsed between the occurrence of a property violation and the detection of the violation) of Π, the availability of auxiliary memory and other system constraints limit the number of times we can apply step 3 to increase the LSP.

3 Hybrid Event-Triggered and Time-Triggered Runtime Verification

In this paper, our goal is to select the monitoring scheme that minimizes the expected total overhead incurred from executing the monitor. In order to formally introduce the problem statement, we need to define the underlining monitoring overhead cost model.

3.1 Overhead Runtime Costs

Broadly, we classify the *overhead costs* incurred from monitoring into three categories:

- C_{event}: the cost incurred to handle each critical event (i.e., in TT mode, this includes the costs of writing and retrieving the history, and the property evaluation; in ET mode, this includes calling the monitor and the property evaluation),
- C_{switch}: the cost incurred from switching between ET and TT modes and vice versa, and
- C_{sample}: the cost incurred from sampling in TT mode.

To derive expressions for the monitoring overhead, the cost of monitoring is broken down into five *elementary cost* values, which capture the costs incurred from performing specific interactions between the program and the monitor:

- c_{ET}: cost of invoking monitor to check a single critical event in ET mode
- c_{hist}: cost of saving a critical event into the history buffer in TT mode
- c_{TT}: cost of processing the history buffer at a sample in TT mode
- $c_{E\to T}$: cost of a switch from ET mode to TT mode
- $c_{T\to E}$: cost of a switch from TT mode to ET mode

Note that these costs are derived in terms of best-case execution time of the corresponding instructions. In particular, we calculate these costs in the same fashion we obtain the arc weights of a control-flow graph (see Definition 1).

3.2 Problem Definition

Let $G = \langle V, v^0, A, w, v^f \rangle$ be the control-flow graph of program P and $V_c \subseteq V$ be the set of critical vertices after computing the longest sampling period LSP through application of 3 steps given in Section 2. We are also given five elementary costs $c_{ET}, c_{hist}, c_{TT}, c_{E\to T}$, and $c_{T\to E}$ as defined in Subsection 3.1. Assuming all execution paths in G are equally likely, our goal is to find a HyRV monitoring scheme M, such that $M_o(G)$ (monitoring overhead of M) is minimum. A HyRV monitoring scheme is

$$M : V \to \{0, 1\} \tag{1}$$

Where 0 denotes that vertex should be monitored using ET monitor whereas 1 indicates TT monitor should be used to monitor the vertex. Note that to uniquely determine the location of a switch, we take domain of V rather than just V_c. For a given path $\pi = v^0 \to v_1 \to \cdots \to v^f$ of G, the overhead of a monitoring scheme is defined as:

$$\begin{aligned} M_o(\pi) = &\sum_{v \in V_c} [c_{ET} \cdot (1 - M(v)) + c_{hist} \cdot M(v)] \\ &+ \sum_{(v_1, v_2) \in A} [c_{E\to T} \cdot (1 - M(v_1)) \cdot M(v_2) + c_{T\to E} \cdot M(v_1) \cdot (1 - M(v_2))] \\ &+ \sum_{\substack{\delta = \langle v_i \to \ldots \to v_j \rangle, \\ \delta \in \Delta_\pi}} \left[c_{TT} \cdot \left(\lceil \frac{\sum_{k=i}^{k=j} w(v_k)}{LSP} \rceil \right) \right] \end{aligned} \tag{2}$$

Where $\varDelta_\pi$ is set of longest subpaths of π whose vertices are monitored using TT scheme. Three sums in equation 2 correspond to C_{event}, C_{switch}, and C_{sample} costs respectively. Let $\varPi$ denotes set of all execution paths of CFG G, the overhead of a monitoring scheme M for program P with CFG G is:

$$M_o(G) = \sum_{\pi \in \varPi} M_o(\pi) \tag{3}$$

3.3 Complexity Analysis

We believe that finding the monitoring scheme 1, which minimizes the overhead cost (Equation 3) for a given CFG, requires knowledge of execution paths of the CFG. This is because depending upon what had happened on a path it may not be beneficial to switch to the optimal monitoring scheme for the rest of the path. Such an interference is not only present in an execution path but also among interacting paths. To illustrate this further consider Figure 4. In an optimal solution, the distribution of critical events on the path $c \rightsquigarrow d$ affects the decision about the monitoring mode (i.e., TT or ET) for vertices on the path $\rightsquigarrow a$ and vice-versa. It may not be correct to choose optimal strategy for the paths $\rightsquigarrow a$ and $c \rightsquigarrow d$ separately if it causes switching on edge (a, c), and the cost of this switching overruns the benefit gained by choosing local optimal solutions for the two paths. This causes intra-path interference among vertices. Note that monitoring mode decision about vertices on the path $\rightsquigarrow b$ is influenced by choice of monitoring mode for virtices on the path $c \rightsquigarrow d$ which in turn gets affected by events on the path $\rightsquigarrow a$. This results into inter-path interference among intersecting paths. The presence of intra- and inter-path interference among vertices indicates that local optimization cannot guarantee overall optimal solution for a given CFG, and all execution paths should be analyzed. However, the presence of unbounded loops makes analysis of all execution paths impossible. Also, even in the absence of unbounded loops, a general CFG can have exponentially many execution paths. This makes the problem of finding the optimal solution intractable.

In order to tackle the high computational complexity of the problem and to make this technique practical, we introduce a heuristic that aims to return a monitoring scheme whose monitoring overhead is equal to or better (i.e. lower) than exclusively in ET or TT schemes. We formulate an *integer linear program* (ILP) as a heuristic for this problem. In order to make this heuristic reflect the realities of the program without computing all execution paths, we assume that function $\mathcal{F} : (u, v) \rightarrow \mathbb{N}$, $(u, v) \in A$, $u, v \in V$ is provided along with CFG of a program P. $\mathcal{F}(u, v)$ defines the expected number of times P will execute the basic block corresponding to v immediately after executing the basic block corresponding to u. Figure 5 illustrates a *CFG*, where the critical vertices are highlighted. The set of numerical values within parentheses defines the function, $\mathcal{F}(u, v)$. We note that this function can be evaluated using standard techniques such as program profiling and symbolic execution. The suboptimality stems from the division of the program into subpaths to estimate the monitoring cost and

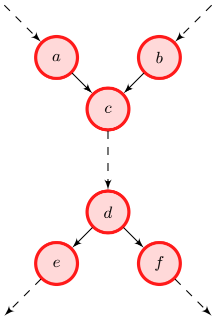


Fig. 4. Intra- and inter-path interference among vertices

the use of function $\mathcal{F}$ which may not represent correct system's behaviour. Computing function $\mathcal{F}$ with high accuracy is desirable because even a small reduction in overhead will have large benefit in the long run of a monitor.

For the rest of this paper, let $CFG = \langle V, v^0, A, w, v^f, \mathcal{F}\rangle$ be a control-flow graph corresponding to a program P. Each vertex corresponds to a critical basic block containing one critical instruction. The definitions of V, v^0, A, w, and v^f correspond to the Definition 1 (see Figure 3(b) for an example).

3.4 The Optimization Problem as an Integer Linear Program

The ILP problem is of the form:

$$\begin{cases} \text{Minimize} \quad c.\mathbf{z} \\ \\ \text{Subject to } A.\mathbf{z} \geq \mathbf{b} \end{cases}$$

where A (a rational $m \times n$ matrix), c (a rational n-vector) and $\mathbf{b}$ (a rational m-vector) are given, and $\mathbf{z}$ is an n-vector of integers to be determined. In other words, we try to find the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities and inequalities can always be put in the above form, implying that this formulation is more general than it might look.

Objective Function. The objective function for our ILP model is:

$$minimize \quad (C_{event} + C_{switch} + C_{sample}) \tag{4}$$

We now describe how we map the optimization objective (Equation 4) by introducing ILP variables and computing each of three costs in terms of these variables and given elementary costs for a CFG.

ILP Variables. We associate two binary variables x_v and y_v for each $v \in V$ in *CFG*. If $x_v = 1$, then the monitor will operate in ET mode whenever the corresponding basic block executes, and if $y_v = 1$, the monitor will operate in TT mode whenever the program is executing the basic block. The following constraint expresses the mutual exclusivity of monitoring modes for $v \in V$:

$$x_v + y_v = 1 \tag{5}$$

Constraint of Handling Critical Events. Equation 6 expresses the cost incurred at each critical event in P:

$$C_{event} = \sum_{v \in V_c} \sum_{\substack{(u,v) \in A \\ u \in V}} [\mathcal{F}(u,v) \cdot (c_{ET} \cdot x_v + c_{hist} \cdot y_v)] \tag{6}$$

where $V_c \subseteq V$ is the set of nodes that correspond to the critical basic blocks in *CFG*. The number of times that P will transit from the set of nodes u to v, where $(u,v) \in A$, determines the expected number of times that the basic block corresponding to v will execute. Equations 5 and 6 guarantee that the cost incurred for the critical event in v is exclusively c_{ET} or c_{TT} if the monitor is operating in ET or TT mode at that point in the program, respectively.

Constraints of Switching Monitoring Mode. The following equation expresses the cost of switching between ET and TT modes:

$$C_{switch} = \sum_{\substack{(v_1,v_2) \in A \\ v_1,v_2 \in V}} [\mathcal{F}(v_1,v_2) \cdot (c_{E \to T} \cdot x_{v_1} \cdot y_{v_2} + c_{T \to E} \cdot y_{v_1} \cdot x_{v_2})] \tag{7}$$

There exists a mode switch between basic blocks v_1 and v_2 when $x_{v_1} = y_{v_2} = 1$ or $y_{v_1} = x_{v_2} = 1$. The former case implies that the monitor switches from ET mode to TT mode and the latter case implies that the monitor switches from TT mode to ET mode. Equation 7 is non-linear; to linearize this expression, we introduce the binary variables p_{v_1,v_2}, q_{v_1,v_2}, r_{v_1,v_2}, and s_{v_1,v_2} and rewrite Equation 7 as:

$$C_{switch} = \sum_{\substack{(v_1,v_2) \in A \\ v_1,v_2 \in V}} [\mathcal{F}(v_1,v_2) \cdot (c_{E \to T} \cdot p_{v_1,v_2} + c_{T \to E} \cdot q_{v_1,v_2})] \tag{8}$$

subject to:

$$x_{v_1} + y_{v_2} + 2r_{v_1,v_2} \geq 2 \tag{9}$$

$$p_{v_1,v_2} + r_{v_1,v_2} = 1 \tag{10}$$

$$x_{v_1} + y_{v_2} - 2(1 - r_{v_1,v_2}) < 2 \tag{11}$$

$$y_{v_1} + x_{v_2} + 2s_{v_1,v_2} \geq 2 \tag{12}$$

$$q_{v_1,v_2} + s_{v_1,v_2} = 1 \tag{13}$$

$$y_{v_1} + x_{v_2} - 2(1 - s_{v_1,v_2}) < 2 \tag{14}$$

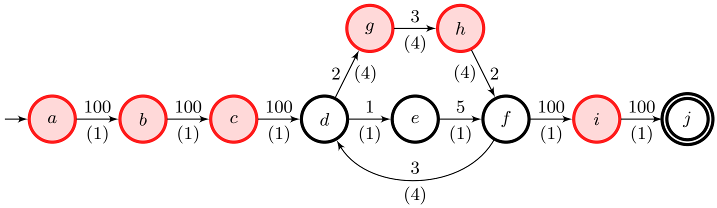


Fig. 5. CFG used for illustrating ILP model

Equations 9 through 11 ensure that if $x_{v_1} = y_{v_2} = 1$, then $p_{v_1,v_2} = 1$, i.e., we incur the cost of switching from ET to TT mode. Similarly, the constraints reflected in Equations 12 through 14 ensure that if there exists a switch from TT to ET mode, then $q_{v_1,v_2} = 1$ and we incur the cost $c_{T\rightarrow E}$.

Constraints of Sampling Cost in TT Mode. Finally, Equation 15 captures the cost incurred from the sampling the monitor does in TT mode:

$$C_{sample} = \sum_{\pi \in \mathit{\Pi}'(CFG)} \left(c_{TT} \cdot \mathcal{F}_{\pi} \cdot N_{samp_{\pi}} \right) \tag{15}$$

where $\mathit{\Pi}'(CFG)$ denotes the set of all subpaths $\pi = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ in CFG that satisfy the following four conditions:

1. $k \geq 2$
2. $\mathit{indegree}(v_i) = \mathit{outdegree}(v_i) = 1,\ 2 \leq i \leq k-1$
3. $\mathit{indegree}(v_1) \neq 1 \vee \mathit{outdegree}(v_1) \neq 1$
4. $\mathit{indegree}(v_k) \neq 1 \vee \mathit{outdegree}(v_k) \neq 1$
5. for each $(v_i, v_j) \in A$, (v_i, v_j) appears in exactly one $\pi \in \mathit{\Pi}'(CFG)$

For example, if we consider the CFG shown in Figure 5, $\mathit{\Pi}'(CFG) = \{\langle a \rightarrow b \rightarrow c \rightarrow d\rangle, \langle d \rightarrow e \rightarrow f\rangle, \langle f \rightarrow d\rangle, \langle d \rightarrow g \rightarrow h \rightarrow f\rangle, \langle f \rightarrow i \rightarrow j\rangle\}$. Moreover, in Equation 15, $\mathcal{F}_{\pi}$ is the expected number of times that π will execute at run time. $\mathcal{F}_{\pi} = \mathcal{F}(v_i, v_j)$, where (v_i, v_j) is any arc on path π. $N_{samp_{\pi}}$ is the number of samples that the monitor takes when P executes π once:

$$N_{samp_{\pi}} = \sum_{\substack{\gamma = \langle v_i \rightarrow \ldots \rightarrow v_j \rangle, \\ \gamma \in \mathit{\Gamma}_{\pi}}} \left[\left(\frac{W(\gamma) + c_{hist} \cdot \sum_{m=i}^{j} y_{v_m}}{SP} \right) \cdot \left(x_{v_{i-1}} \cdot x_{v_{j+1}} \cdot \prod_{l=i}^{j} y_{v_l} \right) \right] \tag{16}$$

where $W(\gamma)$ returns the sum of weights of all arcs on the path $\gamma \in \mathit{\Gamma}_{\pi}$; v_{i-1} and v_{j+1} denote the immediate predecessor and successor of $v_i, v_j \in V$, respectively; and SP is the allowed sampling period of the monitor when it is operating in TT mode. Γ_π is the enumerated set of paths in $\pi \in \Pi'(CFG)$ of length 2 or greater. Using $\Pi'(CFG)$ for the CFG shown in Figure 5, if we consider the subpath $\pi = \langle d \to g \to h \to f\rangle$, then $\Gamma_\pi = \{\langle d \to g \to h \to f\rangle, \langle d \to g \to h\rangle, \langle g \to h \to f\rangle, \langle d \to g\rangle, \langle g \to h\rangle, \langle h \to f\rangle\}$. Note that $|\Gamma_\pi| = \Theta\left(|\pi|^2\right)$. If v_{i-1} does not exist in π, $x_{v_{i-1}} = 1$. Similarly, $x_{v_{j+1}} = 1$ if v_{j+1} does not exist in π. Considering the example where $\pi = \langle d \to g \to h \to f\rangle$, if $\gamma \in \Gamma_\pi$ starts with d or ends with f, then we would ignore the terms $x_{v_{i-1}}$ and $x_{v_{i+1}}$ by substituting them with the value of 1, respectively. N_{samp_π} is linearized by the linearization technique employed for C_{switch}.

4 Implementation and Experimental Results

We empirically tested and verified our hybrid monitoring approach for a subset of programs from the SNU Real-time benchmark suite [1] on an embedded development platform with real-time guarantees. In Subsection 4.1, we describe the experimental setup and the toolchain. Then, in Subsection 4.2, we present and analyze the results of our experiments.

4.1 Experimental Setup

Figure 6 depicts the constructed toolchain used to generate instrumentation schemes from the model described in Section 3. The toolchain generates the program's CFG with estimated execution times of basic blocks by statically analyzing the program's source code with clang and llvm [18]. We use the tool CodeSurfer [9] to determine the location of the critical events the monitor should track at run time. The model generator takes this information along with the estimated monitoring costs to produce the corresponding model for the program. The toolchain then uses Yices [23], an SMT solver, to identify a solution (i.e., an instrumentation scheme) to the optimization problem described in Section 3. A script then takes the instrumentation scheme and instruments the program source with the necessary instructions required to monitor the program accordingly.

The monitor and programs were compiled and executed on the Keil µVision simulator that emulates the behavior of the MCB1700 development platform, which sports an ARM Cortex-M3 processor. We emphasize that the observed execution time across multiple runs of the experiment remains constant because the hardware platform provides accurate timing behavior of instructions, and in each experiment, the only tasks running were the program under inspection and the monitor. Therefore, it is safe to present the results without reporting statistical measures.

We used SNU-RT [1] benchmark suite for the performance analysis. We selected six programs from the suite with different sizes: `bs`, `fibcall`, `insertsort`, `fir`, `crc`, and `matmult`. The largest program has 250 lines of code, and the smallest has 20. We picked two sets of variables for monitoring for each program: (1) a set containing frequently changing variables and (2) a set containing rarely

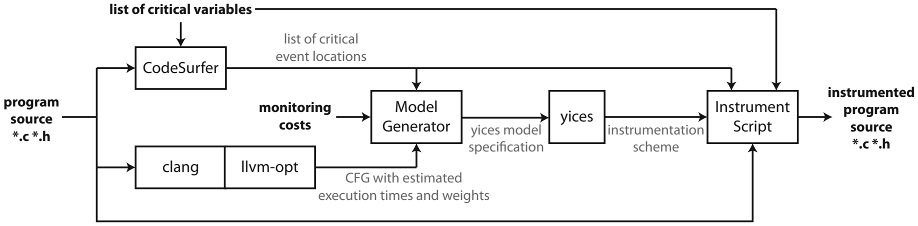


Fig. 6. HyRV instrumentation toolchain for C applications

Table 1. Monitor cost configurations [clock cycles]

Configuration	c_{hist}	c_{ET}	c_{TT}	$c_{E\to T}$	$c_{T\to E}$
1	50	100	100	100	100
2	50	100	100	150	150
3	50	150	150	100	100
4	50	150	150	150	150
5	50	250	250	100	100
6	50	250	250	150	150

changing variables. Instructions that potentially change the value of these variables form the set of critical instructions monitored in the experiments. For each program, the monitoring overheads were measured using the cost configurations (listed in Table 1) and associated instrumentation schemes. The cost configurations depend on the implementation of the monitor (e.g., running on the same processor, distributed). We use the configurations in Table 1 to demonstrate that the instrumentation schemes may change as a result of the relative differences in the elementary monitoring costs.

4.2 Experimental Results

We classify the results of our experiments based on the generated instrumentation scheme and runtime overhead:

1. The first class consists of cases, where our ILP model suggests a hybrid monitor and the monitor indeed significantly outperforms an ET or TT monitor in practice (see Figure 7).
2. The second class consists of cases where the ILP model suggests either an ET or TT monitor and the suggested solution indeed outperforms other monitoring modes (see Figure 8).
3. The third class consists of cases where the solution to the ILP model either exhibits slight improvement over other monitoring modes or it slightly underperforms in practice (see Figure 9).

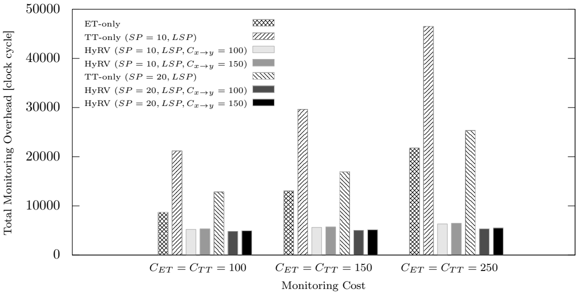


Fig. 7. Monitoring overhead of `crc` for three monitoring modes under all cost configurations

In the rest of this section, we will discuss the experimental results and focus on one program from each class. We note that the three other programs not specifically discussed in this section exhibit similar results.

Hybrid Monitor with Significant Improvement. The program representing this class (i.e., `crc` with CFG of the size 65 vertices and 82 arcs) has two characteristics: it has (1) two tight loops, each containing one critical instruction, and (2) a relatively large initialization function that contains only non-critical instructions. Intuitively, if the program is monitored by an ET monitor, then the tight loops in the program will cause monitor invocations for each iteration. This is an instance where a burst of events creates a large overhead over a short period of time (similar to the timeline in Figure 1). In such cases, an ET monitor suffers.

On the contrary, the large initialization function does not contain critical events; hence, a TT monitor would suffer from redundant sampling overhead. We hypothesize that the combination of these two characteristics can exploit the benefits of employing a hybrid monitor. The graph in Figure 7 validates our hypothesis. As can be seen, in all cost configurations, the hybrid monitor incurs significantly less overhead than both the ET monitor and TT monitor operating with the same sampling period. Another interesting observation is that increasing the cost of ET and TT monitor invocations does not greatly increase the overhead of the hybrid monitor. This is because the hybrid monitor only samples when the program reaches its tight loop, which reduces the cost of monitoring frequently occurring critical events by buffering them into memory before sampling. In addition, the monitoring scheme reduces the number of redundant samples by letting the monitor run in ET mode when critical events are infrequent. In such cases, the behavior of a hybrid monitor is quite robust when the cost of monitor invocation increases.

Time-Triggered Monitor with Significant Improvement. The common characteristic of the member programs of this class (i.e., `bs`, `fibcall`, `insertsort`,

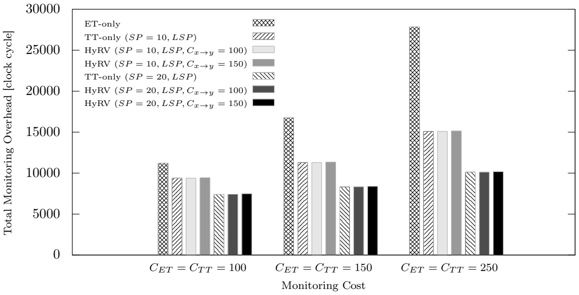


Fig. 8. Monitoring overhead of `insertsort` for three monitoring modes under all cost configurations

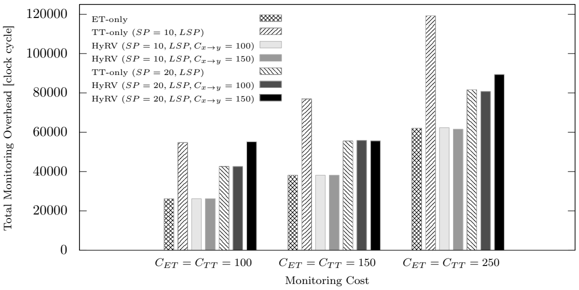


Fig. 9. Monitoring overhead of `fir` for three monitoring modes under all cost configurations

and `matmult`) is that the programs have dense and evenly distributed critical instructions in their respective CFG. This makes the use of TT mode a suitable choice to monitor this class of programs. Figure 8 shows the overhead of monitoring `insertsort` with three monitoring modes (ET-only, TT-only, and hybrid) for all cost configurations. The rest of the programs in this class also exhibit similar monitoring overhead patterns. From Figure 8, one can observe that the corresponding ILP model correctly detects the even distribution of events and the solution suggests monitoring exclusively in TT mode as its solution for all cost configurations. Another observation in these experiments is that the number of redundant samples for these programs is either zero or close to zero. The low number of redundant samples again validates the choice of monitoring these programs using the time-triggered method.

Hybrid Monitor with Mixed Behavior. The program representing this class (i.e. `fir` with CFG of the size 24 vertices and 27 arcs) does not clearly belong to the previous two classes. The number of redundant samples for this program reduces by a factor of six as the sampling period increases from 10 × *LSP* to 20 × *LSP*. This brings the overheads of ET and TT modes to a comparable level and makes the ILP model outcome highly sensitive to the elementary monitoring costs. Figure 9 shows the monitoring overhead of `fir` under the three modes of monitoring for different cost configurations. One can observe that when the sampling period is $10 \times LSP$, the model correctly chooses ET mode for the monitoring schemes. However, if we set the sampling period to $20 \times LSP$, then the ILP model provides a hybrid solution for all three cost configurations. The proposed hybrid solutions have slightly higher overheads in comparison to ET mode, but perform as good as TT mode except for two cases in practice. The reason for this discrepancy lies in the fact that our approach is a heuristic algorithm and, hence, finds suboptimal solutions in some cases. Note, however, that this discrepancy does not dramatically affect the usefulness of our approach.

5 Related Work

In classic runtime verification [21], a system is composed with an external observer, called the monitor. This monitor is normally an automaton synthesized from a set of properties under which the system is scrutinized. From the logical and language point of view, runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) properties [8, 10–12, 25] and, in particular, safety properties [14, 22]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [15,16,27]. [6] considered runtime verification of ω-languages. In [7], the authors address runtime verification of safety-progress [4, 20] properties.

The main focus in the literature of runtime verification is on *event-triggered* monitors [17], where every change in the state of the system triggers the monitor for analysis. Alternatively, in *time-triggered* monitoring [2, 3], the monitor samples the state of the program under inspection at regular time intervals. The time-triggered approach involves solving an optimization problem that aims at minimizing the size of auxiliary memory required so that the monitor can correctly reconstruct the sequence of program state changes. Several heuristics were introduced to tackle

Finally, in [13], the authors propose a method to control the overhead of software monitoring using control theory for discrete event systems. In this work, overhead control is achieved by temporarily disabling involvement of monitor, thus avoiding the overhead to pass a user-defined threshold. Another relevant work to this line of research is [24], where the authors propose sampling using state estimation. In particular, they use hidden Markov models to estimate future reachable states for deciding whether or not the monitor must sample the program under inspection. However, the methods in [13] and [24] do not guarantee correct state reconstruction because the monitor is unaware of all program state changes that may occur between samples.

6 Conclusion

In this paper, we concentrated on combining two techniques in the literature of runtime verification to reduce the overhead: (1) the traditional event-triggered

(ET) approach, and (2) the time-triggered (TT) method for real-time systems. We showed that one can effectively exploit the advantages of both approaches to reduce the overhead of runtime monitoring. To this end, we formulated an optimization problem that takes into account the cost of different monitoring interactions (i.e., monitor invocation in ET, sampling and building history in TT, and mode switching). In particular, the objective of the problem is to minimize the cumulative overhead in all execution paths using the aforementioned costs. Since solving the general problem can be computationally unsolvable (e.g., due to the existence of unbounded loops) or intractable, we proposed a heuristic that finds suboptimal but effective solutions to the problem by transforming it into an instance of the integer linear programming problem. Our experimental results on the SNU-RT benchmark suite showed that our technique effectively reduces the overhead as compared to selecting the ET or TT method in an ad-hoc manner.

There exist several interesting future research directions. We plan to employ symbolic execution techniques to implement a more accurate and realistic prediction function used for conditional and loop statements (see Section 3). Another open problem is to design other heuristics with lower time complexity that eliminate subpath generation. Examples include techniques that exploit static analysis such as graph density and dynamic analysis such as feedback control.

Acknowledgements. This research was supported in part by NSERC Discovery Grant 418396-2012, NSERC Strategic Grant 430575-2012, NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, CFI 20314, CMC, and the industrial partners associated with these projects.

References

1. SNU Real-Time Benchmarks, `http://www.cprover.org/goto-cc/examples/snu.html`.
2. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 88–102. Springer, Heidelberg (2011)
3. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Time-triggered runtime verification. Formal Methods in Systems Design (FMSD) 43(1), 29–60 (2013)
4. Chang, E.Y., Manna, Z., Pnueli, A.: Characterization of Temporal Property Classes. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 474–486. Springer, Heidelberg (1992)
5. Colin, S., Mariani, L.: 18 Run-Time Verification. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 525–555. Springer, Heidelberg (2005)
6. D'Amorim, M., Roşu, G.: Efficient Monitoring of omega-Languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)
7. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime Verification of Safety-Progress Properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009)

8. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Automated Software Engineering (ASE), pp. 412–416 (2001)
9. GrammaTech Inc. CodeSurfer®, `http://www.grammatech.com/products/codesurfer/`.
10. Havelund, K., Rosu, G.: Monitoring Programs Using Rewriting. In: Automated Software Engineering (ASE), pp. 135–143 (2001)
11. Havelund, K., Roşu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
12. Havelund, K., Rosu, G.: Monitoring Java Programs with Java PathExplorer. Electronic Notes in Theoretical Computer Science 55(2) (2001)
13. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. Software Tools for Technology Transfer, STTT (2011) (to appear)
14. Havelund, K., Rosu, G.: Efficient Monitoring of Safety Sroperties. Software Tools and Technology Transfer (STTT) 6(2), 158–173 (2004)
15. Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, Checking, and Steering of Real-Time Systems. Electronic Notes in Theoretical Computer Science 70(4) (2002)
16. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: A Run-Time Assurance Approach for Java Programs. Formal Methods in System Design (FMSD) 24(2), 129–155 (2004)
17. Kupferman, O., Vardi, M.Y.: Model Checking of Safety Properties. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999)
18. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization, p. 75 (2004)
19. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. Journal of Logic and Algebraic Programming (JLAP) 78, 293–303 (2009)
20. Manna, Z., Pnueli, A.: A Hierarchy of Temporal Properties. In: Principles of Distributed Computing (PODC), pp. 377–410 (1990)
21. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
22. Roşu, G., Chen, F., Ball, T.: Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008)
23. SRI. Yices: An SMT Solver (1.0.34), `http://yices.csl.sri.com/index.shtml`
24. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)
25. Stolz, V., Bodden, E.: Temporal Assertions using Aspectj. Electronic Notes in Theoretical Computer Science 144(4) (2006)
26. Zee, K., Kuncak, V., Taylor, M., Rinard, M.: Runtime checking for program verification. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 202–213. Springer, Heidelberg (2007)
27. Zhou, W., Sokolsky, O., Loo, B.T., Lee, I.: *MaC*: Distributed Monitoring and Checking. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 184–201. Springer, Heidelberg (2009)

A Scala DSL for Rete-Based Runtime Verification

Klaus Havelund*

Jet Propulsion Laboratory
California Institute of Technology, California, USA

Abstract. Runtime verification (RV) consists in part of checking execution traces against formalized specifications. Several systems have emerged, most of which support specification notations based on state machines, regular expressions, temporal logic, or grammars. The field of Artificial Intelligence (AI) has for an even longer period of time studied rule-based production systems, which at a closer look appear to be relevant for RV, although seemingly focused on slightly different application domains, such as for example business processes and expert systems. The core algorithm in many of these systems is the Rete algorithm. We have implemented a Rete-based runtime verification system, named LogFire (originally intended for offline log analysis but also applicable to online analysis), as an internal DSL in the Scala programming language, using Scala's support for defining DSLs. This combination appears attractive from a practical point of view. Our contribution is in part conceptual in arguing that such rule-based frameworks originating from AI may be suited for RV.

1 Introduction

Runtime Verification (RV) consists of monitoring the behavior of a system, either online as it executes, or offline after its execution, for example by analyzing log files. Although this task seems easier than verification of all possible executions, this task is challenging. From an *algorithmic* point of view the challenge consists of efficiently processing events that carry data. When a monitor receives an event, it has to efficiently locate what part of the monitor is relevant to activate, as a function of the data carried by the event. This is called the *matching problem.* From an *expressiveness* point of view, a logic should be as expressive as possible. From a *elegance* point of view a logic should be easy to use and succinct for simple properties. The problem has been addressed in several monitoring systems within the last years. Most of these systems implement specification languages which are based on state machines, regular expressions, temporal logic, or grammars. The most efficient of these, for example [11], however, tend to have limited expressiveness as discussed in [3].

* The work described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 322–327, 2013.

It can be observed that rule-based programming seems like an attractive approach to monitoring, as exemplified by the RULER system [5]. Rules are of the form *lhs* $\Rightarrow$ *rhs*, where the left-hand side are conditions on a memory of *facts*, and the right-hand side is an action that can add or remove facts, or execute other code, including yielding error messages. This model seems very well suited for processing data rich events, and is simple to understand due to its operational flavor. Within the field of Artificial Intelligence (AI) rule-based production systems have been well studied, for example in the context of expert systems and business rule systems. The RETE algorithm [6] is a well-established efficient pattern-matching algorithm for implementing such production rule systems. It maintains a network of nodes through which facts filter to the leaves where actions (rule right-hand sides) are executed. It avoids re-evaluating conditions in a rule's left hand side each time the fact memory changes. This algorithm has acquired a reputation for "extreme difficulty". Our primary goal with this work has been to understand how well this algorithm serves to solve the runtime verification task, and hence attempt to bridge two communities: formal methods and artificial intelligence. An initial discussion of this work was first presented in [8]. We have specifically implemented a rule-based system, named LOGFIRE, based on the RETE algorithm in the SCALA programming language as an internal DSL, essentially extending SCALA with rule-based programming. We have made some modifications to the algorithm to make it suitable for the RV problem, including fitting it for *event* processing (as opposed to *fact* processing) and optimizing it with fast indexing to handle commonly occurring RV scenarios. Early results show that the algorithm performs reasonably, although not as optimal as specialized RV algorithms, such as MOP [11].

