Walter Binder
Eric Bodden
Welf Löwe (Eds.)

LNCS 8088

Software Composition

12th International Conference, SC 2013
Budapest, Hungary, June 2013
Proceedings

ifip

Lecture Notes in Computer Science 8088

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Walter Binder Eric Bodden
Welf Löwe (Eds.)

Software Composition

12th International Conference, SC 2013
Budapest, Hungary, June 19, 2013
Proceedings

Springer

Volume Editors

Walter Binder
University of Lugano
Faculty of Informatics
6904 Lugano, Switzerland
E-mail: walter.binder@usi.ch

Eric Bodden
Technische Universität Darmstadt
European Center for Security and Privacy by Design (EC SPRIDE)
64293 Darmstadt, Germany
E-mail: bodden@ec-spride.de

Welf Löwe
Linnaeus University
Department of Computer Science
351 95 Växjö, Sweden
E-mail: welf.lowe@lnu.se

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39613-7 e-ISBN 978-3-642-39614-4
DOI 10.1007/978-3-642-39614-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013942550

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

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

The 12th International Conference on Software Composition (SC 2013) provided researchers and practitioners with a unique platform to present and discuss challenges of how composition of software parts may be used to build and maintain large software systems. Co-located with the STAF 2013 Federated Conferences in Budapest, SC 2013 built upon a history of a successful series of conferences on software composition held since 2002 in cities across Europe.

We received 21 full submissions co-authored by researchers, practitioners, and academics from 14 countries. One paper was desk rejected for obviously being out of scope. All other papers were peer-reviewed by at least three reviewers, and discussed by the Program Committee. Based on the recommendations and discussions, we accepted 9 papers, leading to an acceptance rate of 43%.

Besides these technical papers, we are excited to have won Sven Apel as keynote speaker for SC 2013, who shared his insights on managing and analyzing software product lines with the combined SC 2013 and STAF 2013 audience.

We are grateful to the members of the Program Committee and the external reviewers for helping us to seek submissions and provide valuable and timely reviews. Their efforts enabled us to put together a high-quality technical program for SC 2013. We are indebted to the local arrangements team of STAF 2013 for the successful organization of all conference and social events. The SC 2013 submission, review, and proceedings process was extensively supported by the EasyChair Conference Management System. We also acknowledge the prompt and professional support from Springer, who published these proceedings in printed and electronic volumes as part of the Lecture Notes in Computer Science series.

Most importantly, we would like to thank all authors and participants of SC 2013 for their insightful works and discussions!

March 2013

Walter Binder
Eric Bodden
Welf Löwe

Program Committee

Danilo Ansaloni	University of Lugano, Switzerland
Sven Apel	University of Passau, Germany
Olivier Barais	IRISA / INRIA / Univ. Rennes 1, France
Alexandre Bergel	University of Chile, Chile
Domenico Bianculli	University of Luxembourg, Luxembourg
Daniele Bonetta	University of Lugano, Switzerland
Lubomír Bulej	Charles University, Czech Republic
Shigeru Chiba	The University of Tokyo, Japan
Ion Constantinescu	Google Inc., USA
Schahram Dustdar	Vienna University of Technology, Austria
Erik Ernst	University of Aarhus, Denmark
Bernd Freisleben	University of Marburg, Germany
Thomas Gschwind	IBM Research, Switzerland
Michael Haupt	Oracle Labs, Germany
Christian Kästner	Carnegie Mellon University, USA
Doug Lea	SUNY Oswego, USA
Karl Lieberherr	Northeastern University, USA
David Lorenz	The Open University, Israel
Hidehiko Masuhara	University of Tokyo, Japan
Oscar Nierstrasz	University of Bern, Switzerland
Jacques Noyé	Ecole des Mines de Nantes, France
Ina Schaefer	Technische Universität Braunschweig, Germany
Andreas Sewe	Technische Universität Darmstadt, Germany
Mario Südholt	INRIA-Ecole des Mines de Nantes, France
Clemens Szyperski	Microsoft Research, USA
Immanuel Trummer	École Polytechnique Fédérale de Lausanne, Switzerland
Alex Villazón	Universidad Privada Boliviana, Bolivia
Eric Wohlstadter	University of British Columbia, Canada
Thomas Würthinger	Oracle Labs, Austria
Cheng Zhang	Shanghai Jiao Tong University, China

Additional Reviewers

Abdelmeged, Ahmed
Caracciolo, Andrea
Chis, Andrei
Ghica, Dan
Lessenich, Olaf
Schulze, Sandro
Siegmund, Janet
Zhang, Sai

Table of Contents

Reusable Components for Lightweight Mechanisation of Programming Languages

Seyed H. Haeri (Hossein) and Sibylle Schupp

Institute for Software Systems, Hamburg University of Technology, Germany
{hossein,schupp}@tu-harburg.de

Abstract. Implementing Programming Languages (PLs) has always been a challenge for various reasons. One reason is the excess of routine tasks to be redone on every implementation cycle. This is despite the remarkable fraction of syntax and semantics usually shared between successive cycles. In this paper, we present a component-based approach to avoid reimplementation of shared PL fractions. We provide two sets of reusable components; one for syntax implementation and another for semantics. Our syntax and semantics components correspond to syntactic categories and semantics rules of a PL specification, respectively. We show how, in addition to their service to reusability in syntax and semantics, our components can cater reusable implementation of PL analyses. Our current level of experimentation suggests that this approach is applicable wherever the following two features are available or can be simulated: Type Constraints and Multiple Inheritance. Implementing a PL using our approach, however, requires some modest programming discipline that we will explain throughout the text.

1 Introduction

Mechanisation of a PL is implementing it for the purpose of experimentally studying its characteristics and conduct. One interacts with the mechanisation to discover otherwise inapparent facts or flaws **in action**. PL mechanisation, however, can become very involved using traditional formal proof systems. Various other frameworks have, therefore, been crafted to help lightweight mechanisation. Different frameworks focus on facilitating different mechanisation tasks.

Mechanisation often enjoys cycles. Repeating implementation upon each cycle can form a considerable burden against mechanisation, especially because consecutive cycles often share a sizeable fraction of their syntax, only differ in few semantic rules, and, rarely add new analyses. Modularity becomes vital in that, upon extension, existing modules ought to be readily reusable.

As such, component-based mechanisation can provide even more reusability by facilitating implementation sharing for individual PL constructs. For example, it is not uncommon for different PLs to be extended using similar constructs (in the syntax or semantics). Component-based mechanisation cancels the need for reimplementing such constructs. In addition, our above interpretation of modularity comes as a side product in that PL modules already composed of components need not to be touched upon the addition of new components.

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 1–16, 2013.

In this paper, we implement language-independent components for syntactic categories to obtain syntax code reuse. We also offer a collection of semantics-lenient derivation rules that are individually executable. These form our highly flexible components that cater various sorts of semantics code reuse upon composition. A particularly interesting consequence of the flexibility in our syntax and semantics components is analysis code reuse. These components elevate the programming level of mechanisation; they encourage coding in terms of themselves (as opposed to exact type constructors), viz. , addition of new type constructors imposes no recompilation on existing code. (C.f. Expression Problem [29].)

Whilst our use of multiple inheritance caters easy extension of mechanisation, type constraints help the compiler outlaw reuse of mechanisation when conceptually inapplicable. Our approach is applicable independent of the formalism used for specification. Yet, a modest programming discipline is required for enjoying our reusability. Most of the burden is, however, on the Language Definitional Framework (LDF). Our approach indeed minimises the PL implementer's effort when an extensive set of our components is available through the LDF.

We choose to embed our approach in Scala for its unique combination of **built-in** features that suit mechanisation [23]. Both multiple inheritance and type constraints have special flavours in Scala that are not shared universally amongst languages. However, we do not make use of those specialities. Hence, the applicability of our approach is deemed to only be subject to the availability of multiple inheritance and type constraints.

We start by reviewing the related literature in Section 2. In Section 3, we provide a minimal explanation of the Scala features we use. We exemplify our approach using five systems for lazy evaluation that we briefly present in Section 4. Next, in Section 5, we demonstrate our components for both syntax and semantics mechanisation. As an interesting extra consequence of our particular design of components, Section 6 shows how analysis mechanisation reuse is gained. Concluding remarks and discussion on future work come in Section 7.[1]

2 Related Work

Funcons of PLanCompS [13] are composable components for PL mechanisation each of which with a universally unique semantics. Funcons are similar to our syntax components except that, due to our decoupling of syntax and semantics, our components can have multiple pieces of semantics. On the other hand, a funcon's semantics is provided using Modular SOS [21] and Action Semantics [20], whilst we do not demand any particular formalism. The GLoo mini parsers enable scope-controlled extensions to its language by desugaring the extended syntax into core GLoo [19]. SugarHaskell [9] provides similar facilities, but in a layout-sensitive fashion and for Haskell. Polyglot [22] users can extend a PL compiler (including that of Polyglot itself) by providing (one or more) compiler passes that rewrite the original AST into a Java one. The difference between the

[1] online source code available at `http://www.sts.tuhh.de/~hossein/compatibility`.

last three works and ours is that we do not specifically target PL specification through syntactic desugaring. Our semantics components can be used with or without a core semantics, and, are not restricted to any particular formalism.

Two important ingredients of our approach are type constraints and multiple inheritance. Kiama [25] is an LDF that is embedded in Scala. Hence, Kiama does already have all the language support required by our approach. Maude [5], K [11], MMT [4], Redex [10], Liga [14], Silver [28], Rascal [16], UUAG [7], JastAdd [8], and Spoofax [15] are LDFs that ship with their own DSL as the meta-level PL. Maude is the only such LDF with support for both multiple inheritance and type constraints. JastAdd, UUAG, and Rascal each only provide built-in support for half the language features that our approach requires. Only a runtime simulation of our approach is possible in K, Redex, Liga, Spoofax, and Silver.

Finally, Axelsson [2] and Bahr [3] provide Haskell libraries to improve different aspects of embedded DSL mechanisation. They both build on Swierstra's *data types à la carte* [26] and proceed by offering a new abstract syntax model.

3 Scala Syntax

This section introduces the parts of Scala syntax that we use in this paper.

```
1  object O {def apply(n: Int) = ...}
2  class C0 {type NT1 = Int; type NT2}
3  class C1[T] {
4    def m[U]:  Int -> Int = ...
5  }
6  class C2[T1 <: T2]
7  class C3[T1 <: T2{type NT}]
8  class C4[+T]
9  class C5[T <: C2[_]]
10 class C6[T <: C0 with C2[T]]
```

The method `apply` (line 1) tells Scala to expand calls like `O(1)` to `O.apply(1)`. The nested types `NT1` and `NT2` of the class `C0` (line 2) can be referred to as `C0#NT1` and `C0#NT2`, respectively. This contrasts with Scala's dot notation for referring to members of a package. We say that `C0` binds `NT1` to `Int`. The nested type `NT2` is abstract in that `C0` itself does not bind it. Class `C1` is parametrised over type `T` (line 3). Likewise, method `m` is parametrised over type `U` (line 4). The type parameter `T1` of `C2` is constrained by an *upper bound* type `T2` (line 6). As a result, one can only instantiate `C2` with types which inherit from `T2`. The types to instantiate `C3` with need to also have a nested type `NT` (line 7). One can also place constraints on nested types of type parameters. The plus used before type parameter `T` in line 8 implies that when `T1` is a subtype of `T2`, `C4[T1]` is considered a subtype of `C4[T2]`. Use of underscore in line 9 indicates that one can instantiate `C5` with any type that inherits from `C2[T']`, *for some type* `T'`. Type parameters can be more than one, in which case they are separated using commas. Multiple nested types demanded by an upper bound are to be put on separate lines, or, separated using semicolons. Class `C6` places two upper bound

constraints on its type parameter `T` (line 10). Namely, the type parameter `T` has to inherit from both `C0` and `C2[T]`. (Note that `T` itself is used in its own latter upper bound.) Traits are like abstract classes, but can be multiply inherited.

4 The Implemented Family of Operational Semantics

Five systems in the family of lazy evaluation are: Abramsky and Ong [1], Launchbury [17], Sinot [24], van Eekelen and de Mol [27], and Haeri [12]. We will be referring to these as $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{S}_1$, and $\mathcal{S}_2$, respectively. [2] Moreover, we will refer to the syntax of a family member as a *member syntax*, and, to its semantics as a *member semantics*. Section 4.1 provides an overview of the family syntax. Section 4.2 exhibits only the parts of family semantics that we refer to in this paper. The reader may refer to the original papers for more explanation. The aim of this section is to give the reader enough understanding from the family so that they can follow our discussions. We chose these particular five systems because their good proximity makes demonstration easy. However, given the flexibility in our components, our approach is well applicable beyond just these five.

4.1 Syntax

Here, we briefly present the syntax of the implemented family. Notationally, our presentation is not exactly the same as the original ones. We unify the original notations and neglect the minor differences.

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{S}_1$	$\mathcal{S}_2$		
					$b ::= x \mid Z(\overrightarrow{x})$	
$e ::= x$	✓	✓	✓	✓	$e ::= b \mid \lambda x.b \mid e\ b \mid \mathsf{let}\ \{b_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$	
$\lambda x.e$	✓	✓	✓	✓	$v ::= \lambda x.b \mid x\ b_1 \ldots b_n$	
$e\ x$	✓	✓	✓	✓		
$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$		✓	✓	✓	$v ::= \lambda x.e$	$\mathcal{L}_0, \mathcal{L}_1, \mathcal{S}_1$
$e_1\ \mathsf{seq}\ e_2$			✓	✓	$v ::= \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda x.e \quad (n \geq 0)$	$\mathcal{S}_2$

$\mathcal{L}_0$= Abramsky and Ong, $\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri

Fig. 1. Syntax for All the Family Members

Figure 1 shows the family syntax where e ranges over expressions, v ranges over values, and, x ranges over variables. The left half of the figure demonstrates the syntax common between $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{S}_2$. With the great degree of commonality between the four, we demonstrate them altogether in a single compact form to avoid repetition. Ticks show the constructors in each member syntax.

To the right, the top half shows complete $\mathcal{L}_2$ syntax, which is a bit different from the previous four. Here, $\overrightarrow{x}$ is a short form for $x_1\ x_2 \cdots x_n$ where $n \geq 2$. In $\mathcal{L}_2$, the *generalised identifier* b ranges over ordinary variables and *meta*variables

[2] $\mathcal{L}$ for lazy evaluation and $\mathcal{S}$ for selective strictness

$(Z(\overrightarrow{x}))$. Function applications are likewise generalised. That is, application of functions is allowed to metavariables as well as variables. λ-abstractions are the only values at $\mathcal{L}_0$, $\mathcal{L}_1$, and $\mathcal{S}_1$. Whereas, in $\mathcal{S}_2$, let-*surrounded* λ-abstractions are also considered values, where $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e$ is a syntactic sugar for $\lambda x.e$ when $n = 0$. In the $\mathcal{L}_2$ syntax, successive applications of a variable to generalised identifiers $(x\ b_1 \cdots b_n)$ is also considered a value. We refer to this syntactic category as the *variable-to-identifier-applications*. In the entire family, subscripts do not impact the syntactic category, and, are allowed to be arbitrary.

4.2 Semantics

Figure 2 shows selected parts of the family semantics. Here, rule labels are of the form $(\mathbf{r})_\ell$ where r is the rule name and ℓ is the list of family members containing r in their semantics. Rules are of the form $\Gamma : e \Downarrow \Delta : v$ where capital Greek letters denote *heaps*. This rule form reads: Evaluation of e in Γ results in v and updates the bindings to Δ. We write $\Gamma : e \Downarrow_\Pi \Delta : v$ to emphasise that Π is the derivation tree for $\Gamma : e \Downarrow \Delta : v$. In $\mathcal{L}_1$'s terminology, heaps are partial functions from variables to expressions. $\mathcal{S}_1$ as well as $\mathcal{S}_2$ inherit the same terminology. In the semantics of $\mathcal{L}_2$, however, the domain of heaps consists of the set of variables **and** metavariables. $e[x/y]$ denotes capture-avoiding substitution of variable x in e by variable y. All the family members have a *distinct-name convention*, i.e., variable names are supposed to be distinct.

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2} \qquad \frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : v}\ (\mathbf{var})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{S}_2,\mathcal{L}_2}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\ x \Downarrow \Theta : v}\ (\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$$

$$\frac{(\Gamma, x_i \mapsto e_i)_{i=1}^{n} : e \Downarrow \Delta : v}{\Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e \Downarrow \Delta : v}\ (\mathbf{let})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2} \qquad \frac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Delta : v_2}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq})_{\mathcal{S}_1,\mathcal{S}_2}$$

$\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri, $\mathcal{L}_2$= Sinot

Fig. 2. Selected Parts of the Family Semantics

5 Components

Figure 3 gives a UML overview of our approach. At the top, elements of our approach for syntax mechanisation are illustrated, and, at the bottom, those of semantics. The left portions are class diagrams. The right portions are use cases for the left portion of the same row. We use a number of non-standard UML notations: The type Exp with which a LazyExp (top left portion) is instantiated has to inherit from LazyExp itself. There are similar constraints on the type parameter Exp of OpSem as well as Exp and OS of the apply method of Executable

Rule (all in the bottom left portion). Abusing the UML notation, we draw generalisation arrows that extend from the right portions (use cases) to the respective left (class diagrams). For instance, L1OpSem inherits from OpSem. Finally, in the top row, we use a non-standard dashed arrow "ntb" to specify that an Expression Trait binds nested types to its type constructors. As an example, L1Exp binds its nested type Val to its type constructor Lam. (See the top right portion.)

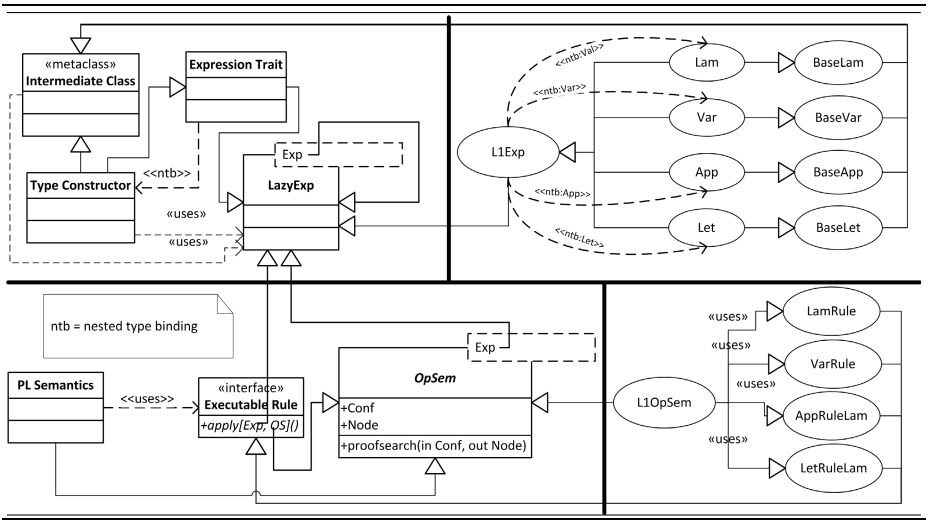

Fig. 3. Architecture of Our Approach

Ideally, of the elements depicted in Figure 3, certain ones ought to be shipped by the LDFs. The PL implementer, then, uses these shipped elements for mechanisation of the desired PL. The Intermediate Class instances, LazyExp, the Executable Rule instances, and OpSem are of the former sort. (In Figure 3, BaseLam, BaseVar, BaseApp, and BaseLet are intermediate classes, whilst LamRule, VarRule, AppRuleLam, and LetRuleLam are executable rules.) The top right portion summarises how to mechanise the $\mathcal{L}_1$ syntax using shipped elements of the left half of the same row. The bottom right portion does the same for the $\mathcal{L}_1$ semantics.

For reasons of improved correctness and reusability that we explain later, we need to check whether a PL syntax contains a certain syntactic category or not. Implementing a syntax using a typical algebraic datatype will, hence, not suffice. This is mainly because an ordinary algebraic datatype does not provide a mechanism for **programmatically** querying its type constructors. In Section 5.1 where our syntax components are presented, we employ nested types as an *extra storage* that make such programmatic queries possible. Our approach enjoys a design-by-contract flavour in that the names and duties of these nested types are dictated by the intermediate classes – at their design time. This flavour will also be available in Section 5.2, where our semantics components are introduced.

5.1 Syntax Components

In an embedded setting, mechanisation of a PL syntax typically involves embedding its abstract and concrete syntax, pretty-printing and the rest of debugging cosmetics, and sanity checks. One can of course perform the same task repeatedly for each member syntax. This entails a total of at least 24 type constructors for Figure 1 with a great amount of reimplementation in the above syntax mechanisation tasks. We instead mechanise a PL syntax in terms of our reusable components that serve as syntactic building blocks. Each of these components – called "intermediate classes" – corresponds to one and only one syntactic category. (See Intermediate Class in the top left portion of Figure 3.) An intermediate class implements its *own part* of an abstract syntax, pretty-printing, and sanity checks. This one-off implementation, then, is readily available to whatever PL syntax that contains the respective syntactic category, and, can be used off-the-shelf. For example, for Figure 1, we have implemented a total of 9 intermediate classes that correspond to variables (`BaseVar`), λ-abstractions (`BaseLam`), function applications (`BaseApp`), let-expressions (`BaseLet`), selective strictness (`BaseSeq`), metavariables (`BaseMVar`), generalised identifiers (`BaseGenIdn`), let-surrounded λ-abstractions (`BaseVal`), and variable-to-identifier-applications (`BaseVarApp`). On the other hand, we embed concrete syntax in terms of `LazyExp` – once and for all. `LazyExp` is our root of expressions for the entire family. (Compare with LazyExp in the top left portion of Figure 3.)

In Section 5.1.1, we first explain how to put our syntax components together to gain a complete syntax mechanisation. We, then, take a deeper look into some internals of our code which made this possible in Section 5.1.2.

5.1.1 Syntax Mechanisation When appropriate intermediate classes and `LazyExp` are at hand, a simple discipline needs to be followed:

- A member syntax is mechanised using its own (algebraic data-) type. Such a type provides its extra storage using binding nested types to its respective type constructors. When a syntax is mechanised in such a fashion, we refer to its implementing datatype as an "expression trait." Furthermore, when a type constructor is bound in such a fashion, we say it is *register*ed (at the expression trait). To inherit the concrete syntax embedding, and to be of use to our semantics mechanisation utilities, an expression trait `T` always derives from `LazyExp[T]`.
- Type constructors themselves need as well to specify which syntactic category they belong to. With their type parameters that will be explained later, we consider our intermediate classes only *half-baked*. We say that an intermediate class gets *fully-baked* for an expression trait `T` when a type constructor of `T` inherits from their instantiation for `T`.

Figure 4 exemplifies our discipline for $\mathcal{L}_1$. Scala uses normal inheritance as a simple facility for extensible algebraic datatypes. Accordingly, `Var`, `Lam`, `App`, and `Let` (lines 11-14) are type constructors of `L1Exp`. They are fully-baked for $\mathcal{L}_1$'s

```
1  package l1
2
3  sealed trait L1Exp extends LazyExp[L1Exp] {
4    type Val = l1.Lam
5    type App = l1.App
6    type Var = l1.Var
7    type Let = l1.Let
8    ...
9  }
10
11 final case class Var(...) extends BaseVar(...)        with L1Exp
12 final case class Lam(...) extends BaseLam[L1Exp](...) with L1Exp
13 final case class App(...) extends BaseApp[L1Exp](...) with L1Exp
14 final case class Let(...) extends BaseLet[L1Exp](...) with L1Exp
```

Fig. 4. Mechanisation of the $\mathcal{L}_1$ Syntax Using our Programming Discipline

expression trait (`L1Exp`) because they inherit from `BaseVar`, `BaseLam[L1Exp]`, `BaseApp[L1Exp]`, and `BaseLet[L1Exp]`, respectively. (See Figure 1 for the syntactic categories of $\mathcal{L}_1$.) These four type constructors are registered in lines 4-7 where they get bound to the nested types `Val`, `App`, `Var`, and `Let` of `L1Exp`, respectively. (C.f. Figure 4 with the top right portion of Figure 3.)

5.1.2 Technicality In order for an intermediate class not to be exclusively suitable to a single PL, it has to be parameterised over the PL syntax. However, not every syntactic category is suitable to every PL syntax. An intermediate class has to act accordingly. Consider `BaseLet`, for example:

```
1 class BaseLet[+Exp <: LazyExp[_]{type Let <: BaseLet[_]}]
2   (val bs: Map[Idn, Exp], val e: Exp) {
3     if(...) //value type == λ-abstractions
4       require(!bs.isEmpty)
5     override def toString() = ...
6 }
```

Lines 3 and 4 above perform a sanity check pertaining to let-expressions. (It is only in $\mathcal{S}_2$ – where let-surrounded λ-abstractions are value types – that empty let-bindings are allowed.) Line 5 handles the pretty-printing. The constructor parameters `bs` and `e` (line 2) embed the abstract syntax part of this intermediate class. Note that the type of the latter parameter is not fixed. Instead, it is typed using the type parameter `Exp` (line 1). The upper bound on `Exp` makes `BaseLet` invariably available to every member syntax so long as `Exp` registers its `BaseLet`-derived type constructor under the name `Let`. (Namely, $\mathcal{L}_0$ is excluded.) Similar type constraints selectively determine the appropriate *classes of syntax*.

It remains to further expand on the role of `LazyExp`. In this section, we focus only on the syntactic parts of its role. Section 5.2 explains its role for semantics mechanisation. We implement all our concrete syntax embedding for `LazyExp` –

once and for all. When applicable, a member syntax reuses the same embedding through inheritance of its expression trait from `LazyExp`. The following table summarises our concrete syntax embedding: In each row, the code on the left gets automatically desugared into a piece of abstract syntax that represents the mathematical expression on the right. (`T` in line 2 is the corresponding expression trait of `e`.)

	code	math
1	`e("x1")...("xn")`	$((e\ x_1)\ \cdots\ x_n)$
2	`\[T]("x1",...,"xn")(e)`	$\lambda x_1 \cdots x_n.e$
3	`let ("x1" -> e1,...,"xn" -> en) in e`	$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$
4	`e1 seq e2`	$e_1\ \mathsf{seq}\ e_2$

Note that the code in line 2 embeds a let-surrounded λ-abstraction in $\mathcal{S}_2$'s syntax and ordinary λ-abstractions in other family members. Likewise, when `e` is a λ-abstraction, the code in line 3 embeds another λ-abstraction for $\mathcal{S}_2$'s syntax, and, let-expressions otherwise. On the other hand, the embedding in line 4 is only applicable when a syntactic category is available for selective strictness. Again, the selectivity on the classes of syntax is enabled by type constraints.

As also explained further above, our convention is that an expression trait `T` must derive from `LazyExp[T]`. (See line 3 in Figure 4 for `L1Exp`, for example.) Given that `T`'s type constructors inherit from it, they also inherit from `LazyExp` by transitivity of inheritance. Note how following our convention for expression traits makes them distinguishable from type constructors. The particular wiring used below for the type parameter `Exp` of `LazyExp` enforces the above convention. Section 5.2 contains an example where this convention comes handy.

```
1  trait LazyExp[+Exp <: LazyExp[Exp]] {...}
```

5.2 Semantics Components

Similar to the case for syntax, it is perfectly possible to program each member semantics separately. That is a total of 23 rules for the entire family, and, with a great deal of code repetition. Instead, we implement a collection of 14 reusable and executable rules, which can be plugged into a PL semantics mechanisation. Our design ships another artefact as well: `OpSem` is our root operational semantics class. This is an abstract base class with a method for distributing the semantics evaluation between executable rules. Yet, `OpSem` is flexible on its input/output to the extent that it allows several semantics mechanisations for a single syntax. In this paper, we only demonstrate the idea for rules which document the entire semantics evaluation, if successful. However, one can easily configure `OpSem` for rules which, for instance, merely work with the PL objects involved in the semantics specification. Although we only demonstrate mechanisation for operational semantics, we have no evidence to doubt the applicability of our approach to other formalisms. After all, it only amounts for the semantic rules to be implemented like our executable rules.

In Section 5.2.1, we first explain how to combine our components to mechanise a PL semantics. A closer look into our components themselves is then

provided in Section 5.2.2. Whilst the latter section targets LDF implementers more, the former one is more useful to the LDF users.

5.2.1 Semantics Mechanisation Assuming the availability of our components, the following programming discipline needs to be followed for semantics mechanisation: A member semantics is implemented as a stand-alone object that derives from `OpSem[T]`, where `T` is the expression trait of the corresponding member syntax. This object needs to implement a method `proofsearch` which distributes evaluation between pertaining executable rules. In such a case, we say the member semantics *plugs* its appropriate executable rules.

```
1  object opsem extends OpSem[LExp] {
2    type Conf = HBConf[LExp]
3    type Node = HBNode[LExp]
4
5    def proofsearch(g: LHeap, e: LExp): HBNode[LExp] = e match {
6      case l: Lam => HBLamRule[LExp, opsem.type](g, l)
7      case a: App => HBAppRuleLam[LExp, opsem.type](g, a)
8      case v: Var => HBVarRule[LExp, opsem.type](g, v)
9      case l: Let => HBLetRuleLam[LExp, opsem.type](g, l)
10   }
11   override def proofsearch(c: Conf): HBNode[LExp] =
12     proofsearch(c._1, c._2)
13 }
```

Fig. 5. Implementing the $\mathcal{L}_1$ Operational Semantics in Isolation

For example, for $\mathcal{L}_1$'s semantics, the method `proofsearch` in Figure 5 takes a heap along with an expression (line 5), and, produces a derivation tree, when successful: Here, the plugged executable rules are `HBLamRule`, `HBAppRuleLam`, `HBVarRule`, and `HBLetRuleLam` (lines 6 to 9, respectively) that we schematically depicted in Figure 3. (In our naming convention, prefix `HB` indicates a *heap-based* system. That includes all the family members in this paper except $\mathcal{L}_0$.) What comes after the executable rule names in square brackets is to guide Scala's type deduction. `HBNode` is the root of our hierarchy for nodes in the heap-based derivation trees. Each node class encapsulates relevant compile time/runtime sanity checks that make it easier to enforce correctness of the executable rules. Armed with such correctness enforcement mechanisms, the compiler would have stopped us, for any of the four cases, had we plugged in a rule which is incompatible with the respective characteristics of either $\mathcal{L}_1$'s syntax or semantics.

5.2.2 Technicality Implementing a rule in a way that is not exclusively useful to a particular PL entails parameterising it over both the syntax and semantics. And, indeed the type parameters of our executable rules *characterise*

both the syntax and semantics they expect. Our rules can be plugged into any semantics so long as their characteristic expectations hold. A compile error will be emitted otherwise. For example, below is our **(let)**$_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$ implementation:

```
1  object HBLetRuleLam {
2    def apply[Exp <: LazyExp[Exp]{type Val <: BaseLam[Exp]
3                                  type Let <: BaseLet[Exp]},
4              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
5                               type Node = HBNode[Exp]}]
6      (g: Heap[Exp], lexp: Exp#Let with Exp)
7      (implicit opsem: OS): HBNode[Exp] = {
8        val (e, bs) = (lexp.e, lexp.bs)
9        val pi = opsem.proofsearch(g ++ bs, e)
10       val (d, z) = (pi.g2, pi.e2)
11       new HBLetNodeLam[Exp](pi, g, lexp, d, z)
12   }
13 }
```

The type parameters `Exp` and `OS` above (lines 2 and 4) signify the expression trait and the operational semantics, respectively. However, not every rule in Figure 2 is a part of every member semantics. For example, this **(let)** rule is only a part of the $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{L}_2$ semantics. The constraint `type Val <: BaseLam[Exp]` (line 2) rules out $\mathcal{S}_2$. This constraint enforces on `Exp` the availability of a nested type `Val` that binds to a class derived from `BaseLam`, i.e., that λ-abstractions is a value type of the syntax. (See Figure 1.) `OS <: OpSem[Exp]` states that `OS` must be an operational semantics type over the expression type `Exp`. (More on `OpSem` shortly.) The constraint `type Conf = HBConf[Exp]` (line 4) on `OS` states that it inputs a pair of heap and expression. Similarly, `type Node = HBNode[Exp]` (line 5) specifies that `OS` outputs a heap-based derivation tree. These constraints rule out the semantics of $\mathcal{L}_0$ too, making `HBLetRuleLam` only applicable to the right family members. Lastly, note how the treatment of type parameters enables `opsem` to take a continuation-passing style role for handling "the rest of the evaluation" – again, only for the correct family members.

Here is a recap on the remaining points: The constraint `type Let <: BaseLet[Exp]` (line 3) ensures that `Exp` registers its `BaseLet`-derived constructor under the name `Let`. Furthermore, `lexp` (in line 6) is required to be an instance of both `Exp` and this registered type. In other words, `lexp` needs to be constructed using a type constructor of `Exp` that corresponds to let-expressions. Recall also that, as seen at the end of Section 5.1, the constraint `Exp <: LazyExp[Exp]` (line 1) ensures that `Exp` is an expression trait. `HBLetNodeLam` inherits from `HBNode` to be the node for let-expressions where λ-abstractions are a value type.

It remains to consider our `OpSem` trait:

```
1  trait OpSem[Exp <: LazyExp[_]] {
2    type Conf
3    type Node <: ProofTree
4    def proofsearch(c: Conf): Node
5  }
```

For each operational semantics, `proofsearch` inputs the initial *configuration*, and, produces the derivation tree according to the rules of the semantics. The abstract type `Conf` represents the type signature of the input (line 2). Likewise, the abstract type `Node` is the derivation tree type an operational semantics outputs (line 3). The type `ProofTree` is our generic ADT for derivation trees.

6 Case Study: Analysis Code Reuse

Like the case of syntax and semantics, one should be able to reuse the code for analyses implemented over previous mechanisation cycles – but, only when they are still conceptually applicable. We address that need based on two facts:

Fact 1. Old code that is implemented in terms of the root of a hierarchy works for new classes that derive from the root.

Fact 2. Code that constraints its type parameters can employ the compiler to prevent its use for wrong types.

Information gathering over derivation tree traversals is the essence of many analyses. A crude idea can, thus, be implementing all the analyses over a single generic tree type. However, such a tree is unaware of the types its nodes contain. One would rather make all the derivation tree types **inherit** from such a generic type. This way, old code which operates on the generic type can remain intact over the addition of new derivation types. (C.f. Fact 1.) More precision can also be gained by giving this hierarchy extra intermediate nodes. On the other hand, by constraining the type parameters of analysis implementations, one can avoid their wrong application. Constraints can enforce applicability of an analysis to all derivation trees that say derive from a certain base. (C.f. Fact 2.) We call the process of organising derivation tree types in a hierarchy and implementing analyses in terms of the suitable hierarchy node "multi-levelling analyses."

Our hierarchy of derivation trees is rooted in `ProofTree` (seen first in Section 5.2). `ProofTree` has a minimal understanding of what it contains. All it knows is that a set of premisses leads to a conclusion using a rule label. `HBNode` and `HLNode` extend `ProofTree` for heap-based nodes and the heap-less ones, respectively. Nodes which represent derivation in the $\mathcal{L}_0$ operational semantics are instances of `HLNode`. All other nodes are of type `HBNode`. Both `HBNode` and `HLNode` provide more specific information. For example, the former also knows that its conclusion is always a 4-tuple for the $\Gamma : e \Downarrow \Delta : v$ scheme. Further down in the hierarchy come nodes that correspond to semantics rules, and hence, executable rules. These latter nodes know the syntactic category of their e in the above scheme. For instance, `HBVarNode` that corresponds to $\textbf{(var)}_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{S}_2,\mathcal{L}_2}$ knows that it works on variables. At the same level are types for the derivation trees of the individual family members. `L1Node`, for instance, is that of $\mathcal{L}_1$. Obviously, `L1Node` has more specific information at hand, e.g., the exact type of expressions/heaps it works with.

Generally, analyses remain invariably useful over several mechanisation cycles so long as they are implemented in terms of the right level at the derivation

tree hierarchy. Most of what makes such a hierarchical craft of derivation trees helpful stems from the high degree of flexibility in executable rules. The `apply` methods of the executable rules presented in this paper all have `HBNode` return types. However, it is trivial to configure executable rules otherwise and still enjoy them as reusable components for semantics mechanisation. We also gain other sorts of analysis reusability from the high degree of reusability in intermediate classes. For example, an analysis which deals with evaluation of a particular syntactic category can remain intact over consecutive mechanisation cycles even though the actual type constructors involved vary across the cycles. We will not demonstrate reusability of this latter sort in this paper.

As a first example, consider an analysis the right level for in our hierarchy is `ProofTree`: Counting the number of rules used over a derivation.

```
1  def rulecount(p: ProofTree): Int = if (p.prems.isEmpty) 1
2                                     else (0 /: p.prems) (_ + rulecount(_))
```

This analysis does not need any knowledge about the types involved over the proof search. It is a simple folding action over the premisses (line 2) with axioms as the basis of induction (line 1). An occasion where this counting might be useful is comparing the cost of designated computations across different semantics which are known to be observationally equivalent.

Our second example is on the analyses in Definition 1, which play a central role in the observational equivalence theorems on $\mathcal{S}_2$ [12]:

Definition 1. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Define* $diff(\Pi) = \{x \in dom(\Gamma) \mid \Gamma(x) \neq \Delta(x)\}$. *Call* x ***atomic*** *in* Γ *when there exist* Δ_x, v_x, *and* Π_x *such that* $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ *and* $diff(\Pi_x) = \{x\}$.

In fact, as opposed to only $\mathcal{S}_2$, *diff* and *atomic* are analyses applicable to any heap-based semantics. Here is how we employ that observation:[3]

```
1  object diff {
2    def apply[Exp <: LazyExp[Exp]](pi: HBNode[Exp]): Set[Idn] =
3      for(x <- pi.g.dom; if(pi.g(x) != pi.d(x))) yield x
4  }//diff(pi) = {x ∈ dom(pi.g) | pi.g(x) != pi.d(x)}
5  object atomic {
6    def apply[Exp <: LazyExp[Exp]{type Var <: BaseVar},
7              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
8                               type Node = HBNode[Exp]}]
9      (g: Heap[Exp], x: Idn)//x is atomic in g when...
10     (implicit opsem: OS, varibliser: Idn => Exp with Exp#Var): Boolean =
11       diff(opsem.proofsearch(g, x)) == Set(x)//... diff(pi_x) == {x},
12 }//where pi_x = opsem.proofsearch(g, x).
```

Notice that `atomic` characterises the syntax and semantics it is applicable to through the constraints on the type parameters (lines 6-8). Consequently, it remains applicable upon extensions of mechanisation so long as the characteristics remain intact. Thanks to our multi-levelling, in the implementation of `diff`,

[3] `varibliser` is our implementation detail in charge of reifying an identifier into an expression of the right type (i.e., a variable).

types are all correctly identified by the compiler. We would have not had such a pleasure, had we implemented it on `ProofTree`, which is oblivious of the types inside it. This static safety becomes clearer in the next example where we examine order of evaluation of variables for any heap-based semantics:

```
1  object EvalList {
2    def apply[Exp <: LazyExp[Exp]{type Var <: BaseVar}]
3      (hbn: HBNode[Exp]): List[Idn] = hbn match {
4        case HBVarNode(pi, g, xvar, d, _) => {//(var)_{L1,S1,S2,L2} in Fig. 2:
5          val prev = EvalList(pi)//what gets evaluated in the premisses...
6          val x = xvar.name
7          if(g(x) != d(x)) (prev:::List(x)) else prev
8        }//... plus x itself when g(x) != d(x).
9        case _ => //Otherwise: union what is evaluated in the premisses.
10         (for(p <- hbn.ps) yield EvalList(p)).toList.flatten
11   }//Note: hbn.ps == premisses of hbn
12 }
```

`EvalList` produces the list of evaluated variables in order. Due to space restrictions, we only report how the above single implementation of `EvalList` remains applicable upon extending mechanisation of $\mathcal{L}_1$ to $\mathcal{S}_2$. Let heap $\Gamma = \{id \mapsto \lambda t.t, y \mapsto (\lambda t_1 t_2.t_1)\ id\ x, x \mapsto (\lambda t_1 t_2.t_2)\ id\}$ be represented by `g`. For $\mathcal{L}_1$, `EvalList(g <::> "y")` produces `List(y)`, whilst `List(x, y)` is produced by `EvalList(g <::> ("x" seq "y"))` for $\mathcal{S}_2$. [4]

Due to multi-levelling, Scala precisely infers the types for `pi`, `xvar`, `g`, and `d`. There is no need for runtime casting. To get multi-levelling, we identified that this analysis is applicable to any heap-based derivation tree on expression traits with a syntactic category for variables. We enforced that by making `EvalList` applicable to any such tree through placing the type constraints at line 2. It is exactly this constraint that creates a flow of type information that automates type inference of the above variables. In the absence of that type information in scope, one has to manually set variable types and/or even resort to runtime casting to calm the type system.

7 Conclusion and Future Work

In this paper, we present our components for syntax and semantics mechanisation. As a driving example, we use five systems for lazy evaluation to show how these components can serve reusability in the mechanisation of PL syntax, semantics, and analysis. We also discuss the internals of our components and how, using type constraints and multiple inheritance, they are engineered for this particular sort of code reuse.

Using our components imposes some modest programming discipline. A PL implementer's part of this discipline is indeed minimal. And, yet, the effort to suit the reusability is incomparably smaller than reimplementation: For syntax, this effort amounts to simply deriving from an extra base class (i.e., intermediate classes for each type constructor) and binding the type constructors under

[4] `g <::> e` abbreviates `os.proofsearch(g, e)` when `os` is an implicit in scope.

some fixed nested type of the expression trait (Figure 4). For semantics, it takes deriving from an abstract base class (e.g., `OpSem`) and implementing a method (like `proofsearch`) that merely distributes the evaluation task between the appropriate executable rules (Figure 5).

Shipping our components is certainly a new burden on LDFs. Implementing our components can sometimes become clunky. This is mainly because, working with constrained type parameters as opposed to exact types makes some extra indirection inevitable. The burden on LDFs magnifies are they to ship an exhaustive set of our components which the PL implementer can freely mix-and-match; that is, after all, likely to take several rounds of refactoring on its way.

Whether or not our approach will scale is a topic for further research. One might also study the classes of extensions in terms of the refactoring they dictate. For example, having had implemented our approach for the other four family members, addition of $\mathcal{L}_2$ dictated some refactoring to our codebase. Regarding further extensions, the effort might vary: For example, adding integer arithmetic as sketched in the $\mathcal{L}_1$'s original paper [17] is routine. Addition of Eden's strict function application [18] would also be relatively easy. However, we anticipate that adding the lazy evaluation material of Danvy et al. [6] needs refactoring.

Our components enjoy composability, but, are not atomic. That is, whilst it is trivial to compose our components to acquire new ones, not every semantic rule can be composed out of existing ones. For example, the subtle difference between $\mathbf{(app)}_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$ and the function application rule of $\mathcal{S}_2$ means that neither can be implemented in terms of another. The study of atomic support for implementing our components is yet another future work.

References

1. Abramsky, S., Ong, C.-H.: Full Abstraction in the Lazy Lambda Calculus. Inf. & Comp. 105(2), 159–267 (1993)
2. Axelsson, E.: A Generic Abstract Syntax Model for Embedded Languages. In: Proc. 17^{th} ACM SIGPLAN Int. Conf. Func. Prog., pp. 323–334. ACM, New York (2012)
3. Bahr, P., Hvitved, T.: Parametric Compositional Data Types. In: Chapman, J., Levy, P.B. (eds.) Proc. 4^{th} W. Math. Struct. Funct. Prog., February 2012. Elec. Proc. Theo. Comp. Sci., vol. 76, pp. 3–24 (2012)
4. Chalub, F., Braga, C.: Maude MSOS Tool. ENTCS 176(4), 133–146 (2007)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 System. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
6. Danvy, O., Millikin, K., Munk, J., Zerny, I.: On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. Theo. Comp. Sci. 435, 21–42 (2012)
7. Dijkstra, A., Fokker, J., Swierstra, S.D.: The Architecture of the Utrecht Haskell Compiler. In: Proc. 2^{nd} ACM SIGPLAN Symp. on Haskell, pp. 93–104. ACM, New York (2009)
8. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: Proc. 22^{nd} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., pp. 1–18 (2007)

9. Erdweg, S., Rieger, F., Rendel, T., Ostermann, K.: Layout-sensitive Language Extensibility with SugarHaskell. In: Voigtländer, J. (ed.) Proc. 5^{th} ACM SIGPLAN Symp. on Haskell, September 2012, pp. 149–160. ACM, New York (2012)
10. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex, pp. 1–502. MIT Press, Cambridge (2009)
11. T. Florin Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, The K Primer (version 2.5), K'11 (M. Hills, ed.), ENTCS, to appear
12. Haeri, S.H.: Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness. In: Proc. Int. Conf. Theo. & Math. Found. Comp. Sci (TMFCS-10), pp. 143–150 (2010)
13. Johnstone, A., Mosses, P.D., Scott, E.: An Agile Approach to Language Modelling and Development. Innovations in Sys. & Soft. Eng. 6, 145–153 (2010)
14. Kastens, U., Waite, M.W.: Modularity and Reusability in Attribute Grammars. Acta Informatica 31, 601–627 (1994)
15. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Proc. 25^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., pp. 444–463. ACM, New York (2010)
16. Klint, P., van der Storm, T., Vinju, J.: EASY meta-programming with rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
17. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: Proc. 20^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., pp. 144–154. ACM, New York (1993)
18. Loogen, R., Ortega-mallén, Y., Peña-marí, R.: Parallel Functional Programming in Eden. J. Func. Prog. 15(3), 431–475 (2005)
19. Lumpe, M.: Growing a language: The gLoo perspective. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 1–19. Springer, Heidelberg (2008)
20. Mosses, P.D.: Theory and Practice of Action Semantics. In: Penczek, W., Szałas, A. (eds.) MFCS 1996. LNCS, vol. 1113, pp. 37–61. Springer, Heidelberg (1996)
21. Mosses, P.D.: Modular Structural Operational Semantics. J. Logic & Alg. Prog. 60–61, 195–228 (2004)
22. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
23. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. 20^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., pp. 41–57. ACM, New York (2005)
24. Sinot, F.-R.: Complete Laziness: a Natural Semantics. ENTCS 204, 129–145 (2008)
25. Sloane, A.: Lightweight Language Processing in Kiama. In: Gener. & Transform. Techs Soft. Eng. III, pp. 408–425 (2011)
26. Swierstra, W.: Data Types à la Carte. J. Func. Prog. 18(4), 423–436 (2008)
27. van Eekelen, M., de Mol, M.: Reflections on Type Theory, λ-Calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60^{th} Birthday, ch. Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101, Radboud U. Nijmegen (2007)
28. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Sci. Comp. Prog. 75(1–2), 39–54 (2010)
29. Wadler, P.: The Expression Problem, Java Genericity Mailing List (November 1998)

Neverlang 2 – Componentised Language Development for the JVM*

Walter Cazzola and Edoardo Vacchi

Department of Computer Science, Università degli Studi di Milano, Milano, Italy
{cazzola,vacchi}@di.unimi.it

Abstract. Traditional compiler development is non-modular. Although syntax extension and DSL embedding is making its way back in modern language design and implementation, componentisation in compiler construction is still an overlooked matter. Neverlang is a language development framework that emphasises modularity and code reuse. Neverlang makes extension, restriction and feature sharing easier, by letting developers define language components in distinct, independent units, that can be compiled independently and shared across different language implementations, even in their compiled form. The semantics of the implemented languages can be specified using any JVM-supported language. In this paper we will present the architecture and implementation of Neverlang 2, by the help of an example inspired by mobile devices and context-dependent behaviour. The Neverlang framework is already being employed successfully in real-world environments.