There are several well-known implementations of the RETE algorithm, including DROOLS [1]. These systems offer external DSLs for writing rules. The DROOLS project has an effort ongoing, defining functional programming extensions to DROOLS. In contrast, by embedding a rule system in an object-oriented and functional language such as SCALA, as done in LOGFIRE, we can leverage the already existing host language features. DROOLS supports a notion of events, which are facts with a limited life time. These events, however, are not as short-lived as required by runtime verification. The event concept in DROOLS is inspired by the concept of *Complex Event Processing* [10]. Two rule-based internal DSLs for SCALA exist: HAMMURABI [7] and ROOSCALOO [2]. HAMMURABI is not RETE-based, and instead evaluates rules in parallel. ROOSCALOO [2] is RETE based, but is not documented in any form other than experimental code. A RETE-based system for aspect-oriented programming with history pointcuts is described in [9]. The system offers a small past time logic, which is implemented with a modification of the RETE algorithm. This is in contrast to our approach where we maintain the core of the RETE algorithm, and instead write or generate rules reflecting specifications. In previous work we designed the internal SCALA DSL TRACECONTRACT for automaton and temporal logic monitoring [4]. An internal SCALA DSL for 'design by contract' is presented in [12].

2 The LogFire DSL

In this section we shall illustrate LogFire by specifying a monitor for the resource management system for a planetary rover. Subsequently we will briefly explain the operational meaning of the specification.

2.1 Specification

Consider a rover that runs a collection of tasks in parallel. A resource arbiter manages resource allocation, ensuring for example that a resource is only used by one task at a time. Consider that we monitor logs containing the events:

$grant(t, r)$	: task t is granted resource r.
$release(t, r)$	: task t releases resource r.
$end()$	: the end of the log is reached.

Consider next the following informal requirement that logs containing instances of these event types have to satisfy:

> *"A resource can only be granted to one task (once) at any point in time, and must eventually be released by that task."*

We shall now formalize this requirement as a LogFire monitor. The main component of LogFire is the class *Monitor*, which any user-defined monitor must extend to get access to constants and methods provided by LogFire. User-defined monitors will contain rules of the form:

name `--` *condition*$_1$ & … & *condition*$_n$ `|->` *action*

A rule is defined by a name, a left hand side consisting of a conjunction of conditions, and a right hand side consisting of an action to be executed if all the conditions match the fact memory. A condition is a pattern matching facts or events in the fact memory, or, as we shall later see, the negation of a pattern, being true if such a fact does not exist in the fact memory. Arguments to conditions are variables (quoted identifiers of the type *Symbol*) or constants. The first occurrence of a variable in a left-hand side condition is binding, and subsequent occurrences in that rule much match this binding. An action can be adding facts, deleting facts, or generally be any Scala code to be executed when a match for the left-hand side is found. Our monitor becomes:

```
class ResourceProperties extends Monitor {
  val grant, release, end = event
  val Granted = fact

 "r1" -- grant('t, 'r) & not(Granted('t, 'r)) |-> Granted('t, 'r)
 "r2" -- Granted('t, 'r) & release('t, 'r) |-> remove(Granted)
 "r3" -- Granted('t, 'r) & grant('_, 'r) |-> fail("double grant")
 "r4" -- Granted('t, 'r) & end() |-> fail("missing release")
 "r5" -- release('t,'r) & not(Granted('t,'r)) |-> fail("bad release")
}
```

Value definitions introduce event and fact names. Rule r_1 formalizes that if a *grant*($'t,' r$) is observed, and no *Granted*($'t,' r$) fact exists in the fact memory (with the same task $'t$ and resource $'r$), then a *Granted*($'t,' r$) fact is inserted in the fact memory to record that the grant event occurred. Rule r_2 expresses that if a *Granted*($'t,' r$) fact exists in the fact memory, and a release event occurs with matching arguments, then the *Granted* fact is removed. The remaining rules express the error situations - r_3: granting an already granted resource, r_4: ending monitoring with a non-released resource, and r_5: releasing a resource not granted to the releasing task. LogFire allows to write any Scala code on the right-hand side of a rule, just as any Scala definitions are allowed in LogFire monitors, including local variables and methods. We can create an instance of this monitor and submit events to it (not shown here), which then get verified for conformance with the rules. Any errors will be documented with an error trace illustrating what events caused what rules to fire.

2.2 Meaning

Each rule definition in the class *ResourceProperties* is effectively a method call, or rather: a chain of method calls (commonly referred to as *method chaining*), which get called when the class gets instantiated (a Scala class body can contain statements). Note that Scala allows to omit dots and parentheses in method calls. As an example, the definition of rule r_2 is equivalent to the statement:

```
R(”r2”).−−(C(’Granted)(’t, ’r)).&(C(’release)(’t, ’r)).|−> {
  remove(’Granted)
}
```

The functions *R* (standing for *Rule*) and *C* (standing for *Condition*) are so-called *implicit functions*. An implicit function in Scala is defined as part of the program (in this case in the class *Monitor*), but is not explicitly called. Such functions are instead applied by the compiler in cases where type checking fails but where it succeeds if one such (unique) implicit function can be applied. In the statement above we have inserted them explicitly for illustration purposes, as the compiler will do. The function *R* takes a string as argument and returns an object of a class, which defines a function `--`, which as argument takes a condition, and returns an object, which defines a method `&`, which takes a condition, and returns an object defining a method `|->`, which takes a Scala statement (passed call by name, hence not yet executed), and finally creates a rule internally.

Creating the rules internally means building the Rete network as an internal data structure in the instantiated *ResourceProperties* object, representing the semantics of the rules. Figure 1 illustrates the network created by the definitions of rules r_2, r_3, and r_4 (rules r_1 and r_5 contain negated conditions which are slightly complicated, and therefore ignored in this short exposition). When events and facts are added to the network, they sift down from the top. For example, a *Granted*$(7, 32)$ event will end up in the lower grey buffer, from which three *join nodes* lead to different actions depending on what the next event is: *release*, *grant*, or *end*. The join nodes perform matching on arguments.

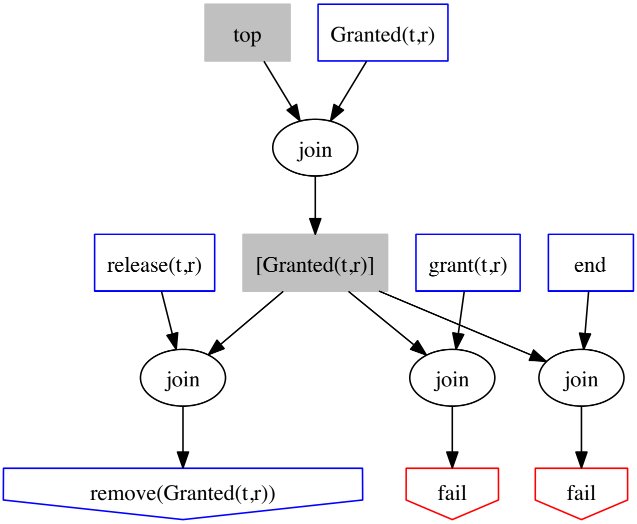


Fig. 1. The RETE network for rules r_2, r_3, and r_4

2.3 Specification Patterns

Rule-based programming as we have seen demonstrated above is an expressive and moderately convenient notation for writing monitoring properties. Although specifications are longer than traditional temporal logic specifications, they are simple to construct due to their straight forward and intuitive semantics. However, the more succinct a specification is, the better. We have as an example implemented a specification pattern in 50 lines of SCALA code in the class *PathMonitor* (not shown here). In a path expression one can provide a sequence of events and/or negation of events. A match on such a sequence anywhere in the trace will trigger a user-provided code segment to get executed. As an example, consider the following formulation of the requirement that a resource should not be granted to a task if it is already granted:

```
class DoubleGrant extends PathMonitor {
  when(”double grant”)(grant('t, 'r),no(release('t, 'r)),grant('_, 'r)) {
    fail ()
  }
}
```

The property states that when a $grant('t,'r)$ is observed, and then subsequently another $grant('\_,'r)$ of the same resource, without a $release('t,'r)$ in between, then the code provided as the last argument is executed, in this case just the reporting of a failure. The function *when* is itself defined as a sequence of rule definitions.

3 Conclusion

We have illustrated how rule-based programming based on the Rete algorithm, integrated in a high-level programming language, can be used for runtime verification. The initial experiments show that the system is very expressive and convenient, and is acceptable from a performance point of view, although not as efficient as optimized specialized RV algorithms.

References

1. Drools website, `http://www.jboss.org/drools`
2. Rooscaloo website, `http://code.google.com/p/rooscaloo`
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata - towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012)
4. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
5. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007)
6. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
7. Fusco, M.: Hammurabi - a Scala rule engine. In: Scala Days 2011. Stanford University, California (2011)
8. Havelund, K.: What does AI have to do with RV? In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 292–294. Springer, Heidelberg (2012)
9. Herzeel, C., Gybels, K., Costanza, P.: Escaping with future variables in HALO. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 51–62. Springer, Heidelberg (2007)
10. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)
11. Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Software Tools for Technology Transfer (STTT) 14(3), 249–289 (2012)
12. Odersky, M.: Contracts for Scala. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 51–57. Springer, Heidelberg (2010)

A Late Treatment of C Precondition in Dynamic Symbolic Execution Testing Tools

Mickaël Delahaye[1] and Nikolai Kosmatov[2]

[1] Université Grenoble Alpes, LIG, 38041 Grenoble, France
firstname.lastname@imag.fr
[2] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

Abstract. This paper presents a novel technique for handling a precondition in dynamic symbolic execution (DSE) testing tools. It delays precondition constraints until the end of the program path evaluation. This method allows PathCrawler, a DSE tool for C programs, to accept a precondition defined as a C function. It provides a convenient way to express a precondition even for developers who are not familiar with specification formalisms. Our initial experiments show that it is more efficient than early precondition treatment, and has a limited overhead compared to a native treatment of a precondition directly expressed in constraints. It has also proven useful for combinations of static and dynamic analysis.

Keywords: test input generation, dynamic symbolic execution, concolic testing, executable preconditions.

1 Introduction

In software testing, the *precondition* of the program under test (often called *test context*) specifies the admissible values of program inputs on which the program is supposed to be executed and should be tested. This ability to select test input domains is essential to test generation. Indeed, it allows concentrating the testing effort on admissible inputs of the program. The most common use of the test precondition is to select inputs for which the behavior of the program is specified. In that particular case, the test precondition corresponds to the specification precondition. Other uses include testing outside the specification to check for unwanted behaviors and partitioning the test domain.

In the case of automated test input generation tools, the precondition offers two interesting challenges. First, one must encode the precondition in a formalism understood by the tool. Second, the tool must take into account the precondition in its test generation process to minimize rejects for test inputs outside the precondition.

PATHCRAWLER [1] is a test input generation tool for C programs. It is based on *dynamic symbolic execution* (DSE), a technique that combines concrete execution and symbolic execution of the program under test. Originally PATHCRAWLER accepted a precondition written in a declarative constraint-based formalism specific to the tool, referred below as *native precondition*. But user feedback encouraged us to find an alternative solution.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 328–333, 2013.

In this paper, we propose a new approach to handle the precondition in a DSE tool, written in the tested language. It is based on a late exploration of the precondition's code during the test generation. This paper also provides an experimental evaluation of the late-precondition technique implemented in PATHCRAWLER. Our approach offers a greater expressiveness than PATHCRAWLER's native precondition. Our experiments also show that the late-precondition approach offers comparable performances to the native precondition and better performances than another alternative approach.

The paper is organized as follows. First, Sec. 2 gives a brief overview of precondition handling. Next, Sec. 3 describes the new method, while Sec. 4 evaluates it experimentally. Finally, Sec. 5 concludes the paper.

2 Related Work

Some test input generation tools allow to express the precondition in the tested language. Tools like Korat [2] are specifically designed to generate valid inputs based on such a precondition without considering the code of the program under test. However, a few code-based test generation tools may also handle the precondition that way. For instance, Java PathFinder [3] (generalized symbolic execution) and CUTE [4] (DSE) allow the user to provide a consistency check as a function in the tested language. The function is first solved, using the normal process of the tool. Similarly, Pex [5], a DSE tool for the .NET platform, treats Code Contracts, an embedded form of specification that is automatically translated into dynamic checks during compilation. For the precondition, assumption statements are placed **before** the code to be tested and handled as any other part of the code. Many DSE tools do not address the precondition problem specifically. However, at the cost of extra test cases, the precondition can be written as a conditional statement around the program code, leading to a solution almost equivalent to previous approaches. Like these tools, our approach proposes to encode the precondition in the tested language. However, it separates and delays the exploration of the precondition in order to minimize the exploration of the program code. To the best of our knowledge, this approach has never been used in other tools.

Another way to enforce the precondition is to describe how to construct a valid input rather than how to check whether a test case is valid. This method, sometimes called *finitization* [2], is dual to a classic precondition. Indeed, some complex structures are simpler to check than to construct, while others are better handled constructively.

The late-precondition method combines multiple symbolic executions. A related combination of symbolic executions is *compositional symbolic execution* proposed by Godefroid [6]. It aims at separating the symbolic execution of called functions in order to maximize reuse and to limit path exploration by generating so-called *function summaries*. Our late-precondition method has another objective. Indeed, it only separates the symbolic execution of two components (program and precondition) and does not create summaries. However, our late-precondition has a remarkable trait: the precondition is considered after the program. This ensures that the program's paths are explored only once without requiring to compute and store function summaries.

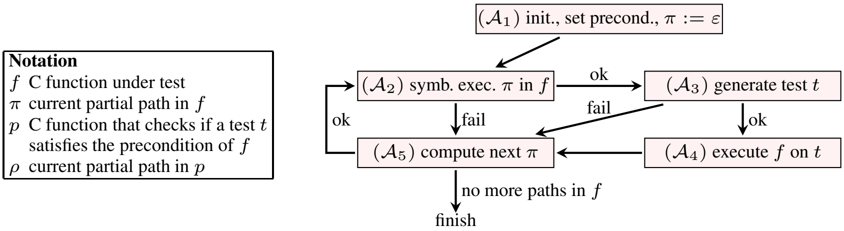

Notation
f C function under test
π current partial path in f
p C function that checks if a test t satisfies the precondition of f
ρ current partial path in p
($\mathcal{A}_1$) init., set precond., $\pi := \varepsilon$
($\mathcal{A}_2$) symb. exec. π in f
ok
($\mathcal{A}_3$) generate test t
fail
ok
fail
ok
($\mathcal{A}_5$) compute next π
($\mathcal{A}_4$) execute f on t
no more paths in f
finish

Fig. 1. Basic PATHCRAWLER test generation method (with a native precondition)

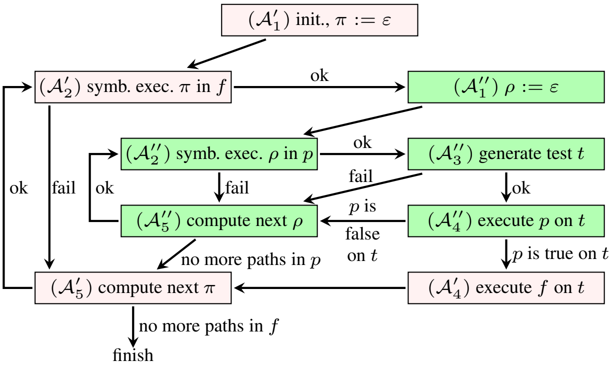

($\mathcal{A}'_1$) init., $\pi := \varepsilon$
($\mathcal{A}'_2$) symb. exec. π in f
ok
($\mathcal{A}''_1$) $\rho := \varepsilon$
($\mathcal{A}''_2$) symb. exec. ρ in p
ok
($\mathcal{A}''_3$) generate test t
ok
fail
ok
fail
fail
ok
p is false on t
($\mathcal{A}''_5$) compute next ρ
($\mathcal{A}''_4$) execute p on t
no more paths in p
p is true on t
($\mathcal{A}'_5$) compute next π
($\mathcal{A}'_4$) execute f on t
no more paths in f
finish

Fig. 2. Our late-precondition method using a precondition defined in a separate C function

3 Late-Precondition Method

Usual Test Generation in PATHCRAWLER In Fig. 1 we briefly present (following [7, Sec.2.1]) the DSE-based test generation method for a C function f implemented in PATHCRAWLER. Step $\mathcal{A}_1$ creates a logical variable for each input of f and posts the constraints for the precondition of f (given in an internal format). The depth-first exploration of program paths (steps $\mathcal{A}_2$-$\mathcal{A}_5$) starts with the empty path ε. $\mathcal{A}_2$ symbolically executes the current partial path π in f and posts corresponding path constraints, solved at $\mathcal{A}_3$ in order to generate a test t activating a path starting with π. If $\mathcal{A}_2$ or $\mathcal{A}_3$ fails, i.e., π is infeasible, then $\mathcal{A}_5$ continues directly to the next partial path in a depth-first search. If a test t is found, $\mathcal{A}_4$ executes f on t and observes the complete executed path and its results. Note that some solvers (e.g., Colibri the constraint solver used in PATHCRAWLER) support incremental constraint solving. That is why, if the constraints are sent to the solver during the symbolic execution and if the solver detects the infeasibility of a path at steps $\mathcal{A}_2$-$\mathcal{A}_3$, the process skips $\mathcal{A}_4$ and goes to $\mathcal{A}_5$.

Late-Precondition Process. Let us assume given the precondition of f defined by a C function p, returning true (a non-zero value) when the inputs are admissible for f. One could suggest to filter inputs by p before exploring f. It can be done when the precondition is a conjunction of elementary conditions (it is the case in the native

N	Example	Key part of	Native		Early		Late		
		the precond.	Time	Paths	Time	Paths	Time	Paths	Precond. calls
1	Merge	$\forall i,\ t_i \leq t_{i+1}$	3m32s	8718	117m43s	73644	4m8s	8142	31415
2	TriangMatrix	$\forall i \geq j,\ M_{ij} = 0$	—		38.6s	4893	27.5s	4093	557
3	PermutOrder	$\forall i \neq j,\ p_i \neq p_j$	17.9s	5153	23s	5179	23.2s	6071	2273
4	PermutOrder	$\forall i, \exists j,\ p_j = i$	—		2m12s	14491	25s	6027	2266

Fig. 3. Late precondition compared to native and early precondition treatment in PathCrawler

precondition format of PathCrawler). The difficulty of treating any C function p is that p can have several paths that may lead to an accepting `return` statement since a C precondition may encode a complex logic formula with disjunctions. How to cover, without repetitions, every program path in f by a test executing an accepting path in p?

Fig. 2 presents our *late-precondition method.* It consists in "exploring p after f", that is, searching, for each partial path π of f, a test accepted by p **after** posting the path constraints of π at $\mathcal{A}'_2$. The steps $\mathcal{A}'_i$ explore the paths of f in the same manner as the classical DSE method, presented above, in Fig. 1, except that the test generation step $\mathcal{A}_3$ is replaced by another DSE-like exploration $\mathcal{A}''_1$-$\mathcal{A}''_5$ for the precondition p. At the steps $\mathcal{A}''_1$-$\mathcal{A}''_5$, the process keeps in the constraint store the constraints for π all the time and adds those for the current partial path ρ in p when necessary. If $\mathcal{A}''_3$ finds a test t satisfying the precondition, t also satisfies the path constraints of π and the exploration of p stops. Otherwise, the process explores all paths of p to check that no admissible test executes the partial path π of f.

This method treats a C precondition in a completely automatic way and is available online [8] (see e.g. example MergePrecond). It never considers again the same path of f. The "exploring p before f" approach cannot guarantee this property, so the same path in f may be covered several times. In addition, our technique allows us to continue to benefit from incremental constraint solving approach (where the constraints of the same partial path π are never re-posted and re-solved again), one of the main forces of the PathCrawler method.

4 Experimental Evaluation

Fig. 3 presents selected experiments of path test generation with PathCrawler for some typical examples and compares our technique with a native precondition (when it can be expressed so in PathCrawler) and with an early C precondition called before (or in the beginning of) the function under test. The indicators include total test generation time, the number of explored (covered and infeasible) paths and, for the late treatment, number of calls to the precondition function (step $\mathcal{A}''_4$ in Fig. 2). The third column illustrates the form of the *essential part* of the precondition (shown in italic below). Ex. 1 is a merge of two given *sorted arrays* t, t' into a third one. Ex. 2 checks for a given *upper triangular matrix* M if $M^2 = 0$. Ex. 3 and Ex. 4 contain the same function computing the order of a given *permutation* p, but this property is ensured in different ways: we check that $p : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ is *injective* in Ex. 3

and *surjective* in Ex. 4. For simpler examples of preconditions (over a few variables without quantifiers), no noticeable difference in performances has been recorded.

Late precondition method appears to be more expressive than native precondition and has a limited overhead. It is more convenient for C developers to write precondition directly in C, using existing code fragments and/or familiar notation. Unlike in the early precondition treatment, our late precondition technique does not explore each path in the function under test several times for each path in the precondition function, that avoids to uselessly consider again already covered paths.

It is natural to expect that this feature brings a valuable benefit for preconditions with disjunctions or existential quantifications (like in Ex. 4) since the precondition likely has multiple accepting paths for a given path in the program under test. Interestingly, even for programs with only conjunctive and universally quantified preconditions, our technique may significantly reduce the global number of paths to be explored (cf. Ex. 1). We also observe that the late precondition treatment appears to be less sensitive to the form of the precondition: as illustrated by Ex. 3–4, writing logically the same precondition in a different way can result in a significant loss of performances for the early precondition while the late precondition treatment is not affected so much. Finally, we notice that a potentially great number of calls to (an efficient executable version of) the precondition does not dramatically slow down test generation (cf. Ex. 1–4).

5 Conclusion

We propose a late-precondition method for dynamic symbolic execution that combines at least two benefits. First, it takes as input an executable precondition written in the tested language, i.e., C for PATHCRAWLER. Such a precondition is easier to write for developers and can be very expressive. Second, the method ensures that paths of the function under test are considered once and only once during test generation. This notably allows high path coverage, where each uncovered path is either infeasible or outside chosen limits (e.g., on the number of loop iterations). This article also gives an initial experimental evaluation of the method. It was made possible through the implementation of the method in PATHCRAWLER. Our experiments report a little overhead with respect to a native precondition and significantly better performances with respect to an early precondition.

Moreover, C precondition appears particularly useful when combining static and dynamic analysis, notably in the SANTE tool [9] and in treating E-ACSL, an executable specification language for C [10]. Indeed, when combining tools with very different views on the program, the program's own language is often the most suitable common language to express a precondition. That is why C precondition is used in those works to encode preconditions given in Pre/Post specifications.

Future work includes further experiments with late precondition and studying the form of precondition for which it is more or less efficient. In case of a precondition filtering out a lot of inputs, one may expect that late precondition could be more expensive than the traditional approach since it may lead to the exploration of many irrelevant program paths. Further improvements (such as a combination with a summarized early precondition, or early treatment of a conjunctive part of the precondition) could improve the performances in more cases.

References

1. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST 2009 (2009)
2. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: ICSE 2007 (2007)
3. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA 2004 (2004)
4. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/ FSE 2013 (2013)
5. Barnett, M., Fahndrich, M., de Halleux, P., Logozzo, F., Tillmann, N.: Exploiting the synergy between automated-test-generation and programming-by-contract. In: ICSE 2009 (2009)
6. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007 (2007)
7. Kosmatov, N.: All-paths test generation for programs with internal aliases. In: ISSRE 2008 (2008)
8. Kosmatov, N.: Online version of PathCrawler (2010–2013), `http://pathcrawler-online.com/`
9. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC 2012 (2012)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC 2013 (2013)

Towards a Generic Non-intrusive Fault Detection Framework

Jukka Julku and Mika Rautila

VTT Technical Research Centre of Finland,
Espoo, Finland
firstname.lastname@vtt.fi

Abstract. Temporal dependencies between programming library API operations form a protocol that can be used to automatically detect incorrect use of abstractions provided by the API. Traditionally, aliasing of abstraction instances is one of the main problems of detecting this kind of protocol violations. In this paper we describe our runtime fault detection approach that uses dynamic data-flow tracking to cope with the aliasing problem. In addition, we present a proof-of-concept fault detection framework for integrating our approach to a development environment.

Keywords: Fault detection, runtime verification, framework, testing.

1 Introduction

Practically all programs use libraries to cope with software complexity and to reduce development effort. Library APIs usually define abstractions of resources and sets of operations that can be applied to them. Often abstractions have a state that the operations may transform. Furthermore, some of the operations may expect an instance of an abstraction to be in a certain state when they are applied. Thus, operations have temporal dependencies. Missing a dependency may result in hard to detect and debug errors.

Nevertheless, verifying that all the required dependencies are satisfied is not always an easy task: use of an abstraction may be scattered not only in time, but also over several source code files, making it difficult to reason about all the possible execution sequences. Furthermore, programmers unfamiliar with APIs often misunderstand them or make false assumptions about their intended usage.

Temporal dependencies between operations form a pattern, or a protocol, that describes how an abstraction should be used. Several tools that utilize such dependencies for detecting violations of specific protocols, such as locking discipline, exist. A more generic approach, typestate analysis [1], was developed as an extension of programming language types to identify syntactically legal, but semantically undefined execution sequences. Originally typestate analyses were mainly static analyses, but because of difficulties in precise static analysis more recent approaches use runtime monitoring when static properties cannot be established [2], [3]. Despite the long history of related research, we are not aware of any of the resulted tools being widely adopted in the industry.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 334–339, 2013.

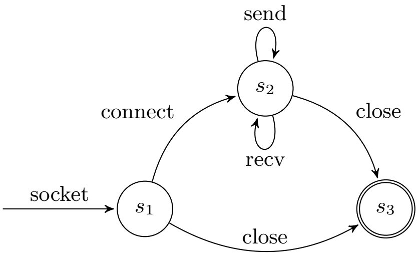


Fig. 1. Partial socket protocol

In this paper we describe our own approach for runtime detection of incorrect use of APIs for C programs. Moreover, we discuss how we think such a tool should be integrated as part of a development environment in order for it to be adopted as a useful tool. We also introduce a proof-of-concept framework that seamlessly integrates our own approach into an existing development environment.

The remainder of the paper is structured as follows. First, in Section 2 we shortly discuss our own approach to detecting protocol violations. Next, in Section 3 we introduce our proof-of-concept implementation of the approach and describe how we have integrated it as a part of a development environment. Finally, in Section 4 we discuss the possible future directions of our work.

2 Our Approach

Sockets are a good example of the abstractions discussed is the previous section. Figure 1 illustrates, in a form of a state machine, a partial protocol for using sockets: before data can be sent or received, a socket must be created and connected to an address. After a socket is not used anymore, it should be closed.

Aliasing is one of the main problems of detecting violations of such protocols. Often abstraction instance is represented by a reference to it, e.g., file descriptor, that may be freely copied. A change in one of the copies should also affect all the others. Compared to static analysis, dynamic analysis simplifies identification of aliases. However, often it is not possible to detect aliases by just comparing references, as implementations may reuse resources. For example, a file descriptor may be reused after a file has been closed. Using the newly allocated file descriptor via an old reference, while possible, is clearly semantically an error.

Our approach originates from dynamic data flow tracking that considers tagging and tracking of interesting data as it propagates through the program during execution. We solve the aliasing problem by using tags to track which memory areas actually represent the same abstraction instances. All these areas share the same tag. With each tag we associate a state and a type. The state represents the current abstract state of the abstraction instance. As an example, that a socket is bound or connected. The type defines which protocol to apply on the state, i.e., the valid state changes.

Correct use of an abstraction is modelled as a state machine. The model consists of a set of states and a set of possible transitions between the states. State transitions are triggered by computation steps that are somehow related to the modelled abstraction, e.g., the functions calls discussed in the socket example above. These computation steps are called input symbols of the model. In addition, models describe preconditions that must hold before input symbols and how each of the input symbols affects the data flow.

At runtime, we track program's data flow to propagate the memory tags. In addition, program's control flow is monitored in order to detect when the execution encounters an input symbol that should be passed to a state machine. The correct state machine instance is determined by the data on which the computation step operates. A fault occurs if an input symbol is detected separately from any state machine instance or if the corresponding state machine instance is in a state for which there is no defined state transition for that particular symbol.

Typestate analysis and our approach have clear similarities, however our approach is in some ways more generic. Typestate analysis associates types with a set of typestates that define allowed operations on each state. Each object of a given type is in one of the related typestates in each program point. Those analyses using residual runtime analysis must, similar to our approach, track object instances. Instead of types, we focus on input symbols, some of which generate tags, others of which change them. Between symbols, the data is considered simply as tagged memory areas. Even though a tagged memory area may equal to an object of a certain type, we do not limit it to do so: a tag could equally well represent objects of different types sharing some property.

Aliasing is not necessarily limited to copies of one abstraction, but instances of different abstractions may also depend on each other. For example, the C API offers two abstractions for file handling: low level file descriptors and higher level buffered streams. An instance of one abstraction can be converted into an instance of the other, and instances of both the abstractions sharing part of the same state may be present simultaneously. Clearly, state transitions, e.g., closing a file, on one instance should be propagated to the others. However, sockets, also represented as file descriptors, can too be converted into streams. Nevertheless, some socket operations are only available via the descriptor. Since not every file used via the stream API is a socket, it makes sense to model protocols for both the abstractions separately. To support modeling of dependencies between abstractions, we allow models to specify links that define effects of a state transition in an instance of one abstraction to the other, related, instances.

3 Fault Detection Framework

In order to use the approach discussed so far, an abstraction must first be modelled and the program must be instrumented to support data flow tracking. Then the program is invoked with a test input and the execution path is automatically checked for faults according to the modelled protocol. Finally, the results are inspected by a tester.

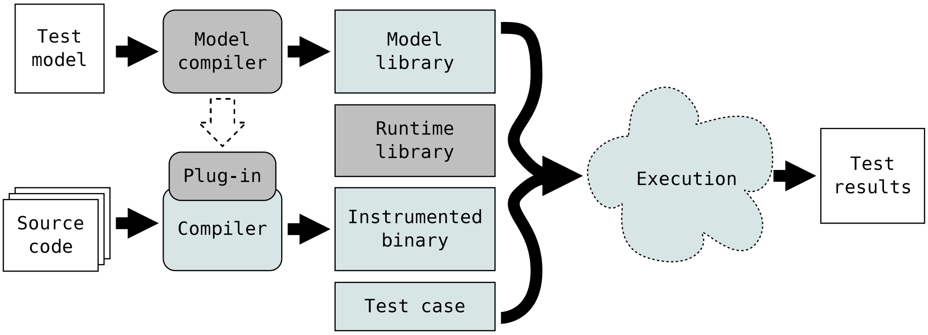


Fig. 2. Fault detection framework

We have built a proof-of-concept framework to support this work-flow. The framework consist of the following three main components that are illustrated in Figure 2. A model compiler is used to transform models into runtime libraries containing description of the model. A compiler plug-in instruments the program under test. At runtime, besides the model specific library, another library is loaded into the program. This library contains generic functionality for dynamic data flow tracking, storing state information, and reporting errors.

We claim that from the software engineering perspective for a tool to be widely adopted it should be as automated as possible in order to reduce the amount of costly human work. Because of this, we find reusability of models, minimum user interaction, and accurate error reporting valuable.

Models can represent library or application specific protocols. Ideally, in order to obtain as accurate models as possible, models should be specified by the authors of an API and distributed with the programming library when possible. However, this clearly won't often be the case. Therefore, models for common APIs, e.g., the socket interface, could also be provided by the fault detection framework developer community. Nevertheless, developers using the framework may sometimes need to specify models themselves because there is no model available for the API they are using, they need application specific models, or wish to check some functionality that has not been modelled at all. Thus, the modelling activity itself should be easy and straightforward. In addition, in order to allow developers to easily specify program specific protocol requirements or behavior, models should be extensible.

The models should be separated from the program's source code for two main reasons. First, models should be reusable across programs, i.e., no changes to a model should be required when testing two programs that use exactly the same API. Second, to allow a tool to be easily taken into use in existing projects, the tool should be non-intrusive from the source code point-of-view, i.e., no source code level changes should be required in order to use the tool.

According to our experience, for a tools to be adopted into use, in addition to the benefits for testing, easy and seamless integration into existing development environments is essential. As we use compiler instrumentation, the compiler is a natural integration point. We use the GCC compiler, also widely used in the industry. Our instrumentation is generated by a plug-in, which can be loaded into a standard compiler. We have successfully integrated our plug-in into the build system of a production environment compiling about million lines of C code by just adding a simple compiler wrapper.

Our current implementation uses extensive program instrumentation in order to track memory tags and do state checks at runtime. During compilation, large amount of detailed program information is saved into the binary and generic instrumentation is added to stop the execution at certain points. At the beginning of the program execution, the saved information and the model description are used to precompute the effect of each basic block to the data flow. This information is then used to interpret changes to the memory tags during execution.

The taken approach avoids the need to recompile for each model, but causes high performance penalty. It can be argued that dynamic binary instrumentation could lead to better overall performance and would also allow monitoring of third party libraries. However, we decided to use compile-time instrumentation as we believe that the precision of the analysis and error reporting can benefit from the rich information extracted from the compiler's intermediate representation.

Many optimizations must be made to our implementation in order to gain acceptable performance for industry use. We plan to add, e.g, static analyses to our framework in order to reduce the runtime overhead. Nevertheless, it is obvious that in general plenty of extra computation is required. However, widely used dynamic analysis tools, such as Valgrind [4], have shown that if the offered benefits are valuable enough, e.g., the tool finds more errors or makes debugging easier, even relatively high overhead can be acceptable. Also, often execution time of a single test case is not what matters, but the total execution time of the test suite. Thus, high overhead can be compensated with other solutions such as test case selection techniques [5] that minimize the amount of tests that need to be run at all.

The proposed approach allows protocol violations to be detected immediately when they happen. However, a protocol violation itself might merely be a symptom of the real bug. Thus, it is important to aid the developer to locate the actual root cause. The rich control-flow information saved into the program binary allows techniques, such as dynamic program slicing, to be incorporated into the framework in order to support debugging.

4 Conclusions and Future Work

In this paper, we proposed an approach for runtime detection of incorrect use of APIs. We further argued that in order for an implementation of such an approach to be adopted by developers, it must be easy to use and seamlessly integrated into existing development environments. Moreover, we presented a framework that integrates our implementation to the widely used GCC compiler.

The work presented in this paper is but just a first step. The most obvious future direction to our research are performance optimizations. Especially static analysis could be used to reduce the runtime overhead of the data flow tracking. Another interesting path to consider is how the runtime analysis could be efficiently implemented in presence of concurrency. Lastly, modelling of protocols opens other interesting questions, including how models, especially input symbols, should be specified, and how they are mapped to the program execution.

Acknowledgements. The research leading to these results was partly conducted in the European ITEA2 project DIAMONDS which was partially funded by TEKES – the Finnish Funding Agency for Technology and Innovation.

References

1. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986)
2. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 124–133. ACM (2007)
3. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE, vol. 1, pp. 5–14. ACM (2010)
4. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 89–100. ACM (2007)
5. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test., Verif. Reliab. 22(2), 67–120 (2012)

A Witnessing Compiler: A Proof of Concept

Kedar S. Namjoshi[1], Giacomo Tagliabue[2], and Lenore D. Zuck[2]

[1] Bell Laboratories, Alcatel-Lucent
kedar@research.bell-labs.com
[2] University of Illinois at Chicago
giacomo.tag@gmail.com, lenore@cs.uic.edu

Abstract. In prior work we proposed a mechanism of "witness generation and propagation" to construct proofs of the correctness of program transformations. Here we present a simpler theory, and describe our experience with an initial implementation based on the LLVM open-source compiler and the Z3 SMT solver.

1 Introduction

Ensuring the correctness of an optimizing compiler is a classic question in computing. Compilers are very large programs – for instance, GCC is over 7 million lines of code, and LLVM is near a million – and they carry out the essential task of transforming other programs (often themselves large) into executable machine code. Ensuring the correctness of compiler transformations is thus a critical question; however, manual inspection is impossible, which has led to much work on the construction of automated proofs of correctness.

In [7] we proposed a methodology for creating such a proof. There, each optimization procedure in the compiler is augmented with an auxiliary *witness generator*. For every instance of optimization, the generator constructs, *at run time*, a *witness relation* between its target and source programs. The conditions for a relation to be a proper witness are checked off-line, using an automated theorem prover. Thus, when a witnessing compiler is used on a source program, it generates a chain of witnesses, one for each optimization, which connects the source program with the final target program. Each link of the chain may be verified *independently* of the compiler code.

Witness generation can be positioned in-between two well known methods of compiler verification: machine-checked proofs of correctness (e.g., [5]) and Translation Validation (TV) (e.g., [10]). It is substantially simpler to define a witness generation procedure than to prove an optimization correct, as the definition does not require one to show (or assume) the correctness of the analysis phase of an optimization. Moreover, as the generating procedure is written with full knowledge of the optimization, one avoids the heuristic constructions which limit the scope of translation validation. The potential drawback to witness generation is the run-time overhead of generation and checking.

In this paper, we report on early experiments with witness generation. The implementation is carried out using the LLVM compiler framework [4]. It currently

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 340–345, 2013.

supports a limited set of instructions (enough to represent **while** programs over the integers) and a small set of transformations (simple constant propagation, dead-code-elimination, loop invariant code motion). The generated witnesses are checked for validity with the Z3 SMT solver [3]. Our experience has been encouraging: the witness generation approach is feasible and requires only small amounts of additional code. The overhead of witness checking is high, but we expect this to reduce with better implementation techniques.

2 Transformations and Witnesses

This section summarizes ideas described in more depth in [7].

Definition 1 (Program). *A program is described as a tuple* $(V, \Theta, \mathcal{T})$*, where*

- V *is a finite set of (typed)* state variables*, including a distinguished* program location variable*,* π*,*
- Θ *is an* initial condition *characterizing the initial states of the program,*
- $\mathcal{T}$ *is a* transition relation*, relating a state to its possible successors.*

A program *state* is a type-consistent interpretation of its variables. The transition relation is denoted syntactically as a predicate on V and V', which is a primed copy of V. For every variable x in V, its primed version x' refers to the value of x in the successor state.

To match the LLVM structure, we consider programs described by a control flow graph (CFG), where each node is a *basic block* (BB) consisting of a single-entry single-exit straight line code. The transition relation of the program can thus be viewed as a disjunction of transition relations $\mathcal{T}_{ij}$, each describing the transition between basic block i (BB_i) and basic block j (BB_j) such that BB_j is an immediate successor of BB_i. The program location variable π ranges over the set of basic block identifiers. We assume that a CFG has a unique *initial* BB with no incoming edges, and a unique *terminal* BB without outgoing edges.

A *witness* relation connects the values of source and target program locations at corresponding basic blocks. In the simplified view, we define a witness relation to have two components:

- A *control mapping* κ from the basic blocks of T to those of S. The function κ maps the initial block of T to the initial block of S, and the terminal block of T to that of S.
- A *data relation*, $\varphi_{i,\kappa(i)}(V_T, V_S)$, which connects the values of target and source variables at corresponding blocks i and $\kappa(i)$. For this paper, it suffices to have relations which are defined as conjunctions of the form $v = e$ where v is a program variable and e is an expression, over variables of either S or T. For instance, one can define equality of corresponding source and target variables by a set of conjunctions of this form.

There are three conditions, shown in Figure 1, which are checked to ensure that a relation is a proper witness (i.e., it ensures the correctness of the transformation). The first checks that the witness relation is a (stuttering) simulation;

the second, that source and target variables match at initial and final blocks. The stuttering simulation check allows infinite stuttering on the source program side; this can be fixed, as described in [7], by generating an auxiliary ranking function. As our current implementation does not do that, we omit it from the rule. The predicate $oeq(V_T, V_S)$ (read as "observably equal") asserts that corresponding target and source variables are equal in value. The correspondence is specific to the optimization. (Hence, a witness is correct up to a correspondence.)

1. For every target block i, the following implication must be valid. $[\varphi_{i,\kappa(i)}(V_T, V_S) \wedge \mathcal{T}^T_{ij}(V_T, V'_T) \;\Rightarrow\; (\exists V'_S : (\mathcal{T}^S_{\kappa(i),\kappa(j)}(V_S, V'_S) \wedge \varphi_{j,\kappa(j)}(V'_T, V'_S))) \vee \varphi_{j,\kappa(i)}(V'_T, V_S)]$
2. For the initial block a, $[(\exists V_S : \varphi_{a,\kappa(a)}(V_T, V_S) \wedge \mathit{oeq}(V_T, V_S))]$ must be a validity.
3. For the final block f, $[\varphi_{f,\kappa(f)}(V_T, V_S) \;\Rightarrow\; \mathit{oeq}(V_T, V_S)]$ must be a validity.

Fig. 1. Witness Checking

Typically, a witness relation encodes invariants about the source and target programs, which are inferred during the analysis phase of an optimization. For instance, constant propagation generates assertions about which variables of the source program are constant, and dead-code elimination depends on a liveness analysis that generates assertions about the live variables at each program point. The witness relation for constant propagation, for example (see [7]), states that $(x_T = x_S)$ for every variable x and that $(x_S = c)$ for those variables x which are known to have constant value c at the source location $\kappa(i)$.

3 Implementation

The source code of the implementation is a fork from LLVM, and is available as a git repository at `https://bitbucket.org/itajaja/llvm-csfv`. Currently, the implementation targets the intra-procedural optimization passes in LLVM, defined over its intermediate representation (IR). Programs in the IR are in SSA (single static assignment) form for each function.

The process that is followed to build a witnessing pass is similar for every pass. The starting base is the LLVM source code for an optimization pass. First, the analysis phase of the pass is augmented – if needed – to store all the invariants found by the analysis for each program location (or basic block). These invariants are used for the witness generation. To validate a witness, it is necessary to build the transition relations for the source and target programs. The validation checks implement the proof rule in Fig. 1 using the Z3 SMT solver. As basic blocks are (guarded) deterministic code fragments, the existential quantification in the simulation check can be eliminated, which simplifies the check.

Table 1. Measurements

Pass	Original LOC	Witness Gen. LOC	Avg. runtime in ms (overhead multiple)
Simple Constant Propagation	99	118	101.36 (12x)
Dead Code Elimination	135	37	41.71 (10x)
Loop Invariant Code Motion	895	65	200.03 (31x)

The framework design is based on the following main components: Optimizer/Analyzer, Witness Generator, Translator, Witness Checker, Invariant Propagator. The *Optimizer/Analyzer* augments the LLVM pass to store the analysis invariants; the *Witness Generator* takes care of generating the optimization-specific witness using the invariants found during the analysis; the *Translator* builds the transition relation of a given CFG and is usually run over the target and the source of every optimization pass; and the *Witness Checker* combines the generated witness and the target and source transition relation to verify that the witness is a stuttering simulation using Z3. In addition, an *Invariant Propagator* uses the witness relation and symbolic manipulations using Z3 to propagate invariants (computed during analysis or externally supplied) from a source program to the target. Out of these five components only the first two are optimization-specific.

Table 1 gives measurements which show (a) the effort required to write a witness generator and (b) the overhead incurred to check the correctness of witnesses. The implemented passes are chosen by their commonality, ease of study and for clearly highlighting some of the critical parts of the framework. The lines of code (LOC) for witness generation are those that are required specifically for that optimization. In addition, there is code which is common to all passes, and implements a witness checker, the invariant propagator, the translator, and basic definitions, amounting in total to approximately 1 KLOC.

The LOC numbers are encouraging: compared to the effort required to define the optimization, the effort required to define a witness generator is high only for the simple constant propagation pass, but is much lower for the other two passes. The run-time overhead measures the overhead of witness generation and checking compared to the optimization time, measured with the `time-passes` tool of the LLVM optimizer. The current runtime overhead for witness checking is very high. However, this is a rough, unoptimized implementation, so we expect this overhead to reduce substantially as the implementation is improved.

4 Conclusion and Related Work

The implementation described here is a work in progress, and is currently at an early stage. Support for the instruction set of the IR is limited to binary operations over integers, return, branch (conditional and unconditional), compare, and ϕ nodes. (This set suffices to describe **while** programs over the integers.) For this reason, it is not possible yet to test the framework against "real" programs that contain many currently unsupported instructions and data types.

Ensuring the correctness of program transformations – in particular, compiler optimizations – is a long-standing research problem. In [6], Leroy gives a nice technical and historical view of approaches to this question. A primary approach is to formally prove each transformation correct, over all legal input programs. This is done, for example, in the CompCert project [5], and in [2], which derives and proves correct optimizations using denotational semantics and a relational version of Hoare's logic Formal verification of a full-fledged optimizing compiler is often infeasible, due to its size, evolution over time, and, possibly, proprietary considerations. *Translation Validation* offers an alternative to full verification. A primary assumption of this approach is that the validator has limited knowledge of the transformation process. Hence, a variety of methods for translation validation arise (cf. [9,8,11,13,14,12]), each making choices between the flexibility of the program syntax and the set of possible optimizations that are handled. As details of the optimization are assumed to be unknown, heuristics are used, which naturally limits the scope of the method. Recently, [1] proposes a method for proving equivalence based on relational Hoare logic; it resembles our witnesses, yet is closer to translation validation and has similar limitations.

Since we assume the optimization process is visible to the witness generator, the generator is able to make use of auxiliary invariants derived by the optimizer in order to produce a witness. This implies that witness generation is, in principle, applicable to any optimization.

Acknowledgements. This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013)
2. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL, pp. 14–25 (2004)
3. 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)
4. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88 (2004), `llvm.org`
5. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM (2006)

6. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
7. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013)
8. Necula, G.: Translation validation of an optimizing compiler. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation, PLDI 2000, pp. 83–95 (2000)
9. Pnueli, A., Siegel, M., Shtrichman, O.: The code validation tool (CVT)- automatic verification of a compilation process. Software Tools for Technology Transfer 2(2), 192–201 (1998)
10. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
11. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proceedings of the Run-Time Result Verification Workshop (July 2000)
12. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI, pp. 295–305 (2011)
13. Zuck, L.D., Pnueli, A., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)
14. Zuck, L.D., Pnueli, A., Goldberg, B., Barrett, C.W., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design 27(3), 335–360 (2005)

Runtime Monitoring of Temporal Logic Properties in a Platform Game

Simon Varvaressos, Dominic Vaillancourt, Sébastien Gaboury,
Alexandre Blondin Massé, and Sylvain Hallé*

Laboratoire d'informatique formelle
Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
shalle@acm.org

Abstract. We report on the use of runtime monitoring to automatically discover gameplay bugs in the execution of video games. In this context, the expected behaviour of game objects is expressed as a set of temporal logic formulæ on sequences of game events. Initial empirical results indicate that, in time, the use of a runtime monitor may greatly speed up the testing phase of a video game under development, by automating the detection of bugs when the game is being played.

1 Introduction

The domain of video games is currently booming; a recent Gartner survey revealed that consumer expenses for video games would raise from 67 billion dollars in 2011 to more than 112 billion by the year 2015 [2]. Similar to all computer systems, video games have not been spared from programming errors making their way to the release of a product. For example, in *Halo Reach* (2010), it is possible for players to go out of the game's map in some places, allowing them to make actions that would otherwise be forbidden [1].

It is therefore important for a designer to detect a maximum of gameplay errors as soon as possible during the development phase of a game, since for some systems, correcting an error using an update after the product's release is technically impossible. Moreover, video games are a special type of *emergent* system: their complexity arises from the combination of multiple simpler parts like the physics engine, the graphics or the graphical user interface. A minor problem can bring a bigger one later in the execution. Therefore, to facilitate debugging, it is important to identify exactly when a bug occurs and report it as fast as possible.

Typically, video game companies hire manual testers, whose hourly salary varies from $20 to $100, with the special purpose of discovering gameplay bugs and manually filing them into a bug tracker database. Obviously, this technique

* With financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de recherche Québec – Nature et technologies (FRQNT).

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 346–351, 2013.

is time-consuming and far from fail-proof: in some cases, gameplay bugs are not immediately apparent to the human eye. In this setting, the use of runtime verification techniques presents the potential for improving the gameplay bug harvesting step. However, video games rely a lot on fast player inputs and are much more sensitive to speed and timing than traditional software; it is therefore important that the use of a monitor does not slow down the game in any noticeable way. This paper presents early results on this approach and illustrates how gameplay bugs of a popular platform game, *Infinite Super Mario Bros.*, can be specified as temporal logic formulæ and efficiently caught at runtime using an off-the-shelf monitor.

2 Gameplay as Temporal Logic Constraints

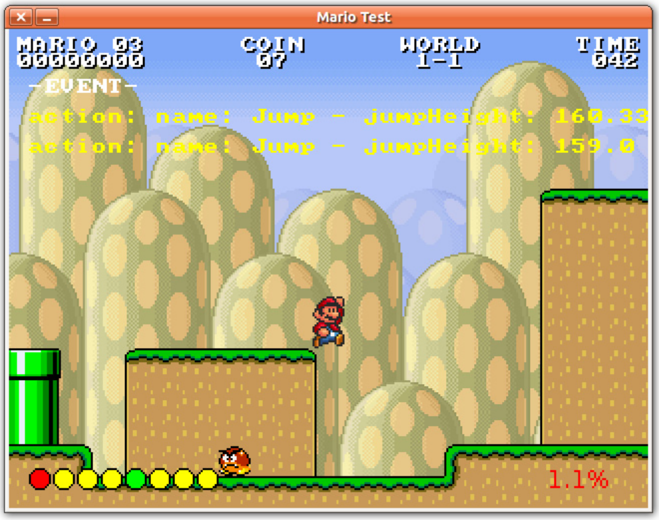
As opposed to most standard software, video games are not entirely driven by the user. Most games include a physics engine and a form of artificial intelligence to update the game environment even in the absence of any input from the player. Moreover, these updates must be executed a minimum of 30 times (called *frames*) per second, with 60 frames per second (fps) being a reasonable target for quality animation. Noticeable disruptions of the frame rate are regarded by players as bugs and have in the past caused the demise of some video game titles. This concept is best exemplified by a well-known game called *Infinite Mario Bros.* (Figure 1), an open-source reimplementation of the popular platform game *Super Mario World*, where various enemies and other game objects move around the game area independently of the player's (i.e. Mario's) actions.

Infinite Mario is made of 6,500 lines of Java code and is available online.[1] It is notable for being the subject of many research works on game testing and applications of Artificial Intelligence algorithms in the past [5]. It has recently been used as a testbed for the automated application of condition-action rules aimed at correcting erroneous game states [7]. A similar approach has been applied to *FreeCol*, a free version of the strategy game *Civilization* [4].

In the following, we push the concept further and attempt to formalize the expected behaviour of various game objects as temporal logic constraints. In this context, events represent various changes of state, both of the player's character and of the surrounding enemies and objects. Each event is represented as a list of parameter-value pairs, and has a parameter called `name`, indicating the type of the event (e.g. Jump, Stomp, EnemyDead, etc.). The number and name of the remaining parameters may differ depending on the event's type. For example, when Mario stomps on an enemy, the unique ID of that enemy will be included in the event; when Mario jumps, the height of the jump will be recorded.

The rules used to express the properties to monitor are represented with LTL-FO$^+$, an extension of Linear Temporal Logic (LTL). For example, the following expression indicates that globally, if an enemy gets hit by a fireball that Mario

[1] `http://mojang.com/notch/mario/`



action: name: Jump - jumpHeight: 160.33
action: name: Jump - jumpHeight: 159.0

Fig. 1. The GUI of the modified version of Infinite Mario Bros

threw (event *name* EnemyFireballDeath), the next event should indicate the disappearance of the fireball so that it does not hit anything else.

$$\mathbf{G}\ (\text{name} = \text{EnemyFireballDeath} \rightarrow \mathbf{X}\ \text{name} = \text{FireballDisappear})$$

The presence of first-order quantifiers is necessary for two reasons: the same parameter may occur multiple times in the same event (such as when multiple enemy IDs are killed by Mario at the same time), and some gameplay properties may affect a single element across multiple events, as in the following expression.

$$\mathbf{G}\ ((\text{name} = \text{Stomp} \wedge \text{isWinged} = \text{true}) \rightarrow \\ \forall x \in \text{id} : \mathbf{X}\ (\text{id} = x \rightarrow \text{name} \neq \text{EnemyDead}))$$

In a normal playthrough, if Mario jumps on a flying enemy, it should lose its wings. In this formula, we make sure that a winged enemy cannot die after Mario stomps on it, as it should only stop flying. Since we are keeping the corresponding `id` in a variable named x, we can check the next event related to the *same* enemy to make sure it's respecting the normal flow of the game. However, one can see that, for this correlation between object IDs to be possible, first-order quantification on event parameters is necessary. As a matter of fact, we discovered early on that *propositional* Linear Temporal Logic is not expressive enough to represent but the simplest gameplay properties.

3 Empirical Results

Once a number of game properties were formalized as LTL-FO$^+$ formulæ, we devised an experimental setup to assess the performance of our runtime monitoring approach in actual runs of the game.[2] As we have seen, any errors caught by a monitor should be identified before the next frame, yielding an upper bound of 17 to 33 ms for the processing of each batch of events. Any processing time slower than this would either slow down the game and cause jerky animation, or have the monitor increasingly lag on the current game state and fill some event buffer.

The BeepBeep runtime monitor[3] [3] was selected to be inserted into the game, since it was developed in Java and uses LTL-FO$^+$ formulæ as its input language. The BeepBeep monitor accepts events in the form of XML strings. Some strings are constant, while others like this one are dynamically created based on the specific parameters of the event (enemy IDs, etc.). For example, the following shows the instrumentation to generate an event indicating that some enemy died:

```
MonitorTimer.Instance().updateWatchers("
 <action>
  <name>
   EnemyDead
  </name>
  <id>
   "+id+"
  </id>
 </action>
");
```

The game's code was manually instrumented to produce these events; about 30 locations in the code had instruction of this kind inserted. We could have chosen AspectJ [6] to facilitate the instrumentation but we decided not to because this solution is Java-specific, and most games use languages like C++ or even unique ones like UnrealScript. Relying on AspectJ would not faithfully represent the restrictions one shall face when monitoring video games in general.

To keep track of the different outcomes for each property, we also added some elements to the game's GUI. First, circles of colour, each representing a property, can be found on the lower left part of the screen. A green dot indicates a property evaluates to true on the sequence of events received so far, while red indicates it evaluates to false. Since each monitor is constantly queried on a finite trace prefix, the value of some properties may not be defined yet; this is indicated by a yellow dot. For debugging purposes, we also print the last two events produced at the top of the window. The lower-right corner displays in real time the overhead incurred by the presence of the monitors.

Finally, in order to make sure that our monitor can actually intercept gameplay bugs, we manually performed modifications to the game's code to create

[2] The instrumented version of Inifinite Mario and the runtime monitor can be downloaded from `http://github.com/sylvainhalle/BeepKitu`

[3] `http://beepbeep.sourceforge.net`

specific problems, such as removing instructions that handle the killing of some enemies. We then performed numerous runs of the game and computed various metrics on the game's and the monitor's execution.

The results were positive. Every formula we used could be monitored using our method, without slowing down the game in any noticeable way. A surprising finding of our study is that in a normal playthrough, the game generated roughly 2.9 events per second (Figure 2a). This event rate is very small compared to rates in typical runtime verification works. One can see that an event could take 9 milliseconds to process for 10 different properties (Figure 2b). It is possible to see a drop in the time required as the game progress, since monitors for properties that evaluate to true or false no longer need any updating. Even if, in a worst case scenario, no properties were resolved and each event took 9 milliseconds to handle, we can safely assume that it would not affect gameplay since 27 spare milliseconds are left before reaching the threshold time for 30 fps.

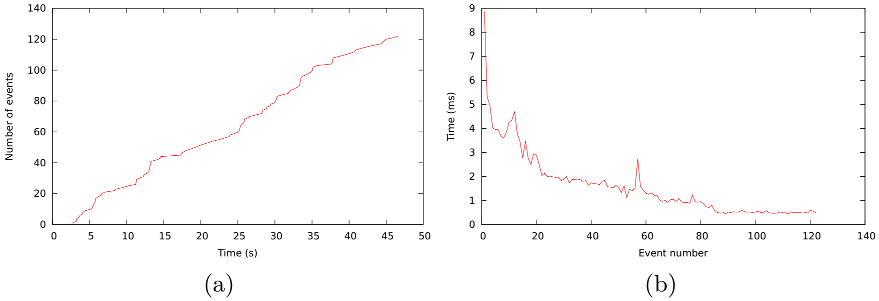


Fig. 2. Experimental results on the monitoring of Infinite Mario Bros. a) Number of events generated in a sample run of the game; b) Monitor processing time for each event.

Finally, since video game companies keep track of their bugs in a database, we implemented a similar functionality using a MySQL server. We instrumented the code in such a way that, upon violation of some temporal property, the monitor sends an SQL query to the server, thereby automatically filing various metadata about the discovered violation: name of property violated, occurrence time, event trace prefix leading to the violation.

4 Conclusion

Overall, the results we obtained were conclusive as a first step in the application of runtime monitoring to video games. We succeeded in the monitoring of different properties using LTL-FO$^+$ in a video game without affecting the game experience. We also provided the game with a GUI that easily shows the outcomes for each monitored property. If one of them becomes violated, indicating a problem in the expected gameplay, the monitor automatically saves information about the bug in a database, something that could help in video game development.

Some improvements to the method could be implemented. For example, manual instrumentation of the game is tedious and error prone; it is believed that one could make good use of the *game loop* present in every video game to simplify its instrumentation: instead of manually finding and inserting the events to monitor, one could keep track of the game objects' state by interpreting the differences from one game loop iteration to the next. Moreover, compiling the monitor within the game does not seem a desirable choice for a larger-scale application of monitoring, as one would have to change the monitor to fit every game's implementation language. Finally, the game's graphical API made it hard to integrate monitor controls within its own GUI and limited the amount of information that could be input from (or displayed to) the user. As future work, we are currently working on a much larger open source game, drawing from the lessons learnt when monitoring Infinite Mario Bros.

References

1. Worst videogame bugs of all time: From game-ending glitches to data-destroying nightmares, `http://bit.ly/GLySq`
2. Biscotti, F., Blau, B., Lovelock, J.-D., Nguyen, T.H., Erensen, J., Verma, S., Liu, V.K.: Market trends: Gaming ecosystem. Technical report, Gartner Group. Report G00212724 (2011)
3. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012)
4. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-based runtime monitoring of JVM hosted applications. ECEASST 44 (2011)
5. Karakovskiy, S., Togelius, J.: The Mario AI benchmark and competitions. IEEE Trans. Comput. Intellig. and AI in Games 4(1), 55–67 (2012)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Commun. ACM 44(10), 59–65 (2001)
7. Lewis, C., Whitehead, J.: Repairing games at runtime or, how we learned to stop worrying and love emergence. IEEE Software 28(5), 53–59 (2011)

SMock — A Test Platform for Monitoring Tools*

Christian Colombo, Ruth Mizzi, and Gordon J. Pace

Department of Computer Science, University of Malta
{christian.colombo,rmiz0015,gordon.pace}@um.edu.mt

Abstract. In the absence of a test framework for runtime verification tools, the evaluation and testing of such tools is an onerous task. In this paper we present the tool SMock; an easily and highly configurable mock system based on a domain-specific language providing profiling reports and enabling behaviour replayability, and specifically built to support the testing and evaluation of runtime verification tools.

1 Introduction

Two of the major challenges in runtime verification, which are crucial for its adoption in industry, are those of the management of overheads induced through the monitoring and the ensuring the correctness of the reported results. State-of-the-art runtime verification tools such as Java-MOP [7] and tracematches [1] have been tested on the DaCapo benchmark[1], but the kind of properties in these case studies were typically rather low level, contrasting with our experience with industrial partners who are more interested in checking business logic properties (see e.g., [4,3]). Whilst we had the chance to test our tool Larva [5] on industrial case studies, such case studies are usually available for small periods of time and in limited ways due to confidentiality concerns. Relying solely on such case studies can be detrimental for the development of new tools which need substantial testing and analysis before being of any use.

To address this lack, we have built a configurable framework which may be used to mock transaction[2] systems under different loads and usage patterns. The key feature of the framework is the ease with which one can define different types of transactions and farm out concurrent instances of such transactions through user-specified distributions. Profiling the overheads induced by different runtime verification tools, thus, enables easier benchmarking and testing for correctness of different tools and techniques under different environment conditions. Although not a replacement of industrial case studies, this enables better scientific evaluation of runtime verification systems.

SMock allows straightforward scripting of case studies, giving control over transaction behaviour, timing issues, load buildup, usage patterns, etc., which can be used to benchmark the correctness and performance of different runtime verification tools. A major issue SMock attempts to address is that of repeatability of the experiments,

* The research work disclosed in this publication has been partially funded by the University of Malta Research Fund 2012, grant number CPSRP04-02.

[1] http://www.dacapobench.org/

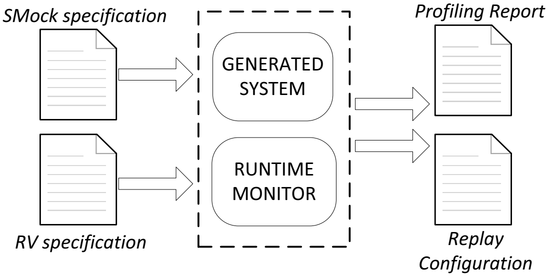
[2] We use *transaction* to refer to an independent, highly replicated unit of processing.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 352–357, 2013.

ensuring the scientific validity of the conclusions. To evaluate the use of the tool, we have used it to build a suite of benchmarks and compared the performance of JavaMop [7] and polyLarva [2] under different mocked scenarios.

2 The Architecture and Design of SMock

SMock has been built in such a manner so as to enable the easy generation of families of transaction systems with two levels of configurability: (*i*) the form of the underlying transactions (possibly including variations); and (*ii*) the scenarios depicting transaction firing as distributed over time. To enable us to achieve this goal, SMock takes a script describing these options and generates a Java mock system behaving in the prescribed manner, generating events which can be captured by monitoring tools. Such a system is then typically monitored using a specification script in a runtime monitoring tool, enabling the measurement of overheads induced by the monitors in such a setting. Since the mock system may include parts which are randomly generated (e.g., launch between 100 and 120 transactions in the first 10 seconds), its execution also tracks information to allow exact replayability. This enables, for instance, comparison of the performance of the different versions of a runtime verification tool, or its performance against that of other tools, or as a regression test following a bug fix. The general usage pattern of SMock is shown below:



3 Scripting Behaviours

One of the strengths of SMock is that it allows users to script scenarios to be used for testing. The scripts are written in a small domain-specific language which provides a number of constructors to enable the description of processes and how they are to be launched. Basic processes are individual actions which are characterised by three parameters — their duration, memory consumption and CPU usage. These parameters can be exact values or probability distributions[3], to enable the mock system to behave within a range of possible behaviours. Actions can be named if one would want to be able to monitor their moment of entry and exit, but can also be left unnamed.

Processes can be combined using the following operators: (*i*) sequential composition; (*ii*) probability weighted choice between processes; and (*iii*) parallel composition of processes. The combinators come in a binary form and also generalised versions which allow the combination of multiple processes together.

[3] The tool currently supports uniform and normal distributions.

Example 1. Consider a document management system which allows users to login, browse, upload documents, edit these documents, etc. The resources used by some of these actions can be characterised as follows:

$$\begin{aligned}
\text{login} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{uniform}(3,5), \text{memory} = \text{normal}(20,40), \text{cpu} = 0.1\rangle\\
\text{browse} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{normal}(5,8), \text{memory} = 300, \text{cpu} = \text{normal}(0.5,0.7)\rangle\\
\text{logout} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{normal}(1,3), \text{memory} = \text{normal}(20,40), \text{cpu} = 0.1\rangle
\end{aligned}$$

One can now generate 300 users acting in one of two possible ways — browsing or editing a number of files:

$$\begin{aligned}
\text{usertype1} &\overset{\text{df}}{=} \text{login};\ \text{browse};\ \text{logout}\\
\text{usertype2} &\overset{\text{df}}{=} \text{login};\\
&\quad\ \text{seq foreach document} \in \{1\ldots 3\}\ \text{do}\\
&\quad\quad\quad \text{open};\ \text{edit};\ \text{save};\ \text{edit};\ \text{close};\\
&\quad\ \text{logout}\\
\text{system} &\overset{\text{df}}{=} \text{par foreach user} \in \{1\ldots 300\}\ \text{do}\\
&\quad\quad\quad \text{choice}\begin{cases} 0.9 \mapsto \mathit{usertype}1\\ 0.1 \mapsto \mathit{usertype}2\end{cases}
\end{aligned}$$

In practice, for a more realistic scenario, we would not want the user transactions to be launched all together at a single point in time, so we would add an (unnamed) action preceding each user transaction, which takes some time to terminate, but does not consume CPU or memory resources. □

As seen in the above simple example, when writing a script, one would usually want to be able to define and reuse transactions, requiring further (non-functional) constructs in the language. Similarly, one may want to add compile-time computations which calculate constants to be used in the rest of the script (e.g., memory usage of a class of actions could be automatically calculated as a function of CPU usage and duration). To avoid having to extend the language with such constructs, we have chosen to build the scripting language as an embedded language [6] in Python. This allowed us to avoid having to build a parser and type-checker for the language, and also allows the user to use Python for function definitions and computation.