Keywords: Domain-Specific Languages, Language Design and Implementation, Composability and Modularity

1 Introduction and Motivations

Compilers are traditionally complex and monolithic entities that only experts can maintain and extend [4]. Even though parsers and compilers for existing programming languages are nowadays often available as source code, they are usually not meant for a developer to adapt or build upon. For instance, even today, the `javac` compiler still relies on a hand-coded LALR parser that, for any developer trying to experiment, represents a high barrier to entry. Although there is an effort to implement the `javac` parser using ANTLR [26], in the context of the OpenJDK project[1], purely generative tools such as ANTLR and the time-honoured `lex` and `yacc` do not really account for modularity and decomposability, making code reuse in compiler development still a challenge. This in turn often translates to duplicate efforts, such as re-implementing the parser for a whole language even when the change is relatively small. Because of this problem, language extensibility is an interesting problem that is currently under research. Microsoft has recently released Roslyn [24], a technology preview of a platform-level API

* This work has been partially supported by MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

[1] `http://openjdk.java.net/projects/compiler-grammar/`

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 17–32, 2013.

to control and extend (by way of AST manipulations) the compiler of C#, and in general any language compiled for the .NET platform. Scala is moving in a similar direction, bringing compiler structures and features at the API level to support reflection and meta-programming [25]. Still, enabling extensibility does not automatically make a language implementation modular, in that extending a language is not the same as sharing a feature across different language implementations. It is not infrequent for languages to have features in common: in recent years, functional programming languages have been cross-pollinating the object-oriented world, and lazy evaluation and higher-order functions are now available in traditionally object-oriented languages such as C#, Scala and Python. Development of new languages often implies to put together the same old concepts and constructs with a different syntax; a typical example of this are conditionals and loops. It follows that development of new languages could benefit from being able to reuse portions of existing compilers; even more if these portions could be shared in a precompiled form. Sharing tested, precompiled components across language implementations, could help minimising the effort and timing required for the development of a new DSL. Moreover, precompiled, runtime loadable components could enable new possibilities, such as hot deployment of new features on running programs.

Our proposed solution is Neverlang [8], a framework designed to assists developers in the implementation of domain-specific languages in a modular way. The last Neverlang implementation we presented had some limitations: the parser generator rebuilt the parse table from scratch for any change in the code base; semantic actions were woven into the AST using the AspectJ compiler, so they had to be rewoven most of the time. In our experience this process took a perceivable amount of time for each rebuilding process. Our older implementation also imposed the choice of the Java language to express semantic actions. Finally, many people brought to our attention that our implementation of the Neverlang compiler had not been written using Neverlang itself.

In this paper we are introducing Neverlang 2. In this new version, we integrated our own compiler generator, which generates and updates LALR parser on-the-fly. Other components can be compiled independently and shared in their pre-compiled form, and, once these components have been finalised, they do not need to be recompiled any more. Pre-compiled semantic actions can be expressed using any language that the JVM supports, and the AspectJ dependency has been completely dropped, favouring instead a manual method dispatching mechanism. The new Neverlang compiler has also been completely rewritten on top of the new runtime library, and thus it is completely self-hosted. We will describe in detail the architecture of Neverlang 2 by the help of a running example inspired by modern sensor-rich mobile devices. The new version of the framework is already successfully solving real-world problems:

1. a Neverlang-generated DSL is now being integrated in TheMatrix [16], a Java framework to query and manipulate Italian administrative databases to produce information on the prevalence of chronic disease and on standards of care across the country;
2. Neverlang is being employed in the implementation of a DSL for ERP software development;
3. the Neverlang compiler has been boostrapped, i.e., it has been developed using Neverlang itself.

```
Set ringer mode to silent between 11:00 PM and 7:00 AM
```

Listing 1: on{X} recipe adapted from `http://onx.ms/recipes/silentAtNight`

```
when time is between 11:00pm and 7:00am : turn ringer off.
```

Listing 2: A DSL using the Recipe DSL

Paper Outline. In Sect. 2 we will describe the running example that we will employ to show Neverlang's features. In Sect. 3 we will describe Neverlang and its architecture, including the incremental parser generator DEXTER [10]. In Sect. 4 we describe the implementation of our running example. In Sect. 5 we discuss the related work and in Sect. 6 we draw our conclusions and describe future work.

2 Running Example

In recent years, mobile devices such as smartphones or tablets have become more and more accessible to the masses. Applications can interact with the great number of sensors of these devices to infer information about the user, and trigger specific actions depending on them. In particular, there are applications that enable users to define *custom actions* to take when a particular condition occurs. On the Android platform, for instance, there are Tasker[2] and Llama[3]. Some of these applications provide the end-user with a graphical user interface to specify the actions and the conditions. Microsoft's on{X}[4] enable users to share *real code snippets* written in JavaScript through a web application. Selected actions can then be synced and deployed to the device. Snippets are made available to programming-illiterate users using a natural-language description (a *recipe*) that can be partially customized. In Listing 1 there is one such recipe (simplified from one really available on the on{X} web site), to put a smartphone in silent mode when time happens to be between a particular, customizable range (in red).

Although the idea is nice, (a) it requires users to know JavaScript to define new actions and (b) code snippets cannot be written directly on the device, but only through the provided web interface. One might want to put the idea further by enabling users to write their own code snippets using a simplified, natural language-like DSL. In this case, users would be writing real code, except it would look similar to a recipe. In Listing 2 we show how the Recipe DSL might look like.

3 Neverlang 2 Architecture

Neverlang [8] is the framework that we developed to implement DSLs using a compositional approach. Our current implementation introduces a number of new features. The previous version of Neverlang employed AspectJ to weave executable code for

[2] `http://tasker.dinglisch.net`
[3] `http://kebabapps.blogspot.com`
[4] `http://onx.ms`

```
module recipe.lang.MainModule {
  import { neverlang.runtime.utils.* }
  role(syntax) {
    Program  <-  RuleList ;
    RuleList <-  Rule RuleList ; RuleList <-  Rule ;
    Rule     <-  "when" ConditionList ":" Action "." ;
    ConditionList <-  Condition ;
    ConditionList <-  Condition "and" ConditionList ;
   }
  role(evaluation) {
    0 .{ $0.rules   = AttributeList.collectFrom($1, "ruleObj"); }.
    7 .{
      List<Condition> conditions = AttributeList.collectFrom($8, "condition")
      $7.ruleObj = new RuleObj(conditions, $9.action);
    }.
  }
}
```

Listing 3: A Recipe program is a list of rules.

semantic actions inside an AST made of several Java class files. In fact, a typical implementation of the *visitor pattern* in an OOP context is to subclass each node of the AST and then invoke some `visit()` method on that node. However, this approach had two main drawbacks: (a) many source files had to be rewoven each time the smallest change affected the code base and (b) involving AspectJ in the building process of the generated compiler took a perceivable amount of time. In Neverlang 2 the AspectJ dependency has been dropped, the AST is generated on-the-fly using the DEXTER LALR parser generator [10] and a *component manager* now loads and dispatches the semantic actions, which now can be also written in any JVM-supported language. The added bonus is that now components can be compiled and possibly distributed separately. Finally the new Neverlang compiler `nlgc` has been bootstrapped. In this section we will briefly describe the concepts that Neverlang 2 has retained from the older version of the framework (for instance, the concepts of **`module`** and **`slice`**), and we will then detail the new architecture in depth.

3.1 Neverlang Components

In Neverlang, a single language component is defined in a *module*. Each module encodes a syntactic feature along with its semantics. For instance, in a C-like programming language a module can define the **`for`** looping construct or the **`if`** branch. C-like languages such as Java, JavaScript, PHP, etc. share most of their syntactic definitions. Writing a compiler or an interpreter using Neverlang, makes possible to share modules between implementations. Each module contains one or more of *roles*.

Modules and Roles. A `syntax` role is a portion of the language's formal grammar. For instance, Listing 3 shows part of the grammar for a Recipe program. Nonterminals are capitalised, and terminals (keywords) are between double quotes[5]. A Recipe program is a list of `Rules`, and each `Rule` is in the form:

```
Rule <- "when" ConditionList ":" Action "."        (1)
```

[5] Terminals can also be defined using regular expressions. In that case, literals are delimited by slashes; e.g., `/[a-z]+/` captures any non-zero-length word of lowercase alphabetic characters.

```
slice foo.bar.MySlice {
  module foo.bar.SomeModule with role syntax
  module baz.qux.AnotherModule with role evaluation type_check
}
```

Listing 4: A slice for a fictional programming language

that is, the `when` keyword, some condition to evaluate, a colon symbol, and some action to take when the condition evaluates to true. Conventionally, the module containing the `Program` nonterminal is considered the main module, and `Program` is always considered the start symbol of the grammar.

Any other **`role`** in the module is a *semantic* role, that is, a compilation phase. Every module can contain as many **`role`**s as needed. The order of evaluation is specified in a separate configuration file. Each **`role`** contains several code sections, introduced by a number. A code section introduced by a number N, binds the corresponding code section to the evaluation of the N-th nonterminal in the syntactic role[6]. Nonterminals are numbered from left to right and from top to bottom. For instance, in Listing 3, 0 would be the first nonterminal in the first production (`Program`), and 7 would be the first nonterminal in the third production (`Rule`). Thus, 0 binds the Java code between the delimiters "`.{`" and "`}.`" to be evaluated when visiting the the 0-th nonterminal in the syntax role, and 7 binds code to the `Rule` nonterminal. Code sections can also refer to nonterminals using the `$N` notation and associate custom attributes to them using a familiar dot-notation. For example, the code

```
$7.rule = new RuleObj(conditions, act);
```

creates an attribute for nonterminal **`$7`** called `rule`, which contains an instance of the class `Rule`. We will see what this code does in more detail in Sect. 4.

Slices and Languages. Once the language has been broken into separate modules, it can be composed together using the **`slice`** and the **`language`** constructs. The **`language`** construct composes together the modules that the developer selects using slices. A **`slice`** imports roles from (possibly) different modules, and it encapsulates a *feature* of the language. Listing 4 shows an example of the syntax: the slice is importing the syntax role for one module, and two semantic roles from a different module. For instance, with respect to our running example, the Recipe language needs at least one slice to define the time condition and one slice to define the action of turning the ringer on or off. Slices can be used to bind semantic roles from one module to the syntax defined in another, so they constitute a powerful mechanism to reuse code in compiler development. We will demonstrate this feature further by supporting a form of localisation in our Recipe language (Sect. 4).

3.2 The Neverlang Compiler

An important part of Neverlang, beside its own DSL, is obviously the Neverlang compiler. The new Neverlang compiler `nlgc` has been developed using the Neverlang runtime (Sect. 3.3) to bootstrap the system. As a result, Neverlang today is completely self-hosted.

[6] It follows that syntactic roles are mandatory for semantic rules to make sense.

The `nlgc` tool acts like a translator from the Neverlang DSL into JVM-supported languages. Each **`slice`** and each **`language`** component is translated into a separate Java file. Modules are broken into several files: one Java class that explicitly declares every role in the module, one Java class containing the translation of the `syntax` role, and then one compile unit for each semantic action binding in each semantic role. For instance, the module `recipe.lang.MainModule` in Listing 3 is translated into 4 independent but logically related classes:

1. `recipe.lang.MainModule`, which lists each sub-component
2. `recipe.lang.MainModule$role$syntax`, which describe the syntactic part of the module
3. `recipe.lang.MainModule$role$evaluation$0`, because a semantic action has been bound to the 0-th nonterminal in the `evaluation` role
4. `recipe.lang.MainModule$role$evaluation$7`, because a semantic action has been bound to the 7-th nonterminal in the `evaluation` role

As we will see in Sect. 3.3 most of the class loading and method dispatching is performed automatically by the Neverlang runtime. Modules and slices have very few interdependencies and thus, they can be compiled *separately*. A change in one module requires to recompile only *that* module from source. Compare this to conventional compiler generation techniques, that, being usually based on source generation, often require a large part (if not all) of the source code to be recompiled anew. This approach streamlines the compiler-generation process by making possible to compile only those components that really need to be rebuilt. Of course, this possibility becomes particularly useful when the compiler becomes large and complex. Moreover, pre-compiled Neverlang components can be bundled together in jars for convenience of distribution, and they can also be shared and imported by different languages independently.

Full JVM Support. We said that `nlgc` translates the Neverlang DSL into JVM-supported languages. In most cases, this means that it generates Java source files. One core goal for the Neverlang 2 runtime was to have very few system requirements. Thus, the Neverlang 2 runtime has been written in Java, and the default language for semantic actions is Java as well. But semantic actions can be implemented using *any* language supported by the Java Virtual Machine, provided that a *translator plug-in* is available. The developer can then hint at the system that semantic actions are being written in a different language. Listing 5 shows an example of the syntax.

The new Neverlang compiler translates each semantic action into a class that implements the simple `SemanticAction` interface:

```
public interface SemanticAction { public void apply(ASTNode n); }
```

As mandated by the syntax-directed translation technique, a semantic action can attach arbitrary attributes to any nonterminal, that Neverlang refers with the dollar notation. This really translates to attaching attributes to the node of an AST: in Listing 3 the `condition` attribute will be attached to the root of any subtree of the AST which has

`Rule` at its root and the nodes `"when"`, `ConditionList`, `":"`, `Action`, `"."` as its children (more on this in Sect. 3.3)[7].

A translator plug-in describes how the occurrences of a nonterminal reference (in dollar notation) in a semantic action should be translated into the internal representation (a call to `n.nchild(int)`). Currently we have implemented support for Java and Scala. Listing 6 shows how this is done for Java. Code for Scala is similar. The plugin itself can be written in any JVM-supported language.

```
<jruby> // switch to jruby in global scope
module foo.bar.Multilang {
  role(syntax) { ... }
  role(role1) { 0 <scala> .{ ... }. } // scala for this action only
  role(role2) <jython> { ... }        // jython is default for this role
}
```

Listing 5: Any language running on the JVM can be supported.

```
public class JavaTranslatorPlugin extends TranslatorPlugin {
  public JavaTranslatorPlugin() {
    language = "java";
    fileExtension = "java";
    fileTemplate  = "public class {0} implements SemanticAction '{'\n"+
                    "  public void apply(ASTNode n) '{'\n{1}\n  '}'\n'}'";

    // when $N is the root of the subtree
    rootAttributeWrite  = "n.setValue(\"{1}\", {2});";
    rootAttributeRead   = "n.getValue(\"{1}\")";
    // when $N refers to a child node
    childAttributeWrite = "n.ntchild({0}).setValue(\"{1}\", {2});";
    childAttributeWrite = "n.ntchild({0}).getValue(\"{1}\")";
  }
}
```

Listing 6: Translator Plug-in for Java

3.3 The Neverlang 2 Runtime

The Neverlang 2 runtime is made of two main parts: the DEXTER [10] incremental parser generator and the component manager. A compiler written using Neverlang implements the well-known syntax directed translation mechanism [1], and implements an adaptive visitor pattern [8]. The component manager is responsible for loading **language**s, **slice**s and **module**s, and for dispatching the correct semantic action to the node of the syntax tree that is being visited in the correct phase (described in a *role*).

The Component Manager. When a Neverlang-generated compiler or interpreter is started, the Neverlang 2 component manager kicks in. To ensure quick loading and interpreting of the directives contained inside the **language**, **slice** and **module** constructs,

[7] Attributes are implemented as a map *attribute* → *value* attached to each node. Setting or getting is implemented as a method call. See Listing 6.

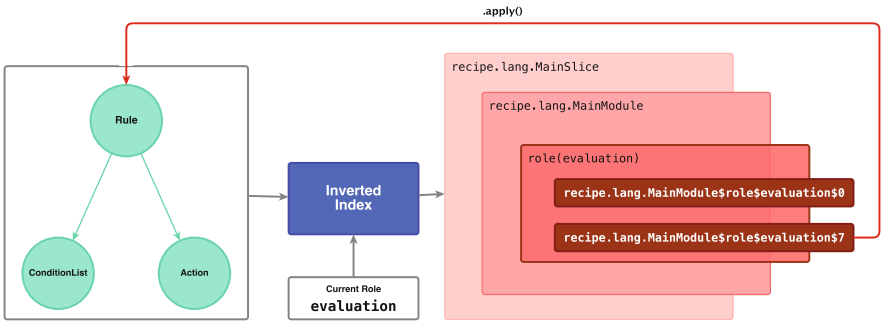

Fig. 1. The Component Manager and the method dispatching procedure

all these components need to be pre-compiled into JVM class files. Compilable source files are translated from Neverlang source files using the `nlgc` tool. These compilable source files can then be given as input to their corresponding native compilers (i.e., `.java` files will be sent to `javac`, `.scala` source files might be compiled using `scalac`, and so on) to generate their own class files.

The compiled **`language`** component implements the *main* interpreter class (a subclass of `Language`), and includes the public API to interact with the interpreter. Using a Neverlang-generated compiler is usually as easy as instantiating this class and then invoking the method `Language.eval(String source)` on a source string, which returns an evaluated AST. The `Language` subclass then loads every declared **`slice`**-related class, which in turn causes every **`module`**-related class to be loaded. Each semantic action is then loaded on-demand, only when the AST visiting procedure requires them to become available (in other words, if certain syntax is never used in an input file, the corresponding semantic action will never be loaded into memory).

Internally, when an input file is given to the generated compiler, the parser constructs an AST. Then, for each role that has been defined in the **`language`** construct, the tree is visited. For each node, if a semantic action has been defined, it should be executed. The component manager is responsible for this form of *method dispatching*: it executes the correct semantic action by invoking its `apply()` method on that node. For instance, if current role is `evaluation`, then, each time a node n containing a nonterminal N, pertaining to a production p is being evaluated, the component manager queries an inverted index to retrieve the slice s from which p has been imported. Then, the corresponding semantic action `sa` of the `evaluation` role imported from s (if any) is applied to the node (`sa.apply(`n`)`). In the process, new modules and new semantic actions might be caused to be loaded from disk. For instance, consider rule (1): the corresponding AST (if we ignore terminals) is like the one in Fig. 1. When the component manager visits the root node of this subtree (labeled with `Rule`), it queries the inverted index for the corresponding semantic action for the `evaluation` role. In this case there is one binding to nonterminal 7. The inverted index returns the right semantic action object, possibly loading the class file from disk, and it then invokes the `apply()` method on the root node.

Incremental Generation of LALR Parsers: DEXTER. In order to support componentisation and runtime composability, we developed DEXTER: the Dynamically EX-

TEnsible Recognizer. DEXTER builds a LALR parser from the production in the modules, parses the input that is given to the compiler and it constructs the AST that the component manager visits. DEXTER implements an in-memory LALR parser generator that can be incrementally extended (*grown*) or restricted (*shrunk*) by adding and removing grammar productions on-the-fly[8] In fact, the `syntax` role of a module is a straight translation from the Neverlang DSL to a series of Java API calls to the DEXTER component. For instance, the production in (1) becomes where `p` is a method that

```
p(nt("Rule"), /* <- */ "when", nt("Condition"), ":", nt("Action"))
```

takes a nonterminal (the left-hand side of a production rule), and then a list of symbols (the right hand side of a rule), and it returns a `Production` instance; `nt` is a method that returns a nonterminal symbol object instance, and string literals are converted to terminal symbol instances. The DEXTER parser generator implements an algorithm that bears some resemblance to those described in [20] and [19]. The algorithm *updates* the LR(0) DFA, which is the basis for many interesting parsers of the LR family, such as GLR and of course LALR, the one adopted in DEXTER. The DEXTER component includes an extensible regex-based lexer that allows to define lexemes at runtime. This subcomponent is called LEXTER. Lexemes are defined inline in a production, whether they are keywords or patterns. Patterns are delimited by slashes, while keywords are delimited by quotes. See Listing 9 for an example of both.

4 Neverlang 2 in Action

In this section we describe the implementation of (part of) the Recipe DSL. We will first show how to support the example in Listing 2, then we will see how Neverlang 2 makes easier to extend the DSL. We will also show that it is possible to change its syntax while still leaving the semantic code unaffected.

4.1 Implementing the Recipe DSL

In Sect. 2, we showed a short snippet from our DSL Recipe. The interpreter for our language will obviously need an internal engine to execute actions when the specified conditions are met. We will not discuss the implementation details of this engine, since they would anyway depend on the particular software platform in use. We will concentrate on the development of the actual language interpreter. In Sect. 3 we described how Neverlang puts together the separate components that make up an interpreter (or a compiler). Our language needs at least three slices. One slice should define the general look of the DSL. A Recipe program is a list of rules, so we expect the first slice to describe this. A rule is supposed to express some truth condition, and then some action to take when the condition holds. As conditions and actions describe single features of our language, it will make sense to write one slice for each one of them. The main step will be to define three modules; then we will write one slice for each one of them, so that the component manager will be able to put them together.

[8] The result of the computation can be still cached to disk for performance, though.

```
slice recipe.lang.MainSlice {
  module recipe.lang.MainModule with role syntax evaluation
}
```

Listing 7: The `MainSlice` slice for Recipe

```
language recipe.lang.Recipe {
  slices recipe.lang.MainSlice recipe.lang.TimeRangeConditionSlice
      recipe.lang.RingerActionSlice
  roles evaluation
}
```

Listing 8: The `language` definition for Recipe

Main Module. In Listing 3 we defined that a `Rule` in our language will be always in the form specified in (1). That is, the keyword `when`, a single condition or a list of conditions separated by the `and` keyword, a colon symbol, and then the action to execute. You may notice that no action has been attached to nonterminals 2, 4, 10, 12. Action 0 contains a reference to `RuleList` (`$1`). Because collecting a list of attributes is a common usage pattern, Neverlang 2 makes available a library method for this purpose. In this case, the built-in library method `AttributeList.collectFrom($1,"ruleObj")` visits the AST subtree rooted at `RuleList` and collects the values attached to the attribute `ruleObj` of each `Rule` node, which is set in action 7. The same method call appears in the action 7, where it collects `"condition"` attributes in a `ConditionList`. It follows that modules that specify conditions shall fill this attribute. In fact, were this attribute not found, the system would raise an exception. For the sake of simplicity, in our example, conditions can be only connected by `and`.

Conditions and Actions. Conditions could be encapsulated into `Condition` object instances that would provide a `boolean check()` method that evaluates to true when the specified condition holds. The "time range" condition has been implemented in Listing 9. In this module, the condition is encapsulated into a `TimeRangeCondition` object (that would be a subclass of `Condition`). The starting and ending times are captured using a *terminal pattern* (described in Sect. 3.3): the *hash* notation `#N` references *N*-th pattern. In this case, numbering is per-rule, starting from 0. Therefore `#0` references the pattern in the second rule[9]. The predefined property `#N.text` contains the matched text.

```
module recipe.lang.TimeRangeConditionModule {
  role(syntax) {
     Condition <- "time" "is" "between" Time "and" Time ;
     Time      <- /[0-9]{1,2}:[0-9]{2}(am|pm)/ ;
  }
  role(evaluation) {
    0 .{ $0.condition = new TimeRangeCondition($1.time, $2.time); }.
    3 .{ $3.time = #0.text; }.
  }
}
```

Listing 9: Time condition

[9] The pattern captures any possible date in the form `HH:MMam/pm`. Of course it might match malformed times such as `27:99pm`; in that case the `TimeRangeCondition` constructor could raise an exception

```
module recipe.lang.RingerActionModule {
  role(syntax) {
     Action <- "turn" "ringer" OnOff ;
     OnOff  <- "on" ;  OnOff  <- "off" ;
  }
  role(evaluation) {
    0 .{ $0.action = new RingerAction($1.ringerIsOn); }.
    2 .{ $2.ringerIsOn = true;  }.
    3 .{ $3.ringerIsOn = false; }.
  }
}
```

Listing 10: Ringer action

```
String script = "when time is between 11:00pm and 7:00am : turn ringer off.";
ASTNode tree = new Recipe().eval(script);
List<RuleObj> rules = tree.getValue("rules");
RecipeSys.registerRules(rules);
```

Listing 11: Retrieving the result of the evaluation of the input string

The "toggle ringer" action could be encapsulated into a subclass of a generic `Action` object that could provide a method `perform()` implementing the logic (in this case, interfacing with the system ringer and turning it on or off). In Listing 10 a `RingerAction` object is instantiated with a boolean representing the ringer status.

Slices and Language. We can now define three slices (one for each module) like that in Listing 7, and then add all of them to the `language` definition for Recipe (Listing 8). Once everything has been passed through `nlgc` and compiled using `javac`, we can already use the `Recipe` object. When the `Recipe.eval(source)` method is invoked on the input string in Listing 2, it puts on the root node of the AST (`Program`) an attribute `ruleList`, which contains a one-element list of `RuleObj` instances (Listing 3). Then each rule can be passed on to the system that will put them into effect at the right time (Listing 11).

4.2 Extending the DSL

The Recipe interpreter can be wrapped up in a package and possibly deployed on the target device. Now, suppose that we want to extend the DSL to support location-based conditions. For instance, mobile devices may allow users to indicate a particular location as *home*. Our original recipe turned off the phone ringer when time was in a customizable range. However, during this time frame the phone user might be away from home, maybe even in a noisy place: in this case, the action should *not* be triggered, because we would prefer the ringer to be on. We would like to add a new feature: a location-dependent condition, "`my position is <location-name>`", that should be evaluated together with the time range condition. We want our Recipe script to turn the ringer off not only when time is in the given range, but when we are also at home (Listing 12). In Listing 13 is the Neverlang code that implements the new feature. Extending the DSL will be as easy as defining a simple slice (similar to that in Listing 7) and adding it to the `language` construct (Listing 8).

```
when my position is home
   and time is between 11:00pm and 7:00am : turn ringer off.
```

Listing 12: Extended Recipe DSL with location support

```
module recipe.lang.LocationModule {
  role(syntax) {
     Condition <- "my" "location" "is" PredefinedLocation ;
     PredefinedLocation <- "home" ;
  }
  role(evaluation) {
    0 .{ $0.condition = new LocationCondition($1.location); }.
    2 .{ $2.location  = RecipeSys.getHomeLocation(); }.
  }
}
```

Listing 13: Location condition

4.3 Localisation

Slices are not just a way to advertise a feature to the component manager. Their real power is to allow to pick features from different modules and mix them together. In this example, we will pick the `evaluation` role defined in the previous `modules`, and apply it to a different (although similar) syntax definition. In particular, we will show that we can *localise* our DSL into another language, with very little effort. In Listing 14 we are showing a Recipe script that has the same meaning as the one in Listing 2 but written in Italian. In Listing 15 we are showing two modules with only one syntax definition each: the first redefines the syntax for the time range condition, and the second redefines the ringer action. As you can see, in both cases no semantic action is specified. In fact, we can reuse the semantic actions we defined in the English modules, because the syntax did not change the order of the nonterminals. Therefore, the action that will be performed when visiting a certain nonterminal will be the same as if the script were written in English. The change does not require the developer to recompile any of the older modules, which are unaffected by the change. In this case, the only components which need compiling are the affected slices and the new modules. Of course, this example is only meant to show that slices give great flexibility to language developers, and not to provide a compelling example for localisation, which is an entirely different matter. In this case, the syntax of the Italian language does not affect the way nonterminals are ordered, but this could very well happen, even in non-natural languages: we are currently working on a solution to this kind of problem (more in Sect. 6). A deeper discussion on how Neverlang supports DSL evolution can be found in [9].

5 Related Work and Discussion

MontiCore [22] is a framework for language composition and extension that provides grammar inheritance and rewriting mechanisms additionally to modularisation features. However the underlying parser generator is still traditional (ANTLR [26]), in the sense that parser-related code has to be recompiled from scratch most of the time the user updates the grammar. The Rats! [17] packrat parser generator makes possible to share and

```
quando l'orario è tra 11:00pm e 7:00am : spegni la suoneria.
```

Listing 14: Localised Recipe DSL

reuse parser components by organising grammar fragments into *modules*. The Rats! module system makes possible to extend and programmatically rewrite existing grammar fragments in a way that resembles a grammar-tailored inheritance system. For instance, a Rats! module can import rules from another module, substitute symbols, and add new productions. However, Rats! is a traditional parser generator that generates Java code. We employed Rats! in the earlier versions of Neverlang, but, of course, the code-generation approach makes impossible separate compilations and sharing precompiled components. Neverlang 2's DEXTER dynamic LALR parser generator does not yet support the same variety of operations on a grammar, but we are currently investigating in this direction. Given the dynamic and in-memory nature of DEXTER-generated parsers, we are confident that implementing features such as namespacing and symbol substitution would require only little effort, while adding rules is already possible.

As seen in [8], the JastAdd [18] compiler construction system was similar to Neverlang's first implementation in that it separates compilation aspects and implemented the AST using the traditional OOP style, generating all the required Java classes that were injected with methods and fields using AOP. The newest Neverlang architecture presents a runtime-generated AST; code is no more injected by way of weaving: instead, the component manager (Sect. 3.3) performs method dispatching depending on the AST node contents. This choice dispenses Neverlang 2 from needing a weaver, and greatly reduced the time to generate and compile the resulting compiler. Moreover, now Neverlang 2 components can be compiled independently and only when needed, and semantic actions can be expressed in any language supported by the JVM. On the other hand JastAdd does not focus on code reuse, it does not separates components nor optimises for pre-compiled code reuse, and it only supports Java.

Several tools deal with the problem of DSL embedding [11], where a host language embeds another language for specific purposes (e.g., SQL or XML literals). For instance, Metafront [5] and Metaborg [6] (part of the Stratego/XT toolset) are tools designed to perform syntactic transformation between programming languages typically to extend a host programming language with an embedded DSL. However, because the problem they are trying to solve is rather different than achieving modularity in the development of one programming language, as in Neverlang, these tools do not really take into account feature or component sharing. Their related literature made still for an interesting read during the development of the DEXTER extensible parser; in particular, in [7] the authors discuss an algorithm for LR parser extension, which is different from DEXTER's, though. In fact, [7] updates the LR(0) ε-DFA, while DEXTER, more similarly to [20,19], applies the updating procedure on the actual LR(0) DFA. This direct updating approach makes possible to avoid an otherwise required additional transformation step, that is, from ε-DFA to LR(0) DFA. This in turn cuts down on the requirement of keeping the intermediate representation available for any subsequent update. SugarJ [15] uses Stratego to provide syntactic transformation to Java programs in the form of library bundles.

```
module recipe.lang.TimeRangeConditionIt {
  role(syntax) {
     Condition <- "l'orario" "è" "tra" Time "e" Time ;
     Time      <- /[0-9]{2}:[0-9]{2}(am|pm)/ ;
  }
}
slice recipe.lang.TimeRageConditionSlice {
  module recipe.lang.TimeRangeConditionIt with role syntax
  module recipe.lang.TimeRangeConditionModule with role evaluation
}
module recipe.lang.RingerActionIt {
  role(syntax) {
     Action <- OnOff "la" "suoneria";
     OnOff  <- "spegni" ;  OnOff  <- "accendi" ;
  }
}
slice recipe.lang.RingerActionSlice {
  module recipe.lang.RingerActionIt with role syntax
  module recipe.lang.RingerActionModule with role evaluation
}
```

Listing 15: Localising the Recipe DSL using Neverlang 2

The xText [14] project is a framework and language workbench for the model-based development of DSLs that tightly integrates with EMF [28]. The framework makes possible to reuse existing grammars and existing meta-models to implement other languages, but it is really meant for model-driven development and therefore is conceptually different from Neverlang. It uses ANTLR to generate the parser. The framework includes xBase [13] a «partial programming language» that can be used as a base for other DSLs, and xSemantics, a DSL for writing «type systems, reduction rules and in general relation rules for languages implemented in xText» [3]. MPS [29] is another language workbench with similar objectives as xText, but it is backed and developed by the JetBrains software company.

LISA [23] is a language workbench and compiler generator that uses inheritance to compose grammars. Similarly to Neverlang it uses attribute grammars to express a language, but it bases on the concept of inheritance to extend and compose syntax and semantics attached the rules. It even includes AOP-like constructs to hook into nonterminals add possibly inject cross-cutting behaviour. Even though inheritance and this kind of AOP construct enable to both layer new semantic actions on top of the others and to override a behaviour altogether, they do not really make possible to define distinct compilation phases. Moreover, LISA is a more traditional compiler generator, in the sense that it outputs Java code using a traditional parser generator; semantic actions are expressed in a Java dialect.

The SPARK toolkit [2] for DSL implementation has similar goals to Neverlang, but it is Python-based. The most interesting part of SPARK is the somewhat curious choice for the parser generator, Earley [12], which is justified by the target audience for the project, that includes users that do not have a background in parser and compiler definition: Earley parsers can handle any context-free grammar, even ambiguous ones. Nevertheless, this comes at the cost of a higher computational complexity than, for instance, LALR. Beside this, SPARK takes a more traditional approach in the definition of the components of a compiler, it does not really account for modularisation or feature sharing and, of course, it is limited to Python.

For completeness, we want to mention π [21], an experimental programming language where the only construct is the *pattern*, i.e., a mapping between a syntax definition and its intended semantic interpretation. Programs written using this language can extend their own syntax and express the new semantics inline. From that point on, the programmer can employ the new constructs anywhere in a program. Something similar can be found, from a more parsing-related perspective, in [27]; in fact, they both employ Earley parsing as well. These proposals differ from Neverlang in that they are more close to metaprogramming techniques and reflective systems; on the other hand, Neverlang is not really a programming language in itself, but rather a framework to define new languages.

6 Conclusions and Future Work

In this paper we described Neverlang 2 and its architecture. The strengths of this new versions are the modular implementation system, which makes possible feature sharing, even when components have been pre-compiled, and the full JVM support, that makes possible to implement the DSL semantics using any language supported by the Java platform.

We are currently working on extending our implementation to make it more robust, with respect to composition. For instance, namespacing and symbol importing may be useful to avoid name clashes when composing grammar fragments. Similarly, symbol renaming could be supported to compose syntax roles while carefully avoiding unexpected behaviour. Programmatic symbol renumbering could be also a way to reuse semantic actions in modules even when keywords in the syntax role occur in a different order (cf. Sect. 4.3). We are currently working on a way to carry on this kind of transformation in a semi-automatic way, by providing a mapping between abstract syntax trees in the composition phase. We are also planning to support layering of roles, that is, not only evaluating distinct roles as distinct phases, but also being able to group roles as part of the same phase (e.g., decorating the `evaluation` role with a `logging` role).

Nevertheless, we believe that Neverlang's current feature set is already promising. In order to stress-test the Neverlang framework, our lab has already implemented a reusable exception handling mechanism, and we are currently developing a modularised Java compiler. The project is already being employed to develop real-world DSLs both in the research and in the industry area: the development of a query DSL for TheMatrix [16] which is in the final testing phase at the time of writing, and the development of a new DSL for ERP software implementation.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Aycock, J.: The Design and Implementation of SPARK, a Toolkit for Implementing Domain-Specific Languages. J. of Computing and Inform. Tech. 10(1), 55–66 (2004)
3. Bettini, L.: Implementing Java-like Languages in xText with xSemantics. In: Proc. of SAC'13, Coimbra, Portugal, March 2013, pp. 1559–1564. ACM Press, New York (2013)
4. Bosch, J.: Delegating Compiler Objects. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 326–340. Springer, Heidelberg (1996)

5. Brabrand, C., Schwartzbach, M.I.: The Metafront System: Safe and Extensible Parsing and Transformation. J. Science of Computer Programming 68, 2–20 (2007)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. J. Science of Comp. Progr. 72(1-2), 52–70 (2008)
7. Bravenboer, M., Visser, E.: Parse Table Composition. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
8. Cazzola, W.: Domain-Specific Languages in Few Steps. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 162–177. Springer, Heidelberg (2012)
9. Cazzola, W., Poletti, D.: DSL Evolution through Composition. In: Proc. of RAM-SE'10, Maribor, Slovenia, June 2010, ACM Press, New York (2010)
10. Cazzola, W., Vacchi, E.: DEXTER and Neverlang: A Union Towards Dynamicity. In: Proc. of ICOOOLPS'12, Beijing, China, June 2012, ACM Press, New York (2012)
11. Dinkelaker, T., Eichberg, M., Mezini, M.: An Architecture for Composing Embedded Domain-Specific Languages. In: Proc. of AOSD'10, Saint-Malò, France, pp. 49–60 (2010)
12. Earley, J.: An Efficient Context-Free Parsing Algorithm. Commun. ACM 13(2), 94–102 (1970)
13. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: xBase: Implementing Domain-Specific Languages for Java. In: Proc. of GPCE'12, Dresden, Germany, Sep. 2012, pp. 112–121. ACM Press, New York (2012)
14. Efftinge, S., Völter, M.: oAW xText: A Framework for Textual DSLs. In: Proc. of the EclipseCon Summit Europe 2006 (ESE'06), vol. 32, Esslingen, Germany (Nov. 2006)
15. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: Library-Based Syntactic Language extensibility. In: Proc. of OOPSLA'11, Portland, OR, USA, Oct. 2011, pp. 391–406 (2011)
16. Gini, R.: Frameworks for Data Extraction and Management from Electronic Healthcare Databases for Multi-Center Epidemiologic Studies: a Comparison among EU-ADR, MATRICE, and OMOP Strategies. Keynote (Nov. 2012)
17. Grimm, R.: Practical Packrat Parsing. TR2004-854, NYU, New York, NY, USA (2004)
18. Hedin, G., Magnusson, E.: JastAdd — An Aspect-Oriented Compiler Construction System. Science of Computer Programming 47(1), 37–58 (2003)
19. Heering, J., Klint, P., Rekers, J.: Incremental Generation of Parsers. IEEE Trans. Softw. Eng. 16(12), 1344–1351 (1990)
20. Horspool, R.: Incremental Generation of LR Parsers. J. Comp. Lang. 15(4), 205–223 (1990)
21. Knöll, R., Mezini, M.: π: A Pattern Language. In: OOPSLA'09, pp. 503–522 (2009)
22. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: A Framework for Compositional Development of Domain Specific Languages. J. SW Tools for Techn. Transfer 12(5), 353–372 (2010)
23. Mernik, M., Žumer, V.: Incremental Programming Language Development. Computer Languages, Systems and Structures 31(1), 1–16 (2005)
24. Ng, K., Warren, M., Golde, P., Hejlberg, A.: The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis. White paper, Microsoft (Oct. 2011)
25. Odersky, M.: Reflection and Compilers. Keynote at Lang.NEXT (Apr 2012)
26. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software—Practice and Experience 25(7), 789–810 (1995)
27. Stansifer, P., Wand, M.: Parsing Reflective Grammars. In: Proc. of LDTA'11, Saarbrucken, Germany, March 2011, pp. 10:1–10:7. ACM Press, New York (2011)
28. Steinberg, D., Budinsky, D., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, Dec. 2008. Addison-Wesley, Reading (2008)
29. Völter, M., Pech, V.: Language Modularity with the MPS Language Workbench. In: Proc. of ICSE'12, Zürich, Switzerland, June 2012, pp. 1449–1450. IEEE Computer Society Press, Los Alamitos (2012)

Preserving Confidentiality in Component Compositions

Andreas Fuchs and Sigrid Gürgens

Fraunhofer Institute for Secure Information Technology SIT
Rheinstr 75, 64295 Darmstadt, Germany
{andreas.fuchs,sigrid.guergens}@sit.fraunhofer.de

Abstract. The preservation of any security property for the composition of components in software engineering is typically regarded a non-trivial issue. Amongst the different possible properties, confidentiality however poses the most challenging one. The naive approach of assuming that confidentiality of a composition is satisfied if it is provided by the individual components may lead to insecure systems as specific aspects of one component may have undesired effects on others. In this paper we investigate the composition of components that each on its own provide confidentiality of their data. We carve out that the complete behaviour between components needs to be considered, rather than focussing only on the single interaction points or the set of actions containing the confidential data. Our formal investigation reveals different possibilities for testing of correct compositions of components, for the coordinated distributed creation of composable components, and for the design of generally composable interfaces, ensuring the confidentiality of the composition.

1 Introduction

Software design and engineering makes strong use of composition in many ways. From the orchestration of web services in a Business Process Engines to the integration of libraries or object files compilers and linkers the principles of composition apply on any of these layers of abstraction.

Beyond the general problems of feature interaction, there exist many specific security related challenges that can introduce serious flaws in a software product. A prominent example for such a flaw is the integration of TLS libraries into the German eID Application [1] that caused the acceptance of update packages by any server with a valid certificate, as the name within the certificate was not checked. In other cases, integrators of TLS libraries do not provide enough entropy for key generation which leads to a series of servers on the Internet with similar private key values [2].

Practical solutions for composition include the provisioning of verbal best-practice catalogues [3], tool-based solution databases [4,5,6] or guides, tutorials and code examples in general. However, there does not exist much research that targets the challenges imposed by composition on a more general and broader scope.

In this contribution we present an approach based on the formal semantics of our Security Modelling Framework SeMF (see e.g. [7,8,9]) that targets the investigation and validation of general component composition regarding the property of data confidentiality. SeMF has available a comprehensive vocabulary for statements of confidentiality that provides the necessary expressiveness to reason about conditions of general composability decoupled from any specific scenario.

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 33–48, 2013.

In the following Section we introduce a scenario that serves as test case for our approach. It is composed of two components that (each on its own) provide a certain confidentiality property, but fail to do so when composed into a joined system. Section 3 gives a brief introduction to the SeMF framework. In Section 4 we introduce our formalization of system composition and demonstrate it using the example scenario. We then explain and formalize the conditions for confidentiality composition in Section 5 and illustrate them by means of the example scenario in Section 6. Section 7 provides an overview of related work on composition of security and Section 8 finalizes the paper and provides an outlook to ongoing and future work.

2 Example Scenario

The provisioning and quality of entropy is a central aspect for many security functionalities. However, the generation of entropy and randomness in computers is a hard problem on its own [10] and at the same time programmers are usually not introduced to its challenges in a correct way. Many code examples and explanations for randomness today advice to use the current time or uptime as seed for a random number generator. This approach may be adequate for desktop applications started by the user at an unforeseeable as well as undetectable point in time. Whenever these conditions do not hold however – as is the case especially in e.g. system service applications or embedded platforms [11] – such date/uptime values do not provide enough entropy. These scenarios rather require specialized entropy sources in CPU through a TPM or a SmartCard.

In our example scenario, we investigate such a case, i.e. a system which is composed of a security library for key generation that targets desktop applications whilst being utilized by an embedded platform system service. The $KeyGenerator$ component hereby uses the current time of the system when being called in order to initialize its random number generator and to create the corresponding key. The $Application$ component of the system represents a system service that is started during boot and calls the $KeyGenerator$ for a key to be generated. Both components have the property of confidentiality for the key that is generated / further used. However, their composition introduces side effects that make the key calculable for a third party.

3 Formal Semantics of SeMF

In our Security modelling Framework SeMF, the specification of any kind of cooperating system is composed of (i) a set $\mathbb{P}$ of agents (e.g. an application and a key generator), (ii) a set Σ of actions, (iii) the system's behaviour $B \subseteq \Sigma^*$ (Σ^* denoting the set of all words composed of elements in Σ), (iv) the local views $\lambda_P : \Sigma^* \rightarrow \Sigma_P^*$, and (v) initial knowledge $W_P \subseteq \Sigma^*$ of agents $P \in \mathbb{P}$. The behaviour B of a discrete system S can be formally described by the set of its possible sequences of actions (which is always prefix closed). An agent P's initial knowledge W_P about the system consists of all traces the agent initially considers possible. This includes a representation of conclusions that an agent may be able to derive; i.e. that the reception of a message implies the sending of this message to have happened before. Finally, an agent's local view essentially captures what an agent can see from the system. Together, the local view and initial knowledge

represent what an agent may know about the system at a given point in time based on what he/she knows in general, has seen and has concluded from this. Different formal models of the same system are partially ordered with respect to the level of abstraction. Formally, abstractions are described by alphabetic language homomorphisms that map action sequences of a finer abstraction level to action sequences of a more abstract level while respecting concatenation of actions. In fact, the agents' local views are expressed by homomorphisms. Note that homomorphisms are in general neither injective nor surjective. For $\Sigma_1 \subseteq \Sigma_2$, the homomorphism $h : \Sigma_2 \longrightarrow \Sigma_1$ that keeps all actions of Σ_1 and maps those in $\Sigma_2 \setminus \Sigma_1$ onto the empty word is called *projection homomorphism*.

In SeMF, security properties are defined in terms of such a system specification. Note that system specification does not require a particular level of abstraction. The underlying formal semantics then allows formal validation, i.e. allows to prove that a specific formal model of a system provides specific security properties.

3.1 Confidentiality in SeMF

Based on the SeMF semantics, we have specified various instantiations of security properties such as precedence, integrity, authenticity and trust (see e.g. [7,12,13]). In this paper however we focus on our notion of parameter confidentiality [8,9]. Various aspects are included in this concept. First, we have to consider an attacker Eve's local view λ_{Eve} of the sequence ω she has monitored and thus the set of sequences $\lambda_{Eve}^{-1}(\lambda_{Eve}(\omega))$ that are, from Eve's view, identical to ω. Second, Eve can discard some of the sequences from this set, depending on her knowledge of the system and the system assumptions, all formalized in W_{Eve}. For example, there may exist interdependencies between the parameter p to be confidential in different actions, such as a credit card number remaining the same for a long time, in which case Eve considers only those sequences of actions possible in which an agent always uses the same credit card number. The set of sequences Eve considers possible after ω is $\lambda_{Eve}^{-1}(\lambda_{Eve}(\omega)) \cap W_{Eve}$. Third, we need to identify the actions in which the respective parameter(s) shall be confidential. Many actions are independent from these and do not influence confidentiality, thus need not be considered. For this we use a homomorphism $\mu : \Sigma^* \longrightarrow (\Sigma_\tau \times M)^*$ that maps actions to be considered onto a tuple $(actiontype, parameter)$.