4 An Application of SMock

SMock supports the generation of mock systems written in the Java language. As a case study SMock has been used to generate a mock document management system — an extension of the example given in Section 3.

In this case study, the generated system was used in conjunction with existing runtime verification tools in order to analyse the effects these tools have on the overall performance of concurrent systems, of which the document management system is an example. Two runtime verification tools, JavaMop [7] and polyLarva [2] were used in this case study. In both instances a sample property: *an edited document must always be saved before closing*, has been monitored.

The programs in Program 4.1 for JavaMop and polyLarva, show simplified excerpts of the specification scripts used to generate runtime monitors for JavaMop and polyLarva. The most relevant feature of these scripts is the definition of noteworthy system events, such as ① and ②, which one must necessarily hook onto, to monitor the required property. Our previous explanation of the SMock specification language has highlighted how these system events are created through the definition of named actions. Both RV tools use AspectJ[4] technology to convert the event specifications to aspect code that can identify the occurrence of these events on the system. The resultant code is woven into the mock system's code to provide runtime monitoring of its execution.

Program 4.1 JavaMop and polyLarva specifications

```
JavaMOP:
SavedDoc(Document d) {
① event open after(Document d):call(* Document.open(..)) && target(d) {}
   ...
   ere: (open save* edit edit* close)  }

polyLarva:
upon { newDocument(doc) } {
  events {
      ② open(doc) = {doc.open();}
     ...  }
  rules { ... }   }
```

The aim of this case study was not to carry out comparison between the runtime monitoring tools; but rather its purpose is that of highlighting the type of analysis that can be carried out using a mock system generated by SMock. In particular we wanted to note the effect of runtime monitoring on the system's performance when it is under considerable load. Changes to the specification affecting the choice settings result in different executions which model differing loads. We generated systems where only 10% of the users are carrying out document editing tasks and then increased this to 50% and 100% for the following executions. Since the property triggers only when a document is manipulated, this affects the overheads induced by the runtime monitors.

The analysis uses the profiling information from the runs of the generated document management system. For each configuration, the mock system was executed multiple times in the following setups: (*i*) without any code instrumentation; (*ii*) using JavaMop; and (*iii*) using polyLarva. Replaying was used to ensure comparison between identical executions.

The table below compares performance of the system execution and demonstrates how profiling data extracted by SMock can give a good indication of the effect that differing system loads can have on the overall system performance and execution time.

[4] http://www.eclipse.org/aspectj/

For each different load of users carrying out document management activities, the table shows the average memory usage and CPU usage across the whole execution together with an indication of system duration.

% users monitored	Average Mem (Kb)	Average % CPU	Total Sys Time (mins)		
			None	MOP	poly
10	13.8	31.3	1.6	1.7	1.7
50	24.3	32.9	3.2	3.3	3.3
100	27.2	33.5	3.4	3.5	5.7

The monitoring overheads are non-trivial in these scenarios and the analysis allows an understanding of how monitoring affects the overall system execution.

5 Related Work

While industry-calibre tools (e.g., jMock[5]) exist for mocking parts of a system under test, to the best of our knowledge, no tools exist which enable one to mock a whole system. Another significant difference of SMock from existing mocking tools is that these do not explicitly support the specification of CPU and memory usage. SMock, not only provides this through a dedicated language, but also provides constructs for the specification of probability distributions over such resources. These differences make SMock ideal to test systems which act upon other systems, e.g., monitoring systems and testing systems.

Another area of somewhat related work is the area of traffic generators (e.g., Apache JMeter[6]) for performance testing. However, such tools assume the existence of a system on which traffic is generated. Since the load on a runtime verification tool occurs *by proxy*, i.e. as a consequence of the load of another system, performance testing tools cannot be used directly for the performance testing of runtime verification tools.

6 Conclusions and Future Work

With the significant advancements in the area of runtime verification in recent years, the availability of mature tools is crucial for the increased adoption in industry. To facilitate the testing and profiling of runtime verification tools we have presented SMock[7], a mock system generator coupled with replay and profiling capabilities. The tool has been successfully applied to two state of the art tools showing the overheads in terms of time taken, memory consumption, and processing resources.

Future improvements to SMock will focus on providing more advanced profiling features such as power consumption measurement and automatic measurement repetition

[5] `http://jmock.org/`

[6] `http://jmeter.apache.org/`

[7] The tool can be downloaded from `http://www.cs.um.edu.mt/svrg/Tools/SMock`

to ensure that results are not affected by external factors such as garbage collection or unrelated operating system activities. Furthermore, we would like to build an appropriate test suite to test runtime monitoring tools for correctness. Although, at the moment we only support the generation of a mock system written in Java, the design of the tool makes it straightforward to extend to other languages — which we plan to do in the near future.

References

1. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. Comput. 20(3), 707–723 (2010)
2. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: Extensible technology-agnostic runtime verification. In: FESCA. EPTCS, vol. 108, pp. 1–15 (2013)
3. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 214–228. Springer, Heidelberg (2010)
4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009)
5. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: SEFM, pp. 33–37 (2009)
6. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4es), 196 (1996)
7. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. JSTTT 14(3), 249–289 (2012)

SPY: Local Verification of Global Protocols

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu

Imperial College London, UK

Abstract. This paper presents a toolchain for designing deadlock-free multiparty global protocols, and their run-time verification through automatically generated, distributed endpoint monitors. Building on the theory of multiparty session types, our toolchain implementation validates communication safety properties on the global protocol, but enforces them via independent monitoring of each endpoint process. Each monitor can be internally embedded in or externally deployed alongside the endpoint runtime, and detects the occurrence of illegal communication actions and message types that do not conform to the protocol. The global protocol specifications can be additionally elaborated to express finer-grained and higher-level requirements, such as logical assertions on message payloads and security policies, supported by third-party plugins. Our demonstration use case is the verification of choreographic communications in a large cyberinfrastructure for oceanography [10].

1 Introduction

The application-level interactions in distributed systems and Web services often involve complex, high-level communication patterns between multiple parties. It is common for implementations of each participant to be written separately, or for a system to be constructed by composing separate services managed by different administrative domains. Implementations are also commonly based on informal protocol specifications, and thus informal verification mechanisms, and can be prone to concurrency errors such as communication mismatch (e.g. the arrival of an unexpected message or request of an unsupported service operation) and deadlock (e.g. party A waits to receive a message from B while B is waiting for a message from A). This is why the need for rigorous description and verification of protocols has been observed in many different contexts.

The Scribble language [6,12] (foundation of the JBoss Savara project [13]) is a formal protocol description language developed towards tackling this challenge. The goal of Scribble is to provide an intuitive engineering language and tools, based on the theory of multiparty session types (MPST) [7], for specifying and reasoning about message passing protocols and their implementations. As a verification technique, the previously published implementations of MPST focus on static type checking of protocol specifications against endpoint processes. Well-typed processes are guaranteed to enjoy properties such as communication-safety (all processes conform to a globally agreed communication protocol) and deadlock-freedom. Static session type checking in these

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 358–363, 2013.

mainstream languages, however, requires support in the form of the language extensions and pre-compiler processing to be tractable.

In this paper, we demonstrate a toolchain (SPY: Session Python) for runtime verification of distributed Python programs against Scribble protocol specifications. Our aim is to adapt the MPST protocol verification techniques to runtime verification in order to be directly applicable to standard mainstream languages. Due to the distributed setting, our toolchain works to enforce a *global* protocol by decomposing it into *local* specifications to be independently monitored at each endpoint. Runtime verification can also be more practical for enforcing advanced protocol features, e.g. we have extended our version of Scribble to support annotations for logical assertions, which would be more complicated to verify statically, even conservatively and with language extensions.

Given a Scribble specification of a global protocol, our toolchain validates consistency properties, such as race-free branch paths, and generates Scribble (i.e. syntactic) local protocol specifications for each participant (role) defined in the protocol. At runtime, an independent monitor (internal or external) is assigned to each Python endpoint. When a session between the endpoints is initiated, each monitor retrieves the local protocol for its endpoint, and generates the corresponding finite state machine (by an extension of the algorithm in [4]) to verify the local trace of communication actions executed during the session. The evaluation of assertions is handled through a third-party engine.

To summarise the main features and characteristics of our toolchain: (1) it is based on a specification language [6,12] with a formal semantics [3,2] (with proof of the soundness of local monitoring of global protocols), and is the first implementation of runtime verification for this theory; (2) protocol specifications can be decorated to perform third-party validation of constraints beyond the core message passing protocol; (3) monitoring is decentralised with each participant verified locally and therefore no synchronisation between monitors is needed; (4) two kinds of monitor, internal (synchronous) and external (asynchronous), are implemented; and (5) the toolchain has been integrated into an industry project [10] for the verification of RPC services and multiagent protocols [11].

The rest of the paper illustrates the key steps of the toolchain, outline its usage requirements and discusses current applications. A discussion of related work and additional examples can be found within the same volume [8]. The source code of the tools and performance benchmarks are available from the project website [9].

2 Multiparty Session Types and Runtime Verification

We illustrate our toolchain through an introductory example, an online payment application, which we call OnlineWallet (Fig. 1). The scenario involves three parties: a Client (C), a Payment Server (S) and a separate Authenticator (A). At the start of a session, C sends its login details to A, and A replies to inform C and S whether the authentication is successful or not. If so, C and

S enter a loop: in each iteration, S sends C the current account status, and C has the choice to make a payment (but only for an amount that would not overdraw the account) or end the session. In the first case, C sends the payee and amount to S, and the loop is repeated. In the other case, or if the authentication failed, the session ends.

Our toolchain performs the verification across several levels, as explained below.

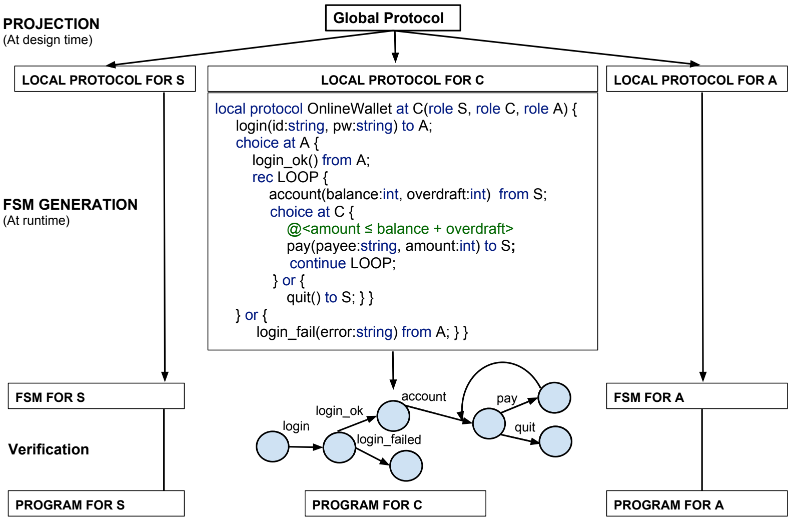
```
global protocol OnlineWallet
      (role S, role C, role A) {
  login(id:string, pw:string)
      from C to A;
  choice at A {
    login_ok() from A to C, S;
    rec LOOP {
      account(balance:int,
        overdraft:int) from S to C;
      choice at C {
        @<amount <= balance+overdraft>
        pay(payee:string, amount:int)
            from C to S;
        continue LOOP;
      } or {
        quit() from C to S; }}
  } or {
    login_fail(error:string)
        from A to C, S; }}
```

Fig. 1. OnlineWallet protocol in Scribble

Global Protocol Correctness. The first level of verification is in the design of the global protocol. The Scribble in Fig. 1 describes interactions between session participants from the global perspective using message passing sequences, branching (choice) and recursion. Each message has an operator (a label) and a payload. The toolchain validates that the protocol is coherent and deadlock-free, and thus projectable [7] for each role. For example, in each case of a `choice` construct, the deciding party (e.g. `at A`) must correctly communicate the decision outcome unambiguously to all other roles involved; a `choice` is badly-formed if the actions of the deciding party would cause a race condition on the selected case between the other roles, or if it is ambiguous to another role whether the decision has already been made or is still pending. The interested reader may refer to [6,12] for a comprehensive overview of the Scribble syntax, a tutorial, and further references to the formal conditions for protocol correctness.

Local Protocol Conformance. The second level is runtime verification to ensure that each endpoint program conforms to the core protocol structure *according to its role*. There are two main factors. First, we verify that the type (operation and payload) of each message matches its specification (operations can be mapped directly to message headers, or to method calls, class names or other relevant artifacts in the program). Second, we verify that the flow of interactions is correct, i.e. interaction sequences, branches and recursions proceed as expected, respecting the explicit dependencies (e.g. `m1() from A to B; m2() from B to C;` imposes an input-output causality at B). These measures rule out errors, such as communication mismatches, that violate the permitted protocol flow.

Fig. 2 outlines the concrete verification steps. First, local protocols are mechanically generated from the validated global protocol. A local protocol is essentially a view of the global protocol from the perspective of one role. The projection algorithm works by identifying the message exchanges where the participant is involved, and disregarding the rest while preserving the overall structure of the global protocol. Each local protocol has a corresponding FSM, generated by the monitor at runtime. When a party requests to start or join a session,



```
local protocol OnlineWallet at C(role S, role C, role A) {
    login(id:string, pw:string) to A;
    choice at A {
        login_ok() from A;
        rec LOOP {
            account(balance:int, overdraft:int)  from S;
            choice at C {
                @<amount ≤ balance + overdraft>
                pay(payee:string, amount:int) to S;
                 continue LOOP;
            } or {
                quit() to S; } }
    } or {
        login_fail(error:string) from A; } }
```

Fig. 2. Global specification to local runtime verification methodology

the initial message specifies which role it intends to play. Its monitor retrieves the local specification based on the protocol name and the role. Fig. 2 gives the local protocol and associated FSM for the client role C (we omit the protocols for S and A). The FSM encodes the flow of local communication actions, with transitions fired by the input and output of the permissible messasges.

Policy Validation. The final level of verification enables the elaboration of Scribble protocols using annotations (@<...> in Fig. 1 and 2). The annotations function as API hooks to the verification framework: they are not verified by the MPST monitor itself, but delegated to a third-party engine. Various policy domains (e.g. security policies) can be enforced by integrating engines for predicates on endpoint state, automata-based properties, etc., as extensions to the core protocol monitor. Our current implementation uses a Python library for evaluating basic predicates (e.g. the overdraft check in Fig. 1), which is sufficient for the application protocols we have developed with [11]. At runtime, the monitor passes the annotation information, along with the FSM state information, to the appropriate policy engine to perform the additional checks or calculations. To plug in an external validation engine, our toolchain API requires modules for parsing and evaluating the annotation expressions specified in the protocol.

3 Toolchain Requirements and Evaluation

3.1 Monitor Requirements

Positioning. The network monitoring in our theory imposes *complete mediation* of communications: no communication action should have an effect unless the message is mediated by the monitor. The tool implements this principal for both inline and outline monitor configurations. Inline monitoring relies on internal message interception: the local conversation runtime, in place at each endpoint, synchronously passes every message (on arrival or prior to dispatch) through the monitor component. Ouline monitoring is realised by dynamically modifying the application-level network configuration to asynchronously route every message through a monitor. Our prototype is built over an Advance Messaging Queue Protocol (AMQP) [1] transport, where we use the AMQP exchange-to-exchange binding functionality to perform the message rerouting. A monitor dispatcher is assigned to each network endpoint as a conversation gateway. The dispatcher can create new routes and spawn new monitor processes if needed, to ensure the scalability of this approach.

Message Format. To monitor Scribble conversations, our toolchain relies on a small amount of message meta data that we refer to as the Scribble header, but is actually embedded into the message payload (for more flexibile interoperability). Messages are processed depending on the message type, as recorded in the header. There are two kinds of conversation messages: *initialisation* (exchanged when a session is started, carrying information such as the protocol name and the role of the monitored process) and *in-session* (carrying the message operation and the sender/receiver roles). Initialisation messages are used for routing reconfiguration, while in-session messages are checked for protocol conformance.

Conversation API. Our toolchain is accompanied by a message-passing library for implementing Python endoint applications, that augments message payloads with the conversation information required for monitoring. The library API concisely exposes the core MPST primitives [3,2] for (1) initiating and joining a conversation and (2) asynchronous message dispatch and consumption by the participants. The API can be used directly by the programmer as a standalone conversation library, or as a complementary support module by another library to handle the formatting of conversation messages for monitoring.

3.2 Evaluation

Our work is applied to and running within the Ocean Observatories Initiative (OOI) [10,11], an ongoing project to establish a cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. The OOI architecture relies on the combination of high-level protocol specifications (to express how the infrastructure services should be used) and distributed run-time monitoring to regulate the behaviour of every application within the system, for which the present toolchain is used. Performance measurements for our current implementation (the project is at release two of a planned four) show a reasonable overhead (13% percent per message call, see [9] for the full benchmarks). The overhead is mostly due to just-in-time FSM generation, which we believe can be reduced by caching or pre-generation of the FSM for each protocol. We also note that the relative overhead due to FSM generation decreases as the length of the conversation increases.

Our collaboration in the OOI project has had interesting impacts on our work and research. First, the practical requirements, emerging from their use cases, led to the several advances of the MPST theory and the Scribble language (interruptible conversations [8], generic protocols [5] and protocol annotations). Second, we found that many OOI use cases can be categorised into a small set of parameterised protocols. As an example, the majority of service-oriented protocols, with diverse message signatures, are now derived from a single parametrised RPC service protocol; rather than requiring a Scribble protocol per application instance, one parameterised protocol can be provided per application library. This is a convenient approach because we have observed that developers are (so far) often not accustomed to writing protocols explicitly and formally. Finally, the integration of our toolchain proceeded from the specification and verification of the smaller, lower-level protocols in the OOI system, such as RPC. In general, the kinds of bugs detected by our toolchain (e.g. messages to/from the wrong participant) did not frequently arise for these smaller protocols; however, this starting point enabled a straightforward, non-intrusive integration (not a single line of existing application code was changed) that eased the adoption of the tool by the developers. The next phase of the ongoing integration is to port the more complex application protocols to Scribble, given the monitoring infrastructure (independent of the protocol size) is already in place: our toolchain is able to verify any Scribble protocol using the single generic monitor implementation.

References

1. Advanced Message Queuing Protocol homepage, `http://www.amqp.org/`
2. Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE 2013. LNCS, vol. 7892, pp. 50–65. Springer, Heidelberg (2013)
3. Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., Yoshida, N.: Asynchronous distributed monitoring for multiparty session enforcement. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 25–45. Springer, Heidelberg (2012)
4. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
5. Honda, K., Hu, R., Neykova, R., Chen, T.-C., Demangeon, R., Deniélou, P.-M., Yoshida, N.: Structuring Communication with Session Types. In: COB 2012. LNCS (2012) (to appear)
6. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
7. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284. ACM (2008)
8. Hu, R., Neykova, R., Yoshida, N., Demangeon, R.: Towards practical interruptible conversations. This volume
9. Session Python (SPY) resource page, `http://www.doc.ic.ac.uk/~rn710/spy/`
10. Ocean Observatories Initiative, `http://www.oceanobservatories.org/`
11. Scribble-OOI collaboration, `https://confluence.oceanobservatories.org/display/CIDev/OOI+Use+Cases+in+Scribble`
12. Scribble project home page, `http://www.scribble.org`
13. JBoss Scribble site, `http://www.jboss.org/scribble`

Instrumenting Android and Java Applications as Easy as abc

Steven Arzt, Siegfried Rasthofer, and Eric Bodden

Secure Software Engineering Group,
European Center for Security and Privacy by Design (EC SPRIDE),
Technische Universität Darmstadt, Germany
{steven.arzt,siegfried.rasthofer,eric.bodden}@ec-spride.de

Abstract. Program instrumentation is a widely used mechanism in different software engineering areas. It can be used for creating profilers and debuggers, for detecting programming errors at runtime, or for securing programs through inline reference monitoring.

This paper presents a tutorial on instrumenting Android applications using Soot and the AspectBench compiler (abc). We show how two well-known monitoring languages –Tracematches and AspectJ– can be used for instrumenting Android applications. Furthermore, we also describe the more flexible approach of manual imperative instrumentation directly using Soot's intermediate representation Jimple. In all three cases no source code of the target application is required.

Keywords: Android, Java, Security, Dynamic Analysis, Runtime Enforcement.

1 Introduction

According to a recent study [1], Android now has about 75% market share in the mobile-phone market, with a 91.5% growth rate over the past year. With Android phones being ubiquitous, they become a worthwhile target for security and privacy violations. Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners use their device both for private and work-related communication [2]. Furthermore, the vast majority of users installs apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

One approach to combat such threats is to augment Android applications obtained from arbitrary untrusted sources with additional instrumentation code. This code alters the behaviour of the target application and can thus enforce certain predefined security policies such as disallowing data leaks of confidential information. Since the instrumentation code runs as an integrated part of the target application, it has full access to the runtime state, thereby avoiding the imprecisions that usually come with static analysis approaches [3–5]. It has full

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 364–381, 2013.

access to environment information, user inputs, and external resources. Policy violations can be captured as they actually occur, thus minimizing the number of false alarms. Furthermore, it has also the advantage that the underlying Android framework does not have to be changed at all, as done so by [6,7].

In many cases, the source code of the target application is not available. Therefore, a mechansim for conveniently analyzing and instrumenting binary applications is required. Soot [8] and the abc compiler [9] for AspectJ both support Android bytecode, both as input and as output for the instrumented application. In this paper, we will give an overview of the two tools, explain how to integrate instrumentation code on various layers of abstraction, and illustrate the mechanisms using examples. Though this paper focuses on the Android platform, many of the tools and concepts presented herein are directly applicable to Java applications as well.

The remainder of this paper is structured as follows. In Section 2, we give an overview of the Android platform, present an example application we will use for instrumentation in the remainder of the paper, and discuss some Android-specific aspects like application signatures. Section 3 is dedicated to high-level instrumentation using AspectJ while Section 4 focuses on Tracematches. In Section 5, we introduce Soot and its Jimple intermediate representation which is then used for manual instrumentation in Section 6. Section 7 concludes the paper.

2 Android Platform Overview

The Android platform is built as a stack with various layers running on top of each other [10]. Lower-level layers provide services to upper-level layers. The lowermost layer is built on a customized Linux system and its libraries. The Android middleware builds an abtraction between the operating system and the user-level application running on the very top of the architecture stack. In this tutorial, we concentrate on instrumenting user-level applications.

Applications are provided to the user via different application markets like the official Google Play Store [11] and various third-party stores. Application developers can also host applications for download on their own websites.

2.1 Application Architecture

Most of the applications are written in the Java programming language[1]. They are compiled to Android's own bytecode format, called the Dalvik executable (dex). On application launch, the Android middleware spawns a new Dalvik Virtual Machine to execute the application's dex file. This enables Android to exploit the process isolation mechanisms of the underlying Linux operating system and ensures that all applications are run inside their own isolated containers. Note that Android applications do not use Java's concept of security managers [12]. Instead, Android implements its own permission systems for certain sensitive function calls. Furthermore, the Dalvik Virtual Machine is not

[1] In this tutorial, we disregard portions written in native code or script languages.

stack-based like the Java VM, but register-based and optimized for resource-constrained mobile devices [13].

Android applications do not have a single entry-point, such as the *main* method in Java. Developers must instead design the application in terms of components, each one adhering to a set of predefined interfaces. Every component is implemented as Java class derived from a specific base class in the Android middleware. Components react to OS events by overwriting the respective methods or calling specific OS methods to register further callbacks that are invoked when e.g. device's physical location changes.

There exist four different kinds of components: *activities*, *services*, *content providers*, and *broadcast receivers* [14]. *Activities* are single focused activities a user can interact with. They are the visible parts of an applications. In contrast, the *services* run in the background and are not interacting with the user directly. They are used for long-running background operations, such as MP3 playback. *Broadcast receivers* react to global events, such as incoming calls or text messages. *Content providers* implement domain-specific databases for, e.g., contacts [15].

The first three component types can communicate via asynchronous messages called *intents*. An intent is an abstract description of an action "intended" to be performed, such as *"launch the following website"*. Intents are a powerful feature in the Android platform that allow communication between components, both inside an application and across application boundaries. Intents are dispatched by the Android middleware, either to a directly specified receipient or to all receivers registered with the system for a specific intent type, e.g., all components capable of displaying a website to a user.

Each of the four different types of components have a distinct lifecyle that defines how the component is created, used and destroyed. The lifecycle is guided using events, i.e., a sequence of methods called by the OS. For instance, the `onCreate()` method gets called when an activity is loaded for the first time [16].

2.2 Android SMS Messenger Example

We next describe a simple Android application implementing an SMS Messenger. The app's user interface simply consists of two user inputs, one for the phone number and one for the message to be sent. When the user clicks on the"Send SMS" button, the application sends the given text message to the given phone number.

Listing 1.1 shows the corresponding source code. The code comprises the two methods `onCreate` and `sendSms`. As described in Section 2.1, the `onCreate` event method gets called when the activity is launched for the first time. The method defines some layout settings (`setContentView`) and prints out some debug information. Section 2.5 will give more details on Android's logging infrastructure.

The `sendSms` callback method is the more interesting part. It is called when the user clicks on the "Send SMS" button. The link between the method and the button is established using a layout XML file, which is a declarative way

```
public class RV2013 extends Activity
{
  private EditText phoneNr, message;
  private SmsManager smsManager = SmsManager.getDefault();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_rv2013);

    Log.i("INFO", "in onCreate");
}

public void sendSms(View v){
  Log.i("INFO", "in sendSms");

  phoneNr = (EditText)findViewById(R.id.phoneNr);
  message = (EditText)findViewById(R.id.message);

  System.out.println("in sendSms");

  smsManager.sendTextMessage(phoneNr.getText().toString(), null,
     message.getText().toString(), null, null);
    }
}
```

Listing 1.1. Source Code of SMS Messenger Example

to register callsbacks for UI components[2]. The button handler again writes out some debug information ("in send Sms"), then extracts the user input in the different text fields using the *findViewById* OS function, afterwards calls the `println` method in the `PrintStream` class with the string "in SendSms" and finally sends out the SMS message, again using an OS function.

2.3 Overview of Android API Calls

The Android middleware provides abstractions for conveniently using device functions like sending SMS messages directly from applications written in Java without having to directly interact with native code libraries on the system level. The most important Android API methods for this paper are the following ones:

- `Log.i(String tag, String msg)`:
  Static methods which writes an `info` message to the log. The log can be browsed using the tool LogCat (c.f. Section 2.5).

  `tag`: usually identifies the class or activity where the log call occurs
  `msg`: the message that should be logged

- `findViewById(int id)`:
  Returns references to GUI objects. In this example, the `findViewById` method is used to get the text field contents for message text and recipient phone number. Note that such calls are generated by the compiler; one usually use the constants in the `R` pseudo class to access GUI elements from app code.

[2] The other alternative would have been to programatically set a listener in *onCreate*.

`id`: the object id to search for

- `SMSManager.sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)`:
  sends a text-based SMS message

  `destinationAddress`: the address to send the message to
  `scAddress`: the service center address; pass `null` to use the default
  `text`: the body of the message to send
  `sentIntent`: a broadcast message to be generated by the system when the message has been sent; pass `null` if not required
  `deliveryIntent`: a broadcast message for the message delivery; pass `null` if not required

2.4 Android.jar: Where Android Lives

The Android middleware consists of predefined Java classes and a set of native libraries. To be able to compile Android apps that make use of this API on a desktop PC, the Java classes of the Android API must be present. Therefore, the Android SDK provides these classes in a file called *android.jar* in its *platforms* directory. There is one such file for every version of the OS. We recommend using the appropriate version of the JAR file since new APIs are added from time to time and old, deprecated ones are removed. The minimum compatible OS version specified in the application's manifest file is usually a good pick.

However, note that these JAR files can only be used to create a somewhat complete callgraph and points-to set in the user code, but they cannot be used to actually run the application. This is because for many methods they only contain stubs and no actual implementations. Stubbed methods just throw a *NotImplementedException*. Obtaining a full *android.jar* file from a real phone is possible, but not trivial, as the Android API is stored in a precompiled and optimized file format. In most cases, such complete JAR files are not needed anyway.

2.5 Useful Tools

The Android SDK provides a number of tools that support a developer during the developement of an Android application. For instance, debugging or running on an emulator is essential during the developement phase. Therefore, we will briefly introduce the two most important tools: Logcat and the Android emulator.

Logcat. Android's logging system provides a mechanism for collecting different kinds of log messages from various applications and system components. These logs can be easily viewed and filtered using the `logcat` tool which is built on top of Android's debug bridge `adb`. Logcat can directly be launched from the command line with `adb logcat`. It supports various settings for filtering and formatting the output as explained in [17]. The Android eclipse plugin provides a more convenient graphical user interface to logcat.

A log entry can be produced by invoking the static methods in the `android.util.Log` class. For instance, the statement `Log.e("TAG", "Ooops")` creates an error line in the log. The `Log.e` method takes two parameters: The tag (first parameter) can be used for filtering and categorization. The error message (second parameter) contains the error message or failure reason.

Android Emulator. As the name already suggests, the Android emulator [18] is a virtual mobile device that runs on a computer and is similar to a real device. It does not contain all the features of a real mobile device such as sending emails, but for most purposes, it is sufficiently complete. However, note that applications may suffer from serious performance penalties when run on the emulator.

With the help of the Android debug bridge [19], it is easy to create a new emulator. The command `android create avd -n <name> -t <targetID>` creates a new virtual device with the given name. The targetID is the API level one needs, e.g., 17 for Android 4.2. The new emulator is started by `emulator -avd <name>`. Afterwards, the virtual device's user interface is shown and one can interact with the emulator through the SDK's command-line tools. A more convenient way for the creation of an emulator is the usage of the grafical interface provided by eclipse (*Android Virtual Device Manager*).

2.6 Managing APKs on the Device

For author identification purposes, the Android framework requries that each application has to be signed with a certificate. This, for instance, allows the system to check whether an application update actually comes from the original application developer. Furthermore, applications signed with the same key are granted special privileges in interprocess communication. On the Android platform, it is common that most of the application certificates are self-signed [20]. When changing the APK file, e.g., by instrumenting the code, the signature is lost and the application must be signed again.

Standard tools like *keytool* and *jarsigner* can be used for signing the application. An example for the generation of a private/public key pair with *keytool* [20] is shown in Listing 1.2.

```
$ keytool -genkey -v -keystore my-release-key.keystore
-alias alias_name -keyalg RSA -keysize 2048 -validity 10000
```

Listing 1.2. Generation of a private/public key pair with keytool [20]

The jarsigner tool can be used for signing the application *my_application.apk* with the private key generated with keytool. Listing 1.3 shows the command for signing an app with jarsigner.

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
my-release-key.keystore my_application.apk alias_name
```

Listing 1.3. Signing my_application.apk with jarsinger [20]

After signing the application, Google [21] recommends to use the *zipalign* tool to optimize the final APK. It ensures that all uncompressed data starts with a particular alignment relative to the start of the file. The zipalign tool comes with the Android SDK and Listing 1.4 shows the corresponding command.

```
$ zipalign -v 4 my_application.apk my_application_release.apk
```

Listing 1.4. Alignment of my_application.apk befor release

3 Instrumentation with abc and AspectJ

In this section, we describe how AspectJ and the AspectBench Compiler abc [9] can be used to declaratively instrument Android applications. Our goal is to modify the example application from Listing 1.1 such that no premium SMS messages to costly 0900 phone numbers can be sent anymore. Instead, an error message shall be written into the log file whenever the target phone number starts with 0900. SMS messages to normal phone numbers should be sent as usual. Obviously, this requires us to inline a monitor since no static analysis can know the target phone number the user is going to enter.

We create a new file *SendPremiumSMS.aj* with the contents shown in Listing 1.5. Note that the name of the file must match the name of the aspect. We first declare a pointcut for the `SmsManager.sendTextMessage` method. We could also have inlined it into the advice definition, but we are using it twice (once for blocking premium-rate SMS messages and once for logging that a message has actually been sent), so we keep it separate. The pointcut matches calls to the SMS sending method in the Android operating system, not our own user code. This way, we ensure that actually all SMS messages are intercepted which is especially useful when instrumenting unknown target applications.

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_PremiumAspect {
  pointcut sendSms(String no) :
     call(void SmsManager.sendTextMessage(..)) && args(no, ..);

  after(): sendSms(*) {
    Log.i("Aspect", "SMS message sent");
  }

  void around(String no): sendSms(no) {
    if (no.startsWith("0900"))
      Log.i("Aspect", "Premium SMS message blocked");
    else proceed(no);
  }
}
```

Listing 1.5. Aspect for blocking premium-rate SMS messages

In general, the aspects as such are written in the well-known AspectJ syntax and are not Android-specific except for the methods intercepted in the pointcuts and the ones called in the pieces of advice. All mapping to the Android platform is done by the abc compiler during the weaving process.

Since we only want a notification when an SMS message has actually been sent, the order of the pieces of advice inside the aspect is important: We place the *around* advice last to give it precedence over the *after* advice which shall only be executed when the *around* advice proceeds, i.e., the target phone number is not a premium-rate number. Otherwise, the SMS message is blocked and thus shall not be logged.

abc supports two different frontends for parsing Java source code: Polyglot and JastAdd. The Polyglot frontend is a bit dated and should not be used for instrumenting Android applications from source. JastAdd can be enabled as an extension using the `-ext abc.ja` command-line option. Also note that abc has its own class path which is independent of the JDK's class path and which must be set using the `-cp` option. It should include both the JRE's `rt.jar` file and abc's own `abc-runtime.jar` file. Since our target application references classes from the Android framework, we also need to include the `android.jar` file.

If applications written for modern versions of the Android API are also supported on older platforms (i.e., have a lower minimum SDK version that one they were developed for), the Android eclipse plugin automatically integrates so-called support classes which add some newer APIs to older platforms. The respective *jar* file can then found in the *libs* directory of the application project and needs to be included in abc's class path as well.

The complete command-line for instrumenting the example is shown in Listing 1.6. The Android support is enabled with the `-android` option, the APK file name is given with the `-injars` switch.

```
java -cp abc-ja-exts-complete.jar abc.main.Main \
  -cp /path/to/rt.jar: \
    /path/to/android-support-v4.jar: \
    /path/to/android.jar: \
    /path/to/abc-runtime.jar \
  -ext abc.ja \
  -android -injars /path/to/RV2013.apk \
  /path/to/SendSMS_PremiumAspect.aj
```

Listing 1.6. abc compiler command-line

Unsigned applications will not run on the Android OS, which is why the instrumented *apk* file still needs to be signed before it can be run on a real phone or the emulator (c.f. Section 2.6).

4 Instrumentation with Tracematches

While AspectJ is rather convenient for describing Android instrumentations (see Section 3), it requires additional manual effort when sequences of actions shall be tracked. Tracematches [22] provide a simple regular-expression based approach to declaratively abstract from such tracking. Let us assume we want to raise an

```
import android.telephony.SmsManager;
import android.app.PendingIntent;

public aspect SMSSpam {
  tracematch(String no) {
    sym sendSms after:
      call (void SmsManager.sendTextMessage(..)) && args(no, ..);

    sendSms[3] sendSms+ {
      System.out.println("SMS spam detected to no: " + no);
    }

  }
}
```

Listing 1.7. Aspect for blocking premium-rate SMS messages

alert when more than three SMS messages are sent to the same phone number by an application, as this might indicate SMS spam. In AspectJ such counting would have to be implemented manually. In Tracematches, we simply define the pattern shown in Listing 1.7. Note that the name of the file and the name of the aspect must match, i.e., `SMSSpam.aj` in this case.

For compiling the tracematch, we again use the AspectBench compiler abc. The command-line is similar to the one shown in Section 3 for AspectJ, we only need to enable the tracematch extension as shown in Listing 1.8.

```
java -cp abc-ja-exts-complete.jar abc.main.Main \
  -cp /path/to/rt.jar: \
    /path/to/android-support-v4.jar: \
    /path/to/android.jar: \
    /path/to/abc-runtime.jar \
  -ext abc.ja.tm \
  -android -injars /path/to/RV2013.apk \
  /path/to/SMSSpam.aj
```

Listing 1.8. abc compiler command-line

5 The Machinery: Soot and Jimple

Soot [8] is an extensible program analysis and optimization framework for Java and Java-like environments such as Dalvik. It supports various input formats including Java source code, Java `class` files, and Dalvik `dex` files and also allows to write out these file formats after transformation. Figure 1 gives an overview of all possible input and output formats.

Code included in an Android application's *apk* file is automatically extracted before analysis. Afterwards, a new *apk* file containing the transformed code is built which can then be signed and executed on a phone or the emulator. abc uses Soot internally to weave aspects or tracematches into Java programs or Android apps.

Soot is organized in phases and packs [23]. Every pack contains an ordered list of phases. The first pack applied to every single method is the Jimple Bodies pack *jb* which translates the respective method's body into an intermediate representation called *Jimple*. Afterwards, if whole-program analysis is enabled, a number

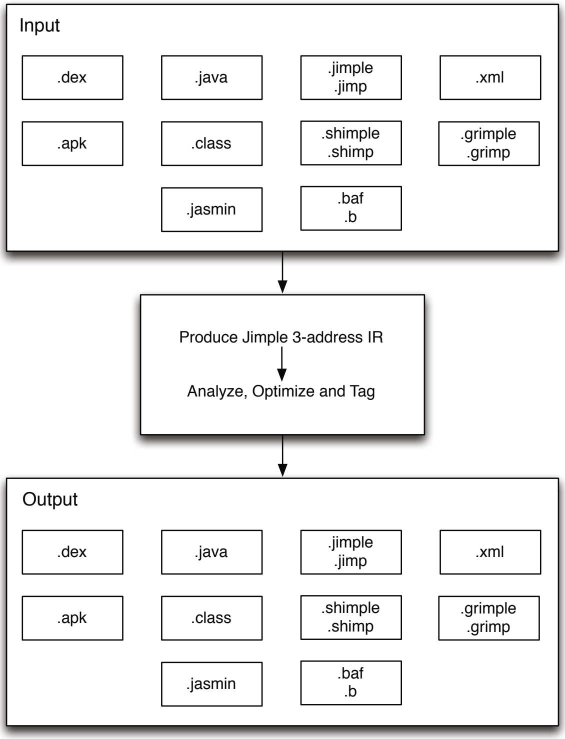


Fig. 1. Input and Output Formats in Soot

of whole-program packs run. They do not target single methods or classes, but the whole so-called *scene* containing all classes that have been loaded. Which classes are loaded depends on Soot's command line options. Consult the online documentation for details [24]. Usually, you only need to enable whole program mode if your analysis requires a complete call graph. If not, you can skip these phases by leaving the whole-program-mode option disabled which can considerably improve performance.

The first whole-program pack to run is the *cg* pack which creates the callgraph. Soot implements various callgraph construction algorithms. In this tutoral, we will use SPARK [25] for maximum precision. In some cases, less precise, but faster algorithms might be more appropriate. Once the callgraph is done, three more whole-program packs (whole-jimple-transformation, whole-jimple-optimization, whole-jimple-annotation) are executed, followed by a sequence of single method-packs (jimple-transformation, jimple-optimization, jimple-annotation).

For our purposes, we leave the whole-porgram-mode disabled and add a new phase to the jimple-transformation pack *jtp* which places our code directly after the Jimple bodies are produced and before all other optimizations like dead code elimination run. This allows us to exploit the transformations done in the latter. If we needed a complete callgraph, we would use the whole-jimple-transformation-pack *wjtp* instead.

In the remainder of this section, we will show how to programatically configure and launch Soot, how to access the Jimple code of a method, and explain how Jimple is structured.

5.1 Jimple: Java, But Simple

Jimple stands between full Java sourcecode on one side and Java/Dalvik bytecode on the other side. While the first is impractically complex for static analysis or program transformations, the latter is quite cumbersome to work with because of its large number of (untyped) instructions. Jimple combines the advantages of both sides: There is only a limited instruction set, data is stored in variables, and statements are generally of a simple three-operand form. More complex statements or expressions are broken up into simple single-operation pieces and a set of intermediate variables. For instance, in Jimple `a=b+c+2` would be transformed to `temp=b+c, a=temp+2` with a new intermediate variable `temp`.

Jimple contains two general concepts: *locals* which are local variables and *units* which are statements. Every method body contains one chain of locals and one ordered chain of units. Units are usually of some type derived from *Stmt*, which in turn can contain references to expressions derived from *Expr*. Jimple generalizes all Java constructs to units and locals. The Java *this* reference, for instance, is assigned to a local at the beginning of an instance method. Afterwards, it behaves just like an ordinary local variable. The same happens with method parameters. These special assignments are called *IdentityStatement*s (c.f. lines 12 and 13 in Listing 1.9).

Assignments between locals, constants, and fields are done using *AssignStatement*s (c.f. line 23). Since Soot represents the AST as an object model in memory, the left and right side of an assignment are references to the objects representing the expressions standing on either side. Programatically traversing Jimple code thus simply means following chains of references. For instance, in line 16, the right-hand side of the assignment is a typecast represented by a *CastExpr* object.

To call methods, Jimple supports four different expressions, depending on the type of the target method. The three most important ones are *VirtualInvokeExpr* for a virtual dispatch invoke to an instance method (lines 15 and 18), *StaticInvokeExpr* for calling a static method (line 14), and *InterfaceInvokeExpr* for calling a method of an object of which only its interface type is known (line 26). Any invoke expression can be part of standalone statement called *InvokeStmt* (lines 14, 22, and 30), but can also serve as the right side of an assignment (i.e. *AssignStmt*, e.g., in line 15) unless the return type is `void`.

```
public void sendSms(android.view.View)
{
  de.ecspride.RV2013 $r0;
  android.view.View $r1;
  java.lang.String $r2, $r3;
  android.widget.EditText $r4;
  int $i0;
  java.io.PrintStream $r5;
  android.telephony.SmsManager $r6;
  android.text.Editable $r7;

  $r0 := @this: de.ecspride.RV2013;
  $r1 := @parameter0: android.view.View;
  staticinvoke <android.util.Log: int
     i(java.lang.String,java.lang.String)>("INFO", "in sendSms");
  $r1 =  virtualinvoke $r0.<de.ecspride.RV2013: android.view.View
     findViewById(int)>(2131165184);
  $r4 = (android.widget.EditText) $r1;
  $r0.<de.ecspride.RV2013: android.widget.EditText phoneNr> = $r4;
  $r1 = virtualinvoke $r0.<de.ecspride.RV2013: android.view.View
     findViewById(int)>(2131165187);
  $r4 = (android.widget.EditText) $r1;
  $r0.<de.ecspride.RV2013: android.widget.EditText message> = $r4;
  $r5 = <java.lang.System: java.io.PrintStream out>;
  virtualinvoke $r5.<java.io.PrintStream: void
     println(java.lang.String)>("in sendSms");
  $r6 = $r0.<de.ecspride.RV2013: android.telephony.SmsManager
     smsManager>;
  $r4 = $r0.<de.ecspride.RV2013: android.widget.EditText phoneNr>;
  $r7 = virtualinvoke $r4.<android.widget.EditText: android.text.Editable
     getText()>();
  $r3 = interfaceinvoke $r7.<android.text.Editable: java.lang.String
     toString()>();
  $r4 = $r0.<de.ecspride.RV2013: android.widget.EditText message>;
  $r7 = virtualinvoke $r4.<android.widget.EditText: android.text.Editable
     getText()>();
  $r2 = interfaceinvoke $r7.<android.text.Editable: java.lang.String
     toString()>();
  virtualinvoke $r6.<android.telephony.SmsManager: void
     sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
            android.app.PendingIntent,android.app.PendingIntent)>($r3,
     null, $r2, null, null);
  return;
}
```

Listing 1.9. Jimple code for sending an SMS message

Method bodies are commonly analyzed by iterating over the units (i.e., statements) they comprise. For program rewriting, the chain of units is patched by removing existing units, inserting new units at the desired positions, or changing the expressions within existing units. All of these changes will be explained in the remainder of this paper.

5.2 Soot Options

Soot provides a lot of different command-line options. An online tutorial [26] gives a good overview of the different kinds of options available. The most important options for instrumenting Android applications are the following:

`-cp` *`pathlist`*: The classpath to be used when loading classes into Soot. Not to be confused with the classpath used by the JVM.

`-pp`: This option prepends the VM's classpath to Soot's own classpath.

`-validate`: Causes sanity checks to be performed on Jimple bodies to make sure the transformations have caused no type errors. This option may degrade Soot's performance, but might be useful for debugging instrumentation code.

`-output-format` *`format`*: Specifies the format of output files Soot should produce, if any. In case of Android instrumentation, the *dex* format has to be set. For debugging purposes, one can use the *jimple* output format to inspect the instrumentation results in the intermediate language. Note, though, that one cannot create outputs in multiple formats at the same time.

`-process-dir` *`dirs`*: Adds all classes in `dirs` to the set of classes to be analyzed and transformed by Soot. The list `dirs` can also contain *jar* or *apk* files.

`-src-prec` *`format`*: Sets format as Soot's preference for the type of source files to read when it looks for a class. In the case of Android, the *apk* format must be set.

`-w`: Tells Soot to enable the whole-program transformation packs. Required if one requires a callgraph or wants to use the *wjtp* pack for performing global transformations spanning multiple methods.

`-allow-phantom-refs`: Allows Soot to model classes not found on the classpath by stubs containing no methods or fields. Useful for saving memory by not including full implementations of some libraries.

These options can either be set via the command line or directly in the Java code via `Options.v()`, e.g., `Options.v().set_whole_program(true)` for enabling whole-program mode if required. Listing 1.10 shows an example of a possible Soot initialization for instrumenting Android applications.

```java
private static boolean SOOT_INITIALIZED = false;
private final static String androidJAR = "./lib/android.jar";
private final static String apk = "./apk/RV2013.apk";

public static void initialiseSoot(){
    if (SOOT_INITIALIZED)
      return;

    Options.v().set_allow_phantom_refs(true);
    Options.v().set_prepend_classpath(true);
    Options.v().set_validate(true);

    Options.v().set_output_format(Options.output_format_dex);
    Options.v().set_process_dir(Collections.singletonList(apk));
    Options.v().set_force_android_jar(androidJAR);
    Options.v().set_src_prec(Options.src_prec_apk);

    Options.v().set_soot_classpath(androidJAR);

    Scene.v().loadNecessaryClasses();

    SOOT_INITIALIZED = true;
}
```

Listing 1.10. Soot Initialization Example for Instrumenting Android Applications

6 Manual Instrumentation

Besides the convenient way of instrumenting Android applications with the help of AspectJ (c.f. Section 3) or tracematches (c.f. Section 4), one can also use Soot to directly manipulate an Android application's code using the Jimple intermediate representation. This is especially important for instrumentations that are not possible with ApsectJ or tracematches. We described a range of such policies in previous work [27]. As a simple example, AspectJ cannot be used to remove debugging outputs including all intermediate computations that are only used in such debugging statements. These computations will remain even if the debug outputs as such are filtered using an *around* advice.

This section demonstrates the two possiblities in direct code modification: removing or adding code. Manipulating existing units is straight-forward given that knowledge. Instead of generating new Jimple units, one simply changes the fields of existing objects. As a first step, we need to configure launch Soot as desribed in section 5. We then register a *jimple-transformation* transformation phase as shown in listing 1.11. As discussed in section 5.2, this is more efficient than using a *whole-jimple-transformation* whole-program phase. Furthermore, we do not need a complete callgraph for our goal, so there is no reason to have Soot create one.

```
PackManager.v().getPack("jtp").add(
    new Transform("jtp.myAnalysis", new MyBodyTransformer()));
PackManager.v().runPacks();
PackManager.v().writeOutput();
```

Listing 1.11. Adding new Phase to Jimple Transformation Pack

The `runPacks()` methods triggers the execution of the packs and calls the overwritten `internalTransform()` method inside the `MyBodyStranformer` class derived from `BodyTranformer`[3]. In order to iterate over all classes and methods in the Android application, one can use the code in listing 1.12 as a starting point. The code must be placed inside `internalTransform()`.

```
for (SootClass c : Scene.v().getApplicationClasses()) {
  for(SootMethod m : c.getMethods()){
    if(m.isConcrete()){
       Body body = m.retrieveActiveBody();
       Iterator<Unit> i = body.getUnits().snapshotIterator();
       while (i.hasNext()) {
         Unit u = i.next();
         //do something
       }
    }
  }
}
```

Listing 1.12. Iterating over the Android Code

[3] In whole-program-mode, we would have used a *SceneTransformer* in the *wjtp* pack.

One important point in this code snippet is the `snapshotIterator()` method that should be used if statements in the method's body will be changed while the loop runs. This avoids `ConcurrentModificationExceptions`.

6.1 Removing Statements

The fact that all statements in the body of a method are stored into a chain makes it very easy to remove a complete statement from the Jimple code. This can be done by just removing it from the chain:

```
body.getUnits().remove(unit);
```

After removing statements, dead code can remain. Soot already offers optimizations for removing such code, propagating definitions that are only used once, and others. If the subsequent Jimple optimization pack (`jop`) is enabled, those optimizations are applied automatically.

6.2 Adding New Statements

In general, the unit chain allows new statements to be placed before (`insertBefore()`) or after (`insertAfter()`) a specific code point. These must be fully-constructed Jimple units containing all required expressions, i.e., the operands in case of a primitive arithmetic operation. The `Jimple.v()` singleton provides factory methods called `Jimple.v().newX` for generating new Jimple statments and expressions where `X` stands for the different kinds of AST elements. An example is `newStaticInvokeExpr()` which creates a new static invoke expression to be used inside an invoke statement or as the right side of an assignment.

Let us go back to our original SMS Messenger Example (c.f. Section 2.2) and insert some checks that prevent the application from sending SMS messages to premium rate numbers. This check has to be placed before the `sendTextMessage()` method in line 30 and could look like the one described in Listing 1.13 for premium rate numbers that start with 0900.

```
if(!phoneNr.getText().toString().startsWith("0900"))
   smsManager.sendTextMessage(phoneNr.getText().toString(), null,
     message.getText().toString(), null, null);
```

Listing 1.13. 0900 Premium Rate SMS Check

Before this statement can be constructed, various expressions must be generated:

- String constant for the number "0900"
- Method call to the `startsWith()` method. The result must be stored in a new local variable that does not conflict with any existing local variable.
- "if" statement with *then* and *else* branch

A complete example for integrating such a check is described in Listing 1.14.

```
private void eliminatePremiumRateSMS(Unit u, Body body) {
  Stmt stmt = (Stmt) u;
  if (stmt.containsInvokeExpr()){
    InvokeExpr iinv = (InvokeExpr) invoke.getInvokeExpr();
    if(iinv.getMethod().getSignature().equals(SEND_SMS_SIGNATURE)){
       Value phoneNumber = invoke.getInvokeExpr().getArg(0);
       if (phoneNumber instanceof Local){
         Local phoneNoLocal = (Local)phoneNumber;

         // Invoke startsWith and save result
         VirtualInvokeExpr inv = generateStartsWith(body, phoneNoLocal);
         Local invRes = generateNewLocal(body, BooleanType.v());
         AssignStmt astmt = Jimple.v().newAssignStmt(invRes, inv);
         body.getUnits().insertBefore(astmt, u);

         //generate condition
         NopStmt nop = Jimple.v().newNopStmt();
         IfStmt ifStmt = Jimple.v().newIfStmt(invRes, nop);

         body.getUnits().insertBefore(ifStmt, u);
         body.getUnits().insertAfter(nop, u);
       }
    }
  }

private InvokeExpr generateStartsWith(Body body, Local phoneNoLocal) {
  SootMethod sm = Scene.v().getMethod(STARTS_WITH_SIGANTURE);
  return Jimple.v().newVirtualInvokeExpr(phoneNoLocal, sm.makeRef(),
    StringConstant.v("0900"));
}

private Local generateNewLocal(Body body, Type type){
  LocalGenerator lg = new LocalGenerator(body);
  return lg.generateLocal(type);
}
```

Listing 1.14. Generation of Jimple Statements for Premium Rate SMS Check

SEND_SMS_SIGNATURE is a string constant containing the method signature of the *sendTextMessage*. STARTS_WITH_SIGNATURE is the signature of the startsWith() method in the String class.

Note that we do not directly create new locals by giving a name and a type. Instead, we defer this task to the LocalGenerator class which automatically creates a unique local name.

Finally, the eliminatePremiumRateSMS() method has to be called inside the code snipped shown in Listing 1.12 so that the instrumentation is performed for all methods that possibly send SMS messages.

7 Conclusion

In this tutorial paper, we have shown how to instrument Android applications using AspectJ, Tracematches and manual imperative instrumentation based on Soot. All these techniques can also be applied to classical Java programs. For Android, there are a number of platform-specific issues to keep in mind such as the need for signing the APK file before running it on a phone or the emulator.

The techniques shown in this paper can not only be used for security purposes, but also for code optimization and analysis in general. Many optimizations like constant propagation or dead code elimination are already built into Soot, making instrumentations easier for the user.

8 Examples

The SMS Messenger example (RV2013) as well as the instrumentation examples can be downloaded from `https://github.com/secure-software-engineering/android-instrumentation-tutorial`

References

1. International Data Corporation: Worldwide quarterly mobile phone tracker 3q12 (November 2012), `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`
2. Bit9: Pausing google play: More than 100,000 android apps may pose security risks (November 2012), `http://www.bit9.com/pausing-google-play/`
3. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240. ACM (2012)
4. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in android applications. In: Proceedings of the Workshop on Mobile Security Technologies (MoST), in Conjunction with the IEEE Symposium on Security and Privacy (2012)
5. Yang, Z., Yang, M.: Leakminer: Detect information leakage on android with static taint analysis. In: IEEE 2012 Third World Congress on Software Engineering (WCSE), pp. 101–104 (2012)
6. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI 2010, pp. 1–6. USENIX Association, Berkeley (2010)
7. Xu, R., Saïdi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012, pp. 27–27. USENIX Association, Berkeley (2012)
8. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop, CETUS 2011 (October 2011)
9. Allan, C., et al.: Abc: the aspectbench compiler for aspectj. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 10–16. Springer, Heidelberg (2005)
10. Android: Android security overview (December 2012), `http://source.android.com/tech/security/`
11. Goolge Inc.: Google play (December 2012), `https://play.google.com/`
12. Bodden, E., Hermann, B., Lerch, J., Mezini, M.: Reducing human factors in software security architectures. In: Future Security Conference (to appear, September 2013)
13. Oh, H.S., Kim, B.J., Choi, H.K., Moon, S.M.: Evaluation of android dalvik virtual machine. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2012, pp. 115–124 (2012)

14. Google Inc.: Application fundamentals (December 2012), `http://developer.android.com/guide/components/fundamentals.html`
15. Google Inc.: Content provider basics (December 2012), `http://developer.android.com/guide/topics/providers/content-provider-basics.html`
16. Google Inc.: Activity (June 2013), `http://developer.android.com/reference/android/app/Activity.html`
17. Google Inc.: Logcat (June 2013), `http://developer.android.com/tools/help/logcat.html`
18. Google Inc.: Android emulator (June 2013), `http://developer.android.com/tools/help/emulator.html`
19. Google Inc.: Android debug bridge (June 2013), `http://developer.android.com/tools/help/adb.html`
20. Google Inc.: Signing your applications (June 2013), `http://developer.android.com/tools/publishing/app-signing.html`
21. Google Inc.: zipalign (June 2013), `http://developer.android.com/tools/help/zipalign.html`
22. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA 2005, pp. 345–364. ACM, New York (2005)
23. Bodden, E.: Packs and phases in soot (November 2008), `http://www.bodden.de/2008/11/26/soot-packs/`
24. Lam, P., Qian, F., Lhoták, O.: Packs and phases in soot (November 2008), `http://www.sable.mcgill.ca/soot/tutorial/phase/`
25. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
26. Patrick Lam, F.Q., Lhoták, O.: Soot command-line options (June 2013), `http://www.sable.mcgill.ca/soot/tutorial/usage`
27. Arzt, S., Falzon, K., Follner, A., Rasthofer, S., Bodden, E., Stolz, V.: How useful are existing monitoring languages for securing android apps? In: 6. Arbeitstagung Programmiersprachen (ATPS 2013). Lecture Notes in Informatics, Gesellschaft für Informatik (February 2013)

On Signal Temporal Logic

Alexandre Donzé

EECS Dept., University of California, Berkeley
donze@eecs.berkeley.edu

Tutorial Description

Temporal Logic (TL) is a popular formalism, introduced into systems design [Pnu77] as a language for specifying acceptable behaviors of reactive systems. Traditionally, it has been used for formal verification, either by deductive methods [MP95], or algorithmic methods (Model Checking [CGP99,QS82]). In this framework, the behaviors in question are typically discrete, that is, sequences of states and/or events. In recent years, several trends suggest alternative ways to use TL. One is due to the state-explosion wall, which limits the size of systems that can be verified especially when dealing with programs involving numerical computations or hybrid (discrete-continuous systems). As a result we can see a proliferation of statistical methods -la Monte-Carlo, where universal quantification is replaced by random simulation, with and sometimes without statistical coverage guarantees. In this framework, related to *runtime verification*, *assertion checking* or *monitoring*, the temporal formula is still used for a rigorous specification of the requirements, but unlike model-checking, it is evaluated on a *single* behavior at a time. Another trend is concerned with *quantitative* evaluation of TL formula. In many real-life applications, especially when dealing with continuous dynamics and numerical quantities, yes/no answers provide only partial information and could be augmented with *quantitative* information about the satisfaction to provide a better basis for decision making. Such notions have been introduced into TL by Fainekos and Pappas [FP09] and later In [DM10] where notions of robustness both in space and time are described. This tutorial is concerned with *signal temporal logic* (STL), a formalism for specifying properties of dense-time real-valued signals, originally introduced in [MN04]. It will review both the fundamentals of STL and the most recent progress, and will introduce the tool Breach [Don10] for illustration and practical applications. It will be organized into three parts:

Part 1 will provide a general introduction to dense-time and real-valued temporal logics

Part 2 will present the STL monitoring algorithm of [MN04], notions of robust semantics [DM10] and robust monitoring algorithms [DFM13]

Part 3 will review various extensions, namely Parametric STL [ADMN11], Time-Frequency Logic [DMB$^+$12]) as well as different applications in particular in the automative industry [JDDS13] and in systems biology [DFG$^+$11,MDMF12].

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 382–383, 2013.

References

ADMN11. Asarin, E., Donzé, A., Maler, O., Nickovic, D.: Parametric identification of temporal properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 147–160. Springer, Heidelberg (2012)

CGP99. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

DFG$^+$11. Donzé, A., Fanchon, E., Gattepaille, L.M., Maler, O., Tracqui, P.: Robustness analysis and behavior discrimination in enzymatic reaction networks. PLoS ONE 6(9) (2011)

DFM13. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring of signal temporal logic. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013)

DM10. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010)

DMB$^+$12. Donzé, A., Maler, O., Bartocci, E., Nickovic, D., Grosu, R., Smolka, S.: On temporal logic and signal processing. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 92–106. Springer, Heidelberg (2012)

Don10. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010)

FP09. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science 410(42) (2009)

JDDS13. Jin, X., Donzé, A., Deshmukh, J., Seshia, S.: Mining requirements from closed-loop control models. In: HSCC 2013 (2013)

MDMF12. Mobilia, N., Donzé, A., Moulis, J.-M., Fanchon, E.: A model of the cellular iron homeostasis network using semi-formal methods for parameter space exploration. In: HSB (2012)

MN04. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)

MP95. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. In: Manna, Z., Peled, D.A. (eds.) Pnueli Festschrift. LNCS, vol. 6200, pp. 279–361. Springer, Heidelberg (2010)

Pnu77. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)

QS82. Queille, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: 5th Int. Symp. on Programming (1982)

Runtime Verification and Refutation for Biological Systems

Hillel Kugler

Microsoft Research, Cambridge UK
hkugler@microsoft.com

Abstract. Advanced computational models are transforming the way research is done in biology, by providing quantitative means to assess the validity of theories and hypotheses and allowing predictive capabilities, raising an urgent need to be able to systematically and efficiently analyze runtime properties of models. In this tutorial I describe key biological applications, modeling formalisms, property specification languages, and computational tools utilized in this domain, survey the techniques and research from the formal verification, machine learning and simulation communities that are currently being used, and outline opportunities for the runtime verification community to contribute new scalable methods.

1 Introduction

Understanding how biological and ecological systems develop and function remains one of the main open scientific challenges of our times, and is key towards developing treatments for disease, helping conserve biodiversity and addressing global environmental challenges . Computational models are becoming increasingly important in biology and ecology, as a way to formulate hypotheses, assumptions and mechanisms and quantitatively assess whether a theory explains the known data, and make new predictions. As the utility of computational models grows in the biological sciences, it is becoming critical to be able to efficiently test whether a biological model explains known data, what assertions and temporal properties hold for a model, and which initial conditions and environment inputs can lead to a specific desired outcome.

Techniques from runtime verification have the potential to be adapted and utilized to address these challenges, especially since many of the biological models are very complex, and therefore a pure testing approach can easily fail to identify important model behavior, whereas formal verification methods do not scale to many of the real-world models. In this tutorial I will survey the state of the art biological research with significant modeling components in areas of developmental biology, immunology, stem cells, DNA computing and global ecological studies, focusing on the challenges related to run time verification and and model refutation.

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 384–385, 2013.

2 Models in Biology

Models are used intensively by experimental biologists to describe their mechanistic understanding of living processes, however until recently they were most often informal pictures and diagrams representing interactions between different components of a biological system, for example signalling between cells or interactions between proteins within a cell. In recent years, new modeling languages and tools have been introduced, with the aim of providing formal semantics to diagrammatic languages that can be used by biologists to describe their systems and then gain the benefits of executable models. To help deal with complexity, methods and languages from the programming languages community have been introduced and adapted for biological applications, utilizing process calculi, statecharts and rule-based formalisms. Overall progress in the field is allowing the construction of exciting models that are increasingly grounded on biological knowledge, and so offering opportunities for predictive capabilities, together with raising a more urgent need to systematically analyze runtime properties.

3 Model Refutation and Verification

In software and system engineering, a main challenge is to improve the confidence that a system satisfies a given specification, where specifications can be described using, e.g., automata, temporal logic, pre- and post-conditions. In studying natural biological systems, the specification is unknown, so this can be viewed as a reverse engineering problem, where the scientific process aims to construct models and theories about how the system works and identify the specification. A model should be able to explain and reproduce the experiments and data, and should typically not produce runs that are in contradiction with known experiments. Thus effective ways to compare a model against known experimental results and hypotheses is crucial to allow the scientific research. Particular challenges for biological models is that they are highly parallel, include probabilistic decisions representing stochastic elements, and are often very large and computationally intensive to simulate. The runtime verification community can contribute to biological research, by designing new specification formalisms, improving runtime verification methods, and targeting new biological areas including recent developments in the ability to construct computational circuits from biological material [2,3], applications to reprogramming cells for medical applications, and global ecological models [1].

References

1. Purves, D., et al.: Ecosystems: Time to model all life on earth. Nature 493(7432), 295–297 (2013)
2. Qian, L., Winfree, E.: Scaling Up Digital Circuit Computation with DNA Strand Displacement Cascades. Science 332(6034), 1196–1201 (2011)
3. Yordanov, B., Wintersteiger, C.M., Hamadi, Y., Kugler, H.: SMT-based Analysis of Biological Computation. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 78–92. Springer, Heidelberg (2013)

A Lesson on Runtime Assertion Checking with Frama-C

Nikolai Kosmatov and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Runtime assertion checking provides a powerful, highly automatizable technique to detect violations of specified program properties. This paper provides a lesson on runtime assertion checking with Frama-C, a publicly available toolset for analysis of C programs. We illustrate how a C program can be specified in executable specification language e-acsl and how this specification can be automatically translated into instrumented C code suitable for monitoring and runtime verification of specified properties. We show how various errors can be automatically detected on the instrumented code, including C runtime errors, failures in postconditions, assertions, preconditions of called functions, and memory leaks. Benefits of combining runtime assertion checking with other Frama-C analyzers are illustrated as well.

Keywords: runtime assertion checking, program monitoring, executable specification, invalid pointers, Frama-C, e-acsl.

1 Introduction

Runtime assertion checking has become nowadays a widely used programming practice [1]. More and more practitioners and researchers are interested in verification tools allowing to automatically check specified program properties at runtime. *Assertions* offer one of the most convenient and scalable automated techniques for detecting errors and providing information about their locations, even for errors that are traversed during execution but do not lead to failures.