Essentially, parameter confidentiality is captured by requiring that for the actions that shall be confidential for Eve with respect to some parameter p, all possible (combinations of) values for p occur in the set of actions that Eve considers possible. What are the possible combinations of parameters is the fourth aspect that needs to be specified, as we may want to allow Eve to know some of the interdependencies between parameters (e.g. in some cases Eve may be allowed to know that the credit card number remains the same, in others we may want to require Eve not to know this). The notion of (L, M)–Completeness captures which are the dependencies allowed to be known within a set of sequences of actions. For the formal definition of (L, M)–completeness, some additional notations are needed: For $f : M \longrightarrow M'$ and $g : N \longrightarrow N'$ we define $(f, g) : M \times N \longrightarrow M' \times N'$ by $(f, g)(x, y) := (f(x), g(y))$. The identity on M is denoted by $i_M : M \longrightarrow M$, while $M^{\mathbf{N}}$ denotes the set of all mappings from $\mathbf{N}$ to M, and $p_\tau : (\Sigma_t \times M) \longrightarrow \Sigma_t$ is a mapping that removes the parameters.

Definition 1 ((L,M)-completeness) *Let $L \subseteq (\Sigma_t \times \boldsymbol{N})^*$ and let M be a set of parameters. A language $K \subseteq (\Sigma_t \times M)^*$ is called (L, M)–complete if*

$$K = \bigcup_{f \in M^{\boldsymbol{N}}} (i_{\Sigma_t}, f)(L)$$

The definition of parameter confidentiality captures all the different aspects described above:

Definition 2 (Parameter Confidentiality) *Let M be a parameter set, Σ a set of actions, Σ_t a set of types, $\mu : \Sigma^* \to (\Sigma_t \times M)^*$ a homomorphism, and $L \subseteq (\Sigma_t \times \boldsymbol{N})^*$. Then M is parameter-confidential for agent $R \in \mathbb{P}$ with respect to (L, M)-completeness if there exists an (L, M)–complete language $K \subseteq (\Sigma_t \times M)^*$ with $K \supseteq \mu(W_R)$ such that for each $\omega \in B$ holds*

$$\mu(\lambda_R^{-1}(\lambda_R(\omega)) \cap W_R) \supseteq p_\tau^{-1}(p_\tau(\mu(\lambda_R^{-1}(\lambda_R(\omega)) \cap W_R))) \cap K$$

Here $p_\tau^{-1} \circ p_\tau$ first removes and then adds again all values of the parameter that shall be confidential, i.e. constructs all possible value combinations. (L, M)–completeness of K captures that R is required to consider all combinations of parameter values possible except for those that it is allowed to disregard (i.e. those that are not in K). Hence the right hand side of the inequality specifies all sequences of actions agent R shall consider as the ones that have possibly happened after ω has happened. In contrast, the left hand side represents those sequences that R actually does consider as those that have possibly happened. For further explanations we refer the reader to [8,9].

Notation: We will use $\Lambda_R(\omega, W_R) = \lambda_R^{-1}(\lambda_R(\omega)) \cap W_R)$ as an abbreviation.

4 Modelling Composition

Based on SeMF we now introduce the definition of the composition of two systems with the same set of agents and a shared interface. Applying this definition, we then specify the composition of the scenario application and key generator.

4.1 Formalizing Composition

The idea of our formalization is to interpret the individual components S_1 and S_2 as homomorphic images of the composed system and to express this system in terms of the inverses of the components with respect to the homomorphisms. Figure 1 illustrates the relationship between the systems: Both components S_1 and S_2 are abstractions (i.e. images of homomorphisms h_1 and h_2, respectively) of their composition S_0, while S_1 and S_2 in turn are abstracted (by homomorphisms h_1^{IF} and h_2^{IF}, respectively) onto their joined interface. Agent P's initial knowledge about the composition does only contain those sequences that P considers possible for both S_1 and S_2, hence it is given by the intersection of the inverses of the two homomorphisms. Further, agents' local views for the composed system need to capture what agents can see in both S_1 and S_2. The projections of S_1 and S_2 into the interface system will be of interest for a theorem to be introduced in Section 6.2. In the following we formalize this composition approach.

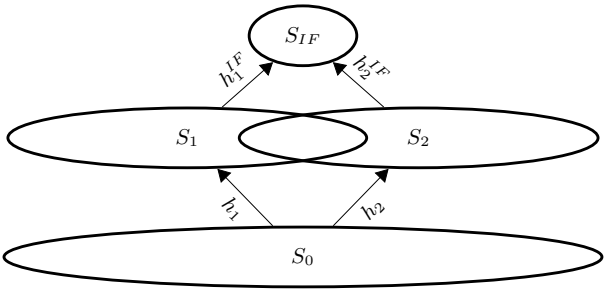

Fig. 1. System relations

Definition 3 (System Composition) *Let S_1 and S_2 be two systems with Σ_i their respective sets of actions, $\mathbb{P}_1 = \mathbb{P}_2$ their set of agents, λ_P^i their agents' local views, and W_P^i their agents' initial knowledge, respectively ($i = 1, 2$). Let further $\Sigma_0 := \Sigma_1 \cup \Sigma_2$, and $h_i : \Sigma_0^* \to \Sigma_i^*$ the projection homomorphisms into Σ_i ($i = 1, 2$). Then the* composition *S_0 of S_1 and S_2 is constructed as follows:*

- $\mathbb{P}_0 := \mathbb{P}_1 = \mathbb{P}_2$,
- $B_0 := h_1^{-1}(B_1) \cap h_2^{-1}(B_2)$
- $W_P^0 := h_1^{-1}(W_P^1) \cap h_2^{-1}(W_P^2)$
- *In order to define the local view of agents in S_0, we define for $i = 1, 2$: $\lambda_P^{i'} : \Sigma_0 \to \Sigma_{P,i}$*
  $\lambda_P^{i'}(a) = \begin{cases} \lambda_P^i(a) \text{ if } a \in \Sigma_i \\ \varepsilon \text{ else} \end{cases}$
  Then the local view of S_0 can be defined as follows:
  $\lambda_P^0(a) := (\lambda_P^{1'}(a), \lambda_P^{2'}(a))$

Further, for $\Sigma_{IF} := \Sigma_1 \cap \Sigma_2$, the projection homomorphisms into Σ_{IF}^* are denoted by $h_i^{IF} : \Sigma_i^* \to \Sigma_{IF}^*$ ($i = 1, 2$).
Note, the above definition is equivalent to $\lambda_P^{1'}(a) = \lambda_P^1(h_1(a))$, $\lambda_P^{2'}(a) = \lambda_P^2(h_2(a))$. Also from the above definition it follows $\Sigma_{0,P} = (\Sigma_{1,P} \times \Sigma_{2,P})$ with $\Sigma_{i,P}$ being the image of λ_P^i ($i = 1, 2$), and $(\lambda_P^0)^{-1}((x, y)) = (\lambda_P^{1'})^{-1}(x) \cap (\lambda_P^{2'})^{-1}(y)$.

4.2 Composing the Scenario Systems

We now model the interface composition of an application (S_1) and a key generation module (S_2) following the above definition. We assume that the application generates a key directly after each system boot. The model for the application is independant from any key generation modul that is actually being used, and abstracts from the actual key generation (this is not part of the application model and happens magically).
The application model S_1 can be specified as follows:

- Agents of this model (and of S_2) are the application, the key genration module, and a third agent that is not allowed to know the key:
  $\mathbb{P}_1 = \{App, KGen, Eve\}$

- The system is booted, the application calls the key generation module, and the key generation module returns a key $key \in \mathcal{K}$, all actions happening at time $t \in T$:
  $\Sigma_1 = \bigcup_{t \in T, key \in \mathcal{K}} \{boot(t), callGenKey(App, t), returnKey(KGen, key, t)\}$
- We assume that Eve can see the time of system boot but can neither see the key generation request nor the key that is returned:
  $\lambda^1_{Eve}(boot(t)) = boot(t); \forall a \in \Sigma_1 \setminus \{boot(t) | t \in T\} : \lambda^1_{Eve}(a) = \varepsilon$
- Eve knows that before a key generation request, the system has been booted. For simplicity we assume the period of time between these two actions to be equal to δ_1. Eve may further know that the time of actions in a sequence is strictly monotonic increasing. This is however not relevant for the given scenario. Hence sequences of actions that contradict this fact are not included in Eve's initial knowledge. Formally:
  $W^1_{Eve} = \Sigma_1^* \setminus \bigcup\limits_{t_j - t_i = \delta_1} (\Sigma_1 \setminus \{boot(t_i)\})^* \{callGenKey(App, t_j)\} \Sigma_1^*$
- We focus on the confidentiality of the key returned to the application, hence μ_1 maps $returnKey(KGen, key, t_j)$ onto $(returnKey(KGen), key)$ and all other actions onto the empty word.

According to this system model, it is easy to see that the returned key is parameter confidential for Eve regarding μ_1 and *(L,M)-completeness* regarding an adequate L and the set of possible keys M.

We now model a concrete key generation module. This module is not able to retrieve a seed for key generation other than the system clock.

- $\mathbb{P}_2 = \{App, KGen, Eve\}$
- The key generation module is called by the application, generates a key, and returns this key, all actions occurring at a specific time $t \in T$:
  $\Sigma_2 = \bigcup_{t \in T, key \in \mathcal{K}} \{callGenKey(App, t), genKey(KGen, key, t), returnKey(KGen, key, t)\}$
- We assume that Eve cannot see any of the actions of the key generation module, hence $\lambda^2_{Eve}(\Sigma_2) = \varepsilon$
- Eve knows that before a key can be generated, the respective key generation call must have happened, and that the time passing between these two actions is at most δ_2. Eve also knows that the key generator only returns keys it has generated before. Eve finally knows that the system time is used as seed for key generation. Formally:

$$W^2_{Eve} = \Sigma_2^* \setminus \bigcup_{t_j - t_i = \delta_2} (\Sigma_2 \setminus \{callGenKey(App, t_i)\})^* \{genKey(App, key, t_j)\} \Sigma_2^*$$

$$\setminus \bigcup_{key_m = key_n} (\Sigma_2 \setminus \{genKey(KGen, key_m, t_j\})^* \{returnKey(KGen, key_n, t_k)\} \Sigma_2^*$$

$$\setminus \bigcup_{key = k(t_j)} (\Sigma_2 \setminus \{genKey(KGen, key, t_j\})^*$$

- As above, we focus on the confidentiality of the key returned to the application, hence μ_2 maps $returnKey(KGen, key, t_j)$ onto $(returnKey(KGen), key)$ and all other actions onto the empty word.

Also in this system model, it is easy to see that the returned key is parameter confidential for Eve regarding μ_2 and *(L,M)-completeness* regarding the same L and set of possible keys M.
Following Definition 3 we can now construct the composed system S_0 with $\Sigma_{IF} = \bigcup_{t\in T, key\in\mathcal{K}}\{callGenKey(App,t), returnKey(KGen,key,t)\}$:

- $\mathbb{P}_0 = \{App, KGen, Eve\}$
- $\Sigma_0 = \bigcup_{t\in T,key\in\mathcal{K}}\{\ boot(t), callGenKey(App,t),$
  $genKey(KGen,key,t), returnKey(KGen,key,t)\}$
- $\lambda^0_{Eve}(boot(t)) = (boot(t),\varepsilon), \forall a \in \Sigma_0 \setminus \bigcup_{t\in T}\{boot(t)\} : \lambda^0_{Eve}(a) = (\varepsilon,\varepsilon)$
- $W^0_{Eve} = \Sigma_0^* \setminus \bigcup\limits_{t_j - t_i = \delta_1} (\Sigma_0 \setminus \{boot(t_i)\})^* \{callGenKey(App,t_j)\}\Sigma_0^*$
  $\setminus \bigcup\limits_{t_k - t_j = \delta_2} (\Sigma_0 \setminus \{callGenKey(App,t_j)\})^*$
  $\{genKey(App,key,t_k)\}\Sigma_0^*$

  $\setminus \bigcup\limits_{key_m = key_n} (\Sigma_0 \setminus \{genKey(KGen,key_m,t_k\})^*$
  $\{returnKey(KGen,key_n,t_l)\}\Sigma_0^*$

  $\setminus \bigcup\limits_{key = k(t_j)} (\Sigma_0 \setminus \{genKey(KGen,key,t_k\})^*$

The question that now needs to be answered is whether or not confidentiality is preserved in this system composition. In the following section, we will introduce theorems that can be used to answer this question.

5 Investigating the Composition of Confidentiality

In this section we provide sufficient conditions under which a composition of two systems preserves the confidentiality properties of each of its components. We start with a very generic approach that is most broadly applicable – however depends on concrete inquiry regarding the satisfaction of the sufficient conditions. Then we provide two more specialized conditions that are less broadly applicable but easier testable.

For each of these cases we first provide a verbal explanation of the concept and then its formal representation. Readers not interested in these formalizations may skip the latter parts. The formalizations all refer to the representation of composition as described in the previous section. An application to the example scenario will be given in Section 6.

For the proofs in this Section we utilize the following lemmata and considerations: The first lemma provides a relation between the local view in the composed system based on the local views from each of the component systems within the integration. This directly reflects the construction rules from Definition 3:

Lemma 1. $(\lambda^0_P)^{-1}(\lambda^0_P(\omega)) = h_1^{-1}((\lambda^1_P)^{-1}(\lambda^1_P(h_1(\omega))))\cap$
$h_2^{-1}((\lambda^2_P)^{-1}(\lambda^2_P(h_2(\omega))))$

Proof. $(\lambda^0_P)^{-1}(\lambda^0_P(\omega)) = (\lambda^0_P)^{-1}((\lambda^{1'}_P(\omega), \lambda^{2'}_P(\omega))$
$= (\lambda^{1'}_P)^{-1}(\lambda^{1'}_P(\omega)) \cap (\lambda^{2'}_P)^{-1}(\lambda^{2'}_P(\omega))$
$= h_1^{-1}((\lambda^1_P)^{-1}(\lambda^1_P(h_1(\omega)))) \cap h_2^{-1}((\lambda^2_P)^{-1}(\lambda^2_P(h_2(\omega))))$

Given a composition we need to find the traces of actions in Component 1 that correspond to those traces in Component 2 – and vice versa. The construction of these relations can be performed via the interface system S^i as well as via the composed system S^c as expressed by the following lemma:

Lemma 2. *Given a system composition as in Definition 3,* $h_1 \circ h_2^{-1} = (h_1^{IF})^{-1} \circ h_2^{IF}$.

Proof. For $x \in \Sigma_2^*$ always holds $h_2^{IF}(x) = h_1(x)$. For $\Diamond$ denoting the shuffle product, $h_1(h_2^{-1}(x)) = h_1(x \Diamond (\Sigma_1 \setminus \Sigma_2)^*) = h_1(x \Diamond (\Sigma_1 \setminus \Sigma_{IF})^*) = h_1(x) \Diamond (\Sigma_1 \setminus \Sigma_{IF})^* = h_2^{IF}(x) \Diamond (\Sigma_1 \setminus \Sigma_{IF})^* = (h_1^{IF})^{-1}(h_2^{IF}(x))$.

For arbitrary sets X and Y and $A, C \subseteq X$, $B, D \subseteq Y$ and a mapping $f : X \longrightarrow Y$ we always have the equality $f^{-1}(B) \cap f^{-1}(D) = f^{-1}(B \cap D)$, but only the inclusion $f(A \cap C) \subseteq f(A) \cap f(C)$. However, for particular intersections we have equality:

Lemma 3. *Let* X, Y *be arbitrary sets,* $f : X \longrightarrow Y$ *a mapping, and* $A \subseteq X, B \subseteq Y$. *Then* $f(A \cap f^{-1}(B)) = f(A) \cap B$.

For the proof of this lemma we refer the reader to [9].

5.1 General Conditions for Confidentiality Composition

The definition of confidentiality in SeMF relies on the extraction and testing of those actions and data that are identified as being confidential. This extraction is applied to every state that the system may take and bases on what an attacker has observed up to this point and what she can deduce from these observations through her initial knowledge.

When two systems that both provide confidentiality are composed into a new system (w.r.t. to some common interface), the conclusion about some data that an attacker may derive at any given state in the composed system is the combination of conclusions she has derived with regards to each of the components. If this combination results in what the attacker is allowed to know in the system composition, then obviously confidentiality is satisfied in the composition.

Within the semantics for confidentiality of SeMF this combination of conclusions about the sequences that may have happened in the individual systems and the value of data used in these sequences is represented as the intersection of these sets – i.e. the smaller a set becomes the more conclusions an attacker can draw, because she considers less values as possible candidates for the confidential data.

It should be noted though that these considerations have to be executed for every state – i.e. every possible sequence of actions – that the system may take. Further, they require a level of detail that would allow for the direct assessment of confidentiality of the composed system instead. However, while these conditions are of less practical relevance, they form the basis for the more restricted conditions presented in the subsequent sections. Formally this approach can be expressed as follows:

Definition 4 *Given a composition as defined in Definition 3, we call* h_1 *confidentiality composable with* h_2 *for R with respect to* μ_0, μ_1 *and* μ_2, *if for all* $\omega \in B_0$ *holds:*

$$\mu_0[\Lambda_{R0}(\omega, W_R^0)] = \mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)] \cap \mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)]$$

Theorem 1 *Given a confidentiality composable composition as defined in Definition 4, if S_1 and S_2 both are parameter confidential for agent R with respect to some μ_1 and μ_2 with $\mu_1 \circ h_1 = \mu_2 \circ h_2$, then S_0 is parameter confidential for R with respect to $\mu_0 := \mu_1 \circ h_1 = \mu_2 \circ h_2$ and $L_0 := L_1 = L_2, M_0 := M_1 = M_2$.*

Proof. S_1, S_2 parameter confidential, $h_1(B_0) \subseteq B_1$, $h_2(B_0) \subseteq B_2$ implies $\forall \omega \in B_0$:
$\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)] \supseteq p_t^{-1}(p_t(\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)])) \cap K$ and
$\mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)] \supseteq p_t^{-1}(p_t(\mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)])) \cap K$.
Taking the intersection of these equations leads to
$\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)] \cap \mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)]$
$\supseteq p_t^{-1}(p_t(\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)])) \cap p_t^{-1}(p_t(\mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)])) \cap K$
$= p_t^{-1}[p_t(\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)]) \cap p_t(\mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)]) \cap K]$
$\supseteq p_t^{-1}(p_t(\mu_1[\Lambda_{R1}(h_1(\omega), W_R^1)] \cap \mu_2[\Lambda_{R2}(h_2(\omega), W_R^2)])) \cap K$.
By assumption of h_1 and h_2 being confidentiality composable it follows that
$\mu_0[\Lambda_{R0}(\omega, W_R^0)] = p_t^{-1}(p_t(\mu_0[\Lambda_{R0}(\omega, W_R^0)])) \cap K$.

5.2 Independantly Testable Conditions for Confidentiality Composition

Testing for the confidentiality of data by analysing data values considered possible by the attacker, as presented in the previous approach, is performed on the same level of detail as the direct assessment of confidentiality. In the approach presented in this section, we instead perform an assessment of the usage of the interface by the composed components regarding observations and knowledge that can be gained by an attacker.

Following this approach it is possible for two component designers to agree about the information regarding the components' interface that an attacker may get and thereby allows for a more distributed development of each of the components.

For a given state (i.e. sequence of actions) in the composed system, the conclusions regarding the interface behaviour that an attacker can draw from her observations and initial knowledge from each of the components must be equal. Consequently, during the design of the interface the component designers must agree on the interface behaviour that shall be considered possible by the attacker when observing the behaviour of the individual components.

The interaction of designers can be further decoupled by overestimating the set of possible states: Instead of considering all possible states / sequences of actions of the composed system, the designers may only define the set of possible sequences of actions at the interface (interface behaviour). This set can then be associated with the sequences considered possible by the attacker in each of the components, which leads to an agreement over the attacker's deductive capabilities.

The component designers can then independently assess if their component fulfils this requirement (equality of interface behaviour concluded from the individual components) by focussing on all sequences of actions that their component can take that will result in one of the agreed interface behaviour sequences.

Definition 5 *A composition following Definition 3 is called confidentiality preserving if the following assumption holds for all $P \in \mathbb{P}_0, \omega \in B_0$:*

$$a)\ h_1^{IF}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1) = h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2)$$

Alternatively, for all $P \in \mathbb{P}_0, \omega \in B^{IF}$, $\omega_1 \in (h_1^{IF})^{-1}(\omega), \omega_2 \in (h_2^{IF})^{-1}(\omega)$*:*

$$b)\ h_1^{IF}((\lambda_P^1)^{-1}(\lambda_P^1(\omega_1)) \cap W_P^1) = h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(\omega_2)) \cap W_P^2)$$

which implies condition a) by overestimation of possible component state combinations.

Theorem 2 *Given a confidentiality preserving composition according to Definition 5 and given that system* S_1 *has a confidentiality property w.r.t. some* μ_1 *and* K *then* S_0 *has the confidentiality property regarding* $\mu_0 = \mu_1 \circ h_1$ *and the same* K*.*

Proof. $\mu_0[\lambda_0^{-1}(\lambda_1(\omega)) \cap W_0] = \mu_1(h_1[\lambda_0^{-1}(\lambda_1(\omega)) \cap W_0])$
$= \mu_1(h_1[\lambda_0^{-1}(\lambda_1(\omega)) \cap h_1^{-1}(W_P^1) \cap h_2^{-1}(W_P^2)])$
Using Lemma 3 leads to equality with
$\mu_1[W_P^1 \cap h_1(\lambda_0^{-1}(\lambda_1(\omega)) \cap h_2^{-1}(W_P^2))]$
$= \mu_1 W_P^1 \cap [h_1(h_1^{-1}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega)))) \cap h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap h_2^{-1}(W_P^2))]$
Again applying Lemma 3 implies equality with
$\mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1(h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap h_2^{-1}(W_P^2))]$
$= \mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1(h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap W_P^2)]$
Applying Lemma 2 leads to equality with
$= \mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1^{IF^{-1}}(h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap W_P^2)]$
which by Assumption equals
$\mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1^{IF^{-1}}(h_1^{IF}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega)))) \cap W_P^1)]$
which is finally equal to
$\mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1]$ which concludes our proof.

5.3 Design of Generally Composable Component Interfaces

This final approach for composition targets the design of interfaces between components. The goal is to design the interface between two components in such a way that no additional considerations have to be made when composing confidentiality properties.

This is for example possible if an interface handles only the single transfer of confidential data. Obviously, if this data is handled in a confidential way by both components, there cannot be any side effects within the interface that may destroy the confidentiality property. This is expressed by testing that for any two combinations of sequences of actions within the interface, the extraction of confidential data from their combination will equal those candidates that result from the combinations of candidates derived independently from each of the sequences.

Most notably in this approach, it is not necessary to assess the capabilities (in terms of local view and initial knowledge) of a possible attacker. The design of the interface will make it impossible for any attacker to gain advantage by the composition of the components as long as they each provide confidentiality of the data. Formally, this is expressed as:

Definition 6 *A composition following Definition 3 has a generally composable interface with respect to some* μ_{IF} *if*

$$\forall A \subseteq h_1^{IF}(W_P^1), B \subseteq h_2^{IF}(W_P^2) : \mu_{IF}(A \cap B) = \mu_{IF}(A) \cap \mu_{IF}(B)$$

Trivially, if μ_{IF} is an isomorphism, the above property is implied.

Theorem 3 *Given a generally composable interface composition as defined in Definition 6, if S_1 and S_2 both are parameter confidential for agent R with respect to some μ_1 and μ_2 , then S_0 is parameter confidential for R with respect to $\mu_0 = \mu_1 \circ h_1 = \mu_2 \circ h_2 = \mu_{IF} \circ h'_1 \circ h_1 = \mu_{IF} \circ h'_2 \circ h_2$.*

Proof. $\mu_0[\lambda_0^{-1}(\lambda_1(\omega)) \cap W_0] = \mu_1(h_1[\lambda_0^{-1}(\lambda_1(\omega)) \cap W_0])$
$= \mu_1(h_1[\lambda_0^{-1}(\lambda_1(\omega)) \cap h_1^{-1}(W_P^1) \cap h_2^{-1}(W_P^2)])$.
Using Lemma 3 leads to equality to
$\mu_1[W_P^1 \cap h_1(\lambda_0^{-1}(\lambda_1(\omega)) \cap h_2^{-1}(W_P^2))]$
$= \mu_1 W_P^1 \cap [h_1(h_1^{-1}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega)))) \cap h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap h_2^{-1}(W_P^2))]$
By Lemma 3 this is equal to
$\mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1(h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega)))) \cap h_2^{-1}(W_P^2))]$
$= \mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1(h_2^{-1}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2))]$
By Lemma 2 this is equal to
$\mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1 \cap h_1^{IF^{-1}}(h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2))]$
As $\mu_1 = \mu_{IF} \circ h_1^{IF}$ and using Lemma 3 leads to equality with
$\mu_{IF}[h_1^{IF}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1) \cap h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2)]$.
By assumption, this equals
$\mu_{IF}[h_1^{IF}((\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1)] \cap \mu_{IF}[h_2^{IF}((\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2)]$
$= \mu_1[(\lambda_P^1)^{-1}(\lambda_P^1(h_1(\omega))) \cap W_P^1] \cap \mu_2[(\lambda_P^2)^{-1}(\lambda_P^2(h_2(\omega))) \cap W_P^2]$
which satisfies Definition 4.

6 Revisiting the Scenarios

In this section, we revisit the scenario composition introduced in Section 4.2 and demonstrate where and how this composition fails with regards to the formal considerations presented in Section 5. From the description in Section 2 it is already known that the example scenario does not preserve confidentiality during composition. In this section we demonstrate how our sufficient conditions, if not met, give hints regarding the possible reasons of confidentiality being violated in the composition, and how the components can be changed in order to preserve confidentiality.

For the following illustrations, we do not require the point in time at which a key is returned for assessing the confidentiality of the key; hence we define $\Sigma_t := \{returnKey(KGen)\}$. For the ease of reading we further simplify the system by restricting it to one single run; i.e. $\forall a \in \Sigma, \omega \in B : card(a, alph(\omega)) = 1$. This results in a considerable reduction of complexity but does not affect the applicability of our methods. Analogous results can be obtained for the full system behaviour.

6.1 General Conditions for Confidentiality Composition

Following the system definitions in Section 4.2 we investigate the preservation of confidentiality in the example composition. We demonstrate that Theorem 1 is not applicable and show how this fact can be used to identify the side effects that violate the confidentiality in the composed system. We use the following sequence of actions:

$$\omega_0 = boot(t_1)\ callGenKey(App, t_2)\ genKey(KGen, key_0, t_3)$$
$$returnKey(KGen, key_0, t_4) \text{ with } t_2 = t_1 + \delta_1 \text{ and } t_3 = t_2 + \delta_2$$

We start by assessing the left hand side of the equation of Definition 4, followed by the two sets for the right hand side.

Given ω_0, we can assess the sequences that Eve considers possible in S_0 (with $pre(\omega)$ denoting the set of prefixes of ω):

$$\Lambda^0_{Eve}(\omega_0, W^0_{Eve}) = \\ pre[\bigcup_{t_x \in \mathcal{T}} \{boot(t_1)\ (callGenKey(App, t_1 + \delta_1) \\ genKey(KGen, key_0, t_1 + \delta_1 + \delta_2)\ returnKey(KGen, key_0, t_x)\}] \\ \setminus \{\varepsilon\}$$

Since Eve knows δ_1 and δ_2, and since the key is completely determined by its time of generation, she only considers one value possible for the returned key

$$\mu_0[\Lambda^0_{Eve}(\omega_0, W^0_{Eve})] = \{(returnKey(KGen), key_0)\} \text{ with } key_0 = k(t_1 + \delta_1 + \delta_2)$$

Regarding the conception of Eve with respect to each of the component systems, we again assess all those sequences that Eve considers possible for the respective images of ω_0 in these systems:

$$\Lambda^1_{Eve}(h_1(\omega_0), W^1_{Eve}) = pre[\bigcup_{t_x \in \mathcal{T}, key_i \in \mathcal{K}} \{boot(t_1) \\ callGenKey(App, t_1 + \delta_1)\ returnKey(KGen, key_i, t_x)] \setminus \{\varepsilon\}$$

$$\Lambda^2_{Eve}(h_2(\omega_0), W^2_{Eve}) = pre[\bigcup_{t_x, t_y \in \mathcal{T}} \{callGenKey(App, t_x) \\ genKey(KGen, key_j, t_x + \delta_2)\ returnKey(KGen, key_j, t_y)\}] \\ \text{with } key_j = k(t_x + \delta_2)$$

This leads to the following sets of values that Eve considers as candidates for the confidential data (as t_x originates from all of $\mathcal{T}$, every $key_i \in \mathcal{K}$ is possible):

$$\mu^1[\Lambda^1_{Eve}(h_1(\omega_0), W^1_{Eve})] = \bigcup_{key_i \in \mathcal{K}} \{(returnKey(KGen), key_i)\}$$

$$\mu^2[\Lambda^2_{Eve}(h_2(\omega_0), W^2_{Eve})] = \bigcup_{key_j \in \mathcal{K}} \{(returnKey(KGen), key_j)\} \cup \{\varepsilon\}$$

Coming back to Definition 4 we can see that the values considered possible by Eve in the composition do not equal the combined (i.e. intersected) knowledge from each of the component systems:

$$\{(returnKey(KGen), key_0\} \neq \big(\bigcup_{key_i \in \mathcal{K}} \{(returnKey(KGen), key_i)\}\big) \\ \cap \big(\bigcup_{key_j \in \mathcal{K}} \{(returnKey(KGen), key_j)\} \cup \{\varepsilon\}\big)$$

implies $\mu_0[\Lambda^0_{Eve}(\omega_0, W^0_{Eve})] \neq \mu^1[\Lambda^1_{Eve}(h_1(\omega_0), W^1_{Eve})] \cap \mu^2[\Lambda^2_{Eve}(h_2(\omega_0), W^2_{Eve})]$

It can be seen however, that if t_1 or δ_1 were unknown to Eve, the confidentiality would be preserved. This relates to the use case as Desktop Application where an attacker does not know at which point in time a user initiates a key generation. It can further be seen that if key was not derived from these values but for example from a non-pseudo random number generator, Eve would also not be able to derive the key's value in the composition.

6.2 Independantly Testable Conditions for Confidentiality Composition

Similarly, Definition 5 can be used to illustrate that the condition of Theorem 2 sufficient for preserving confidentiality does not hold. Using the same ω_0 as in the previous section results in the same sets $\Lambda^1_{Eve}(h_1(\ \omega_0), W^1_{Eve})$ and $\Lambda^2_{Eve}(h_2(\omega_0), W^2_{Eve})$. We now investigate the projections of these sets into the interface system in order to compare the interface expectations of both components.

$$h_1^{IF}[\Lambda^1_{Eve}(h_1(\omega_0),W^1_{Eve})] = pre[\bigcup_{t_x\in\mathcal{T},key_i\in\mathcal{K}} \{ callGenKey(App, t_1+\delta_1)\ returnKey(KGen, key_i, t_x)] \setminus \{\varepsilon\}$$

$$h_2^{IF}[\Lambda^2_{Eve}(h_2(\omega_0),W^2_{Eve})] = pre[\bigcup_{t_y,t_z\in\mathcal{T}} \{ callGenKey(App, t_y)\ returnKey(KGen, key_i, t_z)\}]$$
$$\text{with } key_i = k(t_y+\delta_2)$$

As we can see, these sets are not equal. The dependence of key_i on the point in time of $callGenKey$ being performed is not expected by the App component, which hints to the confidentiality preservation error.

In order to avoid such a situation, the developers of the components could have agreed a priory to a common assumed interface behaviour when they agreed on the interface design. Following option b) of Definition 5 this could have been

$$B^{IF} = pre[\bigcup_{t_x<t_y-\delta\in\mathcal{T},key_i\in\mathcal{K}} callGenKey(App, t_x)\ returnKey(KGen, key_i, t_y)]$$

In this case the developer of the key generator would have needed to alter his/her implementation to reflect the functional independence of t_x and key_i, leading to a confidentiality preserving composition.

6.3 Design of Generally Composable Component Interfaces

Finally, we demonstrate that our example scenario does not satisfy the sufficient condition specified in Definition 6 and show how in particular scenarios the system specification can be corrected in order for the condition to hold and thus confidentiality to hold as well in the composition. We choose the following two sequences of actions from the respective sets:

- $h_1^{IF}(W_{Eve}^1) \ni A = \{callGenKey(App, t_1)\ returnKey(KGen, key_A, t_y)\}$ with $key_A \in \mathcal{K}$ (key_A can be chosen independently of t_1).
- $h_2^{IF}(W_{Eve}^2) \ni B = \{callGenKey(App, t_2)\ returnKey(KGen, key_B, t_y)\}$ with $key_B = k(t_2 + \delta_2)$ according to S_2.

Obviously, as for $t_1 \neq t_2$ A and B are distinct sets, $\mu_{IF}(A \cap B) = \emptyset$. However, for $key_A = k(t_2 + \delta_2) = key_B$, it follows $\mu(A) = \mu(B) = \{(returnKey(KGen), key_A)\} = \mu_{IF}(A) \cap \mu_{IF}(B)$.

In order to construct a system that fulfills the condition for a generally composable interface, S_{IF} must be designed in such a way that μ_{IF} is an isomorphism. This is the case e.g. if the interface only consists of a stream of generated keys that are handed over from the key generator to the application with $\Sigma_{IF} = \{provideKey(KGen, key_i)\}$. As there exists no functional relation from App to $KeyGen$ there cannot be side-effects that destroy the confidentiality property on the key generator's side during composition.

7 Related Work

The model based composition of systems is a field of growing research activity in the last decade. Tout et al. [14] have developed a methodology for the composition of web services with security. They use the Business Process Execution Language (BPEL) for the specification of web services composition and expand it in order to specify the security properties independently from the business logic based on policy languages using a UML Profile for specifying the required security properties. Their approach focusses on how to specify security requirements of web service compositions and does not address verification of security properties in such compositions. Sun et al. propose in [15] a service decomposition-based approach for service composition in which the utility of a composite service can be computed from the utilities of component services, and the constraints of components services can be derived from the constraints of the composite service. Their approach manages the selection of each component service, leading to more scalability and more flexibility for service composition in a dynamic environment. However, this approach focusses on maximizing the utility of the composition and does not address security properties. A method for composing a system from service components with anonymous dependencies is presented by Sora et al. in [16]. They specify component descriptions by means of semantic-unaware properties, an application-domain independent formalism for describing the client-specific configuration requests in terms of desired properties, and propose a composition algorithm. Using a different approach, Lei Zhang and Jun Wu [17] analyse the relationship between trustworthiness attributes and propose models of these attributes and their relationship. They use a Trustworthy Software Composition Architecture (TSCA) software as evaluation method.

Rossi presents in [18] a logic-based technique for verifying both security and correctness properties of multilevel service compositions. Service compositions are specified in terms of behavioural contracts which provide abstract descriptions of system behaviours by means of terms of a process algebra. Multi-party service compositions are modelled as the parallel composition of such contracts. Modal mu-calculus formulae are used to characterize non-interference and compliance (i.e. deadlock and livelock free) properties. The well-known concepts of non-interference or information flow control address

confidentiality with respect to actions. In the above approach, these concepts are used to specify that public synchronizations (i.e. actions concerned with the communication between services) are unchanged as confidential communications are varied. Hence it is not clear how this approach can be extended to cover cases in which satisfaction of confidentiality depends solely on whether specific parameters of an action are visible.

Universal Composability is another prominent branch of research addressing the composition of cryptographic protocols while preserving certain security properties (see for example [19,20,21]). A common paradigm in this area of research is that a protocol that "securely realizes" its task is equivalent to running an idealized computational process (also called "ideal functionality") where security is guaranteed. A main disadvantage of the Universal Composability approach seems to be that for every property that shall be proven, a new ideal process has to be constructed whose interactions with the parties result in providing this property.

Pino et al. present in [22] an approach for constructing secure service compositions, making use of composition patterns and security rules. They prove integrity and confidentiality of service compositions based on specific security properties provided by the individual components of such a composition. While the proofs are based on the same formal framework as the one presented in this paper, their approach uses an intermediate orchestration component. We in contrast focus on the direct composition of any type of components, deriving security proofs from specific conditions concerning the component interfaces.

8 Conclusions & Future Work

In this paper we presented the formalization of the composition of two systems that allows to formally reason about the preservation of confidentiality properties. The central idea is to view each of the systems as an abstraction of their composition, and to describe each aspect of the composition (e.g. its behaviour, agents' local views and initial knowledge) in terms of these abstractions. We then introduced conditions that allow to prove that a specific confidentiality property holds for the composition if it holds for the individual components. Using the composition of an application with a key generation module as scenario, we then demonstrated that the fact that these conditions do not hold reveals side effects with non-trivial implications regarding confidentiality. In particular, we presented a general sufficient condition for preservation of confidentiality that is of more theoretical interest, and derived two more specific conditions that are applicable in distributed system engineering and point to particular aspects of the two components that need to be taken into consideration by the developers. The first concerns additional agreements on interface level between component developers that can be independently tested for each component, the second provides sufficient conditions regarding the interface itself that rules out side effects during composition and thereby guarantees the preservation during composition of any two components that implement these interfaces.

Currently we are working on other types of conditions sufficient for proving confidentiality of a system. Finding relations of these conditions to the ones presented in this paper may broaden their scope of application. Future work includes the application of

the foundations layed out in this paper to general software engineering by projecting the semantic knowledge onto rules and guidelines for composition of software components.

References

1. Heise News: Neuer Personalausweis: AusweisApp mit Lücken. heise.de (2010)
2. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: Proceedings of the 21st USENIX Security Symposium (August 2012)
3. Anderson, R.: Security Engineering: A guide to building dependable distributed systems. Wiley, Chichester (2010)
4. SERENITY Consortium: Serenity (2006), `http://www.serenity-project.org`
5. TERESA Consortium: Trusted computing Engineering for Resource constrained Embedded Systems Applications (2009), `http://www.teresa-project.org/`
6. SecFutur Consortium: SecFutur (2010), `http://www.secfutur.eu/`
7. Gürgens, S., Ochsenschläger, P., Rudolph, C.: On a formal framework for security properties. In: International Computer Standards & Interface Journal (CSI), Special issue on formal methods, techniques and tools for secure and reliable applications **27**(5) (), June 2005, pp. 457–466 (2005)
8. Gürgens, S., Ochsenschläger, P., Rudolph, C.: Parameter confidentiality. In: Informatik 2003 - Teiltagung Sicherheit, Gesellschaft für Informatik (2003)
9. Gürgens, S., Ochsenschläger, P., Rudolph, C.: Abstractions preserving parameter confidentiality. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 418–437. Springer, Heidelberg (2005)
10. Kerrisk, M.: LCE: Don't play dice with random numbers. LWN.net (2012)
11. Corbet, J.: Random numbers for embedded devices. LWN.net (2012)
12. Fuchs, A., Gürgens, S., Rudolph, C.: A Formal Notion of Trust – Enabling Reasoning about Security Properties. In: Proceedings of Fourth IFIP WG 11.1 International Conference on Trust Management (2010)
13. Fuchs, A., Gürgens, S., Rudolph, C.: Formal Notions of Trust and Confidentiality - Enabling Reasoning about System Security. Journal of Information Processing 19, 274–291 (2011)
14. Tout, H.: e.a.: Towards a bpel model-driven approach for web services security. In: International Conference on Privacy, Security and Trust, PST'12 (2012)
15. Sun, S., Zhao, J.: A decomposition-based approach for service composition with global qos guarantees. In: Journal of Information Sciences (2012)
16. Sora, L.: Automatic composition of systems from cponents with anonymous dependencies specified by semantic-unaware properties. In: Technology of object-oriented languages, systems and architecture (2003)
17. Zhang, L., Wu, J.: Research on trustworthy software composition architecture. In: International Conference on Consumer Electronics, Communications and Networks (2012)
18. Rossi, S.: Model checking adaptive multilevel service compositions. In: International Workshop of Formal Aspects of Component Software (2010)
19. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology 13(1), 143–202 (2000)
20. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: focs, Published by the IEEE Computer Society, 136 (2001)
21. Canetti, R., Herzog, J.: Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In: Theory of Cryptography, pp. 380–403 (2006)
22. Pino, L., Spanoudakis, G.: Constructing secure service compositions with patterns. In: IEEE Eighth World Congress on Services (SERVICES), pp. 184–191. IEEE, Los Alamitos (2012)

Method Shells: Avoiding Conflicts on Destructive Class Extensions by Implicit Context Switches

Wakana Takeshita[1] and Shigeru Chiba[1]

The University of Tokyo, Japan
www.csg.ci.i.u-tokyo.ac.jp

Abstract. We propose *method shells*, which is a module system for avoiding conflicts on customization by language mechanisms such as aspects in AspectJ and open classes in Ruby. These mechanisms allow programmers to customize a library without rewriting original source code but by only describing differences in a separate file. We call these mechanisms *destructive class extensions*. A problem with destructive class extensions is conflicts on customization. Different customizations may differently modify the same class. To address this problem, we propose a new module system named *method shells*. With this system, programmers can avoid conflicts since the module system automatically switches a set of customizations that has to be applied together according to the contexts declared by programmers. We present the idea of this module system and then its formal semantics. We also present an extension of Java that supports method shells.

1 Introduction

Aspect-oriented programming (AOP) [1] is a programming paradigm where crosscutting concerns can be separated into different modules called *aspects*. Aspects can modify the behavior of the code contained in a different module so that they will implement their concerns. This mechanism, however, can be used for not only implementing a crosscutting concern but also customizing an existing class library or framework to fit an application program. A class library (or framework) can be extended by subclassing but subclassing does not enable all kinds of customization. Some kinds of customization need to directly modify a class contained in the library. To modify such a class, aspects are useful language constructs from the viewpoint of software maintenance.

Aspects are not only the mechanism for customizing a class library without directly changing the library source code. For this purpose, several other mechanisms have been proposed such as open classes in Ruby [2] and refines in AHEAD [3]. In this paper, we call this category of mechanisms *destructive class extensions* [4] since they directly modify the behavior of existing classes.

A problem with destructive class extensions is that extensions often conflict with each other. This problem has been actively studied in the context of AOP

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 49–64, 2013.

but these studies have focused on conflicts observed when the implementations of different crosscutting concerns are woven at the same join point [5, 6]. On the other hand, in the context of library customization, an important issue is to deal with a library customized by destructive class extensions (or aspects) as a *black box*. This is non-trivial if two libraries require the same sub-library but differently customize it by destructive class extensions. An application program using the two libraries together will cause conflicts on the customization of the sub-library and thus the programmer has to be aware of the customization by the libraries; the libraries cannot be considered as black boxes.

To avoid conflicts on the customization in this scenario, we propose a new module system named *method shells* and present an extension of Java that supports this module system.[1] As a mechanism for destructive class extension, we use a reviser [4]. A method shell is a module that can contain classes and revisers. It can include other method shells to extend and customize their classes. Furthermore, it can *link* to other method shells. The linked method shells are dealt with as black boxes. A method shell can invoke the code in the method shells linked to it but the invoked code is executed in a separate context so that it will not be affected by the customization effective in the method shell invoking the code. No unexpected conflicts happen between method shells linking to each other.

In the rest of this paper, we first show a motivating example and then present method shells. We also present the formal semantics of the method shells and a brief sketch of their implementation. Finally, we mention related work and conclude this paper.

2 Destructive Class Extensions

When programmers need to customize a class library, the customization would be convenient if they can modify it without directly modifying the original source code by only describing differences in a separate file. Such customization is modular and easy to maintain. Even if the customization includes a bug, they can easily obtain the prior code by deleting the file describing the customization.

Although subclassing is often used to describe such customization, it does not perfectly fit the aim. Suppose that the library contains a class C and she wants to modify a method in that class. Describing a subclass of C that overrides the method is not sufficient to make a customized library. All the classes in the library that create an instance of C must be modified to create an instance of that subclass. Subclassing is, therefore, not the perfect approach for customizing an existing library in a modular fashion.

For modular customization of existing libraries, several language mechanisms have been proposed in languages like Ruby [2], AspectJ [7], AHEAD [3], Multi-

[1] The first author submitted a summary of this work to ACM Student Research Competition (SRC) held in March at AOSD 2013. The submission will be reviewed and oral and poster presentations will be scheduled in March. Submitting a full paper to another conference is permitted by the SRC moderators.

```
1  // in the browser library
2  revise WebPage {
3    void popup(HTML text) {
4      warning("disabled");
5    }
6  }
```

Fig. 1. A reviser for the WebPage class

```
1  // in the HTML-renderer library
2  class WebPage {
3    void popup(HTML text) {
4      // show a popup window.
5    }
6    void onClick(Mouse m) {
7      URL url = m.getURL();
8      if (isPopup())
9         popup(url);
10     ...
11   }
12     ...
13 }
```

Fig. 2. WebPage class

Java [8], Jiazzi [9], and GluonJ [4]. In this paper, we call these mechanisms *destructive class extensions* since they are mechanisms for directly modifying existing implementation. They allow programmers to append new methods to an existing class and substitute a new implementation of an existing method. The new implementation is described in a separate source file and thus the original source files are not modified. Describing customization in a separate module is not sufficient for scalable modular customization. It also has to enable modular reasoning; the customization must change the original implementation only through a public interface or well-designed extension points. Some languages such as AspectJ provide a powerful mechanism like pointcuts and thus their ability for modular reasoning is controversial [10, 11]. Since they enable changes of any parts of module, preserving modularity in large scale software is not straightforward. On the other hand, in other languages like GluonJ, the customization changes the implementation by redefining public methods and hence they enable as modular reasoning as normal object-oriented programming.

However, even in the latter languages, enabling modular customization is not easy. The customization through public methods will not scale as the number of methods increase. Different customizations may conflict on the same method. For better scalability, a scoping mechanism must be introduced so that the customization will be effective only within a limited space. This is the aim of this paper.

Suppose that we have a library l_1 for rendering an HTML text and we write another library l_2 for constructing a web browser that will be embedded in an application software. For code reuse, the library l_2 should be implemented on top of the former library l_1. Since an embedded web browser should not show a popup window, which will surprise application users, we have to customize the library l_1 so that a popup window will be blocked.

A mechanism of destructive class extension allows this customization without modifying the source code of the library l_1. Figure 1 shows the code for that modification. In this paper, we use the syntax of GluonJ [4]. The code modifies the original implementation of the WebPage class shown in Figure 2, which is contained in the library l_1. It directly replaces the original implementation of the popup method with a new one in Figure 1. If the library l_2 contains the code in Figure 1, which is called *a reviser*, the behavior of the library l_1 is revised and no popup window will not be displayed when l_2 uses l_1. The original source code of popup does not have to be modified.

We next write the third library l_3, which provides an audited viewer of local files written in HTML. The viewer shows a popup dialog for an alert when a confidential file is opened. To show a popup dialog, we use the popup method in the WebPage class supplied by the library l_1 for rendering an HTML text. Furthermore, we modify several methods by revisers, for example, the getBorder method in the WebPage method so that the rendered HTML text will be shown in a specially decorated window.

An application using either the library l_2 or l_3 will work well. However, since both l_2 and l_3 commonly use the library l_1 but differently modify the classes in l_1, an application using both of them will not work. The revisers in l_2 and l_3 will conflict. For example, the reviser in l_2 disables to show a popup window by the popup method whereas the library l_3 needs that the popup method shows a popup window as its original implementation does.

This conflict is a well-known problem with destructive class extensions [6, 12, 13] but it is more crucial than usual when destructive class extensions are used for customizing a library. A library is usually dealt with as a black-box; library users should be unaware of which other libraries are internally used by that library and how those other libraries are customized. It should be hidden that both the libraries l_2 and l_3 internally use l_1 and they differently customize l_1. Thus, a conflict on l_1 between l_2 and l_3 will be a surprise to their user programmers. This is a similar problem happening when an application requires two libraries and the two require other libraries that are different versions of the same library. A library providing basic functionality is often included by other third-party libraries but, if it is popular and being actively developed, these third-party libraries often require different versions of it. Such third-party libraries are difficult or impossible to use together. Our scenario of conflicts on customization can be regarded as a conflict between two versions of the library l_1, each of that is implemented by revisers describing differences from the base version.

3 Method Shells

To address the problem presented in the previous section, we propose a new module system named *method shells*. With this module system, a set of revisers that must be applied together is implicitly switched to fit execution contexts during runtime. As a prototype of method shells, we have developed an extension of Java. In this extended Java, a new language construct called *a method shell* is available. It is a construct similar to package and it specifies a module that classes and revisers in the source file belong to. Figure 3 presents a renderer method-shell. The first line is a methodshell declaration, which declares that the following WebPage class is contained in the method shell named renderer. This method shell represents the HTML-renderer library l_1 shown in the previous section.

```
1  methodshell renderer;
2
3  class WebPage {
4    void popup(HTML text) {
5      // show a popup window.
6    }
7    void onClick(Mouse m) {
8      URL url = m.getURL();
9      if (isPopup())
10        popup(url);
11     ...
12   }
13     ...
14 }
```

Fig. 3. The renderer method shell

```
1  methodshell browser;
2  include renderer;
3
4  revise WebPage {
5    void popup(HTML text) {
6      warning("disabled");
7    }
8  }
9
10 public static void main(String[] args){
11   WebPage w = new WebPage();
12   w.popup("Available?"); // not shown
13 }
```

Fig. 4. The browser method-shell

Include Declarations and Revisers

In the previous section, the HTML-renderer library l_1 was used by the web-browser library l_2. With the method shells, this relation is represented by an include declaration. Figure 4 presents a reviser in the browser library l_2 reimplemented with method shells. The second line is an include declaration. It represents that the browser method-shell includes the renderer method-shell. All the classes and revisers contained in the renderer method-shell are also contained in the browser method-shell. This relation by include declarations is transitive.

The reviser in Figure 4 belongs to the browser method-shell and it modifies the implementation of the WebPage class. The WebPage class is called *a target class* and it must be in the same method shell that the reviser belongs to. Since include declarations constructs transitive relations, the WebPage class could be in a method shell included by the method shell that the reviser belongs to. A method shell is a scope of the visibility of classes and revisers. Classes and revisers can refer to only the class names contained in the same method shell.

The implementations of the methods declared in a reviser substitute the original ones in the target class or they are appended to the target class if they are new methods. The reviser in Figure 4 substitutes the implementation of the popup method in the WebPage class. Although the source code of the original implementation is not modified, the modification by the reviser is directly applied to the target class. This is a difference from subclassing. The modification by a subclass of WebPage will not affect the instances of WebPage but the modification by a reviser for WebPage affects the instance of the target class WebPage.

A method shell can contain a special function main. It is a main method where the whole program starts. In Figure 4, the main method makes an instance of WebPage and calls the popup method on it. Since this method shell contains a reviser for WebPage, the implementation of popup in the reviser is selected and executed. A popup window is not displayed.

When the program starts from the main method in a method shell S, it runs with the modifications by the revisers contained in S unless the program contains link declarations mentioned later. For clarity, if a program is running with the modifications by a method shell S, we call S *the current context*. Note

```
1  methodshell viewer; include renderer;
2
3  class Viewer {
4    void check(File f) {
5      if (isConfidential(f))
6        new WebPage().popup("<b>Confidential</b>");
7    }
8      ...
9  }
10 revise WebPage {
11   Border getBorder() {
12     // return a decorated window border
13   }
14 }
```

Fig. 5. The viewer method shell

that all revisers contained in the same method shell are applied together to classes in that method shell. If multiple revisers share the same target, they are applied in the precedence order given by the programmer. For backward compatibility, our extended Java language allows classes in a source file without a methodshell declaration. Such classes belong to a special method shell that are implicitly included by any method shell. We call this special method shell *the global context*.

Link Declarations

In the previous section, the library l_1 was also used by the audited-viewer library l_3. Figure 5 shows one of the source files of the library l_3 after being reimplemented with method shells. It contains a class and a reviser as well as the include declaration for including the renderer method-shell. This reimplementation of l_3 will work correctly if it is used independently. However, if we define a new method shell that naively includes both l_2 and l_3, the customizations of l_1 by the revisers in the two libraries l_2 and l_3 will conflict as we already saw in the previous section.

To address this conflict, the method shells provide a link declaration so that programmers can deal with a method shell as a black box. Here, being a black box means that the mere *users* of a method shell are not aware of its internals: which sub method-shells are included and how they are customized. For the developers who customize a method shell, it is still a gray-box; its internals are partly visible and customizable through a public interface. In large-scale applications, we believe that this sense of being a black box and/or a gray box would be necessary. It would be error-prone to construct such a large application by combining only gray-box libraries while manually avoiding conflicts.

The method shell linked by a link declaration is not included but the classes and the revisers in that method shell become visible. See Figure 6. This source file belongs to the application method-shell and it includes the browser method-shell by the include declaration. Since the third line is a link declaration, the application method-shell does not include the viewer method shell. However, the main method in the application method-shell can refer to the Viewer class, which belongs to the viewer method-shell.

```
1  methodshell application;
2  include browser;
3  link viewer;
4
5  public static void main(String[] args) {
6    new WebPage().popup();
7    new Viewer().check(new File("secret.txt");
8  }
```

Fig. 6. The application method-shell and the link declaration

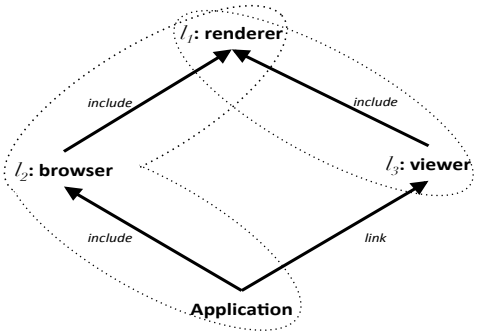

Fig. 7. The method shell commonly shared

A unique feature of link declarations is the current context during the execution of the code in the linked method-shell. While a method implementation contained in the linked method-shell is executed, the current context is set to that linked method-shell. For example, when the main method in Figure 6 calls the check method, since the implementation of check is in the viewer method-shell, the current context is changed from the application method-shell to the viewer method-shell. During the execution of the check method, therefore, only the revisers in the viewer method-shell are effective. The revisers in the application method-shell are not effective. The reviser for the WebPage class in the browser method-shell, which is included by application, is not effective and thus the check method can execute the original implementation of the popup method. The current context is switched back to the application method-shell when the execution of the check method finishes.

A link declaration allows a program to execute a method in a space separated from other modules'. In our example scenario, the original implementation of popup method is in the l_1 library, or the renderer method-shell. It is included in the application method-shell through the library l_2 and l_3, or the browser method-shell and the viewer method-shell, respectively. The problem is that popup is modified differently by the paths through l_2 and l_3 and our solution is to make two versions of popup for each path. One is for the path from application to l_2 and l_1 while the other is for the path from l_3 to l_1. The former version is modified by using the application method-shell as the current context while the latter is by using the viewer method-shell. The two versions are switched when a method implementation in the linked method shell is invoked. This problem and the solution are similar to the diamond inheritance problem [14] and its solution.

4 Semantics and Implementation

This section presents the formal semantics of method shells. It also presents the sketch of the implementation technique of method shells. This technique was used to develop a prototype compiler of our extended Java, which supports method shells. This compiler was developed by using the JastAddJ framework [15].

4.1 Syntax

We first present a simple calculus for the formalization. It is an extension of Featherweight Java (FJ) [16] and GluonFJ [4]. The syntax is given as follows:

SL	::= `methodshell` S; $\overline{\text{IL}}$ $\overline{\text{LL}}$ (CL $\|\|$ RV) $*$ MF	method-shell declaration
IL	::= `include` S;	include declaration
LL	::= `link` S;	link declaration
CL	::= `class` C `extends` C{$\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$}	class declaration
RV	::= `revise` C{$\overline{\text{M}}$}	reviser declaration
M	::= C m($\overline{\text{C}}$ $\overline{\text{x}}$){`return` e;}	method declaration
MF	::= `void main`(){`return` e;}	main function declaration
e	::= x $\mid$ e.f $\mid$ e.m($\overline{\text{e}}$) $\mid$ `new` C($\overline{\text{e}}$) $\mid$ e `in` S;S_C	expressions
v	::= `new` C($\overline{\text{v}}$)	values

The metavariables S and T range over method shell names; B, C, and D range over class names; f range over field names; m ranges over method names; x ranges over parameter names; K ranges over constructor declarations; v and w range over values. The syntax of K and body is not shown here but it conforms to FJ. In this syntax, we use an overline to represent a sequence. For example $\overline{\text{x}}$ equals to "$\text{x}_1, \text{x}_2, ..., \text{x}_\text{n}$" and $\overline{\text{C}}$ $\overline{\text{f}}$ equals to "C_1 f_1, C_2 $\text{f}_2, ..., \text{C}_\text{n}$ f_n".

SL is a method shell. It consists of its name, include declarations, link declarations, class declarations CLs and reviser declarations RVs, and a main-function declaration MF. A class declaration CL consists of its name, its super class, fields, a constructor, and methods. A reviser declaration RV consists of its name and methods. A reviser cannot have a field. An expression e may take a new form e `in` S;S_C, which is used to mark which method shell e originates from and the current contexts in the operational semantics.

We denote the class and reviser table by CRT. It is a mapping from a pair of a method shell S and a class name C to a class declaration CL or a reviser declaration RV. A program is a pair of CRT and a method-shell name. The program execution starts from the main function included in the method shell with that name.

4.2 Lookup Semantics

The reduction relation is of the form S;$\text{S}_\text{C} \vdash \text{e} \rightarrow \text{e}'$, reading "expression e reduces to expression e′ in one step in a method shell S and the current context S_C. If a program starts from the main function in a method shell S, then the program execution is to reduce its expression e in the method shell S and the

current context S. Most reduction rules are given in a straightforward manner from FJ's and GluonFJ's. Interesting rules are the followings:

$$\frac{\mathtt{T;T_C} \vdash \mathtt{e_0} \longrightarrow \mathtt{e_0}'}{\mathtt{S;S_C} \vdash \mathtt{e_0\ in\ T,T_C} \longrightarrow \mathtt{e_0}'\ \mathtt{in\ T;T_C}} \quad \text{(R-In)}$$

$$\frac{\mathtt{mbody(m,C,S,S_C)} = \bar{\mathtt{x}}.\mathtt{e_0\ in\ T;T_C}}{\mathtt{S;S_C} \vdash \mathtt{new\ C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{w}}) \longrightarrow ([\bar{\mathtt{w}}/\bar{\mathtt{x}},\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e_0})\ \mathtt{in\ T;T_C}} \quad \text{(R-Invk)}$$

The first rule is straightforward. $\mathsf{e_0}$ is reduced in $\mathsf{T};\mathsf{T_C}$ although $\mathsf{S};\mathsf{S_C}$ are given. The second rule is for method invocation. Unlike FJ's, a function to look up a method body, named *mbody*, takes four parameters. It looks up a method body by referring to the method name m, the class of the target object C, the method shell S that the expression originates from, and the current context $\mathsf{S_C}$. Both S and $\mathsf{S_C}$ are ones at the caller-side. If a method body e with parameters $\bar{\mathsf{x}}$ is found in a method shell T and the new current context is set to $\mathsf{T_C}$, then the method body is executed with the arguments $\bar{\mathsf{w}}$ in T and $\mathsf{T_C}$.

The definition of *mbody* is presented in Figure 8. $mbody(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ returns the body of m called on the C class from the method shell S with the current context $\mathsf{S_C}$, written $\bar{\mathsf{x}}.\mathsf{e}\ \mathsf{in}\ \mathsf{T};\mathsf{T_C}$, where $\bar{\mathsf{x}}$ are parameters, e is the method body, T stands for the method shell where the body is found, and $\mathsf{T_C}$ stands for the current context used to execute the body.

mbody uses a few auxiliary functions. $includings(\mathsf{S})$ returns a set of method shells directly included by S. $linked\text{-}shells(\mathsf{S})$ returns a set of method shells linked by S. $mbodyshell(\mathsf{m},\mathsf{C},\mathsf{S})$ is a function to search the method shell S. It returns the body of method m in class C found in S. It first searches the method bodies directly contained in S and then recursively searches ones in method shells included by S. Finally, $mbodyglobal(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ searches the global context, which contains classes in source files without methodshell declarations. The global context is a special method shell implicitly included by any method shell. In Figure 8, Global stands for the global context. Note that, if the body of m in the class C is not found in the global context, *mbodyglobal* searches the method bodies declared in a super class of C by recursively calling *mbody*.

$mbody(\mathsf{m},\mathsf{C},\mathsf{S},\mathsf{S_C})$ searches in the following order. First, it searches the method shells *linked* by S. If a method body m is found, the new current context is set to the linked method shell where the body is found. Otherwise, *mbody* searches the current context $\mathsf{S_C}$. Note that it does not search the method shell S, which the method-call expression originates from. S is used only for obtaining the linked method shells searched at the first step. If a method body is not found in either the linked method shells or the current context, then *mbody* searches the global context. Finally, if a method body directly declared in the class C is not found in any method shells, *mbody* looks up a method body declared in a super class of C. The current context does not change except the first step.

4.3 Implementation

Our prototype compiler transforms a program using method shells into plain Java program, which is then compiled into Java bytecode. During the transformation, the methods in revisers are copied into the declaration of the target

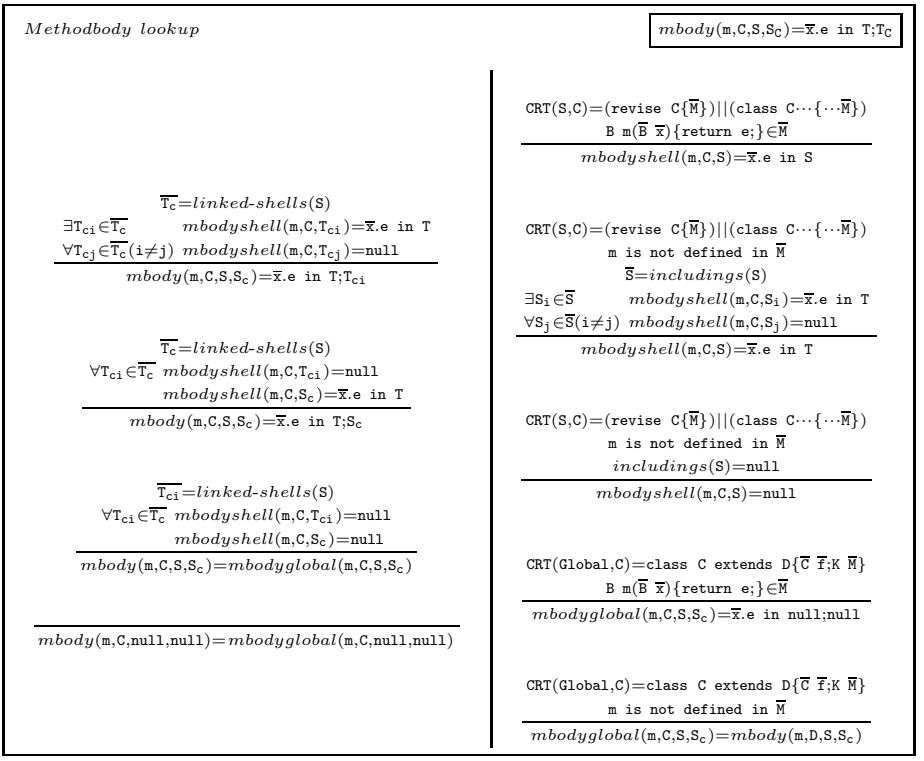

Fig. 8. A function to look up a method body

class. If the method already exists in the target class, the method copied from the reviser substitutes the original one. Our prototype compiler/language has not supported a mechanism to invoke the overridden implementation of a method by a call on super.

However, if multiple revisers in different method shells modify the same method, the naive approach above will not work. Our compiler hence copies a method implementation after renaming the method into the name mangled from the original method name and the method-shell name. A method m declared in a reviser for a class C is copied into the declaration of C after the method name is changed into the name m_S mangled from the method name m and the method shell S, which the reviser is contained in.

When a method m is called on an instance of a class C, the appropriate implementation is selected among the available versions m_{S1}, m_{S2}, m_{S3}, ... for each method shell S_i. According to the semantics we showed above, the selection depends on the method shell that the method-call expression originates from and the current context. The former one is statically determined but the latter one is not. Thus, a naive implementation will have to check the latter one at runtime for method dispatch. This will cause a runtime penalty.

To minimize runtime penalties due to method shells, our prototype compiler duplicates a method implementation for different current contexts and transforms the method body to customize. Therefore, a method $\mathsf{m_{Si}}$ is duplicated into $\mathsf{m_{Si \cdot T1}}, \mathsf{m_{Si \cdot T2}}, \mathsf{m_{Si \cdot T3}}, ...$, where $\mathsf{T_i}$ is a current context. The body of the method $\mathsf{m_{Si \cdot Tj}}$ is transformed for $\mathsf{S_i}$ and $\mathsf{T_j}$. The transformation for a method shell S and a current context $\mathsf{S_C}$ under type environment Γ is written $\mathsf{S}; \mathsf{S_C}; \Gamma \vdash \mathsf{e} \Longrightarrow \mathsf{e'}$. For most expressions, the transformation is trivial. Only method calls must be changed:

$$\frac{\mathtt{S;S_C;}\Gamma\vdash\mathtt{e_0}\Longrightarrow\mathtt{e_0}' \quad \mathtt{S;S_C;}\Gamma\vdash\bar{\mathtt{e}}\Longrightarrow\bar{\mathtt{e}}' \quad \mathtt{S;S_C;}\Gamma\vdash\mathtt{e_0:C} \quad mname(\mathtt{m,C,S,S_C})=\mathtt{m}'}{\mathtt{S;S_C;}\Gamma\vdash\mathtt{e_0.m}(\bar{\mathtt{e}})\Longrightarrow\mathtt{e_0}'\mathtt{.m}'(\bar{\mathtt{e}}')}$$

Here, $\mathsf{S}; \mathsf{S_C}; \Gamma \vdash \mathsf{e_0} : \mathsf{C}$ is read "expression $\mathsf{e_0}$ is given type C under type environment Γ." In other words, the static type of $\mathsf{e_0}$ is C. *mname* is a function to look up a method like *mbody*. It is defined as following:

$$\frac{mbody(\mathtt{m,C,S,S_C})=\bar{\mathtt{x}}\mathtt{.e\ in\ null;null}}{mname(\mathtt{m,C,S,S_C})=\mathtt{m}}$$

$$\frac{mbody(\mathtt{m,C,S,S_C})=\bar{\mathtt{x}}\mathtt{.e\ in\ T;T_C}}{mname(\mathtt{m,C,S,S_C})=\mathtt{m_{T \cdot T_C}}}$$

If the method implementation is selected from the global context, the method name is not changed during transformation.

Finally, the body of the main function is transformed so that the appropriate version of methods will be called. If the program starts with the main function in a method shell S, then the body is transformed for S and S.

5 Related Work

In the context of AOP, a number of researchers have been studying conflicts of advices, or *aspect interference*. Aksit et al. proposed a mechanism for detecting aspect interference by using graph transformation [6]. Several linguistic constructs have been proposed to resolve the interference. Douence et al. proposed a new composition operator of aspects [12,13]. It allows programmers to describe a safely-composable aspect. Airia provided a new kind of around advise called *resolvers* for resolving the interference [17]. A uniqueness of our work is that we have designed a language construct specialized for a specific use-case scenario where aspects are used for building a custom library to be used as a black box.

AOP and destructive class extensions can be regarded as a special case of virtual classes [18,19], where all base-level classes are implicitly contained as virtual classes in a single enclosing class and all aspects (or corresponding constructs like revisers) are in a subclass of that enclosing class, if the differences in how to specify the target base-level classes are ignored (the targets in AOP are specified by pointcuts while ones in virtual classes are by the super-class names). Our method shells are an approach to introduce a scope mechanism into destructive class extensions. In the analogy above, this approach allows programmers to use more than one enclosing classes in programming with destructive class extensions (*i.e.* AOP) as they can do in programming with virtual classes. Therefore, the resulting language mechanism is similar to ones for virtual classes but it still

has some unique features since it is originated from destructive class extensions. For example, in method shells, programmers are less aware of the existence of enclosing classes. When a program refers to a class contained in a different enclosing class, it does not have to explicitly specify the name of that different enclosing class. That class is implicitly selected by the link declaration so that the obliviousness property [20] is somewhat preserved. Another example is that, in method shells, the method selected for a method call on the same target object changes depending on the caller's contexts. Furthermore, method shells allow multiple shells to be included at the same time like mixing.

In Newspeak [21], all class names are virtual and a subclass can override them. This overriding mechanism corresponds to the include declaration in method shells. On the other hand, Newspeak does not provide a mechanism corresponding to the link declaration, with which the method-lookup context is changed after a method in the linked method-shell is selected. Although Newspeak is dynamically typed, method shells are statically typed and we present a technique for reducing their method-lookup overhead.

The idea of method lookup depending on runtime contexts is found not only in our method shells but also other languages such as JPred [22]. JPred supports predicate dispatch, with which a method is selected by referring to calling contexts such as method arguments and caller objects.

Us [23] allows programmers subjectivity-based programming. In Us, every method call explicitly takes a method-lookup context called a perspective. In method shells, every call does not take such a context, which is declared by a include or link declaration at the beginning of the source file.

Context-oriented programming (COP) [24] is a paradigm where a class definition can be changed depending on the contexts during runtime. A class declaration is divided into multiple parts, which are called layers, and different layers may contain different implementations of the same method. Layers are dynamically switched by with and without clauses. Within the with clause, the specified layer is effective while in the without clause it is ineffective. A layer provides the same ability for destructive class extension as our revisers but a layer must be contained in the declaration of the target class although a reviser is described separately from the target class. Despite this difference, method shells and COP share the idea of changing class definition to fit the current context.

However, the with and without clauses are not adequate for addressing the problem mentioned in this paper. Programmers in COP languages cannot deal with a layer as a black box. They have to understand the dependency among all layers and classes used in their programs. Figure 9 shows a program that is equivalent to the program in Figure 6 but is written in ContextJ, a COP extension of Java [25]. This program starts from the main method in the App class. Since it uses the renderer, the browser, and the viewer, it first activates all the three by with (line 10 to 12). However, while the check method in Viewer is executed, the browser layer must be deactivated since it needs a popup window. In Figure 9, the browser layer is deactivated within the body of the check method by without (line 23) but this requires the programmer of Viewer to be aware of the

```
1  class WebPage{
2    layer(renderer){
3      void popup(HTML text){
4        // show a popup window
5      }
6      void onClick(Mouse m){
7        URL url = m.getURL();
8        if(isPopup()) popup(url);
9          ...
10       }
11     }
12   }
13   layer(browser){
14     void popup(HTML text){
15       warning("disabled");
16     }
17   }
18 }
19 class Viewer{
20   layer(viewer){
21     void check(File f){
22       if(isCirfidential(f)){
23         without(browser){
24           new WebPage.popup
25             ("<b>Confidencial</b>");
26         }
27       }
28     }
29   }
30 }
31 }
```

```
1  class App{
2    void run(){
3      new WebPage().popup();
4            // a popup is disabled
5      new Viewer().check
6        (new File("secret.txt");
7            // a popup is needed
8    }
9    public void main(String[] args){
10     with(renderer){
11       with(browser){
12         with(viewer){
13           new App.run();
14         }
15       }
16     }
17   }
18 }
```

Fig. 9. A program in ContextJ

browser layer, which might be independently developed from the Viewer. Another approach is to deactivate the browser layer within the body of the run method in App, for example, just before calling the check method at line 5. However, this requires the programmer of App to be aware that the browser layer must be deactivated while a Viewer is running. The programmer cannot deal with Viewer as a black box. A recent version of ContextJ supports Reflection API [26] and hence the problem above is fairly overcome. In this language, a program can obtain all the layers currently activated and then deactivate them. However, unlike method shells, the programmer still has to be aware of unnecessary layers and explicitly deactivate them.

Classboxes [27, 28] are a module system that also provides a scoping mechanism for destructive class extensions. In Classbox/J, related classes are modularized into a module called a classbox. It can include other classboxes and partly modify them by refine, which corresponds to a reviser in our language. However, Classboxes do not provide a mechanism corresponding to our link declarations and thus they cannot handle the scenario shown in this paper. If the library l_1 is included through multiple paths, all the refines on the paths are applied together.

In Java, every class loader has its own name space. Hence, distinct implementations of the same class can coexist in one program if they are loaded into different class loaders. This is useful to partly address the problem discussed in this paper but moving an object beyond the boundary between class loaders is significantly restricted. In method shells, such restriction known as *the ver-*

sion barrier is not applied. To enable such movement between class loaders, the Java virtual machine has to be modified and support a mechanism like sister namespaces [29].

We have already proposed method shelters, which is a mechanism similar to method shells [30]. Although the two mechanisms share the same approach, method shelters are for a dynamically typed language Ruby. The destructive class extension in Ruby is performed by open classes, which is different from revisers we used in this paper. Furthermore, the design of method shelters is more complicated than that of method shells. A method shelter, which is a unit of module, is divided into an exposed chamber and a hidden chamber. Programmers have to carefully choose which chamber a reviser should be placed to control its visibility. On the other hand, a method shell is simpler but equivalently expressive; it is not divided into smaller containers but a single container. Programmers can intuitively control the visibility of revisers by choosing either an include declaration or a link declaration.

6 Conclusion

We proposed method shells, which are a module system for avoiding conflicts on destructive class extensions. The destructive class extensions are mechanisms for modifying class definitions from a separate module, which include aspects in AspectJ, open classes in Ruby, and revisers in GluonJ. A method shell is a module consisting of classes and revisers. It can include other method shells and the revisers in the included method shells are applied together as well as the revisers in the method shells including them. A unique feature is that a method shell can link to other method shells. The code included in the linked method shells can be invoked but it is executed in a context where only the revisers in the linked method shell are effective. Thus, a linked method shell is dealt with as a black box.

Our contribution is to propose a mechanism for avoiding conflicts on destructive class extensions when we use the extensions for customizing a class library or a framework. The resulting library or framework after customization can be dealt with a black box. The main idea is link declarations. The language automatically switches effective revisers when the thread of control crosses over to a linked method shell. Showing the formal semantics and implementation strategy of method shells is also contribution.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
2. : Ruby programming language., `http://www.ruby-lang.org/`
3. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6) (2004)

4. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '10, pp. 539–554. ACM Press, New York (2010)
5. Dinkelaker, T., Mezini, M., Bockisch, C.: The art of the meta-aspect protocol. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. AOSD '09, pp. 51–62. ACM Press, New York (2009)
6. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. AOSD '09, pp. 39–50. ACM Press, New York (2009)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
8. Millstein, T., Reay, M., Chambers, C.: Relaxed multijava: balancing extensibility and modular typechecking. In: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications. OOPSLA '03, pp. 224–240. ACM Press, New York (2003)
9. Mcdirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '11, pp. 211–222. ACM Press, New York (2001)
10. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 49–58. Springer, Heidelberg (2006)
11. Steimann, F.: The paradoxical success of aspect-oriented programming. ACM SIGPLAN Notices 41(10), 481–497 (2006)
12. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 173–188. Springer, Heidelberg (2002)
13. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Proceedings of the 3rd international conference on Aspect-oriented software development. AOSD '04, pp. 141–150. ACM Press, New York (2004)
14. Malayeri, D., Aldrich, J.: Cz: multiple inheritance without diamonds. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. OOPSLA '09, pp. 21–40. ACM Press, New York (2009)
15. Ekman, T., Hedin, G.: The jastadd extensible java compiler. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. OOPSLA '07, pp. 1–18. ACM Press, New York (2007)
16. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
17. Takeyama, F., Chiba, S.: An advice for advice composition in AspectJ. In: Baudry, B., Wohlstadter, E. (eds.) SC 2010. LNCS, vol. 6144, pp. 122–137. Springer, Heidelberg (2010)
18. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Abstraction mechanisms in the beta programming language. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '83,, pp. 285–298. ACM Press, New York (1983)

19. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: Conference proceedings on Object-oriented programming systems, languages and applications. OOPSLA '89, pp. 397–406. ACM Press, New York (1989)
20. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.) Aspect-Oriented Software Development, pp. 21–35. Addison-Wesley, Reading (2005)
21. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 405–428. Springer, Heidelberg (2010)
22. Millstein, T.: Practical predicate dispatch. In: Proc. of ACM OOPSLA, pp. 345–364. ACM, New York (2004)
23. Smith, R.B., Ungar, D., Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS 2, 161–178 (1996)
24. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
25. Appeltauer, M., Hirschfeld, R., Masuhara, H.: Improving the development of context-dependent java applications with contextj. In: International Workshop on Context-Oriented Programming. COP '09, vol. 5, pp. 5:1–5:5. ACM, New York (2009)
26. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: Contextj: Context-oriented programming with java. Information and Media Technologies 6(2), 399–419 (2011)
27. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions. In: Computer Languages, Systems and Structures (2005)
28. Bergel, A.: Classbox/j: Controlling the scope of change in java. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pp. 177–189. ACM Press, New York (2005)
29. Sato, Y., Chiba, S.: Loosely-separated "sister" namespaces in Java. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 49–70. Springer, Heidelberg (2005)
30. Akai, S., Chiba, S.: Method shelters: avoiding conflicts among class extensions caused by local rebinding. In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development. AOSD '12, pp. 131–142. ACM Press, New York (2012)

Separating Obligations of Subjects and Handlers for More Flexible Event Type Verification

José Sánchez and Gary T. Leavens

University of Central Florida, Dept. of EECS, Orlando, FL 32816, USA
{sanchez,leavens}@eecs.ucf.edu

Abstract. Implicit invocation languages, like aspect-oriented languages, automate the Observer pattern, which decouples subjects (base code) from handlers (advice), and then compound them together in the final system. For such languages, event types have been proposed as a way of further decoupling subjects from handlers. In Ptolemy, subjects explicitly announce events at certain program points, and pass the announced piece of code to the handlers for its eventual execution. This implies a mutual dependency between subjects and handlers that should be considered in verification; i.e., verification of subject code should consider the handlers and vice versa.
However, in Ptolemy the event type defines only one obligation that both the handlers and the announced piece of code must satisfy. This limits the flexibility and completeness of verification in Ptolemy. That is, some correct programs cannot be verified due to specification mismatches between the announced code and the handlers' code. For example, when the announced code does not satisfy the specification of the entire event and handlers must make up the difference, or when the announced code has no effect, imposing a monotonic behavior on the handlers.
In this paper we propose an extension to the specification features of Ptolemy that explicitly separates the specification of the handlers from the specification of the announced code. This makes verification in our new language PtolemyRely more flexible and more complete, while preserving modularity.

Keywords: Event type, specification, verification, Ptolemy language

1 Introduction

Event types [12], and other similar approaches like XPIs [16], AAI [10], Open Modules [1,11], IIIA with Join Point Types [15] and Joint Point Interfaces (JPI) [8,5,4], have been proposed as a way to further decouple subjects from handlers in implicit invocation and aspect-oriented languages. The verification systems for such languages should, as usual, strive to be as complete as possible while staying sound. In this work we propose some enhancements to the Ptolemy language and its specification and verification system for making it more complete while keeping it sound.

1.1 Completeness as a Measure of Usefulness

We work in the framework of a partial-correctness Hoare logic [7]. A judgement of the form $\Gamma \vdash \{P\}S\{Q\}$ means that the Hoare-triple $\{P\}S\{Q\}$ is provable using the type

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 65–80, 2013.

environment Γ. The judgement $\Gamma \vdash \{P\}S\{Q\}$ is *valid* iff for every state σ that agrees with the type environment Γ, if P is true in σ (written $\sigma \models P$) and if the execution of S terminates in a state σ', then $\sigma' \models Q$. Such a logic is *sound* if whenever a judgment $\Gamma \vdash \{P\}S\{Q\}$ is provable, then it is valid. Conversely, such a logic is *complete* if whenever such a judgment is valid, then it is provable in the logic.

To compare two logics, one can ask if both are sound, and if so one can compare how complete they are. Logic A is *strictly more complete than* logic B if there is some valid judgment that is provable in A but not in B, but every judgment that is provable in B is provable in A. Given that both logics are sound, then a more complete logic is potentially more useful for users, as they will be able to prove more programs correct.

1.2 A Brief on Ptolemy Language

Ptolemy's [12] event type concept decouples subjects (*base code*), which explicitly announce *events*, from the *handlers* that process these events. The event type establishes the contract every handler must satisfy. In this way the base (or announcing) code can be modularly reasoned about using the contract, instead of using each handler's code. The contract not only defines the precondition and postcondition every handler method should satisfy, but also the abstract algorithm they must refine, called a *translucid* contract [3]. In the body of a translucid contract, *specification expressions* can abstract away details of particular implementation expressions, by only specifying their effects. *Invoke expressions* in the contract's body show where a handler triggers the execution of the next handler in the *execution chain* (until eventually reaching the originally announced code that stands at the end). In the base code, *announce expressions* are used to explicitly announce occurrences of events, starting the *execution chain* and passing the *announced code* to it. All this is schematized in Figure 1.

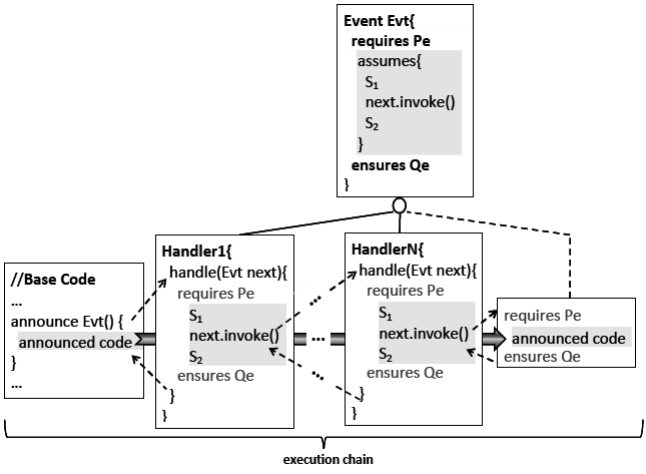

Fig. 1. Event, handlers and announced code

The *invoke expressions* in the contract make visible the control effects of the handlers. Active handlers are registered as such using *register expressions* and handlers are bound to the corresponding event by *when expressions*.

In Ptolemy, every handler method must be verified to satisfy the contract's pre- and postconditions and also to structurally refine (*see section 2*) the translucid contract's body, providing conforming implementations for every specification expression.[1] The announced code is also verified to satisfy the same contract's pre- and postconditions.

1.3 The Billing Example

The *billing* system example in Figure 2 illustrates the basic concepts of Ptolemy and motivates our proposed extension. In this system, each bill includes the amount (a) to be paid and the extra charges (c) like taxes. When the base code totals a bill, adding the charges to the principal amount (line 7), the corresponding event is announced (lines 6-8). This gives registered handlers (maybe $PaymentHandler$ or $ShippingHandler$) the chance to do some adjustments, like adding some extra charges. In this case we register just one handler at random (line 5) to emphasize the fact that the reasoning is based on the event definition, instead of the particular implementation of any specific handler. The *TotalingEvent* definition specifies the behavior and abstract algorithm of every admissible handler. The **`requires`** (line 14) and **`ensures`** (line 21)[2] clauses specifies the behavior: every handler requires (line 14) that the existing charges are not negative and ensures (line 21) that the resulting amount of the bill is greater than or equal to the sum of the original amount plus the original charges. The excess, if any, is due to the extra charges added by the handlers. The *translucid contract* (lines 16-19, inside **`assumes`**$\{\ldots\}$) forces the handlers to make the charges greater than or equal to their current value, but allows charges to be added by each handler in any consistent way. The specification expression (lines 17-18) must be refined by each conforming handler, with code that satisfies the stated pre-post conditions. Also any **`invoke`** expression must be made explicit in the translucid contract (as on line 19). This allows modular verification of control effects, using the specification of the announced event.

This example is verified by Ptolemy's proof system. Both handlers refine the event's translucid contract. The specification expression in this contract (lines 17-18) is refined by $PaymentHandler$ by increasing the charges ($c' = c + 1$, line 27), and by $ShippingHandler$ by leaving the charges the same ($c' = c + 0$, line 38). Considering the above and the effect of the **`invoke`** expression, it can be seen that both handlers satisfy the event specification ($a' \geq a + c$, line 21), and so both are proven valid. The announced code ($a' = a + c$, line 7) also satisfies the event specification ($a' \geq a + c$, line 21), as required by Ptolemy's proof system, so the complete **`announce`** expression (lines 6-8) is proven valid. With the handlers and the **`announce`** expressions proven valid, the entire program is proven valid in Ptolemy.

---

[1] Ptolemy is an expression language.

[2] When summarizing assertions, we adopt the Z [14] convention of denoting the new value of a variable with a prime (like a'), and use unprimed variables to stand for their pre-state values.

```
1  public class Base {
2    public void run(){
3      Bill bill=new Bill(100,8);
4      Bill old=new Bill(bill.a(),bill.c());
5      registerHandler();// Randomly register one handler
6      announce TotalingEvent(bill) {  // event Qe : a' ≥ a + c
7         bill.setA(bill.a()+bill.c());// code Qs : a' = a + c
8      }
9      //assert bill.a()>old.a()+old.c(); //a' > a + c ??
10 } }
11
12 public void event TotalingEvent { // handlers: a' ≥ a + c
13   Bill bill;
14   requires (bill.c()>=0) // Pe: c ≥ 0
15   assumes{
16     // specification expr.: requires c ≥ 0 ensures c' ≥ c
17     requires (next.bill().c()>=0)
18       ensures (next.bill().c()>=old(next.bill().c()));
19     next.invoke(); // control flow: proceed with next handler
20   }
21   ensures (bill.a()>=old(bill.a())+old(bill.c())) //Qe: a' ≥ a + c
22 }
23 public class PaymentHandler { // Payment Processing Fee Handler
24   public void handleTotaling(TotalingEvent next)throws Throwable{
25     refining requires (next.bill().c()>=0)
26              ensures (next.bill().c()>=old(next.bill().c())){
27       next.bill().setC(next.bill().c()+1);  // c' = c + 1
28     }
29     next.invoke();
30   }
31   when TotalingEvent do handleTotaling;
32   public PaymentHandler(){ register(this); }
33 }
34 public class ShippingHandler { // Shipping Fee Handler
35   public void handleTotaling(TotalingEvent next)throws Throwable{
36     refining requires (next.bill().c()>=0)
37              ensures (next.bill().c()>=old(next.bill().c())){
38       next.bill().setC(next.bill().c()+0); //c' = c + 0 NO FEE NOW
39     }
40     next.invoke();
41   }
42   when TotalingEvent do handleTotaling;
43   public ShippingHandler(){ register(this); }
44 }
```

Fig. 2. Billing example in Ptolemy

1.4 Completeness Issues: Enforcing the Billing "Increasing" Property

Now we consider a variation on the *billing* system. A new "business rule" requires us to enforce the "increasing" property: that all the handlers for *TotalingEvent* must strictly increase the total amount, by adding to the charges. Currently $PaymentHandler$ satisfies this condition (line 27) but $ShippingHandler$ does not (line 38). If this property were met, **the assertion on line 9 could be proven true**, since no matter which handler were registered (line 5) the charges would have been incremented.

We have to guarantee that any handler H bound to the event *TotalingEvent*, satisfies the required property, while keeping the program valid.[3] For doing that we can adjust the event specification and the handlers.

Definition 1. *An implementation of the* ***billing*** *program* ***satisfies the "increasing" property*** *if for each binding clause of the form* **`when`** $TotalingEvent$ **`do`** m *appearing in a class* C*: if* $H = bodyOf(C, m)$ *then* $\Gamma' \models \{c \geq 0\}H\{a' > a + c\}$.

The current *TotalingEvent* specification does not guarantee the above property, as its postcondition $(a' \geq a + c)$ does not imply $(a' > a + c)$. The way for the $billing$ system to satisfy this property is by having an event postcondition Q_e such that $Q_e \Rightarrow (a' > a + c)$. However in Ptolemy this Q_e must be such that $(a' = a + c) \Rightarrow Q_e$, to meet the requirement of Ptolemy's proof system that the announced code (line 7) satisfies the event specification. The fact that these two implications result in a contradiction shows that the above property cannot be proved in Ptolemy. This shows the incompleteness of Ptolemy's proof system, that is incapable of *modularly* proving the assertion in line 9.

In section 3 we propose an extension to Ptolemy that makes verification more flexible and complete, and in particular able to enforce the "increasing" property and verify the aforementioned assertion. First, we explain Ptolemy verification in more detail.

2 Verification in Ptolemy

In Ptolemy, event types state the obligations that handlers should satisfy. In the general case that was presented in Figure 1, the event *Evt*'s declaration specifies the precondition (P_e) and postcondition (Q_e) that handlers should conform to, and also the translucid contract (assumes clause) that they should refine.

Verification in Ptolemy is straightforward [3]. Every handler body H for an event and every piece of announced code S for that event must satisfy the same pre-post obligations [3, Figure 11], declared in the event's **`requires`** and **`ensures`** clauses. Besides that, the handlers must also refine the event's translucid contract. This is expressed in the requirement that a program is conformal, meaning that each handler conforms to the corresponding event declaration's specification.

Definition 2. *A Ptolemy program Prog* is conformal *if and only if for each declaration of an event type,* Evt*, in Prog, and for each binding clause of the form* **`when`** Evt **`do`** m *appearing in a class* C *of Prog: if* $(P_e, A, Q_e) = ptolemySpec(Evt)$ *and* $H = bodyOf(C, m)$*, then there is some type environment* Γ' *such that* $\Gamma'(\texttt{next}) = closure\ Evt$*,* $\Gamma' \vdash A \sqsubseteq H$ *and* $\Gamma' \models \{P_e\}H\{Q_e\}$.

[3] The auxiliary function $bodyOf(C, m)$ returns the body of method m in class C.

In the above, the formula P_e is the event's precondition, Q_e is its postcondition, and A is the body of the **assumes** clause (the "translucid contract" [3]), which in our notation is written $(P_e, A, Q_e) = ptolemySpec(Evt)$. Similarly, $bodyOf(C, m)$ returns the code that is the body of method m in class C.[4] The structural refinement relation $\sqsubseteq$ is explained below. Furthermore, we say that a Hoare-triple $\{P\}S\{Q\}$ is *valid*, written $\Gamma \models \{P\}S\{Q\}$, if in every state (typable by Γ) such that P holds, whenever S terminates normally, then Q holds in the resulting state.

In Ptolemy, the verification of handlers is done modularly and separately from the announcements. The body of each handler must structurally refine the translucid contract from the event specification. A handler body, H, *structurally refines* a translucid contract A, written $A \sqsubseteq H$, if one can match each expression in H to an expression in A [13]. The matching of most expressions are exact (only the same expression matches) with the exception of specification expressions of the form **requires** P **ensures** Q, which can occur in A and must each be matched by expressions in H of the form **refining requires** P **ensures** Q { S }, where S is the code implementing the specification expression. In Ptolemy structural refinement is checked by the type checking phase of the compiler [3].

To summarize, according to the work on translucid contracts for Ptolemy [3], the way that one proves that a program is conformal is by proving, for each handler body H for an event Evt such that $(P_e, A, Q_e) = ptolemySpec(Evt)$: $\Gamma' \vdash A \sqsubseteq H$ and $\Gamma' \vdash \{P_e\}H\{Q_e\}$. In order to guarantee soundness, the body of each **refining** expression must satisfy the given specification, as in the (REFINING) rule of Figure 3.