This tutorial paper presents a lesson on runtime assertion checking using Frama-C [2,3], an open-source platform dedicated to the analysis of C programs and developed at CEA LIST. It has an extensible architecture organized as a kernel with several plug-ins for individual analyses which may collaborate with each other. Frama-C offers various analyzers [3] such as control-flow graph construction, abstract-interpretation-based value analysis, dependency analysis, program slicing, automatic test generation, impact analysis, proof of programs, etc.

All static analyzers of Frama-C share a common specification language, called acsl [4]. To bridge the gap between static and dynamic tools, a recent work [5,6] specified e-acsl, an expressive sub-language of acsl that can be

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 386–399, 2013.

a)

```
 1 #include<assert.h>
 2 int absval( int x ) {
 3   return ( x < 0 ) ? x : (-x);
 4 }
 5
 6 void main(){
 7   int n;
 8   n=absval(0);   // test 1
 9   assert(n==0);
10   n=absval(3);   // test 2
11   assert(n==3);
12   // other tests...
13 }
```

b)

```
 1 // returns absolute value of x
 2 int absval( int x ) {
 3   return ( x < 0 ) ? x : (-x);
 4 }
 5
 6 void main(){
 7   int n;
 8   n=absval(0);   // test 1
 9   //@ assert n==0;
10   n=absval(3);   // test 2
11   //@ assert n==3;
12   // other tests...
13 }
```

Fig. 1. Function `absval`, specified **a)** with **`assert`** macros, **b)** with E-ACSL assertions

translated into C, compiled and used as executable specification. An automatic translator into C has been implemented in the E-ACSL plug-in [7] of FRAMA-C.

This paper is organized as follows. Section 2 presents the executable specification language E-ACSL and illustrates how the E-ACSL plug-in can be used to automatically translate the specified C code into an instrumented version suitable for runtime verification of specified properties. Section 3 shows how this solution can be used to automatically detect and report various kinds of errors such as wrong assertions, wrong postconditions, function calls in a wrong context (unsatisfied precondition) and memory leaks. The benefits of combining runtime assertion checking with proof of programs are discussed in Section 4. Section 5 illustrates how existing FRAMA-C analyzers can make runtime assertion checking even more efficient. On the one hand, automatic generation of assertions for frequent runtime errors may help to thoroughly check the program for these kinds of errors without manually writing assertions. On the other hand, some assertions may be statically validated by a static analysis tool, reducing the number of assertions to be checked. Section 6 shows how program monitoring with E-ACSL can be customized in order to define particular actions to be executed whenever a failure is detected. Finally, Section 7 concludes the paper and presents future work.

2 Executable Specifications in E-ACSL

Various ways of specifying assertions have been proposed in the literature [1]. One of the simplest ways to insert an assertion into a C program and to check it at runtime is to use the **`assert`** macro. Unless assertion checks are switched off (usually, by setting the preprocessor option NDEBUG), the condition specified as the argument of the **`assert`** macro is evaluated and whenever it is false, the execution stops and the failure is reported. Fig. 1a illustrates this approach for the function `absval` returning the absolute value of its argument, that is tested by the function `main`. The code contains an error (wrong inequality at line 3) that would be reported by the second assertion check (line 11).

a)

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
       (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int absval( int x ) {
  return ( x >= 0 ) ? x : (-x);
}
```

b)

```
#include<limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing;
    behavior pos:
      assumes x >= 0;
      ensures \result == x;
    behavior neg:
      assumes x < 0;
      ensures \result == -x;
*/
int absval( int x ) {
  return ( x >= 0 ) ? x : (-x);
}
```

Fig. 2. Two ways to specify function absval with an E-ACSL contract

Specifying assertions in a programming language like C has several drawbacks. First, complex properties (e.g. with quantifications and different behaviors) can be difficult to specify and to check, may require significant additional programming effort and appear to be error-prone themselves. Comparing computed values with initial or intermediate ones may need storing some values in additional auxiliary variables. Some arithmetic properties (e.g. absence of overflows after an arithmetic operation) are simpler to express in a specification language with mathematical integers than using bounded machine integers in C. Second, clear separation of the code required only for assertion checking and the functional code may be desirable to optimize performances of the final release, but it often remains manual, even if preprocessor or configuration options may help to exclude assertion related statements in a particular build. Mixing both can make source code more heavy and less readable. Third, absence of unintended side-effects cannot be automatically ensured for assertion checking code written in C. Finally, the usage of resulting C assertions is limited to runtime verification and cannot be extended to verification techniques requiring formal specifications (such as proof of programs).

Other specification formalisms offer more expressive notations compatible with formal verification, for example, Eiffel [8], JML [9], SPEC# [10], SPARK 2014 [11] (see [1] for other references). One of them is the E-ACSL specification language [5,6].

E-ACSL can express the C assertions of Fig. 1a using the **assert** clause as shown in Fig. 1b. However, E-ACSL language is much more expressive than C assertions. Indeed, in addition to C expressions and occasional assertions (like in Fig. 1b), an E-ACSL specification can include function contracts with preconditions and postconditions, behaviors, first-order logic predicates (with quantifications over finite intervals of integer values), mathematical integers (translated into C using GMP library[1] when required), references to the value of a variable or an expression at a particular program point (e.g. a label), memory-related clauses (specifying pointer validity, offset, memory block properties), etc. An E-ACSL clause cannot contain side-effects, so there is no risk to introduce an unintended

[1] http://gmplib.org

```
#include "stdlib.h"

typedef int* matrix;

/*@
requires size>=1;
requires \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(a+i*size+j);
requires \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(b+i*size+j);
ensures \forall integer i,j; 0<=i<size && 0<=j<size ==> \valid(\result+i*size+j);
ensures \forall integer i,j; 0<=i<size && 0<=j<size ==>
  \result[i*size+j] == a[i*size+j]+b[i*size+j];
*/
matrix sum(matrix a, matrix b, int size) {
  int i, j, k, idx;
  matrix c = (matrix)malloc(sizeof(int) * size * size );
  for(i = 0; i < size; i++)
    for(j = 0; j < size; j++) {
      idx = i * size + j;
      c[idx] = a[idx] + b[idx];
    }
  return c;
}
```

Fig. 3. File `sum.c` with function `sum` returning the sum of two given square matrices of given size in a new allocated matrix

modification of program behavior inside annotations. At the same time, specific *ghost code* statements can be used when side-effects are necessary. In addition, E-ACSL has been designed as a subset of ACSL, a specification language for C programs shared by each static analyzer of Frama-C. Therefore, an E-ACSL specification can also be used for instance for proof of programs as illustrated by Section 4.

Since E-ACSL statements are not limited to assertions, we will now use the term *annotation* (rather than *assertion*) to refer to any E-ACSL statement.

Fig. 2 illustrates two equivalent ways to specify the contract of the function `absval`. The contract of Fig. 2a contains a precondition and a postcondition. The precondition (**`requires`** clause) prevents the risk of an overflow since the value -INT_MIN cannot be represented by a machine number of type **`int`**. The postcondition contains an **`ensures`** clause and a frame clause **`assigns`** specifying that the function cannot modify any non-local variables. An equivalent specification using two behaviors is shown in Fig. 2b.

Another example of an E-ACSL contract is given in Fig. 3. Given two square matrices `a` and `b` with `size` rows and `size` columns, function `sum` allocates and returns a new matrix of the same size containing the sum of `a` and `b`. The validity clause **`\valid`**`(p)` specifies that the memory location pointed by `p` is valid. For more detail on C code specification in ACSL, we refer the reader to [4,12].

3 Detecting Errors at Runtime with E-ACSL Plug-In

In this section, we illustrate how the E-ACSL plug-in can be used for runtime verification of C programs. We consider several kinds of errors: failures in asser-

```
#!/bin/sh
share=`frama-c -print-share-path`

frama-c -pp-annot -machdep x86_64 $1 -e-acsl -then-on e-acsl -print -ocode out.c

gcc -DE_ACSL_MACHDEP=x86_64 out.c $share/e-acsl/e_acsl.c
   $share/e-acsl/memory_model/e_acsl_bittree.c
   $share/e-acsl/memory_model/e_acsl_mmodel.c -o runme -lgmp

./runme
```

Fig. 4. Script `check.sh` that instruments the file given in its argument with E-ACSL plug-in, compiles and executes it

tions and postconditions, function calls in a context that does not respect the callee's precondition, potential segmentation faults and memory leaks.

3.1 E-ACSL Plug-In and Assertion Failures

We assume that the Frama-C platform with E-ACSL plug-in[2] and the `gcc` compiler have been installed on the machine. Suppose the file `absval.c` contains the program of Fig. 1b. This program can be instrumented with the E-ACSL plug-in, compiled and executed on a Linux machine by the command `./check.sh absval.c` using the script given at Fig. 4. The options `-machdep x86_64` and `-DE_ACSL_MACHDEP=x86_64` should be used for a 64-bit machine (a 32-bit architecture `x86_32` is currently assumed by default).

The instrumentation translates the E-ACSL specification into executable C code that performs corresponding runtime checks and reports any failure. For Fig. 1b, two assertions at lines 9 and 11 will be checked at runtime exactly as for Fig. 1a. The first one is verified (producing no output), while the second one fails due to a wrong inequality at line 3. The program exits with an explicit message:

```
Assertion failed at line 11 in function main. The failing predicate is: n == 3.
```

Section 6 shows how the user can customize the actions when checking the validity of a predicate.

3.2 Function Contracts

Providing function contracts makes runtime assertion checking even more systematic and powerful. When a function is specified with an E-ACSL contract (with a pre- and/or a postcondition), the E-ACSL plug-in performs their systematic checks on entry (for the precondition) and on exit (for the postcondition) each time the function is called. Suppose we complete the file of Fig. 2a (or Fig. 2b) with a simple function

```c
int main(){
  absval(0);
  absval(3);
  absval(INT_MIN);
  return 0;
}
```

[2] Downloads and installation instructions available at http://frama-c.com/

and instrument and run it using the script of Fig. 4. For each call of `absval`, the precondition is checked before the call and the postcondition is checked after the call without requiring any additional assertions. The checks for the first two calls succeed, while the call `absval(INT_MIN)` violates the precondition, so the program exits and reports:

```
Precondition failed at line 2 in function absval.
The failing predicate is: x > -2147483647-1.
```

If the inequality at line 8 of Fig. 2a was erroneously written again as "<", the program would exit after the call `absval(3)` and explicitly report a postcondition failure. In this way, the preconditions and postconditions are automatically ensured by the instrumented code for each function call. In particular, the precondition check guarantees that the function is called on admissible inputs and prevents from calling it in an inappropriate context. The current release of E-ACSL does not yet support runtime checking of the **assigns** clause, this feature will be integrated in a future version.

3.3 Segmentation Faults

Segmentation faults represent one of the most important issues in C programs. Some of them may lead to runtime errors, others may remain unnoticed and provoke memory corruption. To address this issue, the E-ACSL plug-in provides a memory monitoring library [13] and allows the user to check specified memory-related properties at runtime. Let us illustrate this feature on the program of Fig. 3 completed with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2}; // one element missing, should be {2,2,2,2}
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

The resulting file can be instrumented by the E-ACSL plug-in and run using the script of Fig. 4. The execution reports a failure in the precondition of `sum` at line 8 of Fig. 3 since the array `b` has 3 elements instead of 4 as expected for a 2 × 2 matrix. Thanks to the precondition, E-ACSL prevents an invalid access to `b[3]` at line 18 of Fig. 3 that would be out-of-bounds. This is an example of a *spatial error*, i.e. an invalid memory access due to an out-of-bounds offset or array index. To correct this error, the array `b` must be initialized with 4 elements.

Let us now illustrate some more subtle errors detected by E-ACSL. Replace the line 15 of Fig. 3 by a local array **int** `c[4];` and complete the resulting program with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  int trace = c[0] + c[2];
  free(c);
  return 0;
}
```

Runtime checking with the E-ACSL plug-in for this example reports a postcondition failure at line 9 of Fig. 3 since the elements of c are not valid after exiting the function sum. Thanks to the postcondition, E-ACSL prevents invalid accesses to the elements c[0] and c[2] in function main. This is an example of a *temporal error*, i.e. an invalid memory access to a deallocated memory object.

Another example of a temporal error is a use-after-free. To give an example, we complete the program of Fig. 3 with the function:

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  //@ assert \valid(&c[0]) && \valid(&c[2]);
  int trace = c[0] + c[2];
  return 0;
}
```

Runtime checking with the E-ACSL plug-in for this example reports an assertion failure since the accesses to c[0] and c[2] in function main become invalid after the array is freed.

Notice that memory access validity is only checked when it is specified by (or required for safe translation of) the provided E-ACSL annotations. We show in Section 5 how to enforce memory safety checks with the E-ACSL plug-in by automatic generation of annotation using the RTE plug-in.

3.4 Memory Leaks

Memory leaks represent another common type of defects in programs with intensive dynamic memory allocation. A *memory leak* appears when an inaddressable memory object remains uselessly stored on the system reducing the amount of available memory. In some cases, memory leaks can be a serious problem for programs running for a long time and/or containing frequent memory allocations. When the amount of available memory decreases, the developer may suspect memory leaks. In this case it can be helpful to define (or to bound) the expected difference of the size of dynamically allocated memory at two program points. If the amount of allocated memory increases unexpectedly, additional annotations specifying the difference between closer and closer points may help to find the precise function where the memory leak occurs. The E-ACSL plug-in provides such a means to precisely control the amount of dynamically allocated memory that helps to detect memory leaks.

To illustrate this feature, consider the program of Fig. 5. It includes the file sum.c of Fig. 3 and defines a function sum3 computing the sum of three given matrices. If the program performs frequent calls to such a function sum3, the amount of available memory will decrease. The size of dynamically allocated memory in bytes can be referred by the variable __memory_size (line 2 of Fig. 5) defined by the memory monitoring library of the E-ACSL plug-in. Line 16 of function main illustrates how the user can specify that the amount of dynamically allocated memory must not change between two points, at label L1 before the call to sum3 and at line 16 after deallocating the returned array d. This assertion fails,

```
 1 #include "sum.c"
 2 extern size_t __memory_size;
 3
 4 matrix sum3(matrix a, matrix b, matrix c, int size) {
 5   matrix x, y;
 6   x = sum(a, b, size);
 7   y = sum(x, c, size);
 8   return y;
 9 }
10
11 void main(void) {
12   int a[] = {1,1,1,1}, b[] = {2,2,2,2}, c[] = {3,3,3,3};
13   matrix d;
14   L1: d = sum3(a, b, c, 2);
15   L2: free(d);
16   //@ assert \at(__memory_size,L2) - \at(__memory_size,L1) <= sizeof(int) * 4;
17 }
```

Fig. 5. Function `sum3` returns the sum of three given square matrices of given size

so the user may investigate this difference between two closer points, replacing the assertion at line 16 by another one:

```
//@ assert \at(__memory_size,L2)-\at(__memory_size,L1) <= sizeof(int)*size*size;
```

indicating that only an array of `size * size` integers can be allocated between `L1` and `L2`. This assertion fails again, indicating that the memory leak probably happens inside the function `sum3`. Inserting a postcondition of function `sum3` at line 3

```
//@ ensures __memory_size - \old(__memory_size) == sizeof(int) * size * size;
```

precisely comparing the size of dynamically allocated memory before and after the function call is another way to find out that the memory leak occurs in function `sum3`. The problem is indeed due to the allocation for the array returned at line 6 of Fig. 5 that becomes inaddressable when function `sum3` exits. To solve this issue, `free(x);` should be inserted before the **`return`** statement at line 8 of Fig. 5.

4 Runtime Checking and Analysis of Proof Failures

As we have shown in the previous section, runtime assertion checking helps to ensure that the program execution respects the provided annotations. When the annotations are supposed to be correct, an annotation failure reveals an error in the program. But runtime assertion checking can be also used in a dual way, to find a potentially incorrect annotation. It can also provide more confidence in conformance of the program to its specification when no failures are detected. This approach can be very helpful during proof of programs.

Indeed, in order to formally prove a program, the validation engineer has to specify it. This is a tedious task, and errors in the first versions of specifications are very common. Moreover, when the program proof fails, the proof failure is not necessarily due to a wrong specification or a wrong implementation, but can be also due to the incapacity of the proving tool to find the proof automatically. Proof failures have to be analyzed and fixed manually.

Runtime assertion checking provides an automatic technique allowing the validation engineer to check the conformance between program and specification at runtime on a number of program executions. The instrumented program can be executed on an available test suite, or generated test inputs (e.g. randomly, or by a structural test generation tool like PathCrawler [14], another Frama-C plug-in). Runtime checking does not give a precise answer in all cases, but it can provide a useful indication. When a failure is detected at runtime, the failed annotation and the corresponding program inputs indicate the case that has not been properly taken into consideration in the implementation or in the specification. The engineer can immediately deduce that the proof failure is not due to the limitations of the prover. For example, if the postcondition of the program of Fig. 2a were erroneously written as

```
ensures (x >= 0 && \result == x) && (x < 0 && \result == -x);
```

or as

```
ensures (x >= 0 && \result == x) || (x < 0 && \result == x);
```

the failure would be detected on a concrete program input as illustrated in Sec. 3.2. Although the specification error is obvious for this simple example, automatic runtime checking can save significant time for more complex programs.

When no failure is detected at runtime, the prover may need additional annotations (e.g. assertions, loop invariants), so the specification effort is worth to be continued (even if the risk of an error cannot be completely excluded since runtime checking was performed only on a final set of tests).

5 Combinations with Other Analyzers

Frama-C is a platform which provides a wide variety of plug-ins. The e-acsl plug-in is only one of them. It is possible to make e-acsl more efficient by combining it with other existing plug-ins. Section 5.1 explains how to automatically generate annotations to be checked by e-acsl, while Section 5.2 shows how to reduce the number of dynamic checks by verifying some of them statically.

5.1 Generating Annotations Automatically

The e-acsl plug-in may be used to dynamically verify the absence of runtime errors in C code, by combining it with the rte plug-in. Indeed, this plug-in generates e-acsl annotations preventing potential runtime errors: if these annotations are proven valid, then we get the guarantee that the code provokes no runtime error. For instance, rte generates `/*@ assert \valid_read(p); */` each time a pointer `p` is read, and two assertions `/*@ assert 0 <= i; */` and `/*@ assert i < 10; */` each time an access `t[i]` in an array `t` of length 10 is performed. If these annotations are translated into C code by the e-acsl plug-in, they will be checked at runtime: either a dynamic check fails and the program reports a clear assertion failure, or no dynamic check fails and the whole program does not fail with a runtime error either. Therefore, a potential runtime error does not remain unnoticed thanks to explicit annotations added by RTE and checked by E-ACSL.

Consider for instance the function `sum` of Fig. 3 in which we modify line 18 `idx = i * size + j;` into the incorrect line `idx = i * size + j + 1;`. Since the index `idx` is now too great, that introduces an access out of the bounds of the matrices `a`, `b`, and `c` when computing the sum. We also complete this program with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

Running the RTE plug-in on this program generates several additional annotations corresponding to potential arithmetic overflows while computing `idx` or the sum of the matrices' elements, and potential invalid memory accesses when reading the matrices' elements. You can see them in the Frama-C GUI by running the following command.

```
frama-c-gui -rte sum.c
```

Below are two examples of such annotations.

```
/*@ assert rte: signed_overflow: i*size <= 2147483647; */
/*@ assert rte: mem_access: \valid(c+idx); */
```

We can now combine the RTE and E-ACSL plug-ins to translate these additional annotations (and the already existing ones) into C code in the following way.

```
frama-c sum.c -rte -machdep x86_64 -e-acsl-prepare -then -e-acsl \
        -then-on e-acsl -print -ocode out.c
```

The special option `-e-acsl-prepare` tells Frama-C to generate an E-ACSL-compatible abstract syntax tree (AST). It is required when the computation of the AST is needed by another analysis than E-ACSL (here by the generation of annotations by RTE).

Now, if we compile and run the resulting program `out.c` as shown in Fig. 4, we get the following output which indicates that an assertion preventing a memory-access error failed when dereferencing `c+idx` at line 19.

```
Assertion failed at line 19 in function sum.
The failing predicate is:
rte: mem_access: \valid(c+idx).
```

5.2 Verifying Annotations Statically

Frama-C comes with two static analyzers which try to verify ACSL specifications: Value, based on abstract interpretation [15], and WP, based on weakest precondition calculus [16].

Consider again the function `sum` of Fig. 3 with the following test function.

```
int main(void) {
  int a[]={1,1,1,1}, b[]={2,2,2,2};
  matrix c = sum(a, b, 2);
  free(c);
  return 0;
}
```

Combination with WP Plug-In. Running WP on this program (with the automatic theorem prover Alt-Ergo [17]) automatically proves the preconditions of function `sum` as you can see, for instance, running the Frama-C GUI:

```
frama-c-gui -wp sum.c
```

It does not prove, however, the postconditions of the function: such a proof with WP requires to write loop invariants for both loops of the function. Nevertheless it is possible to combine WP and E-ACSL: WP proves the preconditions[3], while E-ACSL establishes that the postconditions are not violated for a given execution. By default, the E-ACSL plug-in does not perform code instrumentation to check already proved properties: if WP proves the preconditions first, the code generated by E-ACSL performs fewer runtime checks and so is more efficient. Such a combination is run by the following command.

```
frama-c -machdep x86_64 -e-acsl-prepare -wp sum.c -then -e-acsl \
       -then-on e-acsl -print -ocode out.c
```

The generated program `out.c` is then linked and executed as usual. In this particular case, it does not report any error since the program is actually correct.

Combination with Value Plug-In. We can run Value on the same example as follows.

```
frama-c -val sum.c
```

Below is a summary of the output.

```
sum.c:6:[value] Function sum: precondition got status valid.
sum.c:7:[value] Function sum: precondition got status unknown.
sum.c:8:[value] Function sum: precondition got status unknown.
sum.c:19:[kernel] warning: out of bounds write. assert \valid(c+idx);
sum.c:9:[value] Function sum: postcondition got status unknown.
sum.c:10:[value] Function sum: postcondition got status unknown.
```

It indicates than Value automatically proves the first precondition `size >= 1`, but does not prove neither the other ones[4] nor the postconditions. Value also generates an ACSL annotation because it detects a potential out-of-bounds access when writing into the array cell `c[idx]` at address `c+idx` . It does not detect similar out-of-bounds accesses for reading `a[idx]` and `b[idx]` because it assumes that the preconditions of the function are valid.

In the same way as for WP, we can combine Value and E-ACSL to dynamically check with E-ACSL only what remains unproved by Value. E-ACSL will also check the additional annotation generated by Value. Such a combination is run by the following command.

[3] The WP's default memory model assumes that `malloc` never returns NULL.

[4] The Value's default memory model assumes that `malloc` may return NULL.

```
frama-c -machdep x86_64 -e-acsl-prepare -val sum.c -then -e-acsl \
        -then-on e-acsl -print -ocode out.c
```

Linking and executing the generated program in the usual way does not report any failure on this correct program.

6 Customization of Runtime Monitoring

By default, as shown in the previous examples, when the evaluation of a predicate fails at runtime, the execution stops with an error message and exit code 1. This behavior is implemented by the function `e_acsl_assert` provided in the file `e_acsl.c` of the E-ACSL library. This function is called each time an annotation is checked. It is fully possible to modify this behavior by providing your own definition of `e_acsl_assert`. The prototype of the function to implement is as follows.

```
void e_acsl_assert(int, char *, char *, char *, int);
```

For each annotation a to be checked, the parameters are the following:

- the first one is the validity status of a (0 if false, non-zero if true);
- the second one is a string describing the kind of a (an assertion, a precondition, a postcondition, etc);
- the third one is the function name where a takes place;
- the fourth one is a textual description of a;
- the fifth one is the line number of a in the source file.

For instance, Fig. 6 provides an implementation which does not stop the program execution, but appends an error message at the end of file `log_file.log` when an annotation is violated.

Then, in the script of Fig. 4, we can replace the file `$share/e-acsl.c` by the one defining the function of Fig. 6. Thus, executing it on the binary generated from the first example of Section 3.3 generates a file `log_file.log` which contains the following lines:

```
Precondition failed at line 8 in function sum.
The failing predicate is:
\forall integer i, integer j;
  (0 <= i && i < size) && (0 <= j && j < size) ==> \valid((b+i*size)+j).
RTE failed at line 11 in function sum.
The failing predicate is:
mem_access:
  \valid_read(__e_acsl_at_7
  +(long long)((long long)((long long)__e_acsl_i_4*(long long)__e_acsl_at_8)
  +(long long)__e_acsl_j_4)).
```

It indicates that there were actually two failed runtime checks. The former was previously described in Section 3.3 and corresponds to the invalid precondition about the array b (which has 3 elements while 4 are expected). The latter corresponds to an out-of-bound error detected when trying to evaluate the postcondition since `b[i*size+j]` tries to access the fourth element of the array b of length 3 (when $i = 1$, $j = 1$ and $size = 2$).

```
#include <stdio.h>

void e_acsl_assert
  (int predicate, char *kind, char *fct, char *pred_txt, int line)
{
  if (! predicate) {
    FILE *f = fopen("log_file.log", "a");
    fprintf(f,
            "%s␣failed␣at␣line␣%d␣in␣function␣%s.\n\
The␣failing␣predicate␣is:\n%s.\n",
            kind, line, fct, pred_txt);
    fclose(f);
  }
}
```

Fig. 6. Modifying the runtime behavior when an annotation is violated

7 Conclusion and Future Work

In this tutorial paper, we have presented how the E-ACSL plug-in of FRAMA-C can be used to perform runtime assertion checking of C programs. The user expresses the expected properties of the program statements and functions in a powerful formal specification language. These properties are then automatically translated into C code in order to be checked at runtime.

In addition to usual runtime assertion checking, E-ACSL may help to debug specifications before proving a program formally. When combined with other FRAMA-C analyzers, E-ACSL is even more efficient: it may automatically check for any runtime errors in C programs, and may discard runtime checks of properties previously statically verified.

Future work includes the support of missing parts of the E-ACSL specification language, in particular assigns clauses, loop invariants and variants, and logic functions and predicates. It also includes the verification of new kinds of properties like additional memory temporal properties, LTL properties or security properties, and new areas of application like combinations of testing and static analysis, security monitoring and teaching formal specification.

References

1. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes 31(3), 25–37 (2006)
2. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual (April 2013), `http://frama-c.com`
3. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
4. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, v1.6. (April 2013), `http://frama-c.com/acsl.html`

5. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language (May 2013), http://frama-c.com/download/e-acsl/e-acsl.pdf
6. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: The 28th Annual ACM Symposium on Applied Computing (SAC 2013), pp. 1230–1235. ACM (2013)
7. Signoles, J.: E-ACSL User Manual (May 2013), http://frama-c.com/download/e-acsl/
8. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc. (1988)
9. 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)
10. 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)
11. AdaCore and Altran UK Ltd: SPARK 2014 Reference Manual (2013), http://docs.adacore.com/spark2014-docs/html/lrm/
12. Kosmatov, N., Prevosto, V., Signoles, J.: A lesson on proof of programs with Frama-C. Invited tutorial paper. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 168–177. Springer, Heidelberg (2013)
13. Kosmatov, N., Petiot, G., Signoles, J.: Optimized memory monitoring for runtime assertion checking of C programs. In: The 4th International Conference on Runtime Verification (RV 2013). LNCS. Springer (2013) (to appear)
14. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: The 4th Int. Workshop on Automation of Software Test (AST 2009), pp. 70–78. IEEE Computer Society (2009)
15. Cousot, P.R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
16. Dijkstra, E.W.: A constructive approach to program correctness. BIT Numerical Mathematics (1968)
17. Conchon, S., et al.: The Alt-Ergo Automated Theorem Prover, http://alt-ergo.lri.fr/

With Real-Time Performance Analysis and Monitoring to Timing Predictable Use of Multi-core Architectures

Kai Lampka

Uppsala University

Motivation

With future societies and individuals possibly becoming more and more dependent on highly capable (control) systems, the ability to carry out precise system analysis and to enforce the expected behaviour of systems at run-time, i. e., during their operation, could become an irrevocable requirement. –*Who would accept that a computer takes full or partial control over one's car without believing that the system is always reacting as expected?*– Correctness of system designs and their implementations is only one aspect. The other major aspect is the capability to build cost-effective systems. Advances in (consumer) electronics have brought about multi-cored micro-controllers equipped with considerably large memory. This will stimulate the building of embedded (control) systems where multiple applications can be integrated into a single controller and thereby effectively lower the per unit costs of a system as a whole.

The integration of embedded control software brings, via the joint use of hardware, the potential for hidden dependencies between applications. The dependencies can provoke unwanted side effects that are difficult to anticipate and potentially corrupting to the system's behaviour. As an examples to this, one may think of a dual core system with a shared L2-cache. The software executing on the different cores will mutually over-write their cache entries. This in turn, will significantly add to their execution times as code segments and data items must be re-fetched from the main memory. However, this interference does not stop at the level of caches. When fetching items from the main memory, cores need to wait for getting access to the memory. As the Dynamic Memory Access (DMA) controller might implement complex access patterns, on top of the non-deterministically arriving access requests, waiting times may also drastically add to the execution time of the software. For these reasons, it might be the case that a (control) software misses its assumed deadline and a control signal does not reach the actor in time. This in turn can do harm to the overall systems stability and provoke its damage or complete loss, not to mention human casualties.

The challenge inherent to the analysis of such integrated systems is to not drastically over-estimate the execution time of a software component, e.g. by pessimistically bounding the number of cache misses and the waiting times of memory accesses. The challenge inherent to the design of the run-time environment of such integrated system is to not waste compute-capacity, but still

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 400–402, 2013.

guarantee that real-time constraints are met. On top of this, the run-time system must be not too complex as this makes any formal system analysis infeasible and thereby neglects the possibility to demonstrate the formal timing correctness of the system, which is of greatest importance when it comes to the implementation of safety-relevant features by means of electronics, e.g. an anti-blocking system of a car. Overall, this makes the analysis and design of embedded (control) systems extremely challenging, particularly when deploying less predictable multi-core processors, originally designed for the consumer electronics market and not for executing safety-relevant applications.

Contents

This tutorial is concerned with the analysis and run-time support for real-time constrained software executing on multi-core processors with shared resources like caches, memory and intra-core connects. In this setting the challenge is to organize resource use in such a way, that the system is good to analyse, but one avoids waste of compute-capacity or other resources. E.g. static resource arbitration or allocation leads to deterministic system behavior, but may yield low resource utilization which in turn limits the number of applications to be concurrently executed. For addressing these topics the tutorial will present

Part 1: Real-Time Performance Analysis. A modelling and analysis methodology for (formally) analysing real-time constrained software deployed on multi-core architectures. This will contain an introduction to Timed Automata [1] and Real-time Calculus [7] and a in-depth presentation of research results concerned with the modelling and analysis of real-time constraint embedded systems, e.g., [4,5,2].

Part 2: Monitoring of Real-Time Workloads. This part will present run-time mechanisms for ensuring timing correctness (a) in the presence of race conditions and complex arbitration schemes to coordinate access to the shared resource and (b) for carrying out online dynamic power management. The presented material is based on the work presented in [3,6]

The co-development of run-time mechanisms and analysis methods is motivated by the fact that the mechanisms which reduce the non-determinism of the system and its performance model can be exploited within the analysis, as well as within the running system.

The presented techniques support the design safety-critical embedded real-time systems on non-customized multi-core chip designs, originally designed for the consumer-electronic market, rather than for being deployed in a safety system's context.

References

1. Alur, R., Dill, D.L.: Automata For Modeling Real-Time Systems. In: Paterson, M. (ed.) Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990). LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Giannopoulou, G., Lampka, K., Stoimenov, N., Thiele, L.: Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In: Proc. International Conference on Embedded Software (EMSOFT), pp. 63–72. ACM, Tampere (2012)
3. Lampka, K., Huang, K., Chen, J.-J.: Dynamic counters and the efficient and effective online power management of embedded real-time systems. In: Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, pp. 267–276. ACM, Taipei (2011)
4. Lampka, K., Perathoner, S., Thiele, L.: Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. Design Automation for Embedded Systems 14(3), 193–227 (2010)
5. Lampka, K., Perathoner, S., Thiele, L.: Component-based system design: analytic real-time interfaces for state-based component implementations. In: International Journal on Software Tools for Technology Transfer, pp. 1–16 (2012)
6. Neukirchner, M., Lampka, K., Quinton, S., Ernst, R.: Multi-mode monitoring for mixed-criticality real-time systems. Accepted at CODES/ISS (2013)
7. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proc. Intl. Symposium on Circuits and Systems, vol. 4, pp. 101–104 (2000)

Dynamic Analysis and Debugging of Binary Code for Security Applications

Lixin Li[1] and Chao Wang[2]

[1] Battelle Memorial Institute, Arlington, Virginia, USA
[2] Department of ECE, Virginia Tech, Blacksburg, Virginia, USA

Abstract. Dynamic analysis techniques have made a significant impact in security practice, e.g. by automating some of the most tedious processes in detecting vulnerabilities. However, a significant gap remains between existing software tools and what many security applications demand. In this paper, we present our work on developing a *cross-platform interactive analysis* tool, which leverages techniques such as symbolic execution and taint tracking to analyze binary code on a range of platforms. The tool builds upon IDA, a popular reverse engineering platform, and provides a unified analysis engine to handle various instruction sets and operating systems. We have evaluated the tool on a set of real-world applications and shown that it can help identify the root causes of security vulnerabilities quickly.

1 Introduction

Dynamic and symbolic execution based techniques have made a significant impact on analyzing the binary code, e.g. to help automate some of the most tedious and yet nontrivial analysis in security practice. One example is white-box fuzzing [1], where the goal is to systematically generate test inputs to exercise all feasible program paths. Another example is taint analysis [2], where the goal is to track how tainted inputs propagate and trigger security vulnerabilities. In addition, these techniques have been used to detect a broad class of zero-day attacks [3,4] and to generate vulnerability signatures [5] in a honey-pot.

Despite the aforementioned progress, however, there are major limitations in existing techniques that prevent them from being widely adopted. First, there is a lack of support for *interactive analysis*. Current research on dynamic binary analysis focuses primarily on fully automated methods, which is undoubtedly important for applications such as software testing. However, security applications such as malware analysis and exploitation analysis often cannot be fully automated. Although automated analysis can serve as the starting point of another round of deeper analysis, human in the loop is still indispensable. For example, an exhaustive white-box fuzzer can merely exercise all feasible program paths and identify the necessary conditions to trigger software bugs, but cannot decide whether the bugs are exploitable. To decide whether a bug is exploitable, the user needs to refine the input along that path to decide whether it is a security vulnerability. During this process, tools that support interactive analysis would be useful.

Second, there is a lack of support for *cross-platform analysis* by existing tools. This is a burning issue as well because software today runs on an increasingly diverse set

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 403–423, 2013.

of microprocessors and operating systems. Even if a software bug is exploitable on one platform – a specific combination of microprocessor and OS – it is not necessarily exploitable on a different platform, and vice versa. The reason is because a working exploit is often highly dependent on the runtime environment (stack layout, memory model, etc.). Similarly, effective protection, such as address space randomization (ASR), non-executable page, and stack/heap hardening, is also highly dependent on the runtime environment. Unfortunately, existing tools rarely support multiple platforms. For example, ARM based processors are popular in smart phones; many network routers and switchers use PowerPC and MIPS; and embedded devices often use some type of RISC chips. But existing dynamic analysis tools such as TEMU [6] and SAGE [1] focus only on the x86 instruction set.

To bridge the gap, we propose a unified framework for binary code analysis, to support both *interactive* analysis and *cross-platform* analysis. Interactive analysis allows for the user to make an assumption about the target program, and then quickly check for evidence that supports or contradicts that assumption. For example, the user can mark certain memory locations or registers as taint sources and then quickly check for other instructions that are either control-dependent or data-dependent on the taint sources. Since the user often needs to review the same execution scenario repeatedly, e.g. from different angles and in varying degree of details, our tool also supports trace replay augmented with dynamic slicing. Along certain program paths, the user can not only review what has happened but also perform *what-if* analysis: to see whether the program would behave differently if it were to take a different branch or input value. Such analysis is supported by applying *on-demand* symbolic execution using SMT solvers.

To support cross-platform analysis, we adopt a unified binary code intermediate representation (IR) of the target programs, and implement the core analysis algorithms on this IR. We also develop various reverse engineering tools that translate the native execution traces of the program into this IR. Since core analysis algorithms such as symbolic execution and taint analysis are made architecture-independent and OS-independent, the maintenance cost is significantly reduced. This is in sharp contrast to most existing tools, which are all tied to specific instruction set architectures (ISAs) and operating systems (OSs). In our approach, native execution traces from different platforms, together with the native program state, are captured and then translated into the architecture-independent IR. Similarly, the analysis results are mapped back to the native platforms before they are presented to the user.

To the best of our knowledge, such cross-platform interactive analysis framework does not exist before. In addition to symbolic execution and taint analysis, our tool supports *deterministic replay*. More specifically, at the operating system layer, we use a *generic debug breakpoint* based mechanism [7] to support trace generation in user mode, kernel mode, and on real devices. It allows us to avoid the limitations of the existing dynamic binary instrumentation (DBI) tools [8,9] and whole-system emulators [10]. Although there exist many replay systems for binary programs (e.g. [11]), they do not seem to integrate well with mainstream security analysis tools and do not support interactive analysis. For example, there are tools that extend the debugger `gdb` to support replay [12], but do not support taint analysis. Reverse engineering tools such as IDA [13] also support replay but not taint analysis. Without taint analysis, replay itself does not provide enough information about the data relations critical for security analysts. Typically, security analysts need to construct the data flow relations manually.

We have implemented the cross-platform interactive analysis system in the popular IDA Pro tool. New features such as symbolic execution, taint tracking, and replay have been integrated seamlessly with the existing features of IDA Pro. We have evaluated the new tool on a set of real world applications with known vulnerabilities, and demonstrated the effectiveness of the tool.

The remainder of the paper is organized as follows. We provide an overview of our tool in Section 2, and present the cross-platform symbolic execution engine, called CBASS, in Section 3. We present the interactive taint analysis engine, called TREE, in Section 4. We present our experimental evaluation in Section 5, review related work in Section 6, and then give our conclusions in Section 7.

2 System Overview

The proposed system, shown in Fig. 1, consists of the following subsystems:

- CBASS (Cross-platform Binary Automated Symbolic execution System), which separates the platform dependent execution trace generation process from the platform independent analysis process.
- TREE (Taint-enabled Reverse Engineering Environment), which provides a unified replay, debugging, and taint tracking environment, allowing security analysts to form a hypothesis and then check it interactively.
- Front-end subsystems that support both *static processing* and *dynamic tracing*. They translate native traces from different platforms to the common intermediate representation (IR) and map the analysis results back.

We provide a brief description of *static processing* and *dynamic tracing* in this section, while postponing CBASS and TREE to Sections 3 and 4, respectively.

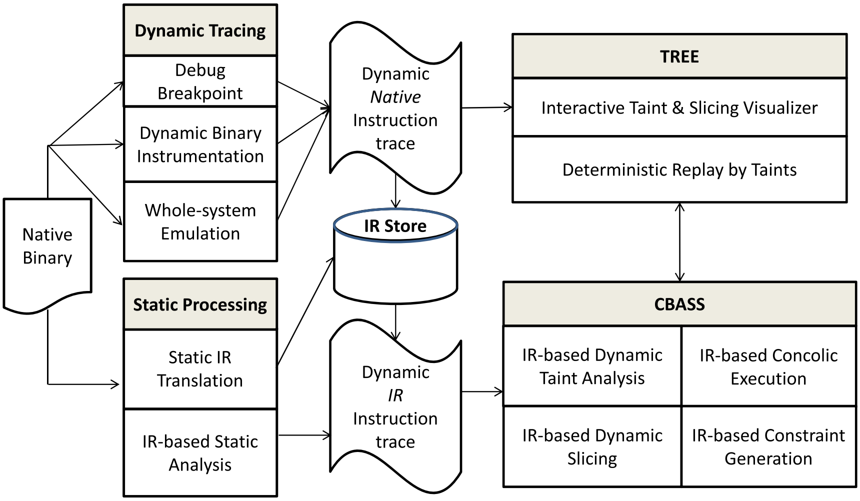
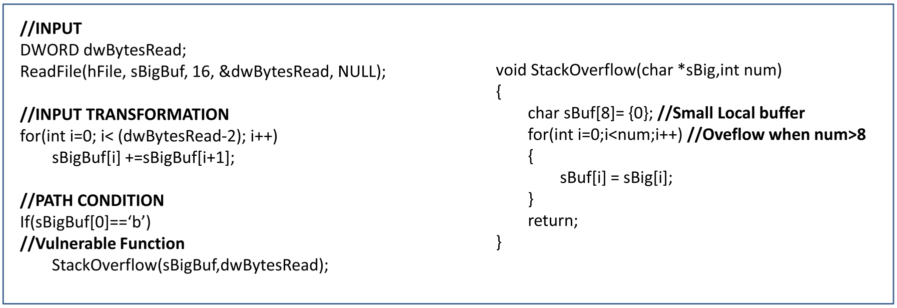

Dynamic Tracing
Debug Breakpoint
Dynamic Binary Instrumentation
Whole-system Emulation
Native Binary
Static Processing
Static IR Translation
IR-based Static Analysis
Dynamic *Native* Instruction trace
IR Store
Dynamic *IR* Instruction trace
TREE
Interactive Taint & Slicing Visualizer
Deterministic Replay by Taints
CBASS
IR-based Dynamic Taint Analysis
IR-based Concolic Execution
IR-based Dynamic Slicing
IR-based Constraint Generation

Fig. 1. The Architecture of our Cross-platform Interactive Analysis System

Static processing and dynamic tracing are crucial components for supporting cross-platform analysis at the instruction set architecture (ISA) level and the operating system (OS) level. ISAs often differ significantly in their encoding and semantics of the instructions. Operating systems often differ in how they use registers to represent high-level data structures. For example, Windows and Linux use `fs` and `gs` segment registers for very different purposes. In our system, however, these differences are mostly removed due to the use of a common IR. In the front-end, only a thin layer needs to deal with remaining subtle differences. In the back-end, all core analysis algorithms are based on the common IR.

We shall use the program called `basicov_plus.exe` in Fig. 2 as the running example. It reads the data inputs from a file and adds each input byte, except for the last two, with its right neighboring byte. If the first byte is `'b'`, the transformed bytes are fed to a vulnerable function called `StackOverflow`. The function is vulnerable in that, if the input is larger than a local buffer inside the function, there will be a buffer overflow, causing the return address to be overwritten. Although the program is small, it consists of all the important elements of a typical security vulnerability: the potentially tainted data source (input), the transformation (addition), the trigger (path condition), and the anomaly manifestation (buffer overflow). In practice, of course, each of these elements can be significantly more complex. For example, the transformation itself may involve not just one instruction but a few millions of instructions.



```
//INPUT
DWORD dwBytesRead;
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

//INPUT TRANSFORMATION
for(int i=0; i< (dwBytesRead-2); i++)
     sBigBuf[i] +=sBigBuf[i+1];

//PATH CONDITION
If(sBigBuf[0]=='b')
//Vulnerable Function
     StackOverflow(sBigBuf,dwBytesRead);
```

```
void StackOverflow(char *sBig,int num)
{
     char sBuf[8]= {0}; //Small Local buffer
     for(int i=0;i<num;i++) //Oveflow when num>8
     {
          sBuf[i] = sBig[i];
     }
     return;
}
```

Fig. 2. Example: A Conditional Buffer Overflow Program

Static Processing. There are two main components for static processing. One component is responsible for pre-processing the binary code statically and building a map from each native instruction to a set of IR instructions. Another component consists of a set of simple static analysis on the resulting IR, e.g. to identify interesting locations that are potential targets of the subsequent dynamic analysis.

Table 1 shows the mapping from a few instructions used by the program in Fig. 2 to the IR instructions. In this table, the native x86 instructions are shown in the first column. The corresponding IR translations are shown in the second column. For example, the native x86 instruction at the address `0x00401073` is mapped to the sequence of REIL instructions from the imaginary address `0x0040107300` to the imaginary address `0x0040107306`. We postpone our detailed presentation of the IR format, called REIL (for reverse engineering intermediate language), to next section. For now, we only show the mapping.

After the REIL IR is constructed, a set of simple static analysis may be conducted. For example, one analysis may be used to measure the Cyclomatic complexity of each function in the IR. The cyclomatic complexity is believed to be useful in identifying a set of functions where bugs most likely hide. Another analysis may be used to detect loops heuristically and annotate the loop counters whenever possible. This is useful because loops, as well as recursive call sites, are places where out-of-bound buffer accesses and non-termination most likely occur.

Table 1. The Mapping from Native Instructions to REIL IR Instructions

Native Instruction (x86)	REIL IR Instruction

00401073 movsx edx, byte ss:[ebp-10]	`40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]`
	`40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1]`
	`40107302: ldm [DWORD t1, EMPTY , BYTE t2]`
	`40107303: xor [BYTE t2, BYTE 0x80, BYTE t3]`
	`40107304: sub [BYTE t3, BYTE 0x80, DWORD t4]`
	`40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5]`
	`40107306: str [DWORD t5, EMPTY , DWORD edx]`

00401077 cmp edx, 0x62	`40107700: and [DWORD edx, DWORD 0x80000000, DWORD t0]`
	`40107701: and [DWORD 98, DWORD 0x80000000, DWORD t1]`
	`40107702: sub [DWORD edx, DWORD 98, QWORD t2]`
	`40107703: and [QWORD t2, QWORD 0x80000000, DWORD t3]`
	`40107704: bsh [DWORD t3, DWORD -31, BYTE SF]`
	`40107705: xor [DWORD t0, DWORD t1, DWORD t4]`
	`40107706: xor [DWORD t0, DWORD t3, DWORD t5]`
	`40107707: and [DWORD t4, DWORD t5, DWORD t6]`
	`40107708: bsh [DWORD t6, DWORD -31, BYTE OF]`
	`40107709: and [QWORD t2, QWORD 0x100000000, QWORD t7]`
	`4010770A: bsh [QWORD t7, QWORD -32, BYTE CF]`
	`4010770B: and [QWORD t2, QWORD FFFFFFFF, DWORD t8]`
	`4010770C: bisz [DWORD t8, EMPTY , BYTE ZF]`

0040107a jnz loc_40108e	`40107A00: bisz [BYTE ZF, EMPTY , BYTE t0]`
	`40107A01: jcc [BYTE t0, EMPTY , DWORD 0x40108e]`

Dynamic Tracing. There are three main components for dynamic tracing. Together, they are responsible for generating a logged execution trace, which will be the starting point of the subsequent offline analysis. Notice that, in our system, there is a clear separation between *online* trace generation and *offline* trace analysis. This makes our trace analysis as platform independent as possible. Among the existing binary analysis tools, some have adopted online analysis [6,14], meaning that the analysis takes place at the time the program is executed, while others have adopted offline analysis [1], meaning that the trace is captured and then analyzed later. However, all of them are tied to a particular platform, making it difficult to maintain and extend to a different platform. In contrast, our system does not have such problems.

In Fig. 1, the components labeled Dynamic Binary Instrumentation and Whole-system Emulation implement the two popular approaches adopted by many existing tools. However, these two components alone doe not meet the demand of our system, for the following reasons. Popular DBI tools, such as PIN and DynamoRIO, provide user mode x86 binary instrumentation but do not support non-x86 ISAs. Valgrind supports non-x86 ISAs such as ARM, PowerPC, and MIPS, but runs only on Linux. None of them provides kernel mode instrumentation. Whole-system emulators can provide kernel instrumentation, but often through an additional instrumentation layer that is not portable to new versions. For example, tools built on the QEMU simulator, such as TEMU [6], DroidScope [15], and S2E [14], have different instrumentation layers. In each case, the implementation is tied to a specific microcode used by QEMU, making it difficult to port. Therefore, although it is well-known that Android builds upon a customized version of QEMU, porting the aforementioned tools to Android is challenging.

In contrast, we propose to use the *debug breakpoint* mechanism [7] for dynamic tracing. This mechanism, already used by interactive debuggers such as `gdb`, is supported by almost all processors and operating systems. Therefore, it provides a unified approach for collecting execution traces from different platforms. It can collect traces in kernel mode. It can also collect traces on real devices such as Cisco routers and Android phones, since almost all of these devices have development tools that provide the breakpoint capability. This *debug breakpoint* approach has significant advantages over DBI tools. Running inside the target process, DBI tools often disturb the behavior of the target program, e.g. by affecting the target's stack and heap layout. This is a serious problem because *interesting* scenarios in security applications tend to manifest only in certain program states.

Our experience shows that breakpoint based tracing is effective for short and interactive analysis. To support long traces, our system leverages existing DBI tools and whole-system emulators, e.g. PIN plug-in for Windows/Linux x86 for trace generation. We have implemented a heuristic algorithm to automatically switch between these techniques, in order to use the best instruction tracer available in each individual application scenario.

Trace Format. The execution trace starts with a snapshot of the program state, which consists of the module, thread, stack, and heap information. The program state is a valuation of the set R of registers for all threads, including privileged registers for kernel mode, and a global memory map M. Therefore, we have the program state represented as $PS = \{R, M\}$.

A tracer on a particular platform would record the finite sequence of *events* starting from the initial state. An event is an execution instance of an instruction that transforms the program state PS into a new program state PS'. Each event in the trace has a unique sequence number. The vast majority of events in a trace are of the form `I = {instInfo, threadID,relevantRegisters, memoryAccess}`, where `instInfo` contains the address of the instruction, the encoding bytes, and the size, `threadId` is the index of the thread that executes this instruction, `relevantRegisters` and `memoryAccess` contain values of the related registers and memory elements before this instruction is executed.

Trace can be optimized to reduce the size while maintaining the same amount of information required by the subsequent analysis. In our implementation, we record only the information that is relevant to the subsequent analysis. For example, for instruction **movsx edx, byte ss:[ebp-10]**, our trace includes the values of registers **edx** and **ebp**. For user mode analysis, we capture the precondition and postcondition of each system call or call to a standard library function as a function summary, to avoid recording the large number of instructions inside the function. For example, after a call to `ReadFile`, we record the address of the input buffer, the input size, and the content of the buffer.

3 Cross-Platform Binary Symbolic Execution System (CBASS)

In contrast to existing symbolic execution tools, where the core analysis algorithms are tied to specific DBI tools or whole-system emulators, CBASS performs symbolic execution on the platform independent REIL IR. This is advantageous because any enhancement to the core analysis algorithms would automatically benefit all platforms.

3.1 The REIL IR

REIL stands for Reverse Engineering Intermediate Language [16]. It is a platform independent intermediate representation of disassembled code, originally designed for supporting static code analysis. We adopt REIL in our system for three reasons:

- Translators for statically mapping the native instruction set to REIL IR are readily available for most of the ISAs, including x86, ARM, PowerPC, and MIPS.
- The REIL instructions are sufficiently close to native instructions on most platforms and therefore can be used to preserve the native register state easily.
- The semantics of REIL instructions can be encoded in SMT formulas precisely by using the bit-vector theory, and therefore is amendable to symbolic analysis.

REIL has only seventeen instructions, each of which has a simple effect on the program state. Each REIL instruction has three operands. The first two operands are always the *source* operands and the last operand is always the *destination* operand. One or more of the operands can be empty. Table 2 summarizes the seventeen REIL instructions. For a more detailed description of REIL, please refer to the online document [17].

Designed for reverse engineering purposes, REIL provides the support to statically translate native instructions in x86, ARM, PowerPC, and MIPS to their IR equivalents for an instruction, a function, or the entire program. More importantly, REIL provides

Table 2. The REIL Instructions and Their Semantics

Category	REIL Instruction	Semantics
Arithmetic	ADD s1, s2, d	d = s1 $+$ s2
	SUB s1, s2, d	d = s1 s2
	MUL s1, s2, d	d = s1 $*$ s2
	DIV s1, s2, d	d = s1$/$s2
	MOD s1, s2, d	d = s1 mod s2
	BSH s1, s2, d	if s2>0 d = s1$*2^{s2}$
		else d = s1 /2^{-s2}
Bitwise	AND s1, s2, d	d = s1& s2
	OR s1, s2, d	d = s1 $\mid$ s2
	XOR s1, s2, d	d = s1 xor s2
Logical	BISZ s1, $\not\epsilon$, d	if s1 = 0, d = 1 else d = 0
	JCC s1, $\not\epsilon$, d	iff s1$\neq$0, set eip = d
Transfer	LDM s1, $\not\epsilon$, d	d = mem[s1]
	STM s1, $\not\epsilon$, d	mem[d] =s1
	STR s1, $\not\epsilon$, d	d = s1
Other	NOP, $\not\epsilon$, $\not\epsilon$, $\not\epsilon$	No op
	UNDEF $\not\epsilon$, $\not\epsilon$,d	Undefined instruction
	UNKN $\not\epsilon$, $\not\epsilon$, $\not\epsilon$	Unknown instruction

a one-to-one mapping of the native instruction address to the imaginary IR address. For example, in Table 1, the x86 instruction **movsx edx, byte ss:[ebp-10]** at address `0x401073` will always be mapped to a list of REIL instructions from `0040107300` to `0040107305`. Therefore, it is easy to map the analysis results back to the native forms before reporting them to the user.

REIL has a simple *register-based* architecture, which can keep native registers and create temporary registers when needed. Preserving native registers is particularly useful for implementing the offline concrete and symbolic (or concolic) execution. Recall that in concolic execution, the program state has to be saved during trace generation and later reconstructed during the offline analysis. At runtime, our trace generator will only save the native program state (related native registers and global memory). During the offline analysis, we can compute the IR program state directly from these native registers and the memory.

In all of the seventeen REIL instructions, the *destination* operand can be represented by a mathematical or logical formula of the *source* operands. Consider the second native instruction **00401077 cmp edx, 0x62** in Table 1. Notice that the REIL instructions use a few basic mathematical and logical operations to precisely compute all the `eflags`; in other words, all the `eflags` can be represented as an expression in terms of `edx` and `0x62`. For example, `ZF = (edx  98) and 0xffffffff`. In some sense, REIL instructions are compatible with the input language of the satisfiability modulo theory (SMT) solver Z3 [18], which supports the theories of integers, bit-vectors, and arrays.

3.2 Symbolic Execution

The symbolic execution procedure consists of three steps:

1. *Mark taint source and symbolize its value.* Here, taint sources refer to the untrusted data in the target program. When a program variable is marked as a taint source, our tool symbolizes the variable, by replacing its concrete value with a symbolic one (a free variable). Traditionally, the taint sources are program inputs. However, during *interactive* security analysis, the user may be interested in tracking other program variables as well. For example, some sensitive data items such as the password and the registry key may become the focus of the analysis. At any time during the program execution, CBASS can mark any byte in any register or at any memory location as the taint source.
2. *Symbolic execution of REIL instructions.* CBASS implements the symbolic execution engine based on the REIL IR. As we have already mentioned, the semantics of REIL instructions can be close to that of the input language of the SMT solvers. Therefore, the symbolic encoding procedure, which takes an IR trace as input and returns an SMT formula, is straightforward. In our implementation of the proposed system, we have used the Z3 SMT solver, which is capable of solving formulas expressed in the theories of bit-vectors and arrays.
3. *Check taint sink to construct constraint.* Depending on the application, security analysts may mark different memory location or register at some interesting point as the taint sinks. For example, to generate potential exploits, the taint sinks are usually registers such as `EIP`. We may create a constraint to steer the execution into a desired code section and make `EIP` equals to the address of that code section.

To detect vulnerabilities, the taint sinks are usually the unexplored branches. When we encounter a branch instruction, we create a path condition if the branch predicate is tainted by a symbolic input.

As shown in Table 2, there are four categories of REIL instructions directly related to symbolic execution. Mathematical and logical instructions perform the corresponding operations on constants, registers, or memory. Memory instructions handle memory read or write operations, which propagate values between registers and memory. Control instructions decide where to jump if the branch conditions are true. During symbolic execution, we use a *concrete and symbolic memory (CSM)* map to represent the memory state. It has both the concrete value and the symbolic value. For memory instructions, if the address is symbolic, also called a symbolic pointer, we have to under-approximate it by using the concrete value derived from the actual execution trace.

3.3 The Running Example

We use the instructions in Table 1 to demonstrate how to construct a path condition during symbolic execution and how to generate the SMT formula. As the IR instructions are fed to the symbolic execution engine, CBASS creates symbolic variables for the taint sources and constructs the symbolic expressions. For each IR instruction, it creates a new symbolic expression for the destination operand if any of the source operands is symbolic. If all the source operands have concrete values, then it uses the concrete value for the destination operand.

Table 3. Example: The REIL IR based Symbolic Execution

Native Instructions	REIL Instructions	Symbolic Execution, with ebp = 0x12ff84 and mem[12ff74] = INPUT
00401073 movsx edx, byte ss:[ebp-10]	40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]	t0 = 0x12ff84+0xfffffff0 = 10012ff74
	40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1]	t1 = t0 and 0xffffffff =0x12ff74
	40107302: ldm [DWORD t1, EMPTY , BYTE t2]	t2 = mem[t1] =INPUT_VAR[8]
	40107303: xor [BYTE t2, BYTE 0x80, BYTE t3]	t3 = INPUT_VAR[8] xor 0x80
	40107304: sub [BYTE t3, BYTE 0x80, DWORD t4]	t4 = (INPUT_VAR[8] xor 0x80) -0x80
	40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5]	t5 = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
	40107306: str [DWORD t5, EMPTY , DWORD edx]	edx = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
00401077 cmp edx, 0x62	40107700: and [DWORD edx, DWORD 0x80000000, DWORD t0]	t0 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff)and 0x80000000
	40107701: and [DWORD 98, DWORD 0x80000000, DWORD t1]	t1 = 98 and 0x80000000 = 98
	40107702: sub [DWORD edx, DWORD 98, QWORD t2]	t2 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) - 98
	Ignore irrelevant temps ...	...
	4010770B: and [QWORD t2, QWORD FFFFFFFF, DWORD t8]	t8= ((((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) 98) and 0xffffffff
	4010770C: bisz [DWORD t8, EMPTY , BYTE ZF]	ZF = ite(t8==0,1,0)
0040107a jnz loc_40108e	40107A00: bisz [BYTE ZF, EMPTY , BYTE t0]	t0 = ite(ZF==0,1,0)
	40107A01: jcc [BYTE t0, EMPTY , DWORD 0x40108e]	eip = ite(t0==1,0x40108e,0x40107c)

Table 3 shows the symbolic execution of the REIL instructions of the three native x86 instructions. Notice that each native instruction is mapped to a sequence of REIL instructions. The REIL instructions take the native registers and memory values as input, transform them by using intermediate registers, and return the results back to the native registers and memory. For example, the instruction at `0x401073` has the native register **ebp** and memory value at address `0x12ff74` as input, and the native register **edx** as output. Just before executing the instruction, the concrete value of **ebp** is assumed to be `0x12ff84` and the memory at the address `0x12ff74` has a symbolic value. From the first two REIL instructions, we have `t1 = 0x12ff74`. The `ldm` instruction sets `t2 = mem[0x12ff74]`, which contains a symbolic value, and then `t2 = INPUT_VAR[8]`.

After carrying out the symbolic execution as shown in Table 3, the branch condition before executing **0040107a jnz loc_40108e** becomes `ite(ite(((((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff)  98) and 0xffffffff)`. This is equivalent to the SMT formula shown in Fig. 3. By negating the path condition and asking the SMT solver for a satisfying solution, we can compute the new input value to be `98`, which corresponds to `sBigBuf[0] == b` in the original code in Fig. 2.

```
(set-logic QF_AUFBV)

(declare-fun _INPUT_VAR () (_ BitVec 8))
(declare-fun EXPR_0 () (_ BitVec 32))
(assert (= EXPR_0 (bvsub ((_ sign_extend 24) (bvxor _ INPUT_VAR (_ bv128 8))) (_ bv4294967168 32))))
(assert (= (ite (not (= (ite (not (= (bvand ((_ extract 63 0) (bvsub ((_ sign_extend 32) (bvand ((_ extract 31 0) EXPR_0)
(_ bv4294967295 32))) (_ bv98 64))) (_ bv4294967295 64)) (_ bv0 64))) ( _ bv1 32) ( _ bv0 32)) (_ bv0 32))) ( _ bv1
8) ( _ bv0 8)) (_ bv0 8)))

(check-sat)

(get-value (_INPUT_VAR))
```

Fig. 3. Example: The Path Constraints in Z3 SMT Formula

4 Taint-Enabled Reverse Engineering Environment (TREE)

To unleash the analysis power of CBASS in security practice, we need to support *interactive* analysis. Toward this end, we have developed the infrastructure that can (1) generate REIL traces on demand, (2) visualize the analysis results on demand, (3) perform taint tracking on demand. Together, these new features form the basis of our taint-enabled reverse engineering environment (TREE).

4.1 Interactive Trace Generation

TREE leverages existing features of IDA, a popular reverse engineering tool, to support on-demand trace generation. IDA is a widely used tool in mainstream security practice. It has become the *de facto* standard tool for conducting vulnerability and malware analysis. IDA can statically disassemble binary code on more than 50 processors and support a wide range of operating systems.

We have implemented the *debug breakpoint* based trace collection framework in IDA and integrated it seamlessly with the existing features of IDA. Our experience shows that the debug breakpoint based approach works well in supporting interactive trace generation, which typically involves short traces. For lengthy traces and large interactive sessions, we rely on the traces generated from the more traditional DBI tools such as PIN, and whole-system emulators such as QEMU.

Compared to the existing tools, the dynamic trace generator in TREE has the following features:

- *Interactive tracing:* The user can select a starting point and an end point at any time during the analysis and request the tool to conduct a deeper analysis on a relatively short trace segment. This feature can be used by security analysts to quickly verify or refute a hypothesis.
- *Kernel tracing:* The trace generator in TREE can generate traces on any platform that supports `windbg` and `gdb` server, allowing kernel mode traces to be generated from both Windows and Linux.
- *Mobile tracing:* The trace generator in TREE can generate traces on Android/ARM platforms through IDA's debug agent. IDA supports real devices such as Android phones and tablets. IDA also supports some versions of iPhone, Windows CE, and Symbian OS, although these platforms have not been integrated with TREE.

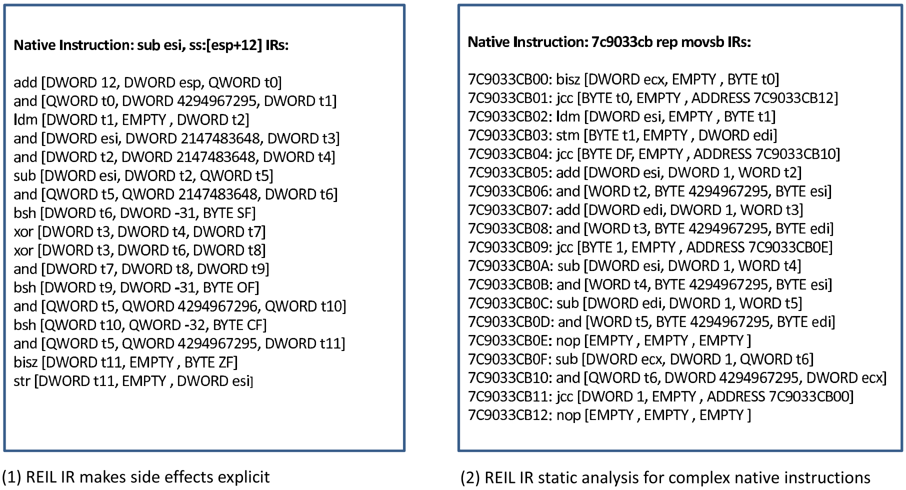
4.2 On-Demand Taint Analysis

Broadly speaking, taint dependencies fall into three categories: data dependency, address dependency, and control dependency.

- Data dependency means that the taint source affects the taint sink through data movement, mathematical operations, or logical operations. The value of the taint source often directly affects the value of the taint sink.
- Address dependency means that the taint source affects the taint sink through its address for read or write, but the taint source does not directly affect the value of the taint sink. One example for address dependency is the use of a tainted data as the index to access a look-up table. Without tracking the address dependency, we would lose track of the tainted data after such a table lookup.
- Control dependency is a form of implicit information flow. Although it can happen in benign programs, it is often more deliberately used by malware. It can be of the form `if x =0 then y=0 else y=1`. If `x` is tainted, the value of `y` is dependent of `x`. But there is no direct link between the value of `x` and the value of `y`.

In security analysis, it is often challenging to keep track of all three types of dependencies. In the remainder of this section, we will show how TREE can make it easier.

The main difficulty in taint tracking for the x86 instruction set is to handle the large number of instructions and their variants, since these native instructions often have complex side effects. REIL provides a unified framework for capturing these side effects, e.g. by breaking down a native x86 instruction into a sequence of simple REIL instructions. Notice that there are only seventeen REIL instructions. Furthermore, each REIL instruction has only one effect, making taint tracking easy to implement. Fig. 4 (1)