For every **announce** expression in a valid program, the announced code S should satisfy the event specification (P_e, Q_e). Then, if the base code guarantees P_e before the

$$\text{(SPECIFICATION-EXPR)}\quad \frac{}{\Gamma \vdash \{P\}\mathbf{requires}\ P\ \mathbf{ensures}\ Q\{Q\}}$$

$$\text{(REFINING)}\quad \frac{\Gamma \vdash \{P\}S\{Q\}}{\Gamma \vdash \{P\}(\mathbf{refining\ requires}\ P\ \mathbf{ensures}\ Q\ \{\ S\ \})\{Q\}}$$

$$\text{(ANNOUNCE)}\quad \frac{(P_e, A, Q_e) = ptolemySpec(Evt), \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \quad \Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}{\Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}\ \mathbf{announce}\ Evt(\boldsymbol{y})\ S\ \{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}$$

$$\text{(INVOKE)}\quad \frac{closure\ Evt = \Gamma(\mathbf{next}), \qquad (P_e, A, Q_e) = ptolemySpec(Evt)}{\Gamma \vdash \{P_e\}\ \mathbf{next.invoke}()\ \{Q_e\}}$$

Fig. 3. Hoare Logic axioms and inference rules for the interesting constructs of Ptolemy

[4] These auxiliary functions query the program, which is treated as a fixed context.

announce expression it can assume Q_e holds afterwards. This constitutes Ptolemy's (ANNOUNCE) rule in Figure 3. In that rule $P_e[\boldsymbol{y}/\boldsymbol{x}]$ means P_e with the actual parameter variables y_i[5] simultaneously substituted for the free occurrences of the x_i, which are the event's formal parameters. Note that the body of the announcement, S, cannot use the event's formal parameters, but only has access to the original type environment, Γ. In the (ANNOUNCE) rule, there is no distinction made regarding the presence or absence of any registered handlers, because the same reasoning applies in either case.

An **invoke** expression in a handler is reasoned about in the same way. That is, the code executing the invoke expression must establish P_e and can assume Q_e afterwards. This is (INVOKE) rule in Figure 3. In this rule, the event's name is obtained from the type of **next**, and this gives access to the specification (P_e, A, Q_e) of that event.

A Hoare logic is *sound* if whenever $\Gamma \vdash \{P\}S\{Q\}$ is provable then every terminating execution of S starting from a state in which P holds ends in a state in which Q holds. Soundness for Ptolemy depends on the program being conformal.

Theorem 1. *Suppose that the Hoare logic for Ptolemy, without using the rules in Figure 3, is sound. Then for conformal programs, the whole logic, including the rules in Figure 3, is sound.*

We omit the proof (which goes by induction on the structure of the proof in the entire Hoare logic). However, the key argument is the same as that for greybox specifications, that structural refinement implies refinement [13].

Ptolemy's design makes both handlers and the announced code have the same pre-post specifications (P_e, Q_e).[6] This design is convenient in some cases, but it limits Ptolemy's flexibility and completeness. For example, it is not possible to use Ptolemy's event type pre and postconditions to specify and verify the "increasing" property of our *billing* system (section 1.4), because the announced code achieves the postcondition $a' = a + c$ and not the event's postcondition $a' > a + c$. However, this property could be considered correct with respect to a more flexible specification that gives different postconditions to the announced code and handlers, which is what we do below. This example shows that verification in Ptolemy is incomplete.

We have other similar examples that show incompleteness of Ptolemy's verification rules. The common theme, like in the *billing* example, is that the effect of the announced code does not match the effect of the handlers.

Another situation that shows Ptolemy's incompleteness occurs when the announced code has no effect (e.g., **skip**). As Ptolemy imposes the event pre-post obligations on the announced code, it requires that the triple $\{P_e\}\mathbf{skip}\{Q_e\}$ holds, or, by Hoare logic, that $P_e \Rightarrow Q_e$. Since these same obligations are imposed on the handlers, thus they are limited to monotonic behaviors; i.e. ones that preserve the precondition P_e. This is a symptom of incompleteness, because in a program where there must be registered handlers, one would not be able to verify an event announcement in which the handlers achieve a postcondition Q_e that is not implied by the event's precondition (P_e).

In the next section we detail our proposed modification to solve these incompleteness issues and analyse its impact regarding modular reasoning.

[5] We use variables in these rules to avoid problems with side effects in expressions, although Ptolemy allows general expressions to be passed as actual arguments to announcements.

[6] We use the convention of denoting by (P, Q) the pre- and postconditions of some code.

3 Explicit Separate Specification

A solution to the incompleteness problems can be found by recognizing that there is a mutual dependency between base code, handlers and announced code, in the execution chain. The base code depends on the behavior of the activated handlers that are triggered by an **`announce`** expression. The handlers depend on the other activated handlers, and on the behavior of the announced code at the end of the chain (Figure 4).

The first change from Ptolemy in our *PtolemyRely* language consists in separating the specification for the handlers (P_e,Q_e) from the specification for the announced code (P_s,Q_s). As before, every handler H is reasoned about using the event requires-ensures specification (P_e,Q_e). But the announced code S is reasoned about using its own specification (P_s,Q_s). (Both cases are depicted in Figure 5). This new approach allows different specifications for the handlers and for the announced code, as in our *billing* example. This also allows announced code that has no effect to be verified without limiting, in any way, the handlers' specification.

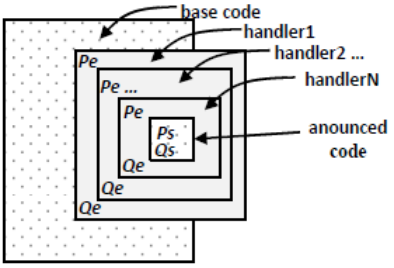

Fig. 4. Mutual dependencies between base code, handlers and announced code

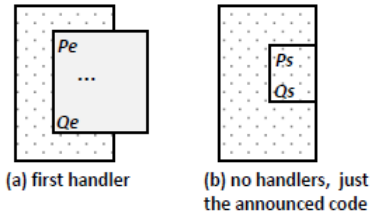

Fig. 5. Reasoning about the base code

In PtolemyRely, the second change is that the verification of both **`announce`** and **`invoke`** expressions is slightly modified. For **`announce`** expressions there are two situations, as shown in Figure 5. If there are registered handlers then the base code interacts with the first of them, which guarantees the event postcondition (Q_e). If there are no handlers then the announced code is executed, ensuring its postcondition (Q_s). This two cases are formalized by the rules (RANNOUNCEHAS) and (RANNOUNCENONE) in Figure 8.

`invoke` expressions are only valid inside the body of a handler, and thus should be analyzed in a context where there are registered handlers. Their effect, instead, depend on the nondeterministic position of the containing handler in the execution chain. If there are other handlers left in the execution chain, the event specification (P_e, Q_e) is used, as all handlers satisfy it. If only the announced code is left, its specification (P_s, Q_s) should be used. However, for modular verification, the problem is that the event declaration, and consequently the handlers, do not know the announced code and thus do not know (P_s, Q_s). To avoid whole-program reasoning, we make a third change, in this case to Ptolemy's event type declarations. Now users also specify, in the

event declaration, the pre-post obligations (P_r, Q_r) for any announced code. Putting this specification in the event type declaration in a new **relies** clause (see Figure 6) allows the handlers to be verified based on that specification, instead of the actual announced code's specification. It also allows one to avoid doing the verification that each handler satisfies the event pre-post specification one handler at time. Instead, that can be done in two separate steps: first, once and for all, verifying that the event's translucid contract satisfies the event's pre-post specification, and then verifying that each handler refines this translucid contract, which in turn guarantees every handler satisfies the event's specification.

To summarize, with our changes in PtolemyRely, the event type declares specifications for the handlers, (P_e, Q_e), and for the announced code, (P_r, Q_r). In the rest of this section, we give the formal details of our approach.

3.1 Syntax

For PtolemyRely, we change the syntax of Ptolemy event declarations by introducing a **relies** clause that establishes the specification for the announced code (P_r, Q_r). This is shown in the event syntax schema, Figure 6.

```
t event Evt {
  t1 f1;...; tn fn;
  relies requires Pr ensures Qr
  requires Pe
  assumes { ... next.invoke(); ...}
  ensures Qe
}
```

Fig. 6. Event syntax schema

sp ::= ... | **handlers** (*c*)
contract ::= ... |
relies requires *sp* **ensures** *sp*
requires *sp*
assumes { *se* }
ensures *sp*

Fig. 7. Formal syntax changes

We make two changes to the formal syntax of Ptolemy [3]. The first adds a predicate **handlers** that returns the number of handlers currently registered for its event argument. The second changes contract definitions, as shown in Figure 7. The nonterminal *c* stands for event names, *sp* stands for specification predicates, and *se* stands for specification expressions (the contract's body in this case).

3.2 Semantics

In PtolemyRely, as stated in the definition of conformance, we check for structural refinement of each handler to the translucid contract, and also check each handler to satisfy the event *requires-ensures* specification.

Definition 3. *A PtolemyRely program Prog* is conformal *if and only if for each declaration of an event type, Evt, in Prog, and for each binding clause of the form* **when** Evt **do** m *appearing in a class C of Prog: if* $(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt)$ *and* $H = bodyOf(C, m)$*, then there is some type environment* Γ' *such that* $\Gamma'(\textbf{next}) = closure\ Evt$*,* $\Gamma' \vdash A \sqsubseteq H$*, and* $\Gamma' \models \{P_e\}H\{Q_e\}$.

The function $eventSpec(Evt)$ returns the specification information from the event type's declaration. The returned 5-tuple consists of the relies clause contract (P_r, Q_r), and the translucid contract: pre and post-conditions (P_e, Q_e) and assumes body A.

The **`announce`** and **`invoke`** expressions are verified using the rules in Figure 8. For **`announce`** expressions there are two rules, depending on whether one can prove that there are registered handlers for the event. (RANNOUNCEHAS) applies when there are registered handlers. In this case the **`announce`** expression is reasoned about using the event's specification (P_e, Q_e). For this rule to be valid, the announced code, S, must satisfy the specification (P_r, Q_r) given in the event's type. (RANNOUNCENONE) applies when there are no registered handlers. In this case only the announced code is executed, and thus the relied on specification (P_r, Q_r) is used.

(RANNOUNCEHAS)

$$\frac{\begin{array}{c}(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt), \\ \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \qquad \Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}] \wedge \mathbf{handlers}(Evt) > 0\} S \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}\end{array}}{\Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}\ (\mathbf{announce}\ Evt(\boldsymbol{y})\ S)\ \{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}$$

(RANNOUNCENONE)

$$\frac{\begin{array}{c}(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt), \\ \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \qquad \Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}] \wedge \mathbf{handlers}(Evt) = 0\} S \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}\end{array}}{\Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}]\}\ (\mathbf{announce}\ Evt(\boldsymbol{y})\ S)\ \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}}$$

(RINVOKE)

$$\frac{closure\ Evt = \Gamma(\mathbf{next}), \qquad (P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt)}{\Gamma \vdash \{P_e \wedge P_r\}\mathbf{next.invoke}\,()\,\{Q_e \vee Q_r\}}$$

Fig. 8. Hoare Logic inference rules for those constructs of PtolemyRely that differ from Ptolemy

The soundness theorem for PtolemyRely states that if a program is conformal, then all provable Hoare triples are valid.

Theorem 2 (Soundness). *Suppose that the Hoare logic for Ptolemy, without using the rules for* **`invoke`** *and* **`announce`***, is sound. Then for conformal PtolemyRely programs, the whole logic, including the rules for those constructs in Figure 8, is sound.*

Proof: Let Γ, P, S and Q be given such that $\Gamma \vdash \{P\}S\{Q\}$ is provable using PtolemyRely's Hoare logic, including the rules in Figure 8. We prove that $\Gamma \models \{P\}S\{Q\}$ (i.e., that this Hoare triple is valid) by induction on the structure of the proof of that triple. In the base case, there are no uses of the rules in Figure 8, so validity follows by the hypothesis. For the inductive case, suppose that the proof has as its last step one of the rules in Figure 8. We assume inductively that all subsidiary proofs are valid. There are three cases. If the last step uses the (RANNOUNCENONE) rule, then the hypothesis that the announced code satisfies the specification (P_r, Q_r) makes the conclusion valid. If the last step uses the (RANNOUNCEHAS) rule, then the hypothesis that the program

is conformal means that, by definition 3, $\Gamma' \models \{P_e\}H\{Q_e\}$ where (P_e, Q_e) is the specification of the handler's from the event type. This again makes the conclusion valid. If the last step uses the (RINVOKE) rule, then there are two sub-cases, and the proof is similar to that given for the previous cases, using the definition of "conformal." ∎

We note that proving that a program is conformal can be done in a simple way, by proving $\Gamma' \vdash \{P_e\}A\{Q_e\}$, where (P_e, Q_e) is the event's pre/post specification, and A is the translucid contract for the event, and then checking that each handler's body, H structurally refines the translucid contract ($\Gamma' \vdash A \sqsubseteq H$). After that, it follows that $\Gamma' \models \{P_e\}H\{Q_e\}$ using techniques from the work of Shaner *et al.* [13].

3.3 Billing Example Revisited (PtolemyRely)

In Figure 9 we show how our *billing* example could be written in PtolemyRely. Here we show how it can be verified using PtolemyRely's rules, how the "increasing" property can be specified and verified and how the assertion in line 9 is now proved.

Contrary to Ptolemy, PtolemyRely allows us to have different specifications for the handlers (P_e, Q_e) and for the announced code (P_s, Q_s). As mentioned before, the specification for the handlers, (P_e, Q_e), goes in the **`requires-ensures`** clauses of the event declaration, meanwhile the minimum specification for any announced code, (P_r, Q_r), goes in the new **`relies`** clause. The specification of the announced code S (line 7) is (P_s, Q_s), that corresponds to $(c \geq 0, a' = a + c)$. We take the expected behavior for the announced code (P_r, Q_r) (lines 13-14) to be the same as the actual behavior for the announced-code (P_s, Q_s). The specification for the handlers (P_e, Q_e) is declared in line 15 and line 22 as $(c \geq 0, a' > a + c)$.

In PtolemyRely we can prove our "increasing" property: that all handlers should strictly increase the total amount of the bill. If a handler H is verified, it means that it satisfies the (P_e, Q_e) specification. In this case: $\Gamma \vdash \{c \geq 0\}H\{a' > a + c\}$, which is exactly what the "increasing" property demands.

Since there are registered handlers (line 5) the (RANNOUNCEHAS) rule applies. It requires $\{P_r\}S\{Q_r\}$, which holds in the **`announce`** expression in lines 6-8. The postcondition in the consequent of this rule, Q_e, corresponds in this case to $a' > a + c$, this immediately proves the assertion in line 9. To reason about **`invoke`** expressions one should use the (RINVOKE) rule, that considers (P_e, Q_e) and (P_r, Q_r). In this case it corresponds to the following:

$$\Gamma \vdash \{(c \geq 0) \wedge (c \geq 0)\}\texttt{next.invoke}\,()\,\{(a' > a + c) \vee (a' = a + c)\}$$

and this is equivalent to $\Gamma \vdash \{c \geq 0\}\texttt{next.invoke}\,()\,\{a' \geq a + c\}$

In this *revisited* version we adjusted *ShippingHandler* to meet the "increasing" property (line 38). Both handlers refine the translucid contract, providing code (line 28 and 38) that correctly refines the specification expression in the contract (lines 18-19). Also both, *PaymentHandler* and *ShippingHandler*, satisfy the handlers specification $(c \geq 0, a' > a + c)$. This can be shown as follows. Both increment the charges, $c' > c$, (line 28 and line 38) and then invoke the next handler. Considering this increment, and the indicated postcondition of the **`invoke`** expression, we have $(c' > c) \wedge (a' \geq a + c')$, and from that we get $(a' > a + c)$, that shows that both handlers satisfy the specification.

```
1  public class Base {
2    public void run(){
3      Bill bill=new Bill(100,8);
4      Bill old=new Bill(bill.a(),bill.c());
5      registerHandler();// Randomly register one handler
6      announce TotalingEvent(bill) {  // event Q_e : a' ≥ a + c
7         bill.setA(bill.a()+bill.c());// code Q_s : a' = a + c
8      }
9      //assert bill.a()>old.a()+old.c(); //a' > a + c ??
10 } }
11
12 public void event TotalingEvent { // handlers: a' ≥ a + c
13   Bill bill;
14   requires (bill.c()>=0) // P_e: c ≥ 0
15   assumes{
16     // specification expr.: requires c ≥ 0 ensures c' ≥ c
17     requires (next.bill().c()>=0)
18       ensures (next.bill().c()>=old(next.bill().c()));
19     next.invoke(); // control flow: proceed with next handler
20   }
21   ensures (bill.a()>=old(bill.a())+old(bill.c())) //Q_e: a' ≥ a + c
22 }
23 public class PaymentHandler { // Payment Processing Fee Handler
24   public void handleTotaling(TotalingEvent next)throws Throwable{
25     refining requires (next.bill().c()>=0)
26              ensures (next.bill().c()>=old(next.bill().c())){
27       next.bill().setC(next.bill().c()+1);  // c' = c + 1
28     }
29     next.invoke();
30   }
31   when TotalingEvent do handleTotaling;
32   public PaymentHandler(){ register(this); }
33 }
34 public class ShippingHandler { // Shipping Fee Handler
35   public void handleTotaling(TotalingEvent next)throws Throwable{
36     refining requires (next.bill().c()>=0)
37              ensures (next.bill().c()>=old(next.bill().c())){
38       next.bill().setC(next.bill().c()+0); //c' = c + 0 NO FEE NOW
39     }
40     next.invoke();
41   }
42   when TotalingEvent do handleTotaling;
43   public ShippingHandler(){ register(this); }
44 }
```

Fig. 9. Billing example revisited (PtolemyRely)

We have showed that the whole program is verified (announce expression and handlers), that the "increasing" property can also be verified and that the assertion in line 9 can be proved in PtolemyRely.

3.4 Extension of Ptolemy

Our new approach extends Ptolemy's, as stated in the following lemma.

Lemma 1. *Let Prog be a program in Ptolemy and S be an expression of Prog. Let Γ be a type environment that types S. Suppose $\Gamma \vdash \{P\}S\{Q\}$ is provable in Ptolemy. Then there is a PtolemyRely program Prog′ in which $\Gamma \vdash \{P\}S\{Q\}$ is provable by the rules for PtolemyRely.*

Proof: The new program *Prog′* in PtolemyRely is constructed by taking each event declaration E declared in *Prog*, and producing a new event declaration E' which is just like E, except that a **`relies`** clause is inserted of the form

`relies requires` P_e **`ensures`** Q_e

where $(P_e, A, Q_e) = ptolemySpec(E)$. Then the rest of the proof proceeds by induction on the structure of S.

If S is not an **`invoke`** or **`announce`** expression, then the proof rules for PtolemyRely are the same as for Ptolemy, so there are only two interesting cases.

When S is an **`invoke`** expression of the form **`next.invoke()`** then, by hypothesis, we have in Ptolemy's proof system $\Gamma \vdash \{P\}$**`next.invoke()`**$\{Q\}$. Thus by the Ptolemy (INVOKE) rule, we must have $\Gamma($**`next`**$) = closure\ Evt$, for some event name Evt, where $(P, A, Q) = ptolemySpec(Evt)$. By construction of $Prog'$, we have $(P, Q, P, A, Q) = eventSpec(Evt)$, so P plays the role of both P_e and P_r in PtolemyRely's (RINVOKE) rule, and Q plays the role of both Q_e and Q_r in that rule. So we have $\Gamma \vdash \{P \wedge P\}$**`next.invoke()`**$\{Q \vee Q\}$. To do this we use the rule of consequence in Hoare logic, since $(P \wedge P) \equiv P$ and $(Q \vee Q) \equiv Q$, to get the desired conclusion in the proof system for PtolemyRely.

When S is an **`announce`** expression of the form **`announce`** *Evt*$(\boldsymbol{y})$ $\{S_0\}$, then using Ptolemy's (ANNOUNCE) rule we have: $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$, and so we also have $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S_0\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$, where Γ is the type environment for expression S, $(P_{Evt}, A, Q_{Evt}) = ptolemySpec(Evt)$ and $\boldsymbol{x} : \boldsymbol{T} = formals(Evt)$. Using PtolemyRely's (RANNOUNCEHAS) or (RANNOUNCENONE) rules, we must prove that: $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$. Since by construction of *Prog′* we have that $(P_{Evt}, Q_{Evt}, P_{Evt}, A, Q_{Evt}) = eventSpec(Evt)$, then P_{Evt} plays the role of P_e and P_r, and Q_{Evt} plays the role of Q_e and Q_r, and so both rules allows us to immediately prove the desired conclusion. One can apply whichever rule is appropriate, or a derived rule with precondition $P_{Evt}[\boldsymbol{y}/\boldsymbol{x}] \wedge P_r[\boldsymbol{y}/\boldsymbol{x}]$ and postcondition $Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}] \vee Q_r[\boldsymbol{y}/\boldsymbol{x}]$, and then use the rule of consequence. ∎

4 Related Work

The original work on Ptolemy [12] addressed the problem of modular reasoning of implicit invocation systems, like AO systems. Many other solutions have also been proposed: XPIs [16], AAI [10], Open Modules [1,11], Join Point Types (JPT) [15] and Joint Point Interfaces (JPI) [8,5,4]. In this work we call attention to the mutual dependency that exists between the base code (*subject*) and the advising code (*handlers*). We

enhanced Ptolemy's event type specifications by clearly separating the obligations imposed on the handlers from the obligations of the announced code, in such a way that both can be reasoned about modularly. Here we review how, if at all, this problem is addressed in the other approaches and if our strategy can be applied to them.

Previous work [2] has shown how the translucid contract concept of Ptolemy can be adapted to other approaches like XPIs, AAI and Open Modules; adding specification and verification capability to them. All these approaches would benefit from our enhancement to the translucid contract concept, in case they adopted it, as they would become more complete and more flexible.

Steimann *et al.* [15] proposed an approach for dealing with Implicit Invocation and Implicit Announcement (IIIA) based on Join Point Types and polymorphic pointcuts. Ptolemy's approach [3], which we extended in this work, is similar to the work of Steimann et al. One important difference, though, is that Ptolemy does not support implicit announcement. On the other hand, Steimann et al. do not treat the issue of specification and verification, suggesting that one can "resort to an informal description of the nature of the join points" [15, p. 9]. Nevertheless, since the IIIA *joinpointtype* concept is very close to the *event* concept of Ptolemy, the translucid contract approach, including our contribution, could be partially applied to join point types.

Joint Point Interfaces (JPI) [8,5] and Closure Joint Points [4] extend and refine the notion of join point types of Steimann *et al.* JPI decouples aspects from base code and provides modular type-checking. Implicit announcement is supported through pointcuts, and explicit announcement through closure join points. JPI, similarly to JPT, lacks specification and verification features. Thus, it could also benefit from the specification and verification approach in Ptolemy and PtolemyRely.

Khatchadourian and Soundarajan [9] proposed an adaptation of the *rely-guarantee* approach used in concurrency, to be applied in aspect orientation. The base code reasoning relies on certain constraints imposed on any applicable advice. These constraints are expressed as a *rely* relation between two states. A conforming piece of advice may only make changes to the state in a way that satisfies the *rely* relation. In this way the reasoning of the base code is stable even in the presence of advice. The event pre-postconditions (P_e, Q_e) that Ptolemy imposes on every handler can be thought as a realization of the *rely* relation: $rely(\sigma_1, \sigma_2) \equiv P_e(\sigma_1) \wedge Q_e(\sigma_1, \sigma_2)$. As observed by those authors, the relation between the base code and the advice is not symmetric, as it is in the case of peer parallel processing. In their approach the base code should just *guarantee* the preconditions required by the advice. PtolemyRely follows a similar strategy, in which the base code guarantees (to the handlers) only the preconditions of the handlers. Thus in PtolemyRely: $guar(\sigma_1, \sigma_2) \equiv P_e(\sigma_1)$. Our key observation in PtolemyRely is that the advice code *might* depend on the piece of base code announced at a given join point, which may be eventually invoked from inside the advice. In PtolemyRely we take ideas from both approaches, Ptolemy and *rely-guarantee*, and declare, as part of the event type, the conditions the advice code relies on, which corresponds to what the base code should *guarantee* to every applicable advice.

5 Conclusions and Future Work

When reasoning about event announcement in AO systems, there exists a mutual dependency between the base code (*subject*) and the advising code (*handlers*). The approach followed in systems like Ptolemy [3], where the same *requires-ensures* obligation is applied to both the handlers and the announced code, limits the flexibility and the completeness of the system.

In this paper we showed an extension to the event type concept in the Ptolemy language that explicitly separates the specification and verification of these obligations. We implemented our proposal as an extension to the Ptolemy compiler and showed that the resulting methodology is more flexible and complete than the original.

We also showed how to make the verification of the handlers more concise. Instead of verifying each handler to satisfy the event pre-post specification, one can verify, once and for all, the translucid contract of the event to satisfy this pre-post specification. Then each handler can be verified to structurally refine this translucid contract. This indirectly guarantees the required behavior of the handlers.

Previous work [2] has shown how the translucid contract concept of Ptolemy can be adapted to other approaches like XPI, AAI and Open Modules; adding specification and verification capability to them. Our work suggests that these approaches, and others like JPT and JPI, would benefit from our enhancement to the translucid contract concept.

Since event subtyping has been recently proposed for Ptolemy [6], a natural future extension to our work would be to apply the added relies clause in the presence of event polymorphism, and to analyse its impact regarding modular reasoning. We also plan to apply our approach to more complex cases, and also to use static checking techniques in the verification process.

Acknowledgments. The work of both authors was partially supported by NSF grant CCF-1017334. The work of José Sánchez is also supported by Costa Rica's Universidad Nacional (UNA), Ministerio de Ciencia y Tecnología (MICIT) and Consejo Nacional para Investigaciones Científicas y Tecnológicas (CONICIT).

References

1. Akai, S., Chiba, S.: Method shelters: avoiding conflicts among class extensions caused by local rebinding. In: AOSD '12: Proceedings of the 11th annual international conference on Aspect-oriented Software Development, pp. 131–142. ACM Press, New York (2012)
2. Bagherzadeh, M., Rajan, H., Leavens, G.T.: Translucid contracts for aspect-oriented interfaces. In: FOAL '10: Workshop on Foundations of Aspect-Oriented Languages workshop, March 2010, pp. 5–14 (2010)
3. Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In: Proceedings of the 10th International Conference on Aspect-oriented Software Development, AOSD '11, pp. 141–152. ACM, New York (2011)
4. Bodden, E.: Closure Joinpoints: Block joinpoints without surprises. In: Proceedings of the 10th International Conference on Aspect-oriented Software Development, AOSD '11, pp. 117–128. ACM, New York (2011)

5. Eric Bodden, Éric Tanter, and Milton Inostroza. Joint point interfaces for safe and flexible decoupling of aspects. ACM Transactions on Software Engineering and Methodology (TOSEM), To appear (2013)
6. Fernando, R.D., Dyer, R., Rajan, H.: Event type polymorphism. In: Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages, FOAL '12, pp. 33–38. ACM, New York (2012)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969)
8. Inostroza, M., Tanter, É., Bodden, E.: Join point interfaces for modular reasoning in aspect-oriented programs. In: ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 508–511 (2011)
9. Khatchadourian, R., Soundarajan, N.: Rely-guarantee approach to reasoning about aspect-oriented programs. In: SPLAT '07: Proceedings of the 5th workshop on Engineering properties of languages and aspect technologies, Vancouver, British Columbia, Canada, p. 5. ACM Press, New York (2007)
10. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proc. of the 27th International Conference on Software Engineering, pp. 49–58. ACM Press, New York (2005)
11. Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., Sittampalam, G.: Adding open modules to AspectJ. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD), March 2006, pp. 39–50 (2006)
12. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
13. Shaner, S.M., Leavens, G.T., Naumann, D.A.: Modular verification of higher-order methods with mandatory calls specified by model programs. In: International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada, October 2007, pp. 351–367. ACM Press, New York (2007)
14. Spivey, J.M.: Understanding Z: a Specification Language and its Formal Semantics. Cambridge University Press, New York (1988)
15. F. Steimann, T. Pawlitzki, S. Apel, C. Kästner. Types and modularity for implicit invocation with implicit announcement. ACM Trans. Softw. Eng. Methodol., 1:1–1:43 (2010)
16. Sullivan, K., Griswold, W.G., Rajan, H., Song, Y., Cai, Y., Shonle, M., TewariModular, N.: aspect-oriented design with xpis. ACM Trans. Softw. Eng. Methodol., 5:1–5:42 (September 2010)

Implementing Feature Interactions with Generic Feature Modules

Fuminobu Takeyama[1] and Shigeru Chiba[2,3]

[1] Tokyo Institute of Technology, Japan
http://www.csg.ci.i.u-tokyo.ac.jp/~f_takeyama
[2] The Univiersity of Tokyo, Japan
http://www.csg.ci.i.u-tokyo.ac.jp/~chiba
[3] JST, CREST, Japan

Abstract. The optional feature problem in feature-oriented programming is that implementing the interaction among features is difficult. Either of the modules for the interacting features cannot contain the code for the interaction if those features are optional. A modular approach for implementing such interaction is separating it into a module called derivative. However, as the number of derivatives increases, it does not scale. This paper shows how derivatives for combinations of features from each group are efficiently implemented. A group of features are implemented by using the inheritance of feature modules. A super feature module works as a common interface to members of that group. It thereby allows to describe a generic derivative applicable for the groups. This paper also presents a feature-oriented programming language, FeatureGluonJ, which provides language constructs for this approach.

1 Feature-Oriented Programming

Feature-oriented programming (FOP) [26] is a programming paradigm where source code is decomposed for each feature. Although it was originally an approach for implementing similar classes, it now refers to an approach for implementing similar software products; such a family of products is called a software product line (SPL). This allows developers by just selecting the features for that necessary product.

In FOP, the code for each feature is separately described in a module called a feature module. A feature module is a collaboration of the classes needed for the feature and/or extensions to the classes belonging to other features. The extensions can be aspects in AspectJ; advices can attach code for the feature to existing code; inter-type declarations can add new fields to an existing class. Several product lines such as the feature-oriented version of Berkeley DB [18] and MobileMedia [30] have been developed in AspectJ. AHEAD Tool Suite [5] has a language construct called a refinement for the same purpose. It enables overriding existing methods and add fields from outside.

A challenge in FOP is the optional feature problem [21]. If multiple optional features interact with each other, any of feature modules for those features should

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 81–96, 2013.

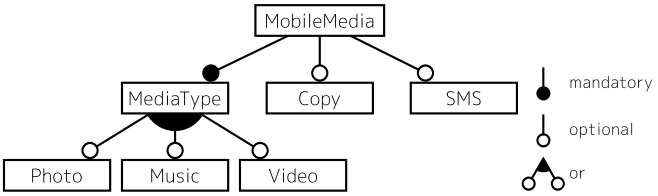

Fig. 1. The feature model of MobileMedia

not contain the code for the interaction since the code must be effective only when those interacting features are selected. Although a possible approach is separating such code into independent modules called *derivatives* [21,22], the number of derivatives tends to be large as the numbers of features increases.

This paper proposes a design principle to reduce the effort in implementing derivatives. A group of features are implemented by inheriting their common super feature module. This module works as an interface common to its sub-features and allows implementing a derivative in a reusable manner for every combination of sub-features. To demonstrate this principle, we developed a new FOP language, FeatureGluonJ, which provides a language construct called a generic feature module for reusable implementation of derivative as well as a feature-oriented module system supporting inheritance.

2 The Optional Feature Problem

This section explains a difficulty in feature-oriented decomposition known as the optional feature problem [21,22]. We can see this problem in the MobileMedia SPL. MobileMedia is a family of multimedia-management application for mobile devices and widely used in the research community of SPLs. This paper uses the six features taken from MobileMedia: MediaType, Photo, Music, Video, Copy, and SMS in Fig. 1. The representation in Fig. 1 is called a feature-model diagram [17]. In this diagram, a node represents a feature, and an edge represents dependence between features. A feature-model diagram also represents constraints on selecting a feature when building a product. A feature indicated by an edge ending with a white circle is called an optional feature. Developers select features from optional features to customize a product. If a feature is not selected for the product, that feature is not implemented in the resulting product. In Fig. 1, Photo, Music, and Video are children of MediaType. The arc drawn among the children represents a *or*-relation and developers must select at least one feature from them if their parent is selected. This paper considers such features as being optional features as well.

In FOP, a feature should be implemented as an independent feature module. If a feature is selected, the corresponding module is compiled together with other selected feature modules. In case of the original MobileMedia implemented in

AspectJ if a feature is selected, aspects belonging to the feature are compiled and linked. If a feature is not selected for a product, its aspects do not affect the product.

However, a group of optional features may interact [9] with another group. Which feature module should implement that interaction? For example, the Photo feature interact with the Copy feature in MobileMedia. If both features are selected for a product, then the command for copying a photo is displayed in the pull-up menu of the screen, i.e., window, showing that photo. This command should not be implemented in a feature module for Photo or Copy since the command is not activated unless Copy is selected. It should not be in a feature module for Copy since Copy may be selected without Photo. If so, the Copy feature module must not add the command to the menu. No matter where the command is implemented, Photo or Copy, the resulting code would cause undesirable dependence among optional features and lower the variability of a product line.

A more modular approach is to implement such interaction into an independent module called a *derivative*. A derivative is a specialized feature module for interaction, which is selected only when all interacting features are selected.[4] A derivative is described as a normal feature module. We show a derivative for the combination of Photo and Copy in List. 1. This code is a part of the MobileMedia Lancaster[5], a MobileMedia implementation in AspectJ [12]. The CopyAndPhoto aspect implements the derivative. It has an advice executed after a constructor call for the PhotoViewScreen in order to add the command for copying a photo.

The scalability of derivatives is, however, still under discussion in the research community. Suppose that n optional features interact with each other. Naively, each of the $2^n - n - 1$ combination of features requires its own derivatives. The composability may reduce the number of derivatives. The paper [26] advocates that if developers provide *lifters*, which can be regarded as derivatives, for every *pair* of interacting features, then the lifters for any combination of the features can be composed by those lifters; the number of necessary lifters are thereby $\frac{1}{2}(n^2 - n)$. In practical product lines, although all features do not interact with each other, a number of derivatives are still required. For example, in Berkeley DB refactored in FOP [18], 38 features have 53 dependencies, which must be separated into derivatives. The paper [20] concludes that the difficulty in implementing features is mainly due to the interaction among the features.

Note that feature interaction is often observed between feature groups. Suppose that two feature groups have n and m features. If a feature from one group interact with one from the other, other pairs between the two groups will also interact with each other due to the similarity of features. Such interaction will require $n \times m$ derivatives in total. Furthermore, these derivatives will be similar

[4] In the original definition in [22], a derivative is a refinement of a method introduced by another feature module.

[5] We show simplified code for explanation. The original code is available from: `http://mobilemedia.sourceforge.net/`.

```
public aspect CopyAndPhoto {
    after(Image image) returning (PhotoViewScreen f):
            call(PhotoViewScreen.new(Image)) && args(image); {
        f.addCommand(new Command("Copy", Command.ITEM, 1));
}}
```

(a) CopyAndPhoto.aj

```
public aspect CopyAndMusic {
    pointcut initForm(PlayMediaScreen mediaScreen):
        execution(void PlayMediaScreen.initForm()) && this(mediaScreen);

    before(PlayMediaScreen mediaScreen): initForm(mediaScreen) {
        mediaScreen.form.addCommand(new Command("Copy", Command.ITEM, 1));
}}
```

(b) CopyAndMusic.aj

List. 1. The derivatives for Copy and Photo/Music written in AspectJ

to each other. They are redundant and should be merged into a single or only a few derivatives.

A group is often represented by a parent-children relation in a feature-model diagram. In Fig. 1, MobileMedia contains a group consisting of Photo, Music, and Video. We call this group the MediaType group named after the parent node. There is another group that the developers of the original MobileMedia did not recognize. It is a group consisting of Copy and SMS, which enable the users to send a photo shown on the screen by SMS. The two groups involve close interaction. Copy interacts with Music as well as Photo. The derivative for Copy and Music in List. 1 (b) is similar to CopyPhoto in List. 1 (a). SMS also interacts with Photo, Music and Video[6]; if these features are selected, a command to send each medium must be added to the menu. Thus, MobileMedia requires 6 derivatives for the two groups.

The MediaType group is an extension point of MobileMedia. One of the goal of FOP is step-wise, *i.e.* incremental, development of large-scale software [5], and hence one of realistic development scenarios is adding a new media type as a new feature. Suppose that developers add plain-text documents as a new medium. Then they will have to implement derivatives for the combination of the plain-text feature and Copy and SMS. The effort to implement derivatives will increase as the size of the product line grows up.

In this paper, the optional feature problem means not only the difficulty in separating feature interaction but also the maintainability problem due to a huge number of derivatives. This paper addresses this optional feature problem by reducing redundancy of derivatives among feature groups. The interactions discussed in this paper are intended ones. Although there are unintended interactions caused by unanticipated advice conflicts, for example, at shared join points [1], discussing unintended interactions is out of scope of this paper.

[6] The original MobileMedia does not support to send a music or a video by SMS. It is not clear that this limitation is caused by the optional feature problem.

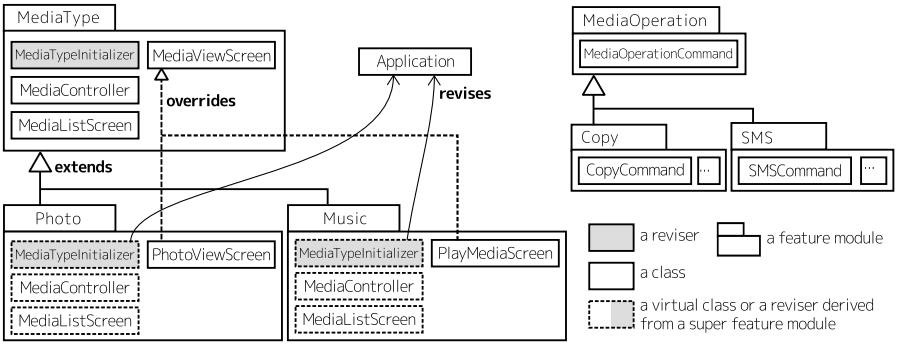

Fig. 2. An overview of feature modules consisting MobileMedia in FeatureGluonJ

3 Implementing Feature Interactions in FeatureGluonJ

Our feature-oriented programming language named *FeatureGluonJ* [7] provides language constructs to reduce redundancy of derivatives among feature groups. FeatureGluonJ is an extension of GluonJ [10], which is a aspect-oriented language based on Java. While GluonJ adds a new language construct called *reviser* to Java as a construct like AspectJ's advice. FeatureGluonJ also adds a generic feature module as a feature-oriented module system.

First, FeatureGluonJ provides an inheritance mechanism for feature modules. Features often make is-a relations [16]. In MobileMedia, the Photo feature *is a* MediaType feature. Thus, in FeatureGluonJ, the Photo feature module, which is the implementation of Photo, is a sub feature module of MediaType as shown in Fig. 2. It can not only add new classes but also redefine the classes contained in the MediaType feature module. The MediaType feature module works as a common interface to this feature group including Photo and Music. The interface represents which classes are commonly available in the feature group. This inheritance mechanism is not novel; it is provided by Caesar [24,25], CaesarJ [4], and Object Teams [14,16]. However, they are not studied in the context of modularity of feature modules [20,29]; this paper focuses on how to use this inheritance mechanism to efficiently implement derivatives.

Another unique mechanism in FeatureGluonJ is a generic feature module. It is a feature module taking feature modules as parameters. Suppose that there are two feature modules. Then the derivatives for combinations of their sub feature modules are often almost identical. For example, in Fig. 2, the derivative for Photo and Copy is almost identical to the derivative for Music and SMS since they are for combinations between MediaType and MediaOperation. A generic feature module enables to describe such derivatives in a generic manner by using the interfaces specified by MediaType and MediaOperation. Note that the task of a typical derivative is to modify the classes in the feature modules that the

[7] The FeatureGluonJ compiler is available from:
`http://www.csg.ci.i.u-tokyo.ac.jp/projects/fgj/`

derivative works for. These classes are often ones specified by the interfaces of the super feature modules such as MediaType and MediaOperation.

3.1 FeatureGluonJ

This section describes the overview of FeatureGluonJ to show how developers can implement an SPL[8]. FeatureGluonJ provides a module system called feature modules. A feature module implements a feature and a derivative. It is represented by two constructs, a feature definition and a feature declaration. A feature definition is described in a separated file, and it defines a feature name and its relation to other features. List. 2 (a) defines the MediaType feature, which is an abstract feature for other features that are to support a media type. The body of this feature is empty in this example, but it may contain import feature declarations, as shown later.

A feature declaration is similar to a package declaration in Java. It is placed at the beginning of a source file and specifies that the classes and revisers in that source file belong to the feature modules. For example, the second lines of the List. 2 (b)–(e) are feature declarations. They declare that those three classes and a reviser belong to the MediaType feature. Note that each class declaration is separated into an independent file.

An abstract feature may represent a group made by *is-a* relationships; a sub feature module of that abstract module defines a feature belonging to that group. Here, the Photo feature module is a sub-feature of MediaType, which is specified in the extends clause in List. 3 (a). Photo reuses the model-view-controller relation defined in MediaType.

After compilation of each feature, developers *select* feature modules needed for a product. Only the selected feature modules are linked together and included in the product. Which features are selected is given at link time. Note that they cannot select abstract features. If an abstract feature module like MediaType must be included in a product, the developers must select a sub-feature of that abstract feature.

To implement feature modules, FeatureGluonJ provides three kinds of class-extension mechanisms: subclasses, virtual classes, and revisers. The difference of those mechanisms is the range of effects. The first one is a normal subclass in Java and affects in the narrowest range. The extended behavior takes effect only when that subclass is explicitly instantiated.

The next class extension mechanism is virtual class overriding [23,11]. Virtual classes enable to reuse a family of classes that refer to each other through their fields or new expressions. All classes in a feature module are virtualized in FeatureGluonJ; a reference to a virtual class is late-bound. A sub feature module can implement a virtual class extending a virtual class in its super-feature. It *overrides* the virtual class in the super-feature with the new class, *i.e.*, class

[8] We cannot describe all of the semantics of our language in detail due to the space limitation. Knowledge on virtual classes and other languages with inheritance for feature module will help to read this section.

```
abstract feature MediaType {
    // MediaType has an empty body.
}
```
(a) MediaType.feature

```
package mobilemedia.controller;
feature MediaType;
import javax.microedition.lcdui.*;
import mobilemedia.ui.*;

public abstract class MediaController {
    protected boolean handleCommand(Command command) {
        if (command == OPEN) {
            open(getSelected());
            return true;
        } else if (...) { ... }
    }

    protected void open(String s) {
        MediaListScreen scr = new MediaViewScreenScreen(s)
        scr.setCommandListener(this);
        Display.setCurrent(scr);
}}
```
(b) MediaController.java

```
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public abstract class MediaViewScreen extends Canvas {
    protected void initScreen() {
        this.add(new Command("Close"));
}}
```
(c) MediaViewScreen.java

```
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public class MediaListScreen extends List {
    // forward command to controller if an item is selected
}
```
(d) MediaListScreen.java

```
package mobilemedia.main;
feature MediaType;
import mobilemedia.ui.MediaListScreen;
import mobilemedia.controller.MediaController;

class MediaTypeInitializer revises Application {
    private MediaListScreen screen;
    private MediaController controller;
    public void startApp() {
        controller = new MediaController();
        screen = new MediaListScreen(controller);
        super.startApp();
}}
```
(e) MediaTypeInitializer.java

```
package mobilemedia.main;

public class Application {
    public static void main() {
        Application app = new Application();
        app.startApp();
    }

    public void startApp() { // initializing this MobileMedia application }
}
```
(f) Application.java

List. 2. The MediaType feature module and the Application class, which has program entry point

```
feature Photo extends MediaType {}
```
(a) Photo.feature

```
feature Photo;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PhotoViewScreen overrides MediaViewScreen {
    public PhotoViewScreen(String s) {
        // : load selected image
    }
    protected void paint(Graphics g) {
        // : draw the selected photo on this screen.
}}
```
(b) PhotoViewScreen.java

List. 3. The Photo feature in FeatureGluonJ

references to the overridden class is replaced with one to the new class. Virtual class overriding is effective only within the enclosing feature module, which includes its super-feature module executed as a part of the sub-feature. It does not affect new expressions in the siblings of the sub-feature.

The syntax of virtual classes in FeatureGluonJ is different from other languages. To override a virtual class, developers must give a unique name to the new virtual class instead of the same name as the overridden class.[9] List. 3 (b) shows the PhotoViewScreen class that overrides MediaViewScreen of MediaType. An overridden class is specified by an overrides clause, placed in the position of an extends clause. Another difference in syntax is that virtual classes cannot be syntactically nested, as separated into each class to a single file.

We adopt lightweight family polymorphism [27] to make the semantics and the type system simple by avoiding dependent types. A feature module cannot be instantiated dynamically. It can be regarded as a singleton object instantiated when it is selected at link time.

The third mechanism is a reviser [10]. A reviser can extend any class in a product; the extended behavior affects globally.[10] A reviser plays a similar role to the one of aspect in AspectJ; its code overrides classes appearing in any other feature module. The class-like mechanism with a keyword revises in List. 2 (e) is a reviser. The reviser has the startApp() method, which replaces the startApp() method in the class specified in its revises clause, *i.e.*, the Application class in List. 2 (f). Whenever the startApp() method is called on an Application object, the reviser's startApp() method is first executed. By calling super.startApp(), the replaced method is executed. A reviser can also add new fields to an existing class. The reviser in List. 2 (e) adds the two fields, screen and controller, to the Application class.

Revisers in a feature module are also virtualized. A feature module derives revisers as well as virtual classes from its super-feature to reuse structure made by revisers and classes defined there. The Photo feature module in List. 3 does not contain any classes and revisers except the PhotoViewScreen class, but it also

[9] Programmers can give the same name by implementing them in a different package.
[10] GluonJ does not support global modification defined in [3].

```
feature Music extends MediaType {}
```
(a) Music.feature

```
feature Music;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PlayMediaScreen overrides MediaViewScreen {
    public PlayMediaScreen(String s) {
        // : load selected image
    }
    protected void paint(Graphics g) {
        // : draw music player
}}
```
(b) PlayMediaScreen.java

List. 4. The Music feature module in FeatureGluonJ

derives virtual classes such as MediaController and the MediaTypeInitializer reviser from MediaType (Fig. 2). A reviser will be executed only if a feature enclosing or deriving that reviser is selected. If Photo is selected, MediaTypeInitializer derived by the Photo is executed in the context of Photo. Within this MediaTypeInitializer, new MediaListScreen(...) will create an object of the class derived by Photo. The expression new MediaViewScreen() in List. 2 (b) on that object will instantiate PhotoViewScreen. The MediaTypeInitializer reviser might be derived by siblings of Photo. Suppose the Music feature module in List. 4 is implemented in the same way as the Photo. If both Photo and Music are selected, two *copies* of MediaTypeInitializer will be executed in the startApp() method but in different contexts.

These class extension mechanisms provided by FeatureGluonJ are an abstraction of the factory method pattern. For virtual-class overriding, each selected feature has its own factory. It receives a name of virtual class and returns an object of the class overriding the given class. Every new expression can be considered as a factory method call. Each virtual class has a reference to such factories. When a factory creates an object, it assigns itself to the object. A factory used in a reviser is given by the linker when its feature module is selected. This factory is one used for virtual classes in the feature module containing or deriving that reviser.

A reviser, on the other hand, can be emulated by a factory shared among all the classes in a product. If a class given to a new expression is not a virtual class, the global factory will create an object. This global factory is also used inside of a factory for each feature. Note that it is unrealistic to manually implement factory methods for every class. Moreover, a factory method pattern degrades type safety.

3.2 Derivatives in FeatureGluonJ

FeatureGluonJ provides two other constructs for referring to virtual classes in other modules. One is import feature declarations. To make coupling of other features explicit, developers have to declare the features required in a derivative

```
feature CopyPhoto {
    import feature c: Copy;
    import feature f: Photo;
}
```

(a) CopyPhoto.feature

```
package mobilemedia.copy;
feature CopyPhoto;
import mobilemedia.ui.*;

class AddCopyToPhoto revises f::PhotoViewScreen {
    protected void initScreen() {
        super.initScreen();
        this.addCommand(new c::CopyCommand());
}}
```

(b) AddCopyToPhoto.java

List. 5. The derivative between Copy and Photo rewritten from List. 1

by import feature declarations. These declarations just open the visibility scope to virtual classes of imported features. Note that when a feature module with import feature is selected, the imported features are also selected. Since an abstract feature module is never selected, it cannot be imported.

An import feature declaration is described in the body of a feature declaration. List. 5 (a) contains two import feature declarations. An identifier after a colon indicates a feature module used in this module. The left one before the colon is an alias to the imported feature module. Then a *feature-qualified access* is available as a reference to a virtual class of the imported feature module. The access is represented by a :: operator. The left of :: must be an alias declared in the feature module and the right of :: is the name of a virtual class in the feature module expressed by the alias. For example, List. 5 implements a derivative straightforwardly rewritten from List. 1. In the AddCopyToPhoto, p::PhotoViewScreen refers to the PhotoViewScreen class in the Photo feature since p is an alias of Photo. The reviser extends PhotoViewScreen and adds a command for copying a medium, which is now represented by the CopyCommand class in the Copy feature module shown in List. 6 (c) and (d).

The reason FeatureGluonJ enforces programmers to use feature-qualified access is that multiple feature modules may contain virtual classes with the same name if they extend the same module. For example, both of Photo and Music contains the MediaController class derived from MediaType, which are distinguished by aliases.

3.3 Generic Feature Modules

We found that if features are implemented by a feature module with an appropriate interface, most derivatives can be implemented by a special feature module that takes the name of required sub-features as parameters. FeatureGluonJ provides a *generic feature module*, which is a reusable feature module to implement derivatives among features extending common feature modules. The Copy feature and the SMS feature, which is not shown but implemented in the same way in List. 6, are sub feature modules of MediaOperation in List. 6 (a) and (b). Now the

```
abstract feature MediaOperation {}
```
(a) MediaOperation.java

```
package mobilemedia.ui;
feature MediaOperation;
import javax.microedition.lcdui.Command;

public abstract class MediaOperationCommand extends Command {
    public MediaOperationCommand(String labelText) {
        super(labelText, ...);
}}
```
(b) MediaOperationCommand.java

```
feature Copy extends MediaOperation {}
```
(c) Copy.feature

```
package mobilemedia.ui;
feature Copy;

public class CopyCommand overrides MediaOperationCommand {
    public CopyCommand(String labelText) {
        super(labelText);
}}
```
(d) CopyCommand.java

```
package mobilemedia.controller;
feature Copy;
import mobilemedia.ui.*;

public class CopyController revises MediaController {
    protected boolean handleCommand(Command command) {
        if (command instanceof CopyCommand) {
            :
        } else { return super.handleCommand(); }
}}
```
(e) CopyController.java

List. 6. The Copy feature module implemented by extending the MediaOperation

generic derivative among sub-features of MediaOperation and MediaType takes sub-features of those modules as parameters and behaves for a derivative among the given features.

A generic feature module is represented by an abstract feature module. It may contains import feature declarations with an abstract keyword. An alias defined by this abstract import feature works as a parameter; the alias is late-bound to a concrete module, which must be a sub-feature of one apparently assigned to the alias. An abstract import feature may import an abstract feature module. List. 7 shows a generic feature modules for derivatives between sub-features of MediaType and MediaOperation. The generic feature modules contains two abstract import declarations that import FileOperation and MediaType with the aliases, t and o, respectively.

The AddCommandToMediaType reviser in List. 7 (b) is almost the same to AddCopyToPhoto expecting for the specific parts to Copy and Photo. The reviser extends a class indicated by t::MediaViewScreen and adds command indicated by o::MediaOperationCommand. Since t and o must be bound to sub-features of the imported features, it is ensured that they provide virtual classes overriding MediaOperationCommand and MediaViewScreen. If those aliases are bound to Copy and Photo, this reviser is semantically the same as AddCopyToPhoto.

```
feature MediaOperationMediaType {
    abstract import feature o: MediaOperation;
    abstract import feature t: MediaType;
}
```

(a) MediaOperationMediaType.java

```
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

class AddCommandToMediaType revises t::MediaViewScreen {
    protected void initForm() {
        super.initForm();
        this.addCommand(new o::MediaOpCommand());
}}
```

(b) AddCommandToMediaType.java

List. 7. A generic derivative implementing common part of derivatives among MediaOperation and MediaType

```
feature CopyPhoto extends MediaTypeFileOp {
    import feature o: Copy;
    import feature t: Photo;
}
```

List. 8. Another derivative for Copy and Photo extending the generic derivative

A feature module can extend a generic feature module and bound aliases declared in its super-feature to concrete feature modules. If a feature module imports another feature module with the same alias as one used in its super-feature, the alias is bound to that feature module also in the super-feature. Suppose another implementation of the CopyPhoto feature module, which is implemented by in List. 8 by extending the generic feature module in List. 7. It assigns o and t to Copy and Photo respectively.

3.4 A Composition Language for Trivial Feature Modules

If each interacting feature is properly implemented, a derivative may not contain its own reviser nor class; we call such derivative is trivial. The derivatives among MediaType and MediaOperation including CopyPhoto in List. 8 are trivial. This is because operations such as copying a medium or sending it by SMS are reduced to operations against streams of bytes.

FeatureGluonJ also provides a composition language to define such trivial derivatives implicitly. It includes a construct defines forevery. If our linker interprets a feature module with defines forevery, it defines sub-features of this derivative automatically at linking time. defines forevery receives one or more aliases to feature modules. If the given alias is declared in an abstract import feature, it represents a set of its sub-features that are selected for the linker. The linker will define and select sub feature modules for every combination from each given set. Let $a_1, a_2, .., a_n$ be aliases given to the defines forevery and $S_i = \{f | f \in Sub(a_i) \cap f \ is \ selected\}$ where $Sub(a)$ is a function returning the

```
feature MediaOperationMediaType defines forevery(o, t) {
    abstract import feature o: MediaOperation;
    abstract import feature t: MediaType;
}
```

(a) MediaTypeFileOp.java

```
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

class AddCommandToMediaType revises t::MediaViewScreen {
    protected void initForm() {
        super.initForm();
        this.addCommand(new o::MediaOpCommand());
}}
```

(b) AddCommandToMediaType.java

List. 9. Our final version of derivatives among MediaOperation and MediaType by defines forevery

set of the sub-features that might be bound to a. A sub-derivative is created for each element of $S_1 \times S_2 \times \ldots \times S_n$. If a is an alias of a concrete feature, $Sub(a)$ returns the set containing the concrete feature only.

List. 9 shows derivatives for sub-features of MediaOperation and MediaType including derivative between Copy and Photo. The defines forevery clause allows programmers to omit concrete feature modules such as one in List. 8. Even when developers add a new feature for a new media type, they would not implement new derivatives if this generic derivative is applicable for the new feature. Otherwise, programmers would implement extra behavior for the specific combinations of feature modules as a new derivative.

3.5 Discussion

We discuss on the limitations of our language. Unfortunately, all derivatives do not become trivial after refactoring. Some derivatives are essential, which must be implemented manually. We can find essential derivatives in the expression product line [20]. Derivatives among a feature for an operator and feature for evaluating expressions is unique to each combination of features. If the feature has redundant parts, FeatureGluonJ allows to reuse it with a generic-feature module.

Although inheritance allow us to implement generic derivatives, it may cause extra effort to implement an SPL. We introduced the common super class between PhotoViewScreen and PlayMediaScreen. As shown in List. 1 (a) and (b), the original derivative uses different methods to add their commands to the menus; in Photo, it is the constructor of PhotoViewScreen, but in Music, it is the initForm() method. We add the common super class MediaViewScreen and its initScreen() method in List. 2 (c) to unify those methods among both features. We also defines Copy and SMS by extending MediaOperationCommand to make the derivatives trivial. The implementations of these feature modules are in a sense composition aware.

Our observation is that whether or not we should implement each feature considering composition is a design decision. The MediaType group and the MediaOperation group are extension points; in other words, a new feature will be added to these groups in the future. The cost of making these features composable is much lower than a large number of derivatives.

4 Related Work

Most of feature-oriented approaches such as AHEAD Tool Suite [5] are based on the idea that a feature is represented as a layer, and a product is linearly stacked layers [7]. FeatureGluonJ and other language with inheritance of feature modules allows to reuse a feature modules including a derivative multiple times in different contexts. ClassBox/J [8] can emulate virtual-classes by refinement and a scope mechanism to control the effects of refinements. Aspectual Mixin Layers [2] provide refinements for modifying advices and pointcuts of aspects defined in other feature modules. However, those languages also do not support to execute such refinements or aspects in different contexts to reuse them.

Delta-oriented programming [28] is a language paradigm where a product is composed of delta modules unique to it. A delta module can modify existing classes as a reviser can. A delta module has a conditional expression specifying when it is applied. Although a derivative is represented by a delta module applied when several features are selected together, delta-oriented programming does not provide mechanisms to reduce the number of delta modules for combinations among groups. We believe that it cannot solve the optional feature problem in this paper.

Caesar [24], CaesarJ [4] are Java-based language supporting both virtual classes and advices from AspectJ. Object Teams [14] is also a language based on Java and provides teams consisting of virtual classes. A virtual class in Object Teams is called a role class and programmers can extend another class so that it plays the role, *i.e.*, behaves as the role class defines. *Callin* binding in Object Teams allows to transfer method call on the extended class, to the role class. In those languages, virtual classes with advices or role classes are also derived from the super-feature like virtual revisers in FeatureGluonJ.

The difference from those languages is language support for generic derivatives. Object Teams provides a dependent team, which behaves polymorphically depending on a given instance of a team. The origin of the dependent team is a dependent class [13]. Dependent revisers could be as expressive as our languages. However current specification of Object Teams [15] does not allow teams dependent to multiple teams. Dependent teams hence cannot be used for derivatives among groups. Those languages may allow to demonstrate our design principle by first class objects of features, but it requires boilerplate code for each product.

In annotation based approaches for SPLs, code regions implementing a feature is annotated with syntactical blocks, #ifdef and #endif, or a color in CIDE [19]. An interaction is indicated by an intersection of these regions [6]. Although

this representation of interactions is more intuitive than a derivative, reusability of code for the interactions is not clear.

5 Conclusion

In this paper, we have shown how derivatives among feature groups are implemented efficiently in FeatureGluonJ. Firstly, designing feature modules hierarchically makes features modular and important for implementing derivatives. FeatureGluonJ facilitates to implement generic derivatives among feature groups represented by the inheritance. Such derivatives are written by using super-features as interfaces.

Our future work includes formal definition of semantics of FeatureGluonJ. FeatureGluonJ is based on GluonJ and light-weight family polymorphism which have formal definitions to prove they are mostly modular and type safe, respectively. The definition of our language will be valuable to show that it derives those properties. Real evaluation of our language requires more SPLs described in FeatureGluonJ.

References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: AOSD, pp. 39–50. ACM, New York (2009)
2. Apel, S., Leich, T., Saake, G.: Aspect refinement and bounding quantification in incremental designs. In: APSEC, pp. 796–804. IEEE Computer Society, Los Alamitos (2005)
3. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. Sci. Comput. Program. 75(11), 1022–1047 (2010)
4. Aracic, I., Gasiunas, V., Awasthi, P., Ostermann, K.: An overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
6. Batory, D., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: GPCE, pp. 13–22. ACM, New York (2011)
7. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. 1(4), 355–398 (1992)
8. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: controlling the scope of change in java. In: OOPSLA, pp. 177–189. ACM, New York (2005)
9. Bowen, T.F., Dworack, F.S., Chow, C.H., Griffeth, N., Herman, G.E., Lin, Y.-J.: The feature interaction problem in telecommunications systems. In: SETSS, pp. 59–62 (1989)
10. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: OOPSLA, pp. 539–554. ACM (2010)

11. Bateni, M.: Family polymorphism. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
12. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F.C., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: ICSE, pp. 261–270. ACM, New York (2008)
13. Gasiunas, V., Mezini, M., Ostermann, K.: Dependent classes. In: OOPSLA, pp. 133–152. ACM, New York (2007)
14. Herrmann, S.: Object teams: Improving modularity for crosscutting collaborations. In: Aksit, M., Awasthi, P., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 248–264. Springer, Heidelberg (2003)
15. Herrmann, S., Hundt, C., Mosconi, M.: OT/J language definition v1.3 (May 2011), `http://www.objectteams.org/def/1.3/s9.html#s9.3`
16. Hundt, C., Mehner, K., Preiffer, C., Sokenou, D.: Improving alignment of crosscutting features with code in product line engineering. Journal of Object Technology 6(9), 417–436 (2007)
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SE-90-TR-021, Carnegie Mellon University (1990)
18. Kästner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: SPLC, pp. 223–232. IEEE Computer Society, Los Alamitos (2007)
19. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE, pp. 311–320. ACM, New York (2008)
20. Kästner, C., Apel, S., Ostermann, K.: The road to feature modularity? In: SPLC, vol. 2, pp. 1–5. ACM, New York (2011)
21. Kästner, C., Apel, S., Rahman, S.S.u., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: SPLC, pp. 181–190. Carnegie Mellon University (2009)
22. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: ICSE, pp. 112–121. ACM, New York (2006)
23. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA, pp. 397–406. ACM, New York (1989)
24. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: AOSD, pp. 90–99. ACM, New York (2003)
25. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: FSE, pp. 127–136. ACM, New York (2004)
26. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
27. Saito, C., Igarashi, A., Viroli, M.: Lightweight family polymorphism. Journal of Functional Programming 18, 285–331 (2008)
28. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
29. Schulze, S., Apel, S., Kästner, C.: Code clones in feature-oriented software product lines. In: GPCE, pp. 103–112. ACM, New York (2010)
30. Young, T., Murphy, G.: Using AspectJ to build a product line for mobile devices. AOSD Demo (2005)

Compositional Development of BPMN*

Peter Y.H. Wong

SDL Fredhopper, Amsterdam, The Netherlands, peter.wong@fredhopper.com

Abstract. Business Process Modelling Notation (BPMN) intends to bridge the gap between business process design and implementation. Previously we provided a process semantics to a subset of BPMN in the language of Communicating Sequential Processes (CSP). This semantics allows developers to formally analyse and compare BPMN diagrams using CSP's traces and failures refinements. In this paper we introduce a comprehensive set of operations for constructing BPMN diagrams, provide them a CSP semantics, and characterise the conditions under which the operations are monotonic with respect to CSP refinements, thereby allowing compositional development of business processes.

1 Introduction

Modelling of business processes and workflows is an important area in software engineering. Business Process Modelling Notation (BPMN) [8] allows developers to take a process-oriented approach to modelling of systems. There are currently over seventy implementations of the notation, but the notation specification does not have a formal behavioural semantics, which we believe to be crucial in behavioural specification and verification activities. Previously a process semantics [12] has been given for a large subset of BPMN in the language of Communicating Sequential Processes (CSP) [10]. This semantics maps BPMN diagrams to finite state CSP processes. Using this semantics, the behaviour expressed in BPMN can be formally verified and BPMN diagrams can be behaviourally compared. Verification and comparison are expressed as CSP traces and failures refinements. Refinements of finite state CSP processes can be automatically verified using a model checker like FDR [4].

1.1 Monotonicity and Refinement

One major problem of verifying concurrent systems by model checking is potential state explosion. To alleviate this one can exploit compositionality. Our constribution is to introduce a comprehensive set of operations to incrementally construct BPMN diagrams. We provide these operations with a CSP semantics, and characterise the conditions under which the operations are monotonic with respect to CSP refinements. These operations are partitioned into the following

* Partly funded by Microsoft Research.

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 97–112, 2013.

categories: sequential composition, split, join, iteration, exception and collaboration.

The combination of monotonicity and refinement allows one to construct and verify behavioural correctness of large complex systems compositionally. Informally, for any two BPMN diagrams C and D, $D \sqsubseteq C$ denotes C is a refinement of D. We let $f(.)$ be an operation over BPMN diagrams. If $f(.)$ is monotonic with respect to the refinement, then by construction $f(D) \sqsubseteq f(C) \Leftrightarrow D \sqsubseteq C$ for all BPMN diagrams C and D. As a result we can incrementally increase the complexity of C and D using the proposed operations without needing further verification.

Moreover, monotonic operations preserve refinement-closed properties. A relation $\oplus$ is refinement-closed if and only if for all BPMN diagrams P and Q such that if $P \oplus Q$, then it is the case that $P' \oplus Q'$ for all $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. One important refinement-closed property is the behavioural compatibility, *compatible*, between BPMN pools [11,13]. Given two BPMN pools P and Q, *compatible*(P, Q) if and only if P and Q are deadlock free and that their collaboration is deadlock free. If we let $f(.)$ be a monotonic operation over BPMN diagrams and *compatible*$(f(C), f(D))$, then by construction *compatible*$(f(C'), f(D'))$ for all C' and D' such that $C \sqsubseteq C'$ and $D \sqsubseteq D'$. As a result we may use the proposed operations to perform independent development while maintaining *any* refinement-closed properties.

1.2 Structure

This paper begins with a brief introduction to Z and CSP in Section 2, and then an overview of the abstract syntax of BPMN in Z and its behavioural semantics in Section 3. Our contribution starts in Section 4: in this section we introduce a set of operations to construct BPMN diagrams and provide them with a CSP semantics. Using this semantics, we characterise the conditions under which these operations are monotonic with respect to CSP refinements. We conclude this paper with a discussion on related work and a summary.

2 Preliminaries

Z The Z notation [14] is a language for state-based specification. It is based on typed set theory with a structuring mechanism: the schema. We write some schema N to make declaration d that satisfies constraint p as $N \mathrel{\widehat=} [d \mid p]$. Z provides a syntax for set expressions, predicates and definitions. Types can either be basic types, maximal sets within the specification, each defined by simply declaring its name [*Type*], or be free types, introduced by identifying each of the distinct members, introducing each element by name $Type ::= E_0 \mid ... \mid E_n$. By using an axiomatic definition we can introduce a new symbol x, an element of S, satisfying constraint p.

$$x : S \mid p$$

CSP In CSP [10], a process is a pattern of behaviour, denoted by events. Below is the syntax of the subset of CSP used in this paper.

$$P, Q ::= P \,|||\, Q \mid P \,[\![A]\!]\, Q \mid P \,[\![A \mid B]\!]\, Q \mid P \,\Box\, Q \mid P \sqcap Q \mid P \fatsemi Q \mid e \rightarrow P \mid \mathit{Skip} \mid \mathit{Stop}$$

Given processes P and Q, $P \,|||\, Q$ denotes the interleaving of P and Q; $P \,[\![A]\!]\, Q$ denotes the partial interleaving of P and Q synchronising events in set A; $P \,[\![A \mid B]\!]\, Q$ denotes parallel composition, in which P and Q can evolve independently performing events from A and B respectively but synchronise on events in their intersection $A \cap B$; we write $\Big\|\, i : I \bullet A(i) \circ P(i)$ to denote an indexed parallel combination of processes $P(i)$ for i ranging over I. $P \,\Box\, Q$ denotes the external choice between P and Q; $P \sqcap Q$ denotes the internal choice between P or Q; $P \fatsemi Q$ denotes the sequential composition of P and Q; $e \rightarrow P$ denotes a process that is capable of performing event e, and then behaves as P. The process *Stop* is a deadlocked process and the process *Skip* is a successful termination. One of CSP's semantic models is the stable failures ($\mathcal{F}$) The stable failures model records the failures of processes. We write $\mathit{traces}(P)$ and $\mathit{failures}(P)$ to denote the traces and the failures of process P. A failure is a pair $(s, X) \in \mathit{failures}(P)$ where $s \in \mathit{traces}(P)$ is the trace of P and X is the set of events of P refuses to do after s. CSP semantics admit refinement orderings such as the stable failures refinement $P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q)$. Refinement under the stable failures model allows assertions about a system's *safety* and *availability* properties.

3 Formalising BPMN

A BPMN diagram is made up of a collection of BPMN elements. Elements can either be events, tasks, subprocesses or control gateways. Elements are grouped in pools. A pool represents a participant in a business process; a participant may be an entity such as a company. Elements in a pool are connected by sequence flows to depict their flow of control in the pool, A BPMN diagram is a collection of pools in which elements between pools may be connected by message flows to depict the flow of messages between pools.

As a running example we consider the BPMN diagram shown in Fig. 1. The diagram describes the business process of an online shop promoting a sale. Specifically, it is a business collaboration between an online shop and a customer, depicted by two pools. The online shop business process begins by sending a message to the customer about a sale offer. This is modelled as a message flow from the task named Send Offer to the message event. The business process then waits until it receives either a confirmation or a decline from the customer. This decision waiting is modelled using the event-based exclusive-or gateway from the Send Offer task to the tasks Receive Confirmation and Receive Decline. If a decline is received, the online shop business process ends. If a confirmation is received, the online shop receives payment from the customer, sends the invoice and dispatches the goods to her. The customer's business process begins by receiving a message from the online shop about a certain promotion item. She

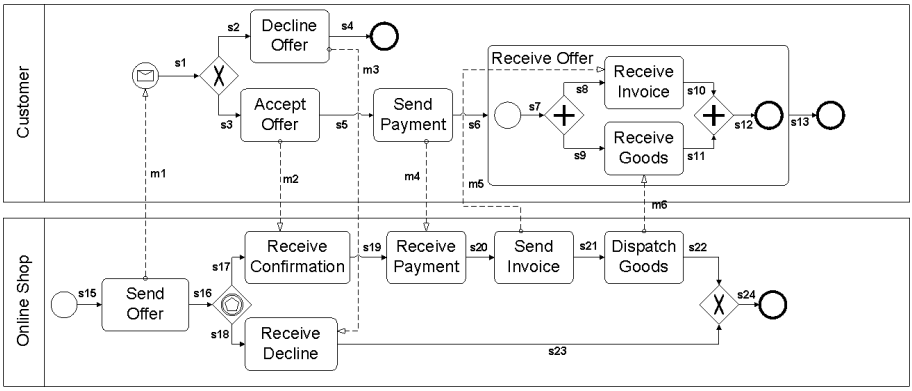

Fig. 1. A running example of a BPMN diagram

may either accept or decline the offer. The decision is modelled using the data-based exclusive-or gateway from the message event to the tasks Decline Offer and Accept Offer. If she decides to accept the offer, she sends payment to the shop, modelled by task Send Payment, then waits for her offer. This is modelled as a subprocess Receive Offer consisting of two tasks: The task Receive Goods models the receiving of goods, while Receive Invoice models the receiving of the invoice. Note these activities may happen in either order and this is modelled using a parallel gateway to both tasks. If she declines the offer, the business process ends.

We now give an overview of BPMN's abstract syntax in Z and its semantics in CSP. For brevity, we highlight only the most important aspects of the syntax and semantics. The complete formalisation of the syntax and semantics can be found in the author's thesis [11, Chapters 4, 5].

In the abstract syntax we let basic types *Sflow* and *Mflow* be types of sequence flows and message flows, and *TaskName* be the type of task names; we use the free type $Type ::= start \mid end \mid ...$ to record the type of an element. The free type $Element ::= atom\langle\langle Atom\rangle\rangle \mid comp\langle\langle Atom \times \mathbb{F}_1\, Element\rangle\rangle$ then records individual BPMN elements, where schema $Atom \mathrel{\widehat=} [t : Type;\ in, ou : \mathbb{F}\, Sflow;\ sn, re : \mathbb{F}\, Mflow \mid in \cap ou = \emptyset \wedge sn \cap re = \emptyset]$ records the type, the sequence flows and the message flows of an element.

We define schema $Pool \mathrel{\widehat=} [proc : \mathbb{F}_1\, Element]$ to record individual pools, it defines component *proc* to record a non-empty finite set of BPMN elements contained in a pool; note that for brevity we have omitted the specification of predicate constraints on that component. We define $InitPool \mathrel{\widehat=} [proc' : \mathbb{F}_1\, Element \mid proc' = \{se, ee\}]$ to be the initial state of *Pool*, where *se* is a start event and *ee* is an end event such that $(atom^{\sim} se).ou = (atom^{\sim} ee).in$. We also specify the structure of a BPMN diagram using the schema $Diagram \mathrel{\widehat=} [pool : PoolId \pfun Pool \mid pool \neq \emptyset]$. It states that a diagram consists of one or more BPMN pools, with each pool being uniquely identified by some *PoolId* value i such that $pool(i)$ gives that pool (where *PoolId* is a basic type). Similar to *Pool*, we define $InitDiagram \mathrel{\widehat=} [pool' : PoolId \pfun Pool \mid pool' = \{p1 \mapsto \langle proc \rightsquigarrow \{se, ee\}\rangle\}]$ to be the initial state of *Diagram*, where $p1$ is a *PoolId*.

We define the semantic function *bToc* that takes a BPMN diagram and returns a CSP process that models communications between elements in that diagram. We present the semantic definition in a bottom up manner, starting with individual BPMN elements.

We first define function *aToc* to model atomic elements, that is, events, tasks and gateways. The function defines the behaviour of the elements by sequentially composing the processes that model the element's incoming sequence flows and message flows, the element's type and the element's outgoing sequence flows and message flows. Note that other than start and end events, *aToc* models other atomic elements as recursive processes.

$$aToc : Element \pfun CSP$$

Our semantics models the communications between elements as a parallel combination of processes, each modelling the behaviour of an element. Our semantics ensures that upon an end event being triggered, the containing BPMN process may terminate if its contained elements can also terminate. This is achieved by insisting that each end event performs a completion event, and that all other elements may synchronise on one of these completion events to terminate.

$$cp : (\mathbb{F}\, Element \times Element) \to CSP$$

The function *cp* defines the execution of a BPMN element as follows: for an atomic element, *cp* applies function *aToc*, and for a compound element, *cp* applies function *mToc*; function *mToc* defines the sequential composition of these processes: the CSP process that models the incoming sequence flows of the compound element; the parallel composition of processes, each modelling an element contained in the compound element, and the CSP process that models the outgoing sequence flows of the element.

$$mToc : CSP \times \mathbb{F}\, Element \times CSP \to CSP$$

A BPMN diagram is modelled by semantic function *bToc*, which defines a parallel composition of processes, each modelling a BPMN pool in the diagram. A pool is modelled by function *pToc*, which defines a parallel composition of processes, each modelling elements in the pool.

$$pToc : Pool \to CSP$$
$$bToc : Diagram \to CSP$$

This semantic function induces an equivalence relationship on the behaviour of BPMN diagrams. Two BPMN diagrams are equivalent when each failures-refines the other. The notion of equivalence is formally defined as follows:

Definition 1. ***Equivalence.*** *Two BPMN diagrams P and Q are equivalent, denoted as* $P \equiv_{BPMN} Q$ *if and only if* $bToc(Q) \sqsubseteq_{\mathcal{F}} bToc(P) \land bToc(P) \sqsubseteq_{\mathcal{F}} bToc(Q)$.

For example, we consider our online shop running example shown in Fig. 1. We define CSP process *CP* in Equation 1 to model the behaviour of the *Customer*

BPMN pool, where αP denotes the alphabet of process P. The start event is identified by sCP while all other BPMN elements in the pool are identified by their incoming sequence flows.

$$
\begin{aligned}
EP &= c.s13 \rightarrow Skip \;\Box\; c.s4 \rightarrow Skip \\
P(sCP) &= (m.m1 \rightarrow s.s1 \rightarrow EP) \;\Box\; EP \\
P(s1) &= (s.s1 \rightarrow (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P(s1)) \;\Box\; EP \\
P(s2) &= (s.s2 \rightarrow w.DO \rightarrow m.m3 \rightarrow s.s4 \rightarrow P(s2)) \;\Box\; EP \\
P(s3) &= (s.s3 \rightarrow w.AO \rightarrow m.m2 \rightarrow s.s5 \rightarrow P(s3)) \;\Box\; EP \\
P(s4) &= (s.s4 \rightarrow c.s4 \rightarrow Skip) \;\Box\; c.s13 \rightarrow Skip \\
P(s5) &= (s.s5 \rightarrow w.SP \rightarrow m.m4 \rightarrow s.s6 \rightarrow P(s4)) \;\Box\; EP \\
P(s6) &= (s.s6 \rightarrow RO \fatsemi s.s13 \rightarrow P(s6)) \;\Box\; EP \\
P(s13) &= (s.s13 \rightarrow c.13 \rightarrow Skip) \;\Box\; c.s4 \rightarrow Skip \\
CP &= \Big\| \, i : \{sCP, s1, s2, s3, s4, s5, s6, s13\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{1}
$$

For presentation purpose, we model sequence flows by prefixing '$s.$', message flows by '$m.$' and completion events by '$c.$'. We abbreviate each task name using the first letter of each word in its name. For example, the CSP event $w.DO$ represents the work done of task Decline Offer, the process $P(s6)$ models the behaviour of the Receive Offer subprocess; the definition of RO is defined in Equation 2.

$$
\begin{aligned}
FP &= c.s12 \rightarrow Skip \\
P(sRO) &= (s.s7 \rightarrow FP) \;\Box\; FP \\
P(s7) &= (s.s7 \rightarrow (s.s8 \rightarrow Skip \;|||\; s.s9 \rightarrow Skip) \fatsemi P(s7)) \;\Box\; FP \\
P(s8) &= ((s.s8 \rightarrow Skip \;|||\; m.m5 \rightarrow Skip) \fatsemi w.RI \rightarrow s.s4 \rightarrow P(s8)) \;\Box\; FP \\
P(s9) &= ((s.s9 \rightarrow Skip \;|||\; m.m6 \rightarrow Skip) \fatsemi w.RG \rightarrow s.s11 \rightarrow P(s9)) \;\Box\; FP \\
P(s10) &= ((s.s10 \rightarrow Skip \;|||\; s.s11 \rightarrow Skip) \fatsemi s.s12 \rightarrow P(s10)) \;\Box\; FP \\
P(s12) &= s.s12 \rightarrow c.12 \rightarrow Skip \\
RO &= \Big\| \, i : \{sRO, s7, s8, s9, s10, 12\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{2}
$$

We may similarly define CSP process OS to model the behaviour of the *Online-Shop* BPMN pool. The CSP process $CP \,[\![\, \alpha CP \,|\, \alpha OS \,]\!]\, OS$ then models the interaction between the customer and the online shop business processes.

4 Constructing BPMN

Using Z we provide a comprehensive set of *operations* for constructing BPMN processes. Specifically these are operation schemas on the state schemas *Pool* and *Diagram*. These operations are partitioned into the following categories: sequential composition, split, join, iteration, exception and collaboration. Informally, sequential composition adds to a BPMN process an activity (a task or a subprocess); split adds a choice to two or more activities; join joins two or more

activities via a join gateway; iteration adds a loop using a exclusive-or split and join gateway; exception adds exception flows to a activity, and collaboration connects two BPMN pools with a message flow.

While these operations are not part of BPMN, we did not need to extend the existing syntax of BPMN for defining these operations. These operations are designed to provide the following benefits:

1. To provide operations to construct business processes. We have chosen a comprehensive set of operations for constructing business processes similar to those in structured programming [1];
2. To ensure the syntactic consistency of business processes by calculating the preconditions of the operations. Precondition calculations can be found in the author's thesis [11, Appendix B];
3. To allow the compositional development of business processes. We provide a CSP semantics to these operations, and characterise the conditions under which these operations are monotonic with respect to CSP traces and failures refinements, and
4. To encourage formal tool support for constructing business processes. The behavioural of BPMN diagrams constructed using the provided operations admits verification such as automatic refinement checking. These operations can be implemented in a BPMN development environment that supports the semantic translation of BPMN to CSP[1] and the automated refinement checking via model checkers such as the FDR.

4.1 Syntax and Semantics

We describe four operations using the combination of Z and CSP; when describing operations, we refer to Fig. 2 for illustration purposes. Diagrams labelled with a number depict an operation's before state and diagrams with a letter depict an operation's after state. Operations whose after states are one of Digrams A and B assume Diagram 1 to be the before state; the operation whose after state is Diagram C assumes Diagram 2 to be the before state, and the operation whose after state is Diagram E assumes Diagram 3 to be the before state. For reason of space, we mainly focus on semantic definitions of these operations and present the syntactic definition of sequential composition (*SeqComp*), while for the other operations, we present their type declaration. We also provide an informal description of these operations.

We provide functions *pf*, *ext*, *int* and *par* to model prefixing, external choice, internal choice and interleaving. We provide functions *type*, *in*, *ou*, *re* and *sd* on BPMN elements to obtain their type, incoming and outgoing sequence flows, and incoming and outgoing message flows respectively. We provide functions *sf*, *mf* and *fn* to model sequence flow, message flow and completion as CSP events. Furthermore, we provide functions *sflow*, *cnt*, *ends*, *eles* and *modify* as follow:

[1] A prototypical implementation of the semantic translation can be found at `http://sites.google.com/site/peteryhwong/bpmn`

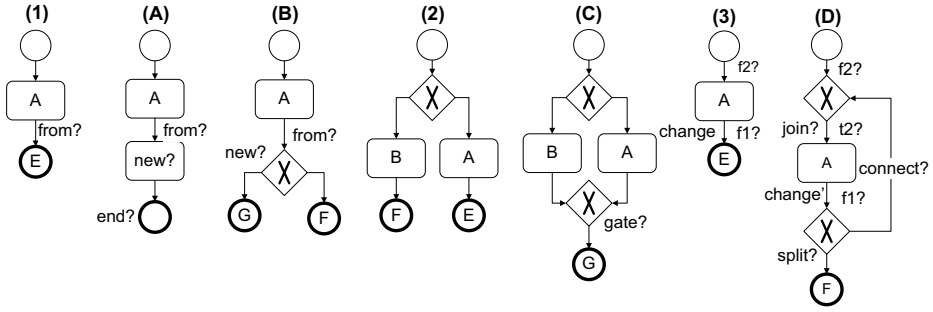

Fig. 2. Operations

sflow(*es*) takes a set of elements *es* and returns all sequence flows of the elements in *es*; *cnt*(*e*) takes element *e* and recursively returns the set of elements contained in element *e*; *ends*(*ps*) takes a set of elements *ps* and returns a partial function that maps each incoming sequence flow of an end event to that event; *nonsend*(*ps*) takes a set of elements *ps* and returns a partial function that maps each incoming sequence flow of an element that is not a task or an intermediate event to that element, and *modify*(*ps*, *ns*, *os*) takes three sets of elements *ps*, *ns* and *os* and returns *ps* with elements of *os* contained in *ps* replaced with elements in *ns*.

SeqComp The schema *SeqComp* adds either a task, a subprocess, or an intermediate message event to a BPMN pool. It takes two elements *new*? and *end*?, and sequence flow *from*? as inputs and replaces the end event in the component *proc* that has incoming sequence flow *from*? with elements *new*? and *end*?.

$$
\begin{aligned}
&\mathit{SeqComp} \mathrel{\widehat=} [\Delta \mathit{Pool};\ \mathit{Commons};\ \mathit{end?} : \mathit{Element} \mid \\
&\quad \#(\mathit{in}\ \mathit{new?}) = 1 \land \#(\mathit{ou}\ \mathit{new?}) = 1 \land \mathit{in}(\mathit{end?}) = \mathit{ou}(\mathit{new?}) \land \\
&\quad \mathit{type}(\mathit{end?}) \in \{\mathit{end}\} \cup \operatorname{ran} \mathit{emsg} \land \#\mathit{in}(\mathit{end?}) = 1 \land \#\mathit{ou}(\mathit{end?}) = 0 \land \\
&\quad \mathit{proc}' = \mathit{modify}(\mathit{proc}, \{\mathit{new?}, \mathit{end?}\}, \{\mathit{ends}(\mathit{proc})\ \mathit{from?}\})]
\end{aligned}
$$

Here *SeqComp* also declares the following schema *Commons*:

$$
\begin{aligned}
&\mathit{Commons} \mathrel{\widehat=} [\mathit{Pool};\ \mathit{new?} : \mathit{Element};\ \mathit{from?} : \mathit{Seqflow} \mid \\
&\quad (\mathit{ou}(\mathit{new?}) \cup \mathit{sflow}(\mathit{cnt}(\mathit{new?}))) \cap \bigcup\{e : \mathit{proc} \bullet \mathit{sflow}(\mathit{cnt}(e))\} = \emptyset \land \\
&\quad \mathit{from?} \in \mathit{in}(\mathit{new?}) \land \mathit{in}(\mathit{new?}) \subseteq \operatorname{dom}(\mathit{ends}\ \mathit{proc})]
\end{aligned}
$$

Schema *SeqComp* is illustrated by Diagrams 1 and A in Fig. 2, where the end event labelled *E* is specified by the expression ((*ends proc*) *from*?). The illustration shows how this operation replaces element *E* with element *new*? and *end*?. Specifically, *new*? is either an intermediate message event (with no message flow) or an activity. That is, *new*? has exactly one incoming and one outgoing sequence flow, and *end*? is an end event. Furthermore, *SeqComp* includes *Commons* to ensure the following constraints: 1) no outgoing sequence flow of *news* as well as of elements contained in *new*? must also be a sequence flow of any element contained the before state component *proc*; 2) *from*? is an incoming sequence flow of *new*?, and 3) *from*? is also an incoming sequence flow of an end event contained in *proc*.

We let e denote the end event $end(proc)\,from?$. The semantics of $new?$ and $end?$ are provided by the processes $P(new?) = cp((proc \setminus \{e\}) \cup \{end?\}, new?)$ and $P(end?) = cp(proc \setminus \{e\}, end?)$ respectively. In general, unless otherwise specified, given a state $Pool$ with component $proc$, we write $P(i)$ to denote the process $cp(proc, i)$, and define $as(es) = \bigcup\{i : es \bullet \alpha P(i)\}$ to return the alphabet of the process semantics of elements in es. The semantics of *SeqComp* is given by the following process,

$$OB \,[\![\, AB \mid as(\{new?, end?\}) \,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid \alpha P(end?) \,]\!]\, P(end?))$$

where $OB = \|\, i : proc \setminus \{e\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup fn(end?)) \circ cp(proc \cup \{end?\}, i)$ and $AB = as(proc \setminus \{e\}) \cup \{fn(end?)\}$.

Split The schema *Split* adds either an exclusive or a parallel split gateway to a BPMN pool.

$$Split \mathrel{\widehat=} [\Delta Pool;\ Commons;\ outs? : \mathbb{F}_1\ Element]$$

The operation takes as inputs gateway $new?$, a set of end events $outs?$, and sequence flow $from?$. Schema *Split* is illustrated by Diagrams 1 and B in Fig. 2, where the end event labelled E is specified by the expression $((ends\ proc)\ from?)$. The illustration shows how the operation replaces E with element $new?$ and the set of elements $outs?$, which contains elements labelled F and G. We now consider the constraints specified by this operation in detail. *Split* includes constraints specified by *Commons* about $new?$, $from?$ and the before state $Pool$. It also specifies the following constraints on all input components: 1) $new?$ must be either an exclusive or a parallel split gateway; 2) $outs?$ must be a non-empty set of end events, in which elements do not share incoming sequence flows; 3) incoming sequence flows of elements in $outs?$ are not sequence flows of elements contained in $proc$ of before state $Pool$, and 4) incoming sequence flows of elements in $outs?$ are exactly the outgoing sequence flows of $new?$.

We let e denote the end event $end(proc)\,from?$. The semantics of $new?$ is then defined as $P(new?) = cp((proc \setminus \{e\}) \cup outs?, new?)$, and the semantics of the set of elements $outs?$ can be modelled by the parallel composition of processes $QS = \|\, o : outs? \bullet \alpha Q(o) \circ Q(o)$, where each process $Q(o)$ is defined as $cp((proc \setminus \{e\}) \cup outs?, o)$. The semantics of *Splits* is then given by the following process,

$$OB \,[\![\, AB \mid as(outs? \cup \{new?\}) \,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid as(outs?) \,]\!]\, QS)$$

where $OB = \|\, i : proc \setminus \{e\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup fn(\!|\ outs?\ |\!)) \circ cp(proc \cup outs?, i)$ and $AB = as(proc \setminus \{e\}) \cup fn(\!|\ outs?\ |\!)$.

Join The schema *Join* adds an exclusive join gateway to a BPMN pool.

$$Join \mathrel{\widehat=} [\Delta Pool;\ gate?, end? : Element]$$

The operation takes input components gateway $gate?$ and end event $end?$. Schema *Join* is illustrated by Diagrams 2 and C in Fig. 2, where the incoming sequence

flows of E and F are the incoming sequence flows of *gate*?. The illustration shows how the operation replaces elements E and F with elements *gate*? and *end*?. Here *end*? is labelled G in the illustration. Specifically, *Join* defines the following constraints: 1) *gate*? is either an exclusive or a parallel join gateway; 2) *end*? is an end event; 3) *gate*?'s incoming sequence flows are incoming sequence flows of some end events contained in either *proc* of the before state *Pool*, or one of the subprocesses contained in *proc* of the before state *Pool*; 4) outgoing sequence flows of *gate*? are exactly the incoming sequence flows of *end*?, and 5) incoming sequence flows of *end*? are not sequence flows of elements contained in *proc* component of the before state *Pool*.

We let $es = end(proc)(\!| in(gate?) |\!)$, and the semantics of *gate*? and *end*? are provided by $P(gate?) = cp((proc \setminus es) \cup \{end?\}, gate?)$ and $P(end?) = cp(proc \setminus es, end?)$ respectively. The semantics of *Join* is then given by the following process,

$$OB \,[\![\, AB \mid as(\{gate?, end?\}) \,]\!]\, (P(gate?) \,[\![\, \alpha P(gate?) \mid \alpha P(end?) \,]\!]\, P(end?))$$

where $OB = \|\, i : proc \setminus es \bullet ((\alpha P(i) \setminus fn(\!| es |\!)) \cup \{fn(end?)\}) \circ cp(proc \cup \{end?\}, i)$ and $AB = (as(proc \setminus es) \setminus fn(\!| es |\!)) \cup \{fn(end?)\}$.

Loop The schema *Loop* adds an exclusive split gateway and an exclusive join gateway to a BPMN pool to construct a loop in the pool. The operation is defined as the conjunction of schemas *ConnectSplit*, *ConnectJoin* and *Connect*,

$$Loop \mathrel{\widehat=} (ConnectSplit \wedge ConnectJoin \wedge Connect) \setminus (change, change')$$

where the declaration of these schemas are shown as follow.

$$ConnectSplit \mathrel{\widehat=} [Commons[split?/new?];\ connect? : Seqflow;\ end? : Element]$$
$$ConnectJoin \mathrel{\widehat=} [Pool;\ ch, ch', join? : Element;\ connect?, f2?, t2? : Sflow]$$
$$Connect \mathrel{\widehat=} [\Delta Pool;\ from?, f2?, t2? : Sflow;\ ch, ch', split?, join?, end? : Element]$$

Briefly, *ConnectSplit* specifies the constraints on the input exclusive split gateway, *ConnectJoin* specifies the constraints on the input exclusive join gateway, and *Connect* specifies the interdependent constraints on the two gateways.

Schema *Loop* is illustrated by Diagrams 3 and D in Fig. 2. Specificially, *Loop* performs a two-step operation: 1) replace the end event in the component *proc* that has incoming sequence flow *from*? with gateway *split*? and the end event *end*?. The constraints of *Loop* ensure *connect*? is one of *split*?'s outgoing sequence flows, and 2) add a join gateway *join*? to the component *proc*. The constraints of *Loop* ensure that *connect*? and $f2?$ are incoming sequence flows of *join*? and $t2?$ is the outgoing sequence flow of *join*?. The constraints also ensure that there is an element contained in the before state of *proc* that has $f2?$ as one of its incoming sequence flows and replaces this element's $f2?$ incoming sequence flow with $t2?$. This element is defined by the expression $(\mu\, p : proc \mid f2? \in in(p) \bullet p)$ and we let m denote this element.

We let $e = end(proc)\, from?$, and $ps = (proc \setminus \{e\}) \cup \{end?\}$. The semantics of *split*?, *join*? and *end*? are provided by the processes $P(split?) = cp(ps, split?)$,

$P(join?) = cp(ps, join?)$ and $P(end?) = cp(ps, end?)$ respectively. We also let $P(m') = P(m)[fn(f2?) \leftarrow fn(t2?)]$, which renames all occurrences of $sf(f2?)$ to $sf(t2?)$ in $P(m)$. The semantics of *Loop* is then given by the following process,

$$
\begin{array}{l}
OB \,[\![\, as(proc \setminus \{e, m\}) \cup \{fn(end?)\} \mid as(\{split?, join?, end?, m'\})]\!] \\
\quad (P(m') \,[\![\, \alpha P(m') \mid as(\{split?, join?, end?\})]\!] \\
\quad\quad (P(split?) \,[\![\, \alpha P(split?) \mid as(\{join?, end?\})]\!] \\
\quad\quad\quad (P(join?) \,[\![\, \alpha P(join?) \mid \alpha P(end?) \,]\!]\, P(end?))))
\end{array}
$$

where $OB = \Big\|\, i : proc \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup \{fn(end?)\}) \circ cp(proc \cup \{end?\}, i)$.