```
Native Instruction: sub esi, ss:[esp+12] IRs:

add [DWORD 12, DWORD esp, QWORD t0]
and [QWORD t0, DWORD 4294967295, DWORD t1]
ldm [DWORD t1, EMPTY , DWORD t2]
and [DWORD esi, DWORD 2147483648, DWORD t3]
and [DWORD t2, DWORD 2147483648, DWORD t4]
sub [DWORD esi, DWORD t2, QWORD t5]
and [QWORD t5, QWORD 2147483648, DWORD t6]
bsh [DWORD t6, DWORD -31, BYTE SF]
xor [DWORD t3, DWORD t4, DWORD t7]
xor [DWORD t3, DWORD t6, DWORD t8]
and [DWORD t7, DWORD t8, DWORD t9]
bsh [DWORD t9, DWORD -31, BYTE OF]
and [QWORD t5, QWORD 4294967296, QWORD t10]
bsh [QWORD t10, QWORD -32, BYTE CF]
and [QWORD t5, QWORD 4294967295, DWORD t11]
bisz [DWORD t11, EMPTY , BYTE ZF]
str [DWORD t11, EMPTY , DWORD esi]
```

(1) REIL IR makes side effects explicit

```
Native Instruction: 7c9033cb rep movsb IRs:

7C9033CB00: bisz [DWORD ecx, EMPTY , BYTE t0]
7C9033CB01: jcc [BYTE t0, EMPTY , ADDRESS 7C9033CB12]
7C9033CB02: ldm [DWORD esi, EMPTY , BYTE t1]
7C9033CB03: stm [BYTE t1, EMPTY , DWORD edi]
7C9033CB04: jcc [BYTE DF, EMPTY , ADDRESS 7C9033CB10]
7C9033CB05: add [DWORD esi, DWORD 1, WORD t2]
7C9033CB06: and [WORD t2, BYTE 4294967295, BYTE esi]
7C9033CB07: add [DWORD edi, DWORD 1, WORD t3]
7C9033CB08: and [WORD t3, BYTE 4294967295, BYTE edi]
7C9033CB09: jcc [BYTE 1, EMPTY , ADDRESS 7C9033CB0E]
7C9033CB0A: sub [DWORD esi, DWORD 1, WORD t4]
7C9033CB0B: and [WORD t4, BYTE 4294967295, BYTE esi]
7C9033CB0C: sub [DWORD edi, DWORD 1, WORD t5]
7C9033CB0D: and [WORD t5, BYTE 4294967295, BYTE edi]
7C9033CB0E: nop [EMPTY , EMPTY , EMPTY ]
7C9033CB0F: sub [DWORD ecx, DWORD 1, QWORD t6]
7C9033CB10: and [QWORD t6, DWORD 4294967295, DWORD ecx]
7C9033CB11: jcc [DWORD 1, EMPTY , ADDRESS 7C9033CB00]
7C9033CB12: nop [EMPTY , EMPTY , EMPTY ]
```

(2) REIL IR static analysis for complex native instructions

Fig. 4. TREE Uses REIL IR for Comprehensive Taint Analysis

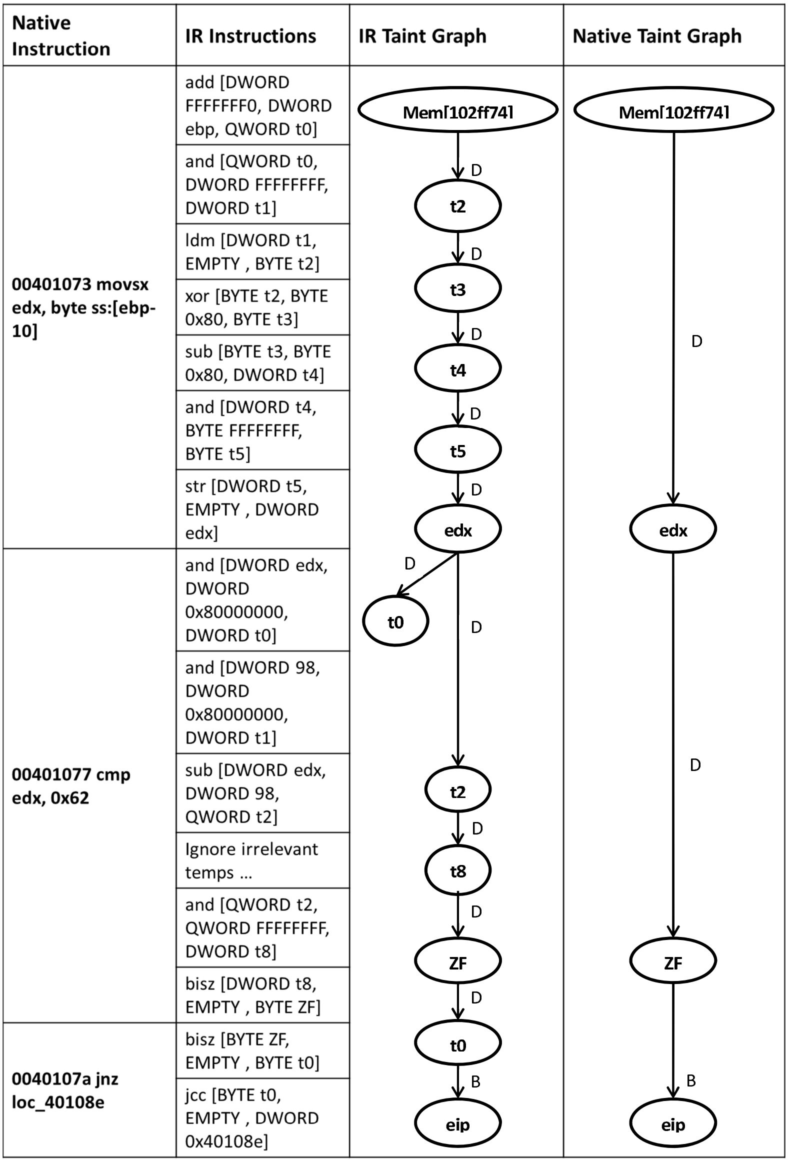
shows a comparison of the native x86 instructions and the corresponding REIL instructions. The REIL instructions capture the side effects of the native instructions on `eflags` including `SF`, `OF`, `CF` and `ZF`.

REIL also supports static analysis that can provide hints for dynamic analysis. They can be useful for x86 instructions that have embedded conditions or loop structures. For example, `cmpxchg` compares the values in the `AL`, `AX` or `EAX` registers with the *destination* operand, and depending on the comparison result, different operands may be loaded into the *destination* operand. Some x86 instructions with prefix such as `rep` behave like a loop. Fig. 4 (2) shows the REIL instructions for x86 instruction `rep movsb`. Since dynamic analysis can only follow one path at a time, in general, it cannot handle the branch and loop dependency. However, a conservative static analysis on REIL IR often can reveal the branch and loop structure. This is the case for `rep movsb` where such analysis can identify `ecx` as the loop counter. We have incorporated such analysis into our REIL-based dynamic taint analysis.

We use the same example for CBASS symbolic execution to show the major steps in dynamic taint analysis. Fig. 5 shows the details of this algorithm. After merging the temporary register nodes, the final taint graph for native instructions is shown in the last column of this table.

4.3 Replay with Taint-Enabled Breakpoints

In an interactive analysis session, the user may want to scrutinize a particular program behavior repeatedly. TREE provides a replay mechanism to support such analysis. One application is to reconstruct the execution states. Comparing to tools such as `gdb` and `IDA`, the replay mechanism in TREE is significantly more powerful. For example, it allows the user to break at any tainted points, after the user marks the initial taint source and specifies the type of impact (taint policy). This new feature of *break by data relation* is key to interactive analysis. It essentially allows the user to break at any point that she



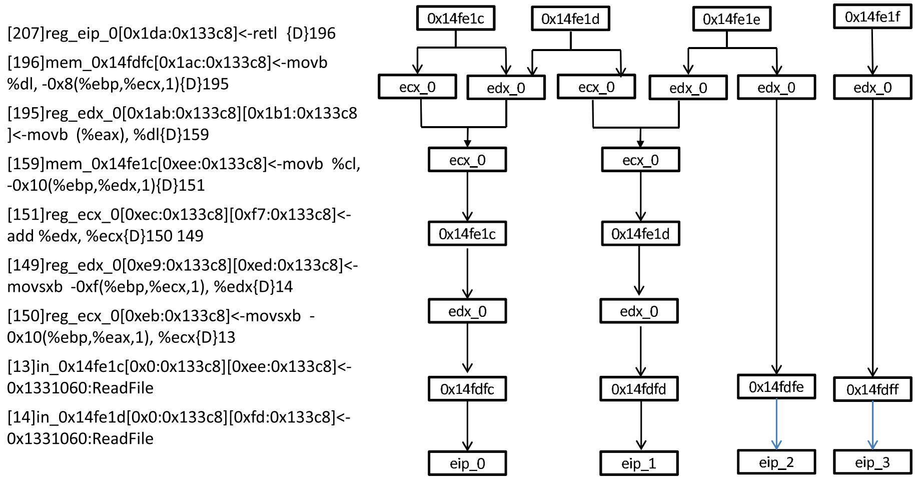
Native Instruction	IR Instructions
00401073 movsx edx, byte ss:[ebp-10]	add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]
	and [QWORD t0, DWORD FFFFFFFF, DWORD t1]
	ldm [DWORD t1, EMPTY , BYTE t2]
	xor [BYTE t2, BYTE 0x80, BYTE t3]
	sub [BYTE t3, BYTE 0x80, DWORD t4]
	and [DWORD t4, BYTE FFFFFFFF, BYTE t5]
	str [DWORD t5, EMPTY , DWORD edx]
00401077 cmp edx, 0x62	and [DWORD edx, DWORD 0x80000000, DWORD t0]
	and [DWORD 98, DWORD 0x80000000, DWORD t1]
	sub [DWORD edx, DWORD 98, QWORD t2]
	Ignore irrelevant temps …
	and [QWORD t2, QWORD FFFFFFF, DWORD t8]
	bisz [DWORD t8, EMPTY , BYTE ZF]
0040107a jnz loc_40108e	bisz [BYTE ZF, EMPTY , BYTE t0]
	jcc [BYTE t0, EMPTY , DWORD 0x40108e]

Fig. 5. Example: Dynamic Taint Analysis

is interested, without the need to construct the chain of events mentally. In addition, TREE can presents the chain of events within the proper semantic context visually.

We illustrate the replay process by using the same buffer overflow example in Fig. 2. When this program runs with a 16-byte input that triggers the `StackOverflow` function, the input bytes at offsets 13 to 16 would overwrite the `EIP` bytes. This chain of events can be tracked by TREE, for which a user-clickable graph is shown in Fig. 6. In this graph, each node represents a byte, annotated by its transformation instruction and followed by its edge type. D is the default edge type that stands for data dependency. The first byte of `EIP` (id =207) is overwritten by input bytes 13 and 14 (id=13,14) after a few steps.

First, these two bytes are added to form a new byte at memory `mem_0x14fe1c(id =159)`. Then the byte is moved to a local buffer at `0x14fdfc` and overflowed the buffer at function `stackOverflow()`. When the call to this function returns, the byte, at the top of the stack at `mem_0x14fdfc[id=196]` is popped into the first byte of register `EIP [id =207]`. For this trivial example, there are already `477` instructions logged in the trace, but only 8 unique instructions are involved in the handling of the input bytes. In such cases, the taint graph allows the user to focus on the most relevant set of instructions quickly.



```
[207]reg_eip_0[0x1da:0x133c8]<-retl  {D}196
[196]mem_0x14fdfc[0x1ac:0x133c8]<-movb
%dl, -0x8(%ebp,%ecx,1){D}195
[195]reg_edx_0[0x1ab:0x133c8][0x1b1:0x133c8
]<-movb  (%eax), %dl{D}159
[159]mem_0x14fe1c[0xee:0x133c8]<-movb  %cl,
-0x10(%ebp,%edx,1){D}151
[151]reg_ecx_0[0xec:0x133c8][0xf7:0x133c8]<-
add %edx, %ecx{D}150 149
[149]reg_edx_0[0xe9:0x133c8][0xed:0x133c8]<-
movsxb  -0xf(%ebp,%ecx,1), %edx{D}14
[150]reg_ecx_0[0xeb:0x133c8]<-movsxb  -
0x10(%ebp,%eax,1), %ecx{D}13
[13]in_0x14fe1c[0x0:0x133c8][0xee:0x133c8]<-
0x1331060:ReadFile
[14]in_0x14fe1d[0x0:0x133c8][0xfd:0x133c8]<-
0x1331060:ReadFile
```

Fig. 6. Taint Graph and Visualization of Running Example

5 Evaluation

We have implemented the proposed *cross-platform interactive analysis* system using the client/server architecture. More specifically, CBASS runs as the back-end server, responding to requests from the front-end. It shares the REIL IR with TREE. TREE is responsible for handling OS level differences and mapping the analysis results back to the native instruction context. The client/server architecture enables parallel development and optimization of CBASS and TREE, and makes it easy to port either subsystem to a different platform without affecting the other.

Currently, CBASS and TREE are able to run on Windows and Linux, and support target programs running on the x86 and Android/ARM platforms. CBASS is written in Jython, a Python-based language that can access Java objects and call Java libraries. CBASS interfaces with REIL through the REIL Java library for IR translation. TREE is implemented as an `IDA Pro` plug-in. TREE also uses Qt/Pyside and extends the IDA graph to support a number of visualization features and user interaction. During the process of developing TREE, we have found a number of bugs in both IDA and REIL related tools. In most cases, the IDA and REIL developers have responded to our bug reports promptly and provided fixes in their latest releases.

In the remainder of this section, we will first provide an overview of our detailed evaluation and then present a case study with a real-world application. Together, they demonstrate the effectiveness of our system in supporting cross-platform interactive security analysis.

5.1 Overview

We have conducted two sets of experiments. The first set consists of unit level tests for the CBASS and TREE subsystems. The second set consists of case studies using real-world applications. At the unit testing level, we have used a large number of binary programs (each around 100 LOC) to check if the core analysis algorithms in TREE/CBASS are implemented correctly. We have designed various transformation functions to process the input (taint source) and created the corresponding test oracles to ensure that TREE and CBASS produce correct results. The test programs are compiled on different platforms (Windows, Linux, and Android) using different compilers (VC, GCC) with various optimization settings. This also allows us to evaluate the effectiveness of our front-end subsystems, which are crucial for the cross-platform analysis.

With real-world applications, the goal of our case study is to evaluate the effectiveness of TREE/CBASS in analyzing vulnerabilities. More specifically, we would like to know whether security analysts, armed with our tool, can quickly discover the chain of critical events leading to the real vulnerability. Toward this end, we have selected a set of Windows/Linux applications with known vulnerabilities. Table 4 shows the statistics of the benchmark programs. In the following, we shall briefly describe each vulnerability and then focus on using WMF (CVE-2005-4560) to explain in details how TREE/CBASS can help reduce the analysis time required to identify the root cause.

The first two columns in Table 4 show the application name, version, and vulnerability identifier. Both the WMF (CVE-2005-4560) and the ANI (CVE-2007-0038) vulnerabilities were present on many Windows versions prior to Windows Vista, and could be triggered by applications including Picture and Fax Viewers, Internet Explorer, Windows Explorer, and various email viewers. Audio Code 0.8.18 has a buffer overflow vulnerability that can be triggered when adding a crafted play list (.lst) file. This vulnerability can enable arbitrary code execution. Streamcast 0.9.75 has a stack buffer overflow, allowing attackers to use the http `User-Agent` field to overwrite the return address of a function call. POP Peeper 3.4.0.0, an email agent, has a vulnerability in

Table 4. Results of Our Analysis on Real World Vulnerabilities

Program Name and Version	Vulnerability Identifier	Binary Code and Trace Size(KB)	Taint Sources (Byte)	Total/Unique Instructions	Total/Unique Tainted Inst.
GDI32.dll 5.1.2600.2180	CVE-2005-4560	272 / 2,422	68	76,618 / 5,677	206 / 115
User32.dll 5.1.2600.2180	CVE-2007-0038	564 / 53,548	4,385	250,534 / 23,868	7,195 / 1,043
AudioCoder 0.8.18	OSVDB-2939	731 / 29,000	620	473,922 / 27,265	12,666 / 66
Streamcast 0.9.75	CVE-2008-0550	804 / 26,541	1,230	83,204 / 3,354	8,351 / 35
POP Peeper 3.4.0.0	BugTraq-34192	1,436 / 68,731	400	182,382 / 8,226	1,106 / 2
PEiD 0.95	OSVDB-94542	214 / 14,163	1,000	32,779 / 9,501	25 / 20
SoulSeek 157	ExploitDB-8777	3,410/147,931	49	4,435,526/142,220	217/121
SoX 12.17.2	CVE-2004-0557	225 /14,441	1,184	180,034 / 2,801	56,138 / 647

its `From` field, where the stack buffer can overflow to overwrite the return address and the Windows Structural Exception Handler (SEH). PEiD is a popular tool for detecting packers, cryptors and compilers found in PE executable files. A carefully crafted EXE file can be used to exploit this vulnerability to run arbitrary code. SoulSeek 157 NS12d, a free file sharing application, has a vulnerability that can be remotely exploited to overwrite SEH. SoX (Sound eXchange) is a sound processing application in Linux. Its `WAV` header handling code has a known buffer overflow vulnerability that can be exploited by the attacker to execute arbitrary code.

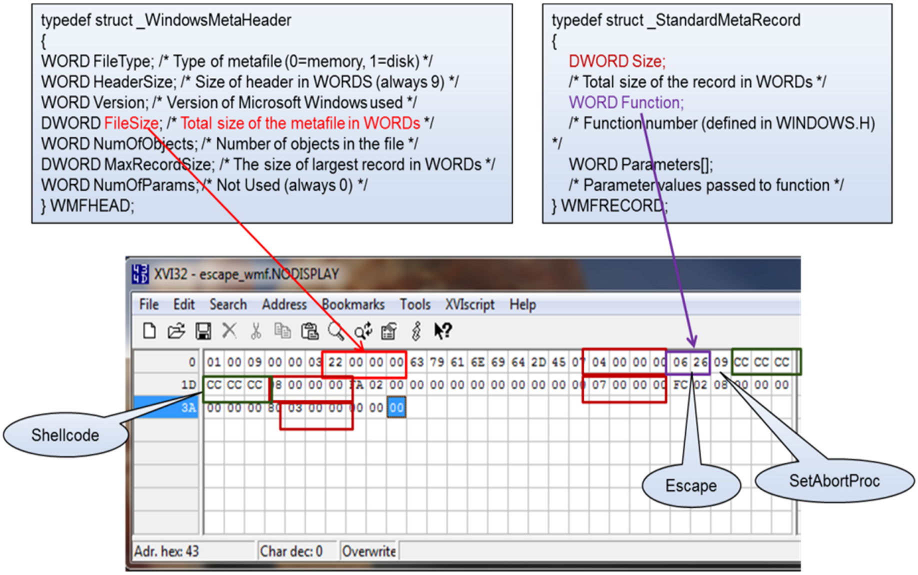
The third column in Table 4 shows the size of the binary code and the size of the trace, respectively. Recall that the on-demand trace logging starts when the target program reads the taint source (input in all these test cases), and stops when the tainted data have taken control of program, e.g. when `EIP` contains a tainted value or the program jumps to the tainted memory location. The fourth column shows the number of bytes of the taint sources, ranging from a few dozen bytes to a few thousand bytes. For all cases, CBASS/TREE can successfully build the taint graph previously described.

For any specific taint sink, the CBASS/TREE system can generate a slice of the tainted instructions from the taint sources to the taint sink. The last two columns in Table 4 show the total and unique instructions in the trace, and the total and unique tainted instructions for all the tainted sources and sinks, respectively. In general, tainted instructions are only a very small portion of the total instructions ($<5\%$). For any specific byte of the tainted target, for example, a tainted register or a tainted memory location, usually only a few dozen tainted instructions are involved.

For more real world vulnerabilities to which we have applied TREE/CBASS, please refer to http://code.google.com/p/tree-cbass/. We will continue our ongoing evaluation process and update the results on this website.

5.2 Case Study: WMF (CVE-2005-4560)

In this section, we will illustrate how TREE/CBASS can support interactive security analysis by using CVE-2005-4560, also known as the `WMF SetAbortProc Escape` vulnerability. WMF stands for Windows Metafile Format. The formal specification of



```
typedef struct _WindowsMetaHeader
{
WORD FileType; /* Type of metafile (0=memory, 1=disk) */
WORD HeaderSize; /* Size of header in WORDS (always 9) */
WORD Version; /* Version of Microsoft Windows used */
DWORD FileSize; /* Total size of the metafile in WORDs */
WORD NumOfObjects; /* Number of objects in the file */
DWORD MaxRecordSize; /* The size of largest record in WORDs */
WORD NumOfParams; /* Not Used (always 0) */
} WMFHEAD;
```

```
typedef struct _StandardMetaRecord
{
   DWORD Size;
   /* Total size of the record in WORDs */
   WORD Function;
   /* Function number (defined in WINDOWS.H)
*/
   WORD Parameters[];
   /* Parameter values passed to function */
} WMFRECORD;
```

XVI32 - escape_wmf.NODISPLAY

```
 0  01 00 09 00 00 03 22 00 00 00 63 79 61 6E 69 64 2D 45 07 04 00 00 00 06 26 09 CC CC CC
1D  CC CC CC 08 00 00 00 FA 02 00 00 00 00 00 00 00 00 00 00 07 00 00 00 FC 02 08 00 00 00
3A  00 00 00 80 03 00 00 00 00 00
```

Adr. hex: 43 Char dec: 0 Overwrite

Fig. 7. Case Study: The WMF Key Data Structures

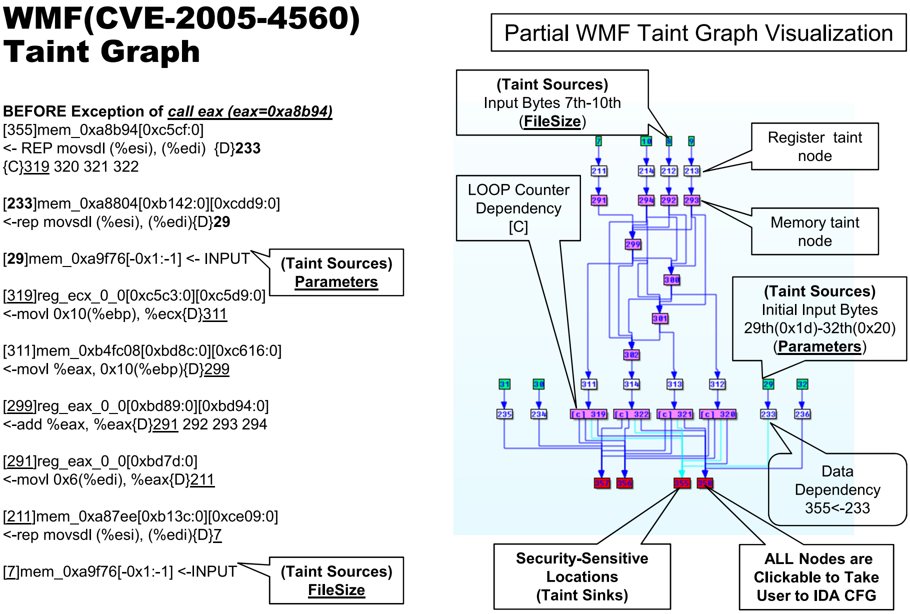
WMF is very complex. In short, the overall WMF file structure has one meta header, followed by zero or more meta records. The key structure of the WMF file format is shown in Fig. 7.

Each meta data record is an encoded Windows GDI (Graphics Device Interface) function call. It is a means of storing and playing back the command sequence that normally would be sent to GDI to display graphics. Among the meta records, one type is called the *escape* record. Although this type of record is deprecated, the code that handles the record has not been removed in a timely fashion. If an *escape* record contains certain values for the Function (0626) and Parameters (09) fields defined in the WMFRECORD structure, the SETABORTPROC escape will inform GDI to call a function provided in the file. This vulnerability allows remote attackers to execute arbitrary code via a WMF format image with a crafted SETABORTPROC GDI Escape function call, related to the Windows Picture and Fax Viewer (SHIMGVW.DLL). It is relatively easy to craft a WMF image file and cause the viewer application to crash.

The lower part of Fig. 7 shows an WMF file with 68 bytes. From the time the viewer program finishes reading the file to the point where an exception happens, 76,618 instructions would be executed. Given that most people do not know WMF format well, we can assume that it is difficult to manually identify which bytes of the WMF file are responsible for the crash, how many instructions are directly involved in rendering the file, from which functions, and under what condition. Without such information, it would be difficult to understand the root cause of this vulnerability. From the exploit development point of view, it would not be obvious which input bytes are critical to a working exploit, and what are the constraints a working exploit must satisfy.

With the dynamic analysis techniques provided by TREE/CBASS, we are able to answer the aforementioned questions in a few minutes. More specifically, the tool can generate a trace that leads to the crash. Furthermore, it can replay the trace by first marking the whole 68 bytes of the file as the taint sources, and then stopping at the tainted points. From the taint graph, we are able to see the connection between the instruction that caused the crash (call `eax` where `eax = 0xa8b94`) and some of the file structures. We have identified 12 unique instructions in WMF that are directly related to moving and processing the file and causing the application to crash. Since our tool can generate an interactive graph, the user can navigate along the chained data and instructions by clicking on each tainted node in the graph.

Fig. 8 shows part of the WMF crash taint graph. The right side is a snapshot taken from the TREE GUI. The nodes in green show the taint source bytes (WMF file), and the nodes in red show the bytes pointed by `eax` in the `call eax` instruction that caused an exception. The left side of the figure shows some internal text representation of the taint graph. For example, the node `355` shows the tainted node of `0xa8b94`. Following the `D` link (highlighted in bold), we can see that it is data-dependent on node 233, which in turn is data-dependent on node 29, an input byte that corresponds to part of the shell-code section. Following the `C` link (highlighted by underline), we can see that it is affected by a loop whose iteration number depends on the values from the 7th to the 10th bytes in the WMF file. When looking back at the `WMFHead` structure, we find that bytes 7-10 actually correspond to the `FileSize` field.



WMF(CVE-2005-4560)
Taint Graph

BEFORE Exception of *call eax (eax=0xa8b94)*
[355]mem_0xa8b94[0xc5cf:0]
<- REP movsdl (%esi), (%edi) {D}**233**
{C}319 320 321 322

[**233**]mem_0xa8804[0xb142:0][0xcdd9:0]
<-rep movsdl (%esi), (%edi){D}**29**

[**29**]mem_0xa9f76[-0x1:-1] <- INPUT

(Taint Sources)
Parameters

[319]reg_ecx_0_0[0xc5c3:0][0xc5d9:0]
<-movl 0x10(%ebp), %ecx{D}311

[311]mem_0xb4fc08[0xbd8c:0][0xc616:0]
<-movl %eax, 0x10(%ebp){D}299

[299]reg_eax_0_0[0xbd89:0][0xbd94:0]
<-add %eax, %eax{D}291 292 293 294

[291]reg_eax_0_0[0xbd7d:0]
<-movl 0x6(%edi), %eax{D}211

[211]mem_0xa87ee[0xb13c:0][0xce09:0]
<-rep movsdl (%esi), (%edi){D}7

[7]mem_0xa9f76[-0x1:-1] <-INPUT

(Taint Sources)
FileSize

Partial WMF Taint Graph Visualization

(Taint Sources)
Input Bytes 7th-10th
(**FileSize**)

Register taint node

Memory taint node

LOOP Counter Dependency [C]

(Taint Sources)
Initial Input Bytes 29th(0x1d)-32th(0x20)
(**Parameters**)

Data Dependency 355<-233

Security-Sensitive Locations (Taint Sinks)

ALL Nodes are Clickable to Take User to IDA CFG

Fig. 8. Case Study: The WMF Crash and Taint Graph

6 Related Work

Independently, Heelan and Gianni [19] have explored the idea of supporting manual vulnerability detection in their work called Pinnacle. However, Pinnacle is limited to taint tracking on the x86 instruction set only. In contrast, our system can handle binary code from multiple platforms. Furthermore, our interactive analysis is significantly broader than the scope of Pinnacle, including not only vulnerability analysis but also exploitation analysis and malware analysis. Our system also supports symbolic execution and replay, which Pinnacle does not. Among the offline binary analysis tools, SAGE [1] is the closest to ours. However, SAGE is designed primarily for white-box fuzzing and works only for the x86 instruction set. It does not focus on interactive analysis and does not support multiple platforms.

Since dynamic taint analysis is independent of the vulnerability specific details, it can analyze a broad class of attacks controllable via input. Therefore, it has become a popular technique for detecting attacks such as buffer overflow and control-flow hijacking. However, online taint analysis often has high runtime overhead and requires intrusive code instrumentation. To make taint analysis more efficient for online intrusion detection, Sekar proposed taint inference [20] for web applications by using approximate string match. Li and Sekar [21] later demonstrated that taint inference could be used to detect buffer-overflow attacks in low-level binary code.

Dytan [2] extended the data-flow based taint tracking to also include control dependency, and developed a framework to support the x86 instruction set. Ganai *et al.* [22] extended this framework to support multithreaded applications. Predictive dynamic analysis provides a new way of conducting trace-based analysis for multithreaded applications [23]. It can detect not only security vulnerabilities in the observed execution traces, but also security vulnerabilities that may appear in some alternative thread interleavings. Wang and Ganai [24] developed a tool for predicting concurrency failures in the generalized execution traces of x86 executables.

Newsome and Song proposed TaintCheck [4], which used dynamic taint analysis for detecting vulnerabilities and for generating vulnerability signatures. TaintCheck was implemented using Valgrind [9]. Portokalidis *et al.* developed Argos [5] based on QEMU to generate fingerprints for zero-day attacks. However, none of these existing tools supports cross-platform interactive security analysis.

7 Conclusions

We have presented a *cross-platform interactive analysis* framework, which integrates state-of-the-art dynamic analysis techniques with a mainstream reverse engineering tool to meet the demand in security practice. Our framework, comprising CBASS and TREE, supports interactive analysis through on-demand symbolic execution and taint tracking. It also supports cross-platform analysis, by separating online trace generation from offline trace analysis and by using a reverse engineering intermediate representation. We have implemented the proposed framework and conducted some preliminary experimental evaluation. Our results have demonstrated its effectiveness in identifying root causes of security vulnerabilities in real applications.

Acknowledgments. We would like to thank James Just for his guidance throughout this project, Xing Li for implementing the IDA debug breakpoint based tracer, and Loc Nguyen for implementing the taint graph visualization. We would like to thank Ilfak Guilfanov and the IDA team for promptly fixing the bugs that we have reported to them and for their suggestions on the GUI integration. We would like to thank Thomas Dullien and Tim Kornau of the Google Zynamics team for making their latest version of REIL available to us. The second author is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

References

1. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (2008)
2. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA, pp. 196–206 (2007)
3. Costa, M., Crowcroft, J., Castro, M., Rowstron, A.I.T., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worm epidemics. ACM Trans. Comput. Syst. 26(4) (2008)
4. Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: NDSS (2005)
5. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: EuroSys, pp. 15–27 (2006)
6. Song, D., et al.: BitBlaze: A new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
7. Paxson, V.: et al.: A survey of support for implementing debuggers (1990), `ftp.ee.lbl.gov:papers/debugger-support.ps.Z`
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: PIN: Building customized program analysis tools with dynamic instrumentation. In: PLDI, pp. 190–200 (2005)
9. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. Electr. Notes Theor. Comput. Sci. 89(2) (2003)
10. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, 41–46 (2005)
11. Bhansali, S., Chen, W.K., De Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: International Conference on Virtual Execution Environments, pp. 154–163. ACM (2006)
12. GNU GDB: Process Record & Replay, `http://sourceware.org/gdb/wiki/ProcessRecord`
13. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, San Francisco, CA, USA (2008)
14. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. 30(1), 2 (2012)
15. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: USENIX Security, p. 29 (2012)
16. Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest (2009)
17. REIL, `http://www.zynamics.com/binnavi/manual/html/reil_language.htm`

18. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
19. Heelan, S., Gianni, A.: Augmenting vulnerability analysis of binary code. In: Annual Computer Security Applications Conference, pp. 199–208 (2012)
20. Sekar, R.: An efficient black-box technique for defeating web application attacks. In: NDSS (2009)
21. Li, L., Just, J.E., Sekar, R.: Online signature generation for windows systems. In: Annual Computer Security Applications Conference, pp. 289–298 (2009)
22. Ganai, M.K., Lee, D., Gupta, A.: DTAM: dynamic taint analysis of multi-threaded programs for relevancy. In: FSE (2012)
23. Wang, C., Kundu, S., Limaye, R., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. Int. J. Formal Aspects of Computing, 1–25 (April 2011)
24. Wang, C., Ganai, M.: Predicting concurrency failures in generalized traces of x86 executables. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 4–18. Springer, Heidelberg (2012)

Author Index