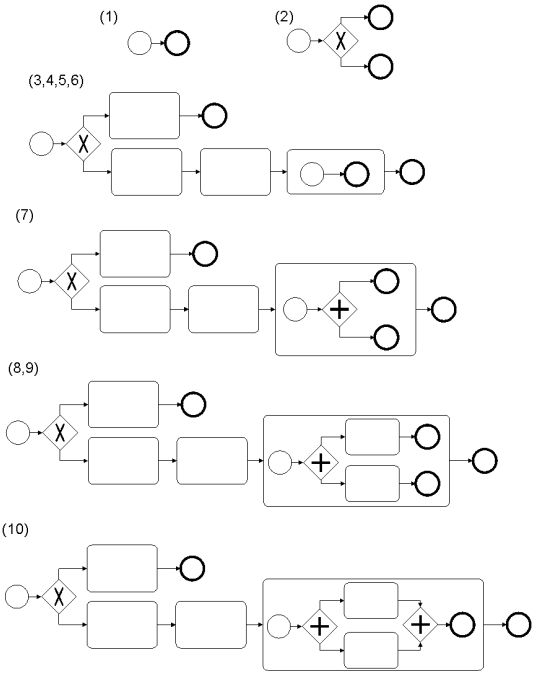

Fig. 3. Construction of the customer business process

Example Here we describe how to construct the customer business process of our online shop example in Fig. 1. A step-by-step illustration of the business process construction is shown in Figure 3. The following describes the steps shown in the figure.

1. Start with a subprocess' initial state, containing a start and an end events (Step 1);
2. Add an exclusive split gateway using *Split* (Step 2);

3. Apply *SeqComp* four times to add three task elements and one subprocess element. The subprocess element is in an initial state, again containing a start and an end events (Steps 3, 4, 5, 6);
4. Apply *Split* to add an exclusive split gateway inside the subprocess element that was added in the previous step (Step 7);
5. Add two task elements inside the subprocess element using *SeqComp* two times (Steps 8, 9) and,
6. Join two end elements inside the subprocess element using *Join* (Step 10).

4.2 Analysis

The combination of monotonicity and refinements allows one to verify behavioural correctness of large complex systems incrementally. We would like to show that the operations considered in the previous section to be monotonic with respect to failures refinement ($\sqsubseteq_{\mathcal{F}}$). However, we observe that *in general* these operations are *not* monotonic.

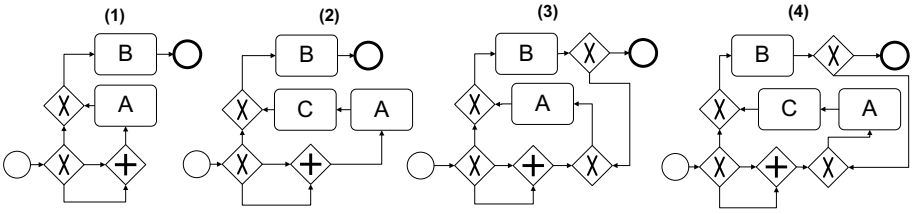

Fig. 4. A non-monotonic scenario

Consider the BPMN processes in Fig. 4. They are constructed by a combination of operations *SeqComp*, *Split*, *Join* and *Loop*. We let $P1$, $P2$, $P3$ and $P4$ denote the BPMN processes shown in Figs. 4(1), (2), (3) and (4) respectively We observe that both $P1$ and $P2$ deadlock, because in both cases not all of the parallel join gateway's incoming sequence flows can be triggered. Moreover, we observe that $P1$ and $P2$ admit the same behaviour, that is, $P1 \equiv_{BPMN} P2$. We now consider $P3$ and $P4$ that are constructed by applying the operation *Loop* to $P1$ and $P2$ respectively. We observe that unlike in $P1$ and $P2$, A can be triggered in both $P3$ and $P4$. However, after performing A, $P4$ can trigger element C, while $P3$ cannot. As a result we have $P3 \not\equiv_{BPMN} P4$. This shows that in general not only the operations are non-monotonic, but more importantly the equivalence $\equiv_{BPMN}$ is not congruent with respect to these operations. To ensure that the operations can be applied monotonically, we assume the following conditions about BPMN diagrams and processes and state that the operations are monotonic with respect to the failures refinement; complete proofs of the following theorems can be found in the author's thesis [11, Appendix C].

(a) Given any two BPMN pools $X, Y : Pool$, if we have $pToc(X) \sqsubseteq_{\mathcal{F}} pToc(Y)$, then the set of elements $X.proc \setminus Y.proc$ can be partitioned into two sets A and B:

(i) Each element $e \in A$ is either an exclusive split gateway or a subprocess such that there exists an element $e' \in Y.proc \setminus X.proc$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$.

(ii) For each element $f \in B$, there exists some exclusive split gateway $e \in A$, such that there exists some element $e' \in Y.proc \setminus X.proc$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$. Moreover, $ou(e') \subset ou(e)$ and either $in(f) \subset ou(e) \setminus ou(e')$ or there exists an element $g \in B$ such that $in(g) \subset ou(e) \setminus ou(e')$ and there exists a sequence of sequence flows connecting g to f.

Furthermore, $Y.proc \setminus X.proc$ can be partitioned into two sets M and N:

(iii) For each element $m \in M$, there exists exactly one element $e \in A$ such that $P(e) \sqsubseteq_{\mathcal{F}} P(m)$,

(iv) Each element $e \in N$ is an exclusive join gateway such that there exists an exclusive join gateway $f \in B$ with the same outgoing sequence flow and whose set of incoming sequence flows is a superset of e's.

(b) Given any two BPMN diagrams $X, Y : Diagram$. If we have $bToc(X) \sqsubseteq_{\mathcal{F}} bToc(Y)$, we have $\mathrm{dom}\, X.pool = \mathrm{dom}\, Y.pool$ and $\forall i : \mathrm{dom}\, X.pool \bullet pToc(X.pool(i)) \sqsubseteq_{\mathcal{F}} pToc(Y.pool(i))$.

Theorem 1. ***Monotonicity.*** *Assuming BPMN diagrams satisfy Conditions (a) and (b), the composition operations are monotonic with respect to $\mathcal{F}$.*

Condition (a) is appropriate: according to our process semantics, only exclusive split gateways and subprocesses have nondeterministic behaviour. This condition ensures that when comparing the behaviour of two BPMN processes, every BPMN element from one BPMN process either is related to an element in the other BPMN process via refinement or is only reachable due to nondeterministic behaviour that is removed due to the refinement in the other BPMN process. This condition ensures that there are no hidden behaviour such as element C in Fig. 4 (2). Condition (b) is also appropriate: it is reasonable to compare behaviour of business collaborations if they have the same participants during compositional development. Participants in a business collaboration are typically decided a priori before designing and refining individual processes. These conditions do not reduce the practical expressiveness of BPMN. In particular we have validated the expressiveness via case studies in which we have constructed and verified complex business processes compositionally [11, Chapter 8].

In general, for any subprocess s, the CSP process that models s's behaviour can be generalised as $C[S]$ where S is the CSP process that models elements directly contained in s. Here $C[.]$ is a CSP process context that models the incoming, outgoing sequence flows and message flows of s. The following result shows that refinements are preserved from S to $C[S]$.

Theorem 2. *Given any subprocess s satisfying Conditions (a) and (b), and that its behaviour is modelled by CSP process $C[S]$, where S is the CSP process that models elements contained in s and $C[.]$ is a CSP process context that models s's sequence flows and message flows. Let t be any subprocess whose behaviour is modelled by CSP process $C[T]$ and T is the CSP process that models elements contained in t. If we have both $S \sqsubseteq_{\mathcal{F}} T$ and $C[S] \sqsubseteq_{\mathcal{F}} C[T]$, then we have $C[S'] \sqsubseteq_{\mathcal{F}}$*

$C[T']$ where S' and T' are the results of applying any one of the composition operations on S and T respectively.

A consequence of Theorems 1 and 2 is that refinement is preserved between a subprocess and the BPMN subprocess or pool that contains it. The following result lifts the composition operations to BPMN diagrams, and follows immediately from the fact that the CSP parallel operator $\|$ is monotonic with respect to refinements.

Corollary 1. *Given a BPMN process X that directly contains some subprocess s, such that X's behaviour is modelled by the CSP process $P(s) \,[\![\, \alpha P(s) \mid \alpha D \,]\!]\, D$, where D models the behaviour of other elements directly contained in X. For any s' such that $P(s) \sqsubseteq_{\mathcal{F}} P(s')$ we have $P(s) \,[\![\, \alpha P(s) \mid \alpha D \,]\!]\, D \sqsubseteq_{\mathcal{F}} P(s') \,[\![\, \alpha P(s') \mid \alpha D \,]\!]\, D$*

Monotonic operations preserve refinement-closed properties. For example, we previously formalized the notion of behavioural compatibility on BPMN pools [11,13] as a binary relation *compatible*(P, Q) on BPMN pools P and Q, such that *compatible*(P, Q) if and only if P and Q are deadlock free and their collaboration is also deadlock free. We further showed that *compatible* is a refinement-closed property. As a result, if we let G be one of the composition operations on *Pools*, if *compatible*$(G(P), G(Q))$, then for all $pToc(P) \sqsubseteq_{\mathcal{F}} pToc(P')$ and $pToc(Q) \sqsubseteq_{\mathcal{F}} pToc(Q')$, *compatible*$(G(P'), G(Q'))$. Furthermore, due to monotonicity, the equivalence $\equiv_{BPMN}$ defined in Definition 1 is a congruence with respect to the composition operations.

Corollary 2. *Assuming BPMN processes satisfy Conditions (a) and (b), the equivalence $\equiv_{BPMN}$ is a congruence with respect to composition operations.*

A congruence relationship allows one to substitute one part of a BPMN process with another that is semantically equivalent and obtain the same BPMN process.

Back to the running example. Fig. 5(1) shows an optimistic version of the customer business process. It is modelled by BPMN pool *OpCustomer*. After receiving an offer from the online shop, the customer eventually always accepts the offer. Equation 3 defines process *OCP* that models pool *OpCustomer*, where for all $i \in \{sCP, s3, s5, s6, s13\}$ process $P(i)$ is defined in Equation 1.

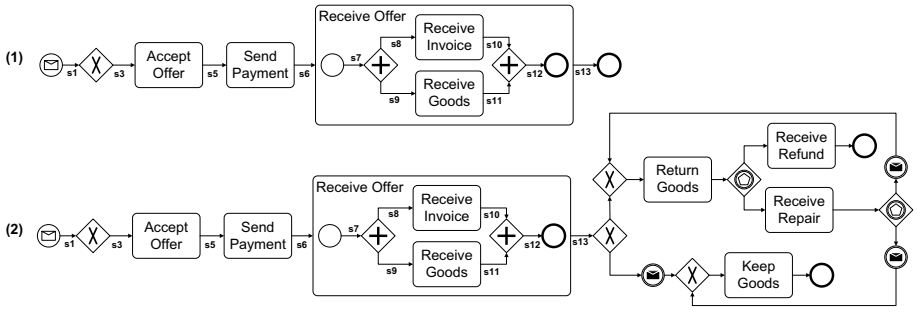

Fig. 5. Variants of the customer business process

$$P(s1) = (s.s1 \rightarrow \mathit{Skip} \fatsemi (s.s2 \rightarrow \mathit{Skip} \sqcap s.s3 \rightarrow \mathit{Skip}) \fatsemi P(s1)) \,\Box\, c.s14 \rightarrow \mathit{Skip}$$

$$OCP = \Big\| i : \{sCP, s1, s3, s5, s6, s13\} \bullet \alpha P(i) \setminus \{c.s4\} \circ P(i) \tag{3}$$

In fact the optimistic customer process is a refinement of the customer process; we verify this by checking the refinement $CP \sqsubseteq_{\mathcal{F}} OCP$ using FDR. Suppose we extend the customer's business process with the following return policy: "After receiving the goods and the invoice, the customer may decide to either keep the goods or return them for repair. Depending on the policy of the online shop, if the customer chooses to return her goods for repair, the shop may either provide a full refund, or repair the goods and deliver them back to the customer. After every repair, the customer has the choice to send the goods back again if further repairs are required.".

Fig. 5(2) shows the result of extending the optimistic customer business process with the above return policy. Here we observe that this extension can be constructed by a combination of operations *SeqComp*, *Split* and *EventSplit*, *Join* and *EventLoop*. Note that the same combination of operations can be applied to the original customer business process to model this return policy. If we let CP' be the resulting CSP process modelling the extended version of CP and OCP' be that of OCP, by Theorem 1, we have $CP' \sqsubseteq_{\mathcal{F}} OCP'$.

5 Related Work

Beside the work described in this paper and our earlier work [12], CSP has been applied to formalize other business process modelling languages. For example Yeung [15] mapped the Web Service Business Process Execution Language (WS-BPEL) and the Web Service Choreography Description Language (WS-CDL) to CSP to verify the interaction of BPEL processes against the WS-CDL description; his approach considers traces refinement and hence only safety properties. There have been attempts to formalize BPMN behaviour using existing formalisms (for example, Dijkman et al. [2]), which focus on the semantic definition of BPMN diagrams rather than their construction. Morale et al. [6,7] designed the Formal Composition Verification Approach (FVCA) framework based on an extended timed version of CSP to specify and verify BPMN diagrams. Similar to us, they achieve compositionality by considering the parallel combination of individual business process participants. However, their approach does not consider the semantics of constructing individual BPMN processes. To the best of our knowledge, Istoan's work [5] is the first attempt at defining composition operations on BPMN. He provided the composition operations to construct a BPMN process by composing two BPMN processes. He also provided these operations with a semantics in terms of Petri Nets [9]. His notion of refinement is that of functional extension while our work considered the reduction of nondeterminism [3]. Moreover, Istoan did not consider the semantic property of the composition operations to allow compositional development.

6 Summary

In this paper we introduced a set of composition operations to construct BPMN diagrams. We provided these operations a CSP semantics, and characterised the conditions under which these operations guarantee monotonicity with respect to $\mathcal{F}$. Refinement-closed properties, such as behavioural compatibility, are preserved by monotonic operations and these operations enable compositional development of business processes.

Acknowledgment. We wish to thank Jeremy Gibbons for suggesting improvements to this paper and Stephan Schroevers for proof reading it. We would like to thank anonymous referees for useful suggestions and comments.

References

1. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured programming. Academic Press Ltd. (1972)
2. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. In: Information and Software Technology (2008)
3. Eshuis, R., Fokkinga, M.M.: Comparing refinements for failure and bisimulation semantics. Fundamenta Informaticae 52(4)
4. Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual (1998), `http://www.fsel.com`
5. Istoan, P.: Defining Composition Operators for BPMN. In: Stinson, D.R., Tavares, S. (eds.) SAC 2000. LNCS, vol. 7306, Springer, Heidelberg (2012)
6. Mendoza, L.E., Capel, M.I.: Automatic Compositional Verification of Business Processes. In: Tse-Yun, F. (ed.) Parallel Processing. LNBIP, vol. 24, Springer, Heidelberg (2009)
7. Morales, L.E.M., Tuñón, M.I.C., Pérez, M.: A Formalization Proposal of Timed BPMN for Compositional Verification of Business Processes. In: Ng, E.W., Ehrig, H., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science and Biology. LNBIP, vol. 73, Springer, Heidelberg (2011)
8. O.: Business Process Modeling Notation, V1.1, Available Specification (Feb. 2008), `http://www.bpmn.org`
9. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Institut fur instrumentelle Mathematik, Bonn (1962)
10. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
11. Wong, P.Y.H.: Formalisations and Applications of Business Process Modelling Notation. DPhil thesis, University of Oxford (2011), Available at `http://ora.ox.ac.uk/objects/uuid:51f0aabc-d27a-4b56-b653-b0b23d75959c`
12. Wong, P.Y.H., Gibbons, J.: A Process Semantics for BPMN. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008)
13. Wong, P.Y.H., Gibbons, J.: Property Specifications for Workflow Modelling. Science of Computer Programming 76(10) (October 2011)
14. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1996)
15. Yeung, W.L.: Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In: Proceedings of 4th European Conference on Web Services (2006)

Building a Customizable Business-Process-as-a-Service Application with Current State-of-Practice

Fatih Gey, Stefan Walraven, Dimitri Van Landuyt, and Wouter Joosen

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
{fatih.gey, stefan.walraven, dimitri.vanlanduyt, wouter.joosen}@cs.kuleuven.be

Abstract. Application-level multi-tenancy is an increasingly prominent architectural pattern in Software-as-a-Service (SaaS) applications that enables multiple tenants (customers) to share common application functionality and resources among each other. This has the disadvantage that multi-tenant applications are often limited in terms of customizability: one application should fit the needs of all customers.
In this paper, we present our experiences with developing a multi-tenant SaaS document processing system using current state-of-practice workflow technologies from the JBoss family. We specifically focus on the customizability w.r.t. the different tenant-specific requirements, and the manageability of the tenant-specific customizations.
Our main experiences are threefold: (i) we were insufficiently able to modularize the activities and compositions that constitute the document processing workflow, (ii) we lacked support for describing tenant-level variations independently, and (iii) the workflow engine we employed is too centralized in terms of control, which limits resilience and thereby endangers scalability of the document processing application.

Keywords: Software-as-a-Service, Business Process, jBPM, Multi-tenancy, Customization, Document Processing

1 Introduction

Application-level multi-tenancy is an increasingly prominent architectural pattern in Software-as-a-Service (SaaS) applications. Different tenants are served simultaneously from the same run-time instance of the application while features of the application remain logically separated on a per-tenant basis. This suits best for applications where all potential tenants have highly similar (non-)functional requirements for the application. In case the tenant requirements differ slightly (or even profoundly), customization is required as an architectural feature to the SaaS application to facilitate efficient incorporation and management of tenant-specific requirements.

In the context of an ongoing project [1], we analysed a multi-tenant SaaS application for document processing of an industrial partner that currently serves a

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 113–127, 2013.

large amount of companies. Although their tenants differ in terms of specific requirements, they do share the common requirement of processing large volumes of documents and data through a multi-step processing scheme, e.g. after document generation additional processing steps such as signing may be required. In summary, the document processing represents a system for workflow-centric processing of batch jobs.

The application that is currently in use follows an ad-hoc software management approach: application functionality for document processing is available as reusable library functions. For each tenant, an individual application is created and executed. This approach suffers from being error-prone and not efficiently manageable (e.g. in case of changes to the document processing system).

In this paper, we present our experiences with the development of a customizable multi-tenant SaaS application that is configured by the tenant, whose workflow is run on top of JBoss' jBPM [2] and whose document processing facilities are modelled as Web services on top of JBoss AS7 [3]. As variability modelling and service variability is out of scope of this paper, we focus on the business process modelling (BPM) and execution aspects of our document processing system.

By showing that batch-oriented business processes with custom requirements (of a particular application domain) can be run as a SaaS application with manageable efforts on state-of-practice tools, we encourage companies with similar settings to migrate their workflow-driven application to a cloud platform. Research has already been performed in adjacent fields, such as feature-oriented domain analysis [4] (variability analysis) and multi-tenant customization [5] (middleware to enable variability in services), but has not been focussing on (practical) studies enlightening the business process aspect of customizable SaaS applications. We believe that this is one reason for low usage of the Cloud paradigm for companies of aforementioned types. Our concept envisions a set of pre-designed workflows provided by the SaaS application developer from which a tenant can simply select and configure the most suitable one and use the business processes execution on-demand as a Service (BPaaS). Simultaneously, by tackling the efficient manageability aspect of a workflow-driven, customizable multi-tenant SaaS application, we also motivate to operate the provider-side of such an application.

This paper is organized as follows: Section 2 introduces the document processing application and motivates the requirements of interest. Section 3 discusses our implementation while Section 4 provides an in-depth discussion of our key decisions and experiences from which we distil challenges and drawbacks that are relevant beyond the scope of this single implementation project. Section 5 discusses related work, and we conclude this paper in Section 6.

2 Problem Illustration and Motivation

In this section we first describe the document processing system that is currently in use by our industrial partner. Then, we highlight the drawbacks in terms

of manageability and summerize requirements for our implementation of the document processing system as a customizable multi-tenant SaaS application.

2.1 Document Processing System

The system of interest in this paper is that of a Belgian SaaS provider. This company, hereafter referred-to as Document Processor (DP), provides a platform for generating, signing, printing, delivering, storing, and archiving documents, and they offer these B2B facilities as a Software-as-a-Service (SaaS) application to their customers (tenants). As a result of adhering to multi-tenancy at the SaaS paradigm, the document processing system is difficult to customize: the benefits of scale inherent to SaaS rely on the fact that the same application can be reused by many different tenants. Nonetheless, as the processing facilities are of relevance to a wide range of companies in very different application domains, several tenant-level variabilities and customizations exist. To illustrate, we present two such tenant companies and their document processing requirements.

$TenantA$ is a temporary employment agency which requires printed payslips to be delivered to its employees. It provides the raw data to DP, with metadata attached to each document. $TenantB$ is in the financial business and uses the document processing facilities for generating invoices and distributing them to their customers (end users). $TenantB$ provides only raw data as input and requires its custom layout to be applied to the documents generated (for branding purposes) and the distribution of documents depending on the end-user's preference (email and printed paper).

2.2 Challenges

In their current document processing offering, the document processing provider uses a set of functional libraries to realize the superset of document processing activities. As each document is processed by a sequence of these activities, the processing logic is realized in the form of Java code in which these libraries are called sequentially. To realize a tenant-specific customization of the application, a variation of this processing logic is created manually – by copy-pasting the existing Java code and making the tenant variations manually.

This approach has several obvious drawbacks: (1) There is no systematic reuse of customization knowledge, and techniques such as copy-paste are error-prone. Moreover, the management complexity of these different variants grows exponentially with the number of supported tenant variations. (2) Because the workflow logic is currently written in a programming language (Java), application administrators are required to be developers skilled in that language in order to set-up new tenants. (3) Whenever the libraries change, these changes ripple through to the different workflow definitions: they needs to be changed manually which does not scale for large number of tenants.

In this paper, we report on our experiences of migrating the existing document processing application to state-of-practice workflow processing techniques (from the JBoss family), and this obviously in the context of multi-tenant SaaS

applications. Specifically, we focused on addressing the key requirements listed below:

- **Manageability of Variations.** In order to remain competitive, the time-to-market of a specific tenant variant is of crucial importance. Therefore, adding new tenants (tenant acquisition), changing tenant configurations, extending tenant variabilities, or modifying the interfaces to the document processing activities have to be more efficient, and the configuration process itself less error-prone.
  Furthermore, the tooling should be suitable to be used by business analysts and domain experts, rather than by developers and programmers. As document processing workflows consist of a set of pre-existing activities out of which a particular sequence is defined, the tool is not required to nor should provide the expressiveness of a general-purpose programming language.
- **Resilience of Workflow Execution.** It is especially important for SaaS applications in a distributed setup, such as for our document processing system, that the workflow execution is resilient against failures of remote services, as failure of nodes is likely, and Service Level Agreements (SLAs) in SaaS contexts tend to approach maximum utilization of resources so that such failures may have severe impact on the fulfilment of SLAs.

Section 3 discusses the relevant implementation decisions. Subsequently, Section 4 provides an in-depth discussion of our main experiences and findings.

3 Implementation

In this section, we describe the implementation of a customizable multi-tenant SaaS application for document processing that we have built to address the challenges discussed in Section 2.1. In line with the scope of this paper, we focus on the business process modelling aspect that we have implemented using the business modelling language *Business Process Modelling and Notation (BPMN)* and its state-of-practice execution engine and modelling tool jBPM.

First we define – for the sake of clarity – the common terminology that is used in the context of jBPM and that we use to describe our experience with the implementation. We then give a short overview of the end-to-end application before discussing its business processing modelling aspects.

3.1 Terminology

A *workflow* is a sequence of (business) *activities*. The persistent artifact in which a workflow is defined, e.g. using BPMN, is a *process definition*. For each execution of a workflow a run-time instance of the process definition, a so called *process instance* is created. The implementation of an activity is called a *task*. Each process instance can have *process instance variables* that may have been set at process instance's creation time and are accessible from within the tasks.

3.2 Overview of the End-to-End Application

The application we illustrate in this section processes document (processing) jobs which are uploaded to the system. Such a job contains a set of input files (either ready-to-distribute documents or raw data for document generation), meta-data for each input-file, and a tenant-ID.

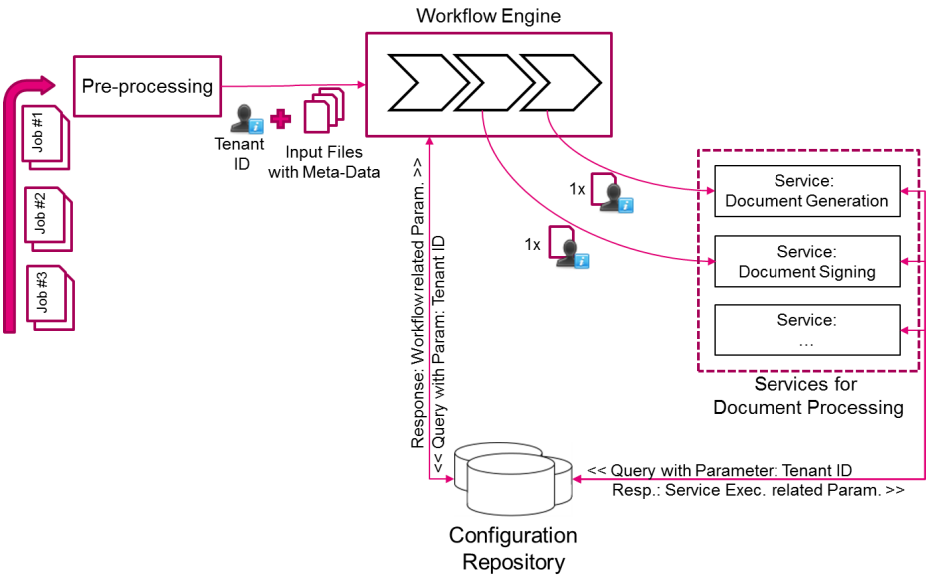

Fig. 1. High-level architecture of the end-to-end application

Figure 1 shows the overall architecture of the document processing SaaS application. A document job is received at the pre-processing component and passed on to the workflow engine. The workflow engine uses the tenant-ID of that job to fetch the corresponding workflow-related tenant-specific configuration from the central configuration repository. For example, tenant A's workflow is configured such that no document generation is executed, but that the input documents should be printed and distributed via postal mail. For each activity of the on-going workflow, e.g. document distribution for tenant A, the corresponding service is called.

Each service fetches its configuration using the tenant-ID, e.g. the template to use for printing tenant A's documents.

3.3 Business Process Modelling

The main business logic of the document processing case is modelled in two different workflows: the outer workflow is represented in Figure 2 and embeds

the inner workflow represented in Figure 3. The outer workflow iterates over the input files of the uploaded document job and invokes the inner workflow for each individual document.

Starting with the Start Event (circle labelled with "S" at the left-top side of Figure 3), the graph depicts the sequence of workflow activities that is run. Activities that are optional have a parallel edge connecting their predecessor with the successors of that activity. Alternative activities are placed as parallel paths to the actual activity. For both variations, `XOR`-typed gateways are used which will proceed the workflow by selecting one of the available outgoing edges depending on dynamically-evaluated code, which we call *switch code*. A switch code may be written in Java code or Drools Rules (a domain-specific language of JBoss for workflows). In addition, gateways of type `AND` are used which result in executing all out-going paths.

As mentioned in Section 3.2, each of the activities of this process definition, when triggered, executes a service call to the document processing services passing the document that is currently being processed and its related data.

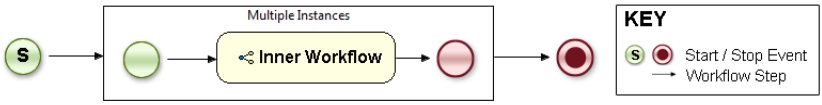

Fig. 2. Process Definition for Document Processing: Outer Workflow

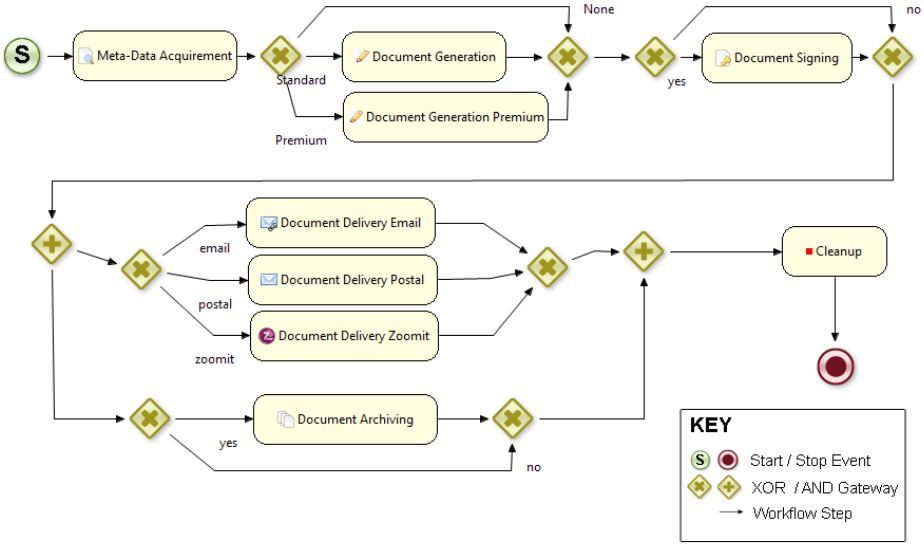

Fig. 3. Process Definition for Document Processing: Inner Workflow

Variability Modelling. As mentioned earlier, `XOR` gateways are used to express optional or alternative activities in the document processing workflow. More specifically, we use them to express tenant-specific variability in the sense that the inner workflow, as shown in Figure 3, depicts the workflow with all tenant-specific variants *included*. Hence, we call this type of artifact a *multi-tenant process definition*.

At the instantiation of a workflow execution (process instance), the tenant-specific parameters required to customize the workflow's multi-tenant process definition at run time are fetched. For tenant A, these parameters are *no document generation* and *delivery method is postal*, and for tenant B the parameters are *document generation method using custom templates* and *delivery method is postal* (or *delivery method is e-mail*, respectively). Those parameters are set as process instance variables which are accessible to all gateways and tasks within the multi-tenant process definition. The aforementioned `XOR` gateways read these parameters and select the tenant-specific options accordingly. For example, for tenant A, the gateway selecting between the delivery methods will read instance variable for tenant A and will select the *postal* option. As a result, workflow executions for each tenant follow only one path from start to finish of a workflow. After a workflow for tenant A has been initialized, no other delivery method than postal delivery is available for the duration of that process instance.

Passing Variables between Tasks and into an Iteration Activity. The BPMN language provides two options for a task to retrieve data: (1) Process instance variables which are variables in the scope of a process instance, i.e. each task can access those, and (2) parameter mapping, a mechanism that can map process instance variables to input parameters of a task or output data from a task to a process instance variable.

Using option 1, all processes would be able to read and write to a scoped global variable space which would limit modularity. In the current version of BPMN, option 2 is limited to map a variable's content to another variable or vice versa, without providing the ability to map a member of a variable (assuming it is an object) to another variable. As a result, using options 2, tasks are expected to know which data are required their successor tasks in order to provide those in separate variables. That is, in case the set of tasks changes, programmatic changes on the tasks are becoming necessary in order to reflect on the changed set of output variables that this task has to fill in.

Option 2 also causes another issue when considering a workflow with iterations. As described earlier (cf. Section 3.3), the outer workflow of the document processing system is triggered with a set of input documents (and per-document meta-data) over which it iterates, calling the inner workflow for each document. A consequence of that architecture is that the document processing system requires the iteration activity of the outer workflow to pass multiple variables (document and meta-data) to the inner workflow, which is not supported by the current version of BPMN. Using option 2, the members of that single variable that is passed into each iteration cannot be mapped to the according parameters within the inner workflow.

As a workaround, we decided to introduce a composite data structure that provides (i) members to store the input document *and* meta-data, and (ii) read- and write-access for additional information. The composite data structure is stored as process instance variable and is accessible by each task (as in option 1). It is used to pass partial results, e.g. the document in question in its (potentially intermediate) current state after each activity, among other book-keeping information, such as a list of so-far completed activities, between tasks. For iterations, it is used to reduce the amount of variables that are required within the inner workflow to one.

4 Discussion

This section discusses our experience and main findings with our implementation (presented in the previous sections) with regard to the requirements we set up for our document processing application (cf. Section 2.2).

4.1 Manageability of Variations

Our findings related to manageability are twofold: (1) Using BPMN to model the document processing workflows, we were forced to create a custom data-structure per multi-tenant process definition and introduce a structural dependency between that data structure and the tasks which decreases reusability of said tasks across process definitions. (2) The Lack of explicit support for multi-tenant customization in BPMN (a) increases the need to add or modify a tenant configuration redundantly at multiple places, limiting the modularity of that configuration, and (b) limits potential future tool support for tenant-specific configuration management. Next, we will elaborate on these findings in detail.

Structural Dependency of Tasks. As discussed in Section 3.3, we have created a composite data structure as a workaround, that is used to pass input document and its meta-data between tasks, because we experienced that BPMN's techniques for passing parameters were not sufficient to realize the requirements set by our document processing application. In order to enable all tenants to read from and write to this data structure, additional structural dependencies between all tasks in our document processing workflow and the common data structure were introduced, i.e. all tasks use (the same) implied knowledge about the common data structure. This workaround is the result of a trade-off in reusability. On the one hand, by introducing this composite data structure and the dependency to the tasks of the multi-tenant process definition, we ensure that the process definition can be efficiently and easily (re-)assembled using existing tasks and graphical tools. On the other hand, in case an additional process definition becomes necessary, tasks of the one process definition cannot be used in the other, as they may rely on different composite data structures for their inter-task communications. For example in the document processing system, tenants with relatively similar requirements are clustered together (tenant

A and tenant B belonging to the same cluster). If however, new tenants show up with very different requirements (thus, belonging to a different cluster), the overall management effort is effectively lower when separating the two clusters in separate process definitions.

Ideally, this problem is addressed at the level of the BPMN language. By supporting the operator to access member variables (in Java, that is the dot operator), the parameter mapping feature, which is configurable using jBPM's graphical tools, *could* be used to pass parameters betweens tasks without the need of additional data structures. Hence, the dependency between tasks in a process definition could be easily managed using graphical tools (to configure the parameter mapping feature) rather than changing the program code of task to comply to the additional data structures.

Explicit Support in BPMN for Tenant-specific Variations. We have observed that BPMN does not explicitly support tenant-specific variations. For the implementation of the document processing application, we therefore borrowed other features of that language to realize the desired level of variability, namely gateways with Java as switch code.

This workaround has two drawbacks: (1) the knowledge about tenant-specific variations for each activity in the document processing workflow is defined within the tenant-specific configuration. As with our implementation, the same knowledge is used when creating the multi-tenant process definition which is a manual process. Thus, our current workaround limits modularity and requires an error-prone manual process. (2) We use the BPMN language item *gateway* to express tenant-specific rather than business-process-driven variability for which it is meant to be used. Therefore, these two semantics become harder to distinguish. As a result, potential tool support for tenant-specific management may be limited.

Note, however, that the lack of explicit support for tenant-specific variations does not affect the manageability of workflow definitions. Placing all tenant-specific variants into a single multi-tenant process definition, i.e. using branches, increases its overall size, and may seem as a bottleneck for (change-)management at first. But, as BPMN supports the partition of workflows into sub-workflows, the size of process definition has no big impact on its practicality in management per-se.

In order to tackle the aforementioned two issues, we envision an extension to BPMN that provides explicit support for tenant-specific variations by introducing two elements. One, an activity that is subject to tenant-specific alternatives should be modelled as variation point[1]. Two, the workflow engine should provide mechanisms to import knowledge about variation points, such as a feature model, and variants from an external source. In our document processing system, this would be the configuration repository. Similar suggestions have been made for the BPEL language but have not been shown in a proof-of-concept implementation, yet [7].

[1] We use the terms *Variation Point* and *Variant* as it is defined in [6].

4.2 Resilience of jBPM's Workflow Execution

The workflow engine jBPM, which we used for our implementation, executes workflows employing dedicated control over the process instance. That is, at the time an execution is triggered, the process definition is loaded into memory. Thereafter, programmatical access to a specific process instance from outside the process instance is very limited, and especially an update of the process definition is not possible.

In addition, for our case, no state during the entire workflow execution is persisted[2]. Technical failures are not considered in the modelling concepts of BPMN. Although jBPM offers technical exception handling[3], it is intended to only run additional procedures in case of exceptions and has no effect on the execution sequence.

Potential Types of Failures. The described properties above lead to following three potential failures of the workflow execution which we will discuss subsequently: (1) A process instance may continue processing on the basis of an out-dated process definition that may lead to task failures or, even worse, to incorrect results of the process. (2) In case a task fails, the entire workflow has to be executed all over. (3) In case the workflow engine crashes, the entire workflow has to be re-run.

The first type of failure can occur when workflow tasks change their scope of activity and, as a result, also the sequence in which the workflow requires to be executed. Example: Assume that a task that was supposed to create and send an e-mail is split into two tasks, one for creating an e-mail, i.e. HTML formatting, BASE64 encoding, etc., and the other for sending the e-mail (talking SMTP with a server). Obviously, process definitions that included this task need to be updated accordingly. Without the ability to update process instances during their execution, all process instances that include that task but have not executed it yet will fail or produce incorrect results.

The second and the third behaviour basically refer to the same issue: In case a task fails, the enclosing workflow is restarted from the beginning. As a result, documents are reprocessed not because of a business process reason, e.g. the document at hand is an exceptional case or contains errors, but purely because of a technical reason. Because we modelled the workflow to process an

[2] The jBPM workflow engine persists workflow state only at so called safe-points. These are phases in which the workflow engine has no further immediate tasks to execute and is waiting for workflow events to continue. For non-interactive workflows that contains only a sequence of subsequent activities, such as the document processing application, no persistence of workflow state is applied during the entire execution.

[3] jBPM distinguishes between two kinds of exceptions: logic and technical exceptions. While logical exceptions refer to exceptional cases in the business logic, e.g. when escalation to the next business hierarchy level is required, technical exceptions can be mapped the exception handling concept found in Java and other programming languages.

entire document processing job (multiple documents) at once, the overhead of a service failure is even higher as already successfully processed documents would be processed again.

This shortcoming is related to jBPM's focus of failure-recovery. It is best suited for situations in which the workflow-engine (or the underlying infrastructure) fails especially when waiting for a particular event to resume the according process instance. This can be a long period when interactive tasks are involved. These phases in which the workflow engine is waiting are called *safe-points*. For our non-interactive and not event-driven workflow, the safe-points are located before the workflow execution has started and after its completion. Thus, our implementation using jBPM makes failures of single tasks expensive[4], as the entire workflow needs to be repeated.

Task Failures in the Context of Distributed SaaS Applications. In the presence of failures with expensive consequences, attention should be paid to the fact that a distributed multi-tenant SaaS application risks multiple natural error sources that may lead activities to fail: First, every distributed system inherently lacks control over the remote machine's state and suffers from occasional data omissions due to network failures. Second, the fact that the benefit from economies-of-scale is a dominant motivation to operate an application on a cloud platform implies that the application is intended to be operated under continuous load. In our case, load refers to document processing jobs that have a SLA-committed completion dates. Thus, large delays in workflow execution are not tolerable from a business perspective.

Conclusion. Therefore, we identify the gap in our document processing SaaS application that it lacks of support for inexpensive failures of tasks. In future work, we plan to elaborate further on safe points that occur between each workflow-step (activity) rendering a process definition to be executed as a set of tasks. Moreover, by removing the centralized control that spans the entire workflow execution and supporting the execution of individual tasks of a workflow from independent workflow engine instances, concurrent execution of semantically parallelizable tasks within a workflow could be enabled. Furthermore, in case a task execution fails, the aforementioned safe points can depict process instance states to resume at when restarting the process instance. In addition, updating the process definition of a running process instance would become simpler, as after each activity the process instance would be in a (persisted) quiescence [8] state and before each activity the process definition is re-read.

Building up on these features, task failures would be less expensive (resume instead of start over) and could therefore be accepted as a planned behaviour of the system and incorporated into the SLA-targeting scheduling strategies. As a result, changes to and failures of the system would be less harmful, and

[4] *Cost* can have multiple dimensions: operational costs, duration (endangering SLA fulfillment) or damage of brand (sending invoices twice and thereby communicating technical error to customers)

the scalability in performance and management overhead (i.e. for re-allocating performance schedules) would benefit significantly.

5 Related Work

Manageability for Business Processes. Modularity is a key concept to support manageability through reuse. Research has been executed to increase the modularity in business process definitions. Geebelen et. al [9] proposed a pre-processing layer for the BPEL standard, that uses a set of concrete parameters to transform a parameterized process definition template into an actual BPEL process definition that can be executed on ordinary BPEL engines. Charfi et. al [10] use aspect orientation to modularize the definition of activities within and across business processes, e.g. an activity that always has to precede another activity can be defined in modularized way. Isotan et. al [11] propose to add composition operators to BPMN in order to facilitate the composition of smaller and reusable definition units into full process definitions. In their work, they are formally modelling operators based on Petri Nets.

In contrast, our document processing application is designed to be operated as SaaS application and, thereby, has a different set of requirements for manageability: We address one application domain at a time by creating a single process definition including all anticipated variabilities. Not dealing with a large amount of separate process definitions, our context benefits less from the kind of modularity that is presented in the related work. We rather lack of reusability of tasks across process definitions, as elaborated in Section 4.

Multi-Tenancy for Business Processes. Pathirage et. al [12] address multi-tenancy mostly at the infrastructure level. They provide a platform on top of which a BPEL engine can be run and that maintains a tenant context during the entire workflow execution.

However, it does not take customization of workflows into account, i.e. all tenants operate on the same workflow, while we focus on workflow customization as well as multi-tenancy.

Variability for Business Processes. The work of Mietzner et. al [13] focus mainly on modelling variability and providing deployment support in that it will choose the set of required components optimizing for the lowest operational costs.

While they present rather generic concepts for modelling workflow variability, our work is based on a practical experience with a concrete state-of-practice framework from which we extract further challenges. We also focus on the business process modelling aspect that is not enlightened in their work.

Geebelen et. al present in a later work [14] a framework for run-time adaptation of workflow processes. They use an application-domain-dependent workflow template at which concrete service calls are weaved in at run-time depending on an external policy engine. Also rollbacks to previous workflow activities are provided.

Even though, they provide manageable flexibility, their scope of workflow customization differs from ours. While they provide a fixed sequence of activities and flexibility in choosing the service to execute an activity, we offer alternative sequences based on tenant-specific requirements.

VxBPEL [7] is an extension to BPEL that explicitly adds alternative task implementations to an activity in the process definition. It is motivated that the knowledge about variations (1) should be obtained from an external source and (2) should be injectable into on-going workflow executions. The implementation of that work does not show those motivated proposals.

In their proposals, the authors argue similarly to us. Yet, we differ in the fact that we use business process modelling for non-interactive batch-processing and variability in the context of multi-tenancy, while they argue on basis of interactive application and introduce variability to achieve higher Quality-of-Service. Furthermore, we use the state-of-practice technology jBPM without modifications in order to comply with cloud providers as well as with existing applications and tools.

Resilience of Workflow Execution. In Section 4.2, we described the kind of resilience for workflow executions that is required for the document processing application, i.e. a task execution failure should not cause the enclosing workflow to be reset to the beginning.

Leymann et. al [15] describe a workflow management system that is based on multiple message queues. They claim that their system, persisting state information about each invoked task, is "forward recoverable". While their system is situated in a local environment with (remote) interactive clients, our context is a non-interactive workflow as distributed SaaS application.

The work of Yu et. al [16] proposes to process BPEL workflows without a central execution engine. One of its key goals is to enable dynamically composed workflows which also addresses changes in the task execution sequence in the presence of service failures. They make use of continuations which are persisted after each task execution of the workflow and that can be picked-up by different workflow engines for continuing execution, and extend the BPEL execution engine. We, the other hand, use state-of-practice tools to model and execute workflows.

6 Conclusion

We presented our experiences with the implementation of a customizable multi-tenant SaaS application for document processing. We discussed the key requirements for this application: multi-tenancy and Software-as-a-Service on the one hand, and customizability to tenant-specific requirements on the other hand. To guarantee the practical relevance of our findings, we addressed these requirements in the context of state-of-practice technologies from the JBoss family. Specifically, we employed Business Process Modelling and Notation (BPMN) language to model tenant-specific customizable business processes of the document processing system and jBPM for their execution.

Our findings can be summarized as follows. First, because of the parameter-passing mechanism that is currently provided in BPMN, it is hard to design individual tasks in a modular manner so that they can be reused across business process definitions. Our second finding is a consequence of the fact that BPMN lacks explicit support for multi-tenancy and variability. We introduce workflow branches to express tenant-specific instead of business-case variabilities as a workaround. Ideally, the workflow modelling language should offer support to describe these tenant-level variabilities explicitly. In our third finding, we have argued that using a centralized run-time instance to control the entire workflow may not provide the necessary resilience in execution, because of the high costs related to recover from task failure. It may therefore only be partially suited for SaaS environments where different kinds of faults are likely to occur regularly. Their occurrence may endanger SLA commitments and thereby limit scalability of the application.

Customization will gain importance in multi-tenant Software-as-a-Service applications as it enables the SaaS provider to fine-tune his offerings to specific tenants without losing the benefits of scale inherent to SaaS. Not only applications and services, but also the composition of those, i.e. a workflow-driven application, need to support customization to facilitate the migration of legacy applications to the cloud.

Acknowledgments This research is partially funded by the Research Fund KU Leuven and by the iMinds project CUSTOMSS. The iMinds CUSTOMSS is a project co-funded by iMinds (Interdisciplinary institute for Technology) a research institute founded by the Flemish Government. Companies and organizations involved in the project are AGFA Healthcare, IBCN/INTEC-UGent, Televic Healthcare, and UnifiedPost.

References

1. iMinds CUSTOMSS Project Consortium: iMinds CUSTOMSS Project (2013), `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=CUSTOMSS`
2. The jBPM Team of the JBoss Community: jBPM (2013), `http://www.jboss.org/jbpm`
3. The JBoss Community: JBoss AS7 (2013), `http://www.jboss.org/as7`
4. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document (1990)
5. Walraven, S., Truyen, E., Joosen, W.: A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 370–389. Springer, Heidelberg (2011)
6. Chang, S.H., Kim, S.D.: A Variability Modeling Method for Adaptable Services in Service-Oriented Computing. In: 11th International Software Product Line Conference, 2007. SPLC 2007, pp. 261–268 (2007)
7. Koning, M., ai˜Sun, C., Sinnema, M., Avgeriou, P.: Vxbpel: Supporting variability for web services in bpel. Information and Software Technology 51, 258–269 (2009)

8. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16, 1293–1306 (1990)
9. Geebelen, K., Michiels, S., Joosen, W.: Dynamic reconfiguration using template based web service composition. In: Proceedings of the 3rd workshop on Middleware for service oriented computing. MW4SOC '08, pp. 49–54. ACM Press, New York (2008)
10. Charfi, A., Awasthi, P.: Aspect-oriented web service composition with AO4BPEL. In (LJ) Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 168–182. Springer, Heidelberg (2004)
11. Istoan, P.: Defining composition operators for bpmn. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 17–34. Springer, Heidelberg (2012)
12. Pathirage, M., Perera, S., Kumara, I., Weerawarana, S.: A multi-tenant architecture for business process executions. In: IEEE International Conference on Web Services (ICWS), pp. 121–128 (2011)
13. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems. PESOS '09, Washington, DC, USA, pp. 18–25. IEEE Computer Society Press, Los Alamitos (2009)
14. Geebelen, K., Kulikowski, E., Truyen, E., Joosen, W.: A mvc framework for policy-based adaptation of workflow processes: A case study on confidentiality. In: 2010 IEEE International Conference on Web Services (ICWS), pp. 401–408 (2010)
15. Leymnn, F., Roller, D.: Building a robust workflow management system with persistent queues and stored procedures. In: 14th International Conference on Data Engineering, Proceedings, pp. 254–258 (1998)
16. Yu, W.: Running BPEL Processes without Central Engines 1, 224 (2007)

Verifying Data Independent Programs Using Game Semantics

Aleksandar S. Dimovski

Faculty of Information-Communication Tech., FON University, Macedonia

Abstract. We address the problem of verification of program terms parameterized by a data type X, such that the only operations involving X a program can perform are to input, output, and assign values of type X, as well as to test for equality such values. Such terms are said to be data independent with respect to X. Logical relations for game semantics of terms are defined, and it is shown that the Basic Lemma holds for them. This proves that terms are predicatively parametrically polymorphic, and it provides threshold collections, i.e. sufficiently large finite interpretations of X, for the problem of verification of observational-equivalence, approximation, and safety of parameterized terms for all interpretations of X. In this way we can verify terms with data independent infinite integer types. The practicality of the approach is evaluated on several examples.

1 Introduction

In this paper we study predicative parametric polymorphism in the setting of game semantics, and its applications to parameterized verification. In order to keep the presentation focussed, we work with Idealized Algol (IA) [1], an expressive programming language combining imperative features, locally-scoped variables and (call-by-name) higher-order functions.

Parametric polymorphism is the notion of treating data from a range of types in a uniform fashion. Its predicative version allows types to be formed from type variables, while its impredicative version is more general and allows universal quantification of type variables. We achieve predicative parametric polymorphism by extending our language with free data type variables X on which only the equality operation is available. Thus, any program term makes sense if any data type is instantiated for X, i.e. terms become parameterized by X. We will want to verify observational-equivalence, approximation, and safety of predicatively parametrically polymorphic terms.

We obtain results which provide threshold collections, i.e. sufficient finite interpretations of the data type variable X, such that if a property holds/fails for those interpretations, then it holds/fails for all interpretations which assign larger sets to the parameter X. Considering the case when an infinite integer type is substituted for X, we obtain a procedure to perform verification of terms which contain infinite integers completely automatically. This is done by replacing integers with threshold collections, i.e. appropriate small finite data types.

W. Binder, E. Bodden, and W. Löwe (Eds.): SC 2013, LNCS 8088, pp. 128–143, 2013.

A useful tool for enabling parameterized verification as above are logical relations [13,14,16]. A logical relation of a language is an assignment of relations to all types such that, the relation for any type is obtained from the relations assigned to data types and type variables, by induction on the structure of types. We will define logical relations for game semantics by lifting the relations on data values through all the constructs of game semantics, in such a way that the Basic Lemma holds for them. It states that logical relations are preserved by game semantics of terms. This proves that the language we consider is parametrically polymorphic, i.e. any term behaves uniformly for different instances of its parameter. The Basic Lemma will be applied to parameterized verification, since it provides direct ways of relating various game semantics interpretations of parameterized terms.

Game semantics [1,2] is a method for compositional modeling of programming languages, which constructs models of terms (open programs) by looking at the ways in which a term can observably interact with its environment. Types are interpreted by *games* (or arenas) between a Player, which represents the term being modelled, and an Opponent, which represents the environment in which the term is used, while terms are interpreted by *strategies* on games. Game semantics is compositional, i.e. defined recursively on the syntax, therefore the model of a larger term is constructed from the models of its constituting subterms, using a notion of strategy composition. Another important feature of this method, also known as external compositionality, is that there is a model for any term-in-context (open program) with undefined identifiers, such as calls to library functions. These two features are essential for achieving modular analysis of larger terms. The model obtained by game semantics is *fully abstract*, which means that it is both sound and complete with respect to observational equivalence of programs, and so it is the most accurate model we can use for a programming language. Although this model is precise, it is complicated and so equivalence and a range of properties are not decidable within it. However, it has been shown that for several language fragments with finite data types, the model can be given certain kinds of concrete automata-theoretic representations [4,5,6,12]. This gives a decision procedure for a range of verification problems, such as observational-equivalence, approximation, safety, and others, to be solved algorithmically.

The paper is organised as follows. Section 2 introduces the language considered in this paper, and its game semantics is defined in Section 3. Logical relations for game semantics are presented in Section 4. Several theorems which provide support for parameterized verification are shown in Section 5. The practicality of this approach is demonstrated in Section 6. In Section 7, we conclude and discuss possible extensions.

Related Work. Predicative parametric polymorphism is known as data independence in the setting of concurrent reactive systems. Some practically important examples of such systems are communication protocols, memory systems, and security protocols. The literature contains efficient algorithms for deciding the parameterized verification problem for data independent systems [10,11].

System F is a typical example of impredicative parametric polymorphism. It is also known as the polymorphic or second-order λ-calculus, and it works with pure type theories, with no notion of ground data types. Game semantics for System F are given in [8,9].

2 Programming Language

Idealized Algol (IA) [1,2] is a simply-typed call-by-name λ-calculus with the fundamental imperative features and locally-scoped variables. We extend IA with predicative parametric polymorphism by allowing data type variables X.

The data types D are finite integers ($\mathsf{int}_n = \{0, \ldots, n-1\}$), booleans, and a data type variable X ($D ::= \mathsf{int}_n \mid \mathsf{bool} \mid X$). The phrase types consists of base types: expressions, commands, variables ($B ::= \mathsf{exp}D \mid \mathsf{var}D \mid \mathsf{com}$) and function types ($T ::= B \mid T \to T$).

Terms of the language are the standard functional constructs for function definition ($\lambda\, x : T.M$) and application (MN) as well as recursion (YM). Expression constants are integers (n), booleans (tt, ff), and data values from the set W which interprets X ($w \in W$). The usual arithmetic-logic operations are employed ($M \,\mathsf{op}\, N$), but equality is the only operation available on X-expressions. We have the usual imperative constructs: sequential composition ($M; N$), conditional ($\mathsf{if}\, M\, \mathsf{then}\, N\, \mathsf{else}\, N'$), iteration ($\mathsf{while}\, M \,\mathsf{do}\, N$), assignment ($M := N$), de-referencing ($!M$), "do nothing" command skip, and $\mathsf{diverge}$ which represents an infinite loop (divergence). Block-allocated local variables are introduced by a *new* construct ($\mathsf{new}_D\, x := v\, \mathsf{in}\, M$), which initializes a variable and makes it local to a given block. The constructor $\mathsf{mkvar}_D MN$ is used for creating "bad" variables.

Well-typed terms are given by typing judgements of the form $\Gamma \vdash_W M : T$, where Γ is a type *context* consisting of a finite number of typed free identifiers, and W is a set of data values used to interpret X, which are allowed to occur in M as expression constants. When it does not cause ambiguity, we may write only $\Gamma \vdash M : T$. Typing rules of the language are those of IA (e.g. [1,2]), where the rules for arithmetic-logic operations are:

$$\frac{\Gamma \vdash_W M : \mathsf{exp}D \qquad \Gamma \vdash_W N : \mathsf{exp}D}{\Gamma \vdash_W M \,\mathsf{op}\, N : \mathsf{exp}D'} \qquad \frac{\Gamma \vdash_W M : \mathsf{exp}X \qquad \Gamma \vdash_W N : \mathsf{exp}X}{\Gamma \vdash_W M = N : \mathsf{expbool}}$$

where $D, D' \in \{\mathsf{int_n}, \mathsf{bool}\}$, and $\mathsf{op} \in \{+, -, *, /, =, \neq, <, >, \wedge, \vee, \neg\}$. For such terms we say that are *data independent* with respect to the data type X.

Any well-typed term can contain equality tests between values of X. We define a condition on terms, which does not allow any equality tests between values of X. A term $\Gamma \vdash_W M : T$ satisfies $\mathbf{(NoEq}_X\mathbf{)}$ condition if for any equality operation $\Gamma' \vdash_W N = N'$ within M, X does not occur in the types of N and N', i.e. $\Gamma' \vdash_W N, N' : \mathsf{exp}\{\mathsf{int_n}, \mathsf{bool}\}$.

The operational semantics of our language is given for terms $\Gamma \vdash_W M : T$, such that all identifiers in Γ are variables, i.e. $\Gamma = x_1 : \mathsf{var}D_1, \ldots, x_k : \mathsf{var}D_k$. It is defined by a big-step reduction relation: $\Gamma \vdash_W M, \mathrm{s} \Longrightarrow V, \mathrm{s}'$, where s, s$'$

represent Γ-*states* before and after reduction. A Γ-state s is a (partial) function assigning data values to the variables $\{x_1, \ldots, x_k\}$. We denote by V terms in *canonical form* defined by $V ::= x \mid v \mid \lambda\, x : T.M \mid \mathsf{skip} \mid \mathsf{mkvar}_D MN$. Reduction rules are those of IA [1,2].

Given a term $\Gamma \vdash_W M : \mathsf{com}$, where all identifiers in Γ are variables, we say that M *terminates* in state s, if $\Gamma \vdash_W M, \mathrm{s} \Longrightarrow \mathsf{skip}, \mathrm{s}'$ for some state s'. Then, we say that a term $\Gamma \vdash_W M : T$ is an *approximation* of a term $\Gamma \vdash_W N : T$, denoted by $\Gamma \vdash_W M \sqsubseteq N$, if and only if for any term-with-hole [1] $C[-] : \mathsf{com}$, such that both $C[M]$ and $C[N]$ are well-typed terms of type com, if $C[M]$ terminates then $C[N]$ terminates. If two terms approximate each other they are considered *observationally-equivalent*, denoted by $\Gamma \vdash_W M \cong N$.

3 Game Semantics

We now give a brief description of game semantics for IA extended with predicative parametric polymorphism. A more detailed presentation of game semantics for IA can be found in [1,2].

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$, where M_A is a countable set of *moves*, $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ is a labeling function which indicates whether a move is by *Opponent* (O) or *Player* (P), and whether it is a *question* (Q) or an *answer* (A). Then, $\vdash_A$ is a binary relation between $M_A + \{*\}$ $(* \notin M_A)$ and M_A, called *enabling* (if $m \vdash_A n$ we say that m enables move n), which satisfies the following conditions: (i) Initial moves (a move enabled by $*$ is called *initial*) are Opponent questions, and they are not enabled by any other moves besides $*$; (ii) Answer moves can only be enabled by question moves; (iii) Two participants always enable each others moves, never their own.

We denote the set of all initial moves in A as I_A. The simplest arena is the empty arena $I = \langle \emptyset, \emptyset, \emptyset \rangle$. Given arenas A and B, we define new arenas $A \times B$, $A \Rightarrow B$ as follows:

$$A \times B = \langle\, M_A + M_B,\, [\lambda_A, \lambda_B],\, \vdash_A + \vdash_B \,\rangle$$
$$A \Rightarrow B = \langle\, M_A + M_B,\, [\overline{\lambda}_A, \lambda_B],\, \vdash_B + \big(I_B \times I_A\big) + \big(\vdash_A \cap (M_A \times M_A)\big)\rangle$$

where $+$ is a disjoint union, and $\overline{\lambda}_A$ is like λ_A except that it reverses O/P part of moves while preserving their Q/A part.

Let W be an arbitrary set of data values, and w be a meta-variable ranging over W. A *parameterized arena* $A_W = \langle M_{A_W}, \lambda_{A_W}, \vdash_{A_W} \rangle$ is defined as follows. The set of moves M_{A_W} is of the form $C_A \cup (P_A \times W)$, where C_A is a set of constant moves that do not depend on W, and P_A is a set of parameterized move-tags so that for any $p \in P_A$ and $w \in W$, (p, w) is a move. Moves of the form (p, w) are called parameterized moves, and we will also denote them as $p(w)$. Each particular parameterized move-tag $p \in P_A$ will generate one partition of M_{A_W},

[1] A term-with-hole $C[-] : \mathsf{com}$ is a term with with zero or more holes $[-]$ in it, such that if $\Gamma \vdash M : T$ is a term of the same type as the hole then $C[M]$ is a well-typed closed term of type com, i.e. $\vdash C[M] : \mathsf{com}$.

denoted as $[p] = \{p(w) \mid w \in W\}$. The partition of M_{A_W} induced by a constant move $c \in C_A$ is a singleton set $[c] = \{c\}$. The partitioning of M_{A_W} induced by W is: $\{[c] \mid c \in C_A\} \cup \{[p] \mid p \in P_A\}$.

All moves of M_{A_W} that belong to a single partition of the form $[p]$ have the same labellings and enablings, i.e. $\lambda_{A_W}(p(w))$ is the same for all $w \in W$, and if $(p(w), n) \in \vdash_{A_W}$ (resp., $(n, p(w)) \in \vdash_{A_W}$) for some $n \in M_{A_W}, w \in W$, then $[p] \times \{n\} \subseteq \vdash_{A_W}$ (resp., $\{n\} \times [p] \subseteq \vdash_{A_W}$).

Now we are ready to give interpretations of the types of our language. The data types are interpreted by sets of values they can contain:

$$[\![int_n]\!]_W = \{0, \ldots n-1\} \qquad [\![bool]\!]_W = \{tt, ff\} \qquad [\![X]\!]_W = W$$

The base types are interpreted by parameterized arenas, where all questions are initial and P-moves answer them.

$$\begin{array}{l}
[\![\mathsf{expD}]\!]_W = \langle\, \{q, v \mid v \in [\![\mathsf{D}]\!]_W\}, \{\lambda(q) = \mathsf{OQ},\ \lambda(v) = \mathsf{PA}\}, \\
\qquad\qquad \{(*, q),\ (q, v) \mid v \in [\![\mathsf{D}]\!]_W\} \,\rangle \\
[\![\mathsf{com}]\!]_W = \langle \{run, done\}, \{\lambda(run) = \mathsf{OQ}, \lambda(done) = \mathsf{PA}\}, \{(*, run), (run, done)\} \rangle \\
[\![\mathsf{varD}]\!]_W = \langle\, \{read, v, write(v), ok \mid v \in [\![\mathsf{D}]\!]_W\}, \{\lambda(read, write(v)) = \mathsf{OQ}, \\
\lambda(v, ok) = \mathsf{PA}\}, \{(*, read), (*, write(v)), (read, v), (write(v), ok) \mid v \in [\![\mathsf{D}]\!]_W\} \rangle
\end{array}$$

In the arena for expressions, there is an initial move q to ask for the value of the expression, and corresponding to it a value from $[\![D]\!]_W$. Note that, the set of moves of $[\![\mathsf{expX}]\!]_W$ has two partitions $\{q\}$ and $\{v \mid v \in W\}$. For commands, there is an initial move *run* to initiate a command, and an answer move *done* to signal successful termination of a command. This arena does not depend on W. In the arena for variables, we have moves for writing to the variable, *write*(v), acknowledged by the move *ok*, and for reading from the variable, a move *read*, and corresponding to it a value from $[\![D]\!]_W$. $M_{[\![\mathsf{varX}]\!]_W}$ has four partitions $\{read\}$, $\{ok\}$, $\{v \mid v \in W\}$, and $\{write(v) \mid v \in W\}$.

A *justified sequence* s_W in arena A_W is a finite sequence of moves of A_W together with a pointer from each non-initial move n to an earlier move m such that $m \vdash_{A_W} n$. We say that n is (explicitly) justified by m, or when n is an answer that n *answers* m. A *legal play* (or play) is a justified sequence with some additional constraints: *alternation* (Opponent and Player moves strictly alternate), *well-bracketed* condition (when an answer is given, it is always to the most recent question which has not been answered), and *visibility* condition (a move to be played is justified by a move from a certain subsequence of the play so far, called view). The set of all legal plays in arena A_W is denoted by L_{A_W}.

A *strategy* σ_W on an arena A_W (written as $\sigma_W : A_W$) is a non-empty set of even-length plays of A_W satisfying: if $s_W \cdot m \cdot n \in \sigma_W$ then $s_W \in \sigma_W$; and if $s_W \cdot m \cdot n, s_W \cdot m \cdot n' \in \sigma_W$ then $n = n'$. A strategy specifies what options Player has at any given point of a play and it does not restrict the Opponent moves. A play is *complete* if all questions occurring in it have been answered. Given a strategy σ_W, we define the corresponding *complete strategy* σ_W^{comp} as the set of its non-empty complete plays. We write $Str_{A_W}^{comp}$ for the set of all complete strategies for the arena A_W.

Composition of strategies is interpreted as CSP-style "parallel composition plus hiding". Given strategies $\sigma_W : A_W \Rightarrow B_W$ and $\tau_W : B_W \Rightarrow C_W$, the *composition* $\sigma_W \fatsemi \tau_W : A_W \Rightarrow C_W$ consists of sequences generated by playing σ_W and τ_W in parallel, making them synchronize on moves in the shared arena B_W. Moves in B_W are subsequently hidden. We now formally define composition of strategies. Let u be a sequence of moves from A_W, B_W, and C_W. We define $u \upharpoonright B_W, C_W$ to be the subsequence of u obtained by deleting all moves from A_W along with all associated pointers from/to moves of A_W. Similarly define $u \upharpoonright A_W, B_W$. Define $u \upharpoonright A_W, C_W$ to be the subsequence of u consisting of all moves from A_W and C_W, but where there was a pointer from a move $m_A \in M_{A_W}$ to an initial move $m \in I_{B_W}$ extend the pointer to the initial move in C_W which was pointed to from m. We say that u is an *interaction* of A_W, B_W, C_W if $u \upharpoonright A_W, B_W \in L_{A_W \Rightarrow B_W}$, $u \upharpoonright B_W, C_W \in L_{B_W \Rightarrow C_W}$, and $u \upharpoonright A_W, C_W \in L_{A_W \Rightarrow C_W}$. The set of all such sequences is written as $int(A_W, B_W, C_W)$. We define:

$$\sigma_W \fatsemi \tau_W = \{u \upharpoonright A_W, C_W \mid u \in int(A_W, B_W, C_W) \wedge u \upharpoonright A_W, B_W \in \sigma_W \wedge u \upharpoonright B_W, C_W \in \tau_W\}$$

The *identity strategy* $id_{A_W} : A_W \Rightarrow A_W$, which is also called *copy-cat*, is defined in such a way that a move by Opponent in either occurrence of A_W is immediately copied by Player to the other occurrence, i.e. we have

$$id_{A_W} = \{s \in L_{A^l_W \Rightarrow A^r_W} \mid \forall\, s' \sqsubseteq^{even} s.\ s' \upharpoonright A^l_W = s' \upharpoonright A^r_W\}$$

where the l and r tags are used to distinguish between the two occurrences of A, $s' \sqsubseteq^{even} s$ means that s' is an even-length prefix of s, and $s' \upharpoonright A^l_W$ is the subsequence of s' consisting of all moves from A^l_W.

Plays in a strategy may contain several occurrences of initial moves, which define different *threads* inside plays in the following way: a thread is a subsequence of a play whose moves are connected via chains of pointers to the same occurrence of an initial move. We consider the class of *single-threaded* strategies whose behaviour depends only on one thread at a time, i.e. any Player move depends solely on the current thread of the play. We say that a strategy is *well-opened* if all its plays have exactly one initial move. It can be established one-to-one correspondence between single-threaded and well-opened strategies. The set of all strategies for an arena forms a complete partial order (cpo) under the inclusion order ($\subseteq$). The least element is $\{\epsilon\}$, and the least upper bound is given by unions. It is shown in [1,2] that arenas as objects and single-threaded (well-opened) strategies as arrows constitute a cpo-enriched cartesian closed category. From now on, we proceed to work only with well-opened strategies.

A type T is interpreted as an arena $[\![T]\!]_W$, and a term $\Gamma \vdash_W M : T$, where $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, is interpreted by a strategy $[\![\Gamma \vdash M : T]\!]_W$ for the arena $[\![\Gamma \vdash T]\!]_W = [\![T_1]\!]_W \times \ldots \times [\![T_n]\!]_W \Rightarrow [\![T]\!]_W$. Language constants and constructs are interpreted by strategies and compound terms are modelled by composition of the strategies that interpret their constituents. Identity strategies are used to interpret free identifiers from Γ. Some of the strategies [1,2] are given

below, where to simplify representation of plays, every move is tagged with the index of type component where it occurs.

$[\![\mathsf{v} : \mathsf{exp}D]\!]_W^{comp} = \{q\, v\}$ $[\![\mathsf{skip} : \mathsf{com}]\!]_W^{comp} = \{run\, done\}$ $[\![\mathsf{diverge} : \mathsf{com}]\!]_W^{comp} = \emptyset$
$[\![\mathsf{op} : \mathsf{exp}D^1 \times \mathsf{exp}D^2 \to \mathsf{exp}D]\!]_W^{comp} = \{q\, q^1\, v^1\, q^2\, v'^2\, (v\ op\ v') \mid v, v' \in [\![D]\!]_W\}$
$[\![; : \mathsf{com}^1 \times \mathsf{com}^2 \to \mathsf{com}]\!]_W^{comp} = \{run\, run^1\ done^1\ run^2\ done^2\ done\}$
$[\![:=: \mathsf{var}D^1 \times \mathsf{exp}D^2 \to \mathsf{com}]\!]_W^{comp} = \{run\, q^2\, v^2\, write(v)^1\, ok^1\, done \mid v \in [\![D]\!]_W\}$

Using standard game-semantic techniques, it can be shown as in [1,2] that this model is fully abstract for observational-equivalence.

Theorem 1. *$\Gamma \vdash_W M \subseteq_\approx N$ iff $[\![\Gamma \vdash M]\!]_W^{comp} \subseteq [\![\Gamma \vdash N]\!]_W^{comp}$.*

Suppose that there is a special free identifier abort of type com in Γ. Let $M[N/x]$ denote the capture-free substitution of N for x in M. We say that a term $\Gamma \vdash_W M$ is *safe* iff $\Gamma \backslash \mathsf{abort} \vdash_W M[\mathsf{skip}/\mathsf{abort}] \subseteq_\approx M[\mathsf{diverge}/\mathsf{abort}]$; otherwise we say that a term is *unsafe*. Since the game-semantics model is fully abstract, the following can be shown (see also [3]).

Lemma 1. *A term $\Gamma \vdash_W M$ is safe if $[\![\Gamma \vdash M]\!]_W^{comp}$ does not contain any play with moves from $M_{[\![\mathsf{com}^{abort}]\!]}$.*

For example, $[\![\mathsf{abort} : \mathsf{com}^{\mathsf{abort}} \vdash \mathsf{skip}\,;\ \mathsf{abort} : \mathsf{com}]\!]^{comp}$ is the set $\{run \cdot run^{abort} \cdot done^{abort} \cdot done\}$, so this term is unsafe.

4 Logical Relations

A binary *relation* between sets W_0 and W_1 is any subset $R \subseteq W_0 \times W_1$. We will use the notation $R : W_0 \longleftrightarrow W_1$ to mean that R is a binary relation between W_0 and W_1, and $w_0\, R\, w_1$ to mean $(w_0, w_1) \in R$, in which case we say that w_0 and w_1 are R-related. The domain of a relation R is the set of the first components of all pairs in R. We say that R is a *partial function* iff $\forall\, w_0, w_1, w_1'.(w_0 R w_1 \wedge w_0 R w_1') \Rightarrow w_1 = w_1'$, and R is *injective* iff $\forall\, w_0, w_0', w_1.(w_0 R w_1 \wedge w_0' R w_1) \Rightarrow w_0 = w_0'$. A special case of relation is the *identity relation* $I_W : W \longleftrightarrow W$, defined by $I_W = \{(w, w) \mid w \in W\}$, i.e. $w\, I_W\, w'$ iff $w = w'$. Next, we define relations on sequences. For any $R : W_0 \longleftrightarrow W_1$, define $R^* : W_0^* \longleftrightarrow W_1^*$ as

$$t\, R^*\, t' \quad \text{iff} \quad t_1\, R\, t_1' \wedge \ldots \wedge t_{|t|}\, R\, t'_{|t'|}$$

where $|\ t\ |$ denotes the length of t, and for any $1 \leq k \leq |\ t\ |$, t_k denotes the k-th element of t. That is, sequences are R-related if they have the same length and corresponding elements are R-related.

Let $R : W_0 \longleftrightarrow W_1$ be a relation. For any data type, we define the relation $[\![D]\!]_{R, W_0, W_1} : [\![D]\!]_{W_0} \longleftrightarrow [\![D]\!]_{W_1}$ as follows.

$$[\![int_n]\!]_{R, W_0, W_1} = I_{[\![int_n]\!]} \quad [\![bool]\!]_{R, W_0, W_1} = I_{[\![bool]\!]} \quad [\![X]\!]_{R, W_0, W_1} = R$$

Next we "lift" the definition of relations to arenas. We define a relational arena $A_{R, W_0, W_1} = \langle M_{A_{R, W_0, W_1}}, \lambda_{A_{R, W_0, W_1}}, \vdash_{A_{R, W_0, W_1}} \rangle : A_{W_0} \longleftrightarrow A_{W_1}$ between two

parameterized arenas induced by R. Let $M_{A_{W_i}} = C_A \cup (P_A \times W_i)$ for $i = 0, 1$. Then, we have:

$$\begin{array}{ll} (m, m') \in M_{A_{R,W_0,W_1}} \text{ iff} & \begin{cases} m = m', \text{ if } m \in C_A \\ w\, R\, w', \text{ if } m \in P_A \times W_0, m = p(w), m' = p(w') \end{cases} \\ \lambda_{A_{R,W_0,W_1}}(m, m') = & \lambda_{A_{W_0}}(m) = \lambda_{A_{W_1}}(m') \\ * \vdash_{A_{R,W_0,W_1}} (m, m') \text{ iff} & (* \vdash_{A_{W_0}} m \wedge * \vdash_{A_{W_1}} m') \\ (n, n') \vdash_{A_{R,W_0,W_1}} (m, m') & \text{iff } (n \vdash_{A_{W_0}} m \wedge n' \vdash_{A_{W_1}} m') \end{array}$$

Let define a relation $L_{A_{R,W_0,W_1}} : L_{A_{W_0}} \longleftrightarrow L_{A_{W_1}}$ between the sets of all legal plays in A_{W_0} and A_{W_1} induced by R.

$$\begin{array}{l} s\, L_{A_{R,W_0,W_1}}\, s' \text{ iff } (i) s\, (M_{A_{R,W_0,W_1}})^*\, s' \\ \qquad\qquad (ii)\, \lambda_{A_{W_0}}(s_i) = \lambda_{A_{W_1}}(s'_i), \text{ for } 1 \leq i \leq |\, s\, | \\ \qquad\qquad (iii) s_i \text{ justifies } s_j \text{ iff } s'_i \text{ justifies } s'_j, \text{ for } 1 \leq i < j \leq |\, s\, | \end{array} \quad (1)$$

We define $Domain(L_{A_{R,W_0,W_1}})$ as the set of all legal plays s from A_{W_0}, such that for any parameterized move $p(w)$ in s we have that w is in the domain of R.

Finally, we define a relation $Str^{comp}_{A_{R,W_0,W_1}} : Str^{comp}_{A_{W_0}} \longleftrightarrow Str^{comp}_{A_{W_1}}$ between complete strategies on A_{W_0} and A_{W_1} induced by R.

$$\begin{array}{l} \sigma\, Str^{comp}_{A_{R,W_0,W_1}}\, \sigma' \text{ iff } \forall\, s \in \sigma. s \in Domain(L_{A_{R,W_0,W_1}}) \Rightarrow \exists\, S' \subseteq \sigma'. S' \neq \emptyset \wedge \\ \qquad\qquad (\forall\, s' \in S'. s\, L_{A_{R,W_0,W_1}}\, s') \wedge Closed(A, R, W_0, W_1, s, S') \end{array} \quad (2)$$

$$\begin{array}{l} Closed(A, R, W_0, W_1, s, S') = \forall\, s' \in S'. \forall\, k. \forall\, w' \in W_1. \lambda^{OP}(s_k) = O \wedge \\ \quad s_k = p(w) \wedge w\, R\, w' \Rightarrow \exists\, s'' \in S'. s''_1 = s'_1 \wedge \ldots \wedge s''_{k-1} = s'_{k-1} \wedge s''_k = p(w') \end{array}$$

That is, two complete strategies σ and σ' are R-related if and only if for any complete play s from σ which is in the domain of the logical relation, there exists a nonempty subset S' of σ' such that s is R-related to any complete play in S' and S' is closed under those choices of Opponent moves which preserve the relation R. We say that S' is *R-closed* with respect to s.

Before we prove the Basic Lemma for logical relations, we first show several useful technical lemmas.

Lemma 2. *Let $\sigma_W : A_W \Rightarrow B_W$ and $\tau_W : B_W \Rightarrow C_W$ be two strategies and $R : W_0 \longleftrightarrow W_1$ be a relation. If we have that $\sigma^{comp}_{W_0}\, (Str^{comp}_{A \Rightarrow B_{R,W_0,W_1}})\, \sigma^{comp}_{W_1}$ and $\tau^{comp}_{W_0}\, (Str^{comp}_{B \Rightarrow C_{R,W_0,W_1}})\, \tau^{comp}_{W_1}$, then*

$$(\sigma_{W_0} \fatsemi \tau_{W_0})^{comp}\ (Str^{comp}_{A \Rightarrow C_{R,W_0,W_1}})\ (\sigma_{W_1} \fatsemi \tau_{W_1})^{comp}$$

Proof. Let $s \in (\sigma_{W_0} \fatsemi \tau_{W_0})^{comp}$. Then there must be some witness to this, i.e. some $u \in int(A_{W_0}, B_{W_0}, C_{W_0})$ such that $s = u \upharpoonright A_{W_0}, C_{W_0}$. By definition of composition, we have that $u \upharpoonright B_{W_0}, C_{W_0} \in \tau^{comp}_{W_0}$, and for every B_W-initial move $i \in u \upharpoonright B_{W_0}, C_{W_0}$ we have $(u \upharpoonright A_{W_0}, B_{W_0}) \upharpoonright i \in \sigma^{comp}_{W_0}$. We denote by $s \upharpoonright i$ the subsequence of s consisting of all those moves which are hereditarily justified (via chains of pointers) by the same initial move i. Since σ_{W_0} is R-related to σ_{W_1}, there is a set $U_i \subseteq \sigma^{comp}_{W_1}$ such that U_i is R-closed with respect

to $(u \restriction A_{W_0}, B_{W_0}) \restriction i$ for any B_W-initial move i in $u \restriction A_{W_0}, B_{W_0}$. Since τ_{W_0} is R-related to τ_{W_1}, there is a set $U \subseteq \tau_{W_1}^{comp}$ such that U is R-closed with respect to $u \restriction B_{W_0}, C_{W_0}$. We can now generate a set U'. For an arbitrary $u_1 \in U$ we create a set of sequences in U' as follows. For any B_W-initial move $i \in u_1$, we choose an arbitrary $u_{i,1} \in U_i$ such that $(u_1 \restriction B) \restriction i = u_{i,1} \restriction B$. Then we generate an interaction sequence $u' \in int(A_{W_1}, B_{W_1}, C_{W_1})$, such that $u' \restriction B_{W_1}, C_{W_1} = u_1$, and $(u' \restriction A_{W_1}, B_{W_1}) \restriction i = u_{i,1}$. We repeat this process for all possible $u_1 \in U$ and $u_{i,1} \in U_i$ for any B_W-initial move $i \in u_1$. Finally, we obtain $S = \{u' \restriction A_{W_1}, C_{W_1} \mid u' \in U'\}$, which is R-closed with respect to s. □

Lemma 3. *Let $id_{A_W} : A_W \Rightarrow A_W$ be an identity strategy and $R : W_0 \longleftrightarrow W_1$ be a relation. Then $id_{A_{W_0}}^{comp}\,(Str_{A \Rightarrow A_{R,W_0,W_1}}^{comp})\,id_{A_{W_1}}^{comp}$.*

Proof. Let $s \in id_{A_{W_0}}^{comp}$. We first generate a complete play $t \in id_{A_{W_1}}^{comp}$, such that s is R-related to t. This is done by choosing for any Opponent move s_k of the form $p(w)$, where k is odd, a R-related move $p(w')$ from $M_{A_{W_1}}$, and setting $t_k = p(w')$, $t_{k+1} = p(w')$. Then we obtain a set S' from t, such that for any odd k and for any $m \in M_{A_{W_1}}$, where $s_k\, M_{R,A_{W_0},A_{W_1}}\, m$, we create a sequence in S': $s' = t_1 \dots t_{k-1} m m t_{k+2} \dots t_{|t|}$. Such S' is R-closed with respect to s. □

Let R be a relation between two cpo $(V, \leq)$ and $(V', \leq')$, and let $\precsim$ be the pointwise ordering on R. We say that R is *complete* iff $(R, \precsim)$ is cpo such that:

- the least element is $(\bot, \bot')$, where $\bot$ (resp., $\bot'$) is the least element of $(V, \leq)$ (resp., $(V', \leq')$).
- for any directed set $D \subseteq R$, its least upper bound consists of pointwise least upper bounds of $(V, \leq)$ and $(V', \leq')$.

Lemma 4. *For any parameterized arena A_W and for any relation $R : W_0 \longleftrightarrow W_1$, we have that $Str_{A_{R,W_0,W_1}}^{comp}$ is complete.*

Proof. The least elements of $Str_{A_{W_0}}^{comp}$ and $Str_{A_{W_1}}^{comp}$ are both $\emptyset$, and the least upper bounds are unions. □

We now present the Basic Lemma of logical relations for our language.

Theorem 2. *Let $\Gamma, z_1 : \mathsf{exp}X, \dots, z_k : \mathsf{exp}X \vdash M : T$ be a term such that M contains no data values of X, and $R : W_0 \longleftrightarrow W_1$ be a relation such that $(w_1, w_1'), \dots, (w_k, w_k')$ are some pairs in R. If either M satisfies ($\mathbf{NoEq}_X$) or R is a partial function and injective, then*

$$[\![\Gamma \vdash M[w_1/z_1, \dots, w_k/z_k]]\!]_{W_0}^{comp}(Str_{[\![\Gamma \vdash T]\!]_{R,W_0,W_1}}^{comp})[\![\Gamma \vdash M[w_1'/z_1, \dots, w_k'/z_k]]\!]_{W_1}^{comp}$$

Proof. The proof is by induction on the structure of terms.

The case of free identifiers holds due to the fact that logical relations are preserved by identity strategies (see Lemma 3).

Let $\Delta = z_1 : \mathsf{exp}X, \ldots, z_k : \mathsf{exp}X$. Consider the case $\Gamma, \Delta \vdash M = N$, where $\Gamma, \Delta \vdash M, N : \mathsf{exp}X$. By induction hypothesis, we have that:

$$[\![\Gamma \vdash M[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}\,(Str^{comp}_{[\![\Gamma \vdash \mathsf{exp}X]\!]_{R,W_0,W_1}})\,[\![\Gamma \vdash M[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1} \quad (*)$$
$$[\![\Gamma \vdash N[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}\,(Str^{comp}_{[\![\Gamma \vdash \mathsf{exp}X]\!]_{R,W_0,W_1}})\,[\![\Gamma \vdash N[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1} \quad (**)$$

where $w_1 R w'_1, \ldots, w_k R w'_k$, and we also use the following abbreviations: $\overrightarrow{w} = (w_1, \ldots, w_k)$, $\overrightarrow{w'} = (w'_1, \ldots, w'_k)$, and $M[\overrightarrow{w}/\Delta] = M[w_1/z_1, \ldots, w_k/z_k]$. Suppose $t \in [\![\Gamma \vdash M = N[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}$. Then either $t = q\,t_1\,t_2\,tt$ or $t = q\,t_1\,t_2\,ff$, where $q\,t_1\,w_1 \in [\![\Gamma \vdash M[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}$ and $q\,t_2\,w_2 \in [\![\Gamma \vdash N[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}$. Let $w_1 \neq w_2$, and so $t = q\,t_1\,t_2\,ff$. Since R is a partial function and injective, for any play $q\,t'_1\,w'_1 \in [\![\Gamma \vdash M[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1}$ related by R with $q\,t_1\,w_1$, and for any $q\,t'_2\,w'_2 \in [\![\Gamma \vdash N[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1}$ related by R with $q\,t_2\,w_2$, it must be that $w'_1 \neq w'_2$. Then all plays of the form $t' = q\,t'_1\,t'_2\,ff \in [\![\Gamma \vdash M = N[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1}$ are R-related with t. The other case, when $w_1 = w_2$ is similar. So we have $[\![\Gamma \vdash M = N[\overrightarrow{w}/\Delta]]\!]^{comp}_{W_0}\,(Str^{comp}_{[\![\Gamma \vdash \mathsf{expbool}]\!]_{R,W_0,W_1}})\,[\![\Gamma \vdash M = N[\overrightarrow{w'}/\Delta]]\!]^{comp}_{W_1}$.

Recursion $\Gamma \vdash YM : T$ is handled by using the following facts:

- $Str^{comp}_{[\![\Gamma \vdash T]\!]_{R,W_0,W_1}}$ is complete by Lemma 4
- the inductive hypothesis $[\![\Gamma \vdash M : T \to T]\!]^{comp}_{W_0}\,(Str^{comp}_{[\![\Gamma \vdash T \to T]\!]_{R,W_0,W_1}})\,[\![\Gamma \vdash M : T \to T]\!]^{comp}_{W_1}$
- and the definition of recursion $[\![\Gamma \vdash YM : T]\!]$ (see [1,2]).

The other cases for language constants and constructs are proved similarly, and we also use the fact shown in Lemma 2 that logical relations are preserved by composition of strategies. □

Remark. In Theorem 2, we have made an assumption that data values of X that occur as expression constants in M must be R-related in the two versions of M with parameters W_0 and W_1. This is so because any complete play s and any play in its R-closed set S' have to be R-related on their Player moves. For example, let $W_0 = W_1 = \{0, 1\}$, and $R = \{(0, 1)\}$. Then $[\![\vdash 0 : \mathsf{exp}X]\!]_{W_0}$ is not R-related to $[\![\vdash 0]\!]_{W_1}$, but is R-related to $[\![\vdash 1]\!]_{W_1}$.

Example 1. Consider the term M_1:

$$f : \mathsf{exp}X^{f,1} \to \mathsf{com}^f, x : \mathsf{exp}X^x \vdash f(x) : \mathsf{com}$$

The model representing this term, when $[\![X]\!] = \{0, 1\}$, is shown in Fig. 1. The model illustrates only the possible behaviors of this term: f may evaluate its argument, zero or more times, then the term terminates with a move *done*. The model makes no assumption about the number of times that f uses its argument. Note that moves tagged with f represent the actions of calling and returning from the function, while moves tagged with $f, 1$ are the actions caused by evaluating the first argument of f.

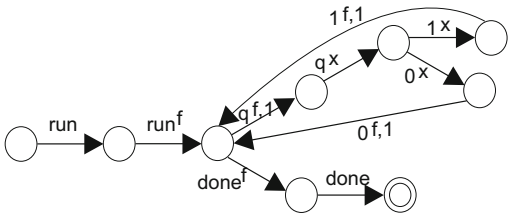

Fig. 1. The strategy for M_1 as a finite automaton

Let $W = \{0,1\}$, $W' = \mathbb{Z}$ (integers), and $R = \{(0,-n),(1,n) \mid n \in \{0,1,\ldots\}\}$. This term satisfies ($\mathbf{NoEq}_X$), so by Theorem 2 we have that $[\![M_1]\!]^{comp}_W$ and $[\![M_1]\!]^{comp}_{W'}$ are related by R. For example, let $s = run\, run^f\, q^{f,1}\, q^x\, 1^x\, 1^{f,1}\, done^f\, done \in [\![M_1]\!]^{comp}_W$. Then, the corresponding R-closed subset of $[\![M_1]\!]^{comp}_{W'}$ is:

$$S' = \{run\, run^f\, q^{f,1}\, q^x\, n^x\, n^{f,1}\, done^f\, done \mid n \in \{0,1,\ldots\}\}$$

Also note that, for $s = run\, run^f\, q^{f,1}\, q^x\, 1^x\, 1^{f,1}\, q^{f,1}\, q^x\, 1^x\, 1^{f,1}\, done^f\, done$, the corresponding R-closed set is:

$$S' = \{run\, run^f\, q^{f,1}\, q^x\, n^x\, n^{f,1}\, q^{f,1}\, q^x\, m^x\, m^{f,1}\, done^f\, done \mid\ n, m \in \{0,1,\ldots\}\}$$

This is so because two occurrences of 1^x in s are Opponent moves, so S' needs to be closed under all alternative choices of these moves which preserve R.

Let $W = \{0,1,2,3\}$, $W' = \{0\}$, and $R = \{(0,0)\}$. Then $[\![M_1]\!]^{comp}_W$ and $[\![M_1]\!]^{comp}_{W'}$ are also related by R. □

Example 2. Consider the term M_2:

$$f : \mathsf{com}^{f,1} \to \mathsf{com}^f, abort : \mathsf{com}^{abort}, x : \mathsf{exp}X^x, y : \mathsf{exp}X^y \vdash f\big(\mathsf{if}\ \neg(x = y)\ \mathsf{then}\ abort\big) : \mathsf{com}$$

The model for this term with parameter $\{tt, ff\}$ is shown in Fig. 2. Let $W = \{0,1,2,3\}$, $W' = \{tt, ff\}$, and $R = \{(0,tt),(1,ff)\}$. This term does not satisfy

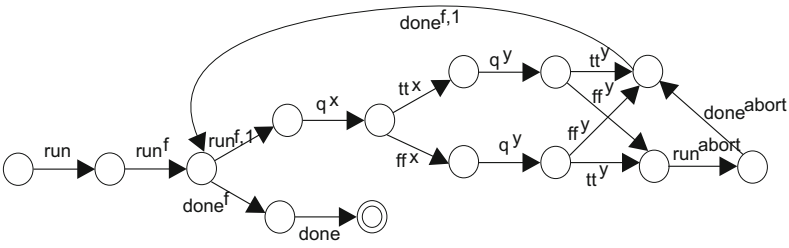

Fig. 2. The strategy for M_2

($\mathbf{NoEq}_X$), so R is a partial function and injective. By Theorem 2, terms M_2 with parameters W and W' are related by R. Let

$$s = run\, run^f\ run^{f,1}\ q^x\ 1^x\ q^y\ 0^y\ run^{abort}\ done^{abort}\ done^{f,1}\ done^f\ done$$

be a complete play for M_2 with parameter W. Then, the corresponding R-closed subset of $[\![M_2]\!]^{comp}_{W'}$ is:

$$S' = \{run\, run^f\ run^{f,1}\ q^x\ ff^x\ q^y\ tt^y\ run^{abort}\ done^{abort}\ done^{f,1}\ done^f\ done\}\ \square$$

5 Threshold Collections

In this section we present theorems which provide sufficient conditions for reducing the verification of properties for all interpretations of X, to the verification of the same properties for finite interpretations of X.

Theorem 3. *Let $\Gamma \vdash M, N : T$ be terms which satisfy ($\mathbf{NoEq}_X$), $W_0 = \{0\}$, and all data values of X in M and N are from W_0.*
(i) If $\Gamma \vdash_{W_0} M \not\cong N$ then $\Gamma \vdash_W M \not\cong N$ for all W.
(ii) If $\Gamma \vdash_{W_0} M$ is not safe then $\Gamma \vdash_W M$ is not safe for all W.
(iii) If $\Gamma \vdash_{W_0} M$ is safe then $\Gamma \vdash_W M$ is safe for all W.

Proof. Let $R : W \longleftrightarrow W_0$ be the unique total function from W to W_0. By Theorem 2, we have that

$$[\![\Gamma \vdash N]\!]^{comp}_W\ (Str^{comp}_{[\![\Gamma \vdash T]\!]_{R,W,W_0}})\ [\![\Gamma \vdash N]\!]^{comp}_{W_0} \quad (1)$$
$$[\![\Gamma \vdash M]\!]^{comp}_{W_0}\ (Str^{comp}_{[\![\Gamma \vdash T]\!]_{R^{-1},W_0,W}})\ [\![\Gamma \vdash M]\!]^{comp}_{W} \quad (2)$$

We will prove (i) by contraposition. Suppose that $\Gamma \vdash_W M \sqsubseteq_{\approx} N$ for some W. Let $t \in [\![\Gamma \vdash M]\!]^{comp}_{W_0}$. By (2), there exists $t' \in [\![\Gamma \vdash M]\!]^{comp}_{W}$ such that $t(L_{[\![\Gamma \vdash T]\!]_{R^{-1},W_0,W}})t'$ (*). Since $\Gamma \vdash_W M \sqsubseteq_{\approx} N$, we have $t' \in [\![\Gamma \vdash N]\!]^{comp}_{W}$. Now by (1), there exists $t^\dagger \in [\![\Gamma \vdash N]\!]^{comp}_{W_0}$ such that $t'\, L_{[\![\Gamma \vdash T]\!]_{R,W,W_0}}\, t^\dagger$ (**). By (*), (**), the fact that $W_0 \times W_0$ is the identity relation, it follows that $t = t^\dagger$. Therefore, $[\![\Gamma \vdash M]\!]^{comp}_{W_0} \subseteq [\![\Gamma \vdash N]\!]^{comp}_{W_0}$, and so $\Gamma \vdash_{W_0} M \sqsubseteq_{\approx} N$. The cases (ii) and (iii) are similar. $\square$

Theorem 4. *Let $\Gamma \vdash M, N : T$ be terms, κ be a nonzero integer such that $W_\kappa = \{0, \ldots, \kappa\}$, and all data values of X in M and N are from W_κ.*
(i) If $\Gamma \vdash_{W_\kappa} M \not\cong N$ then $\Gamma \vdash_{W_{\kappa'}} M \not\cong N$ for all $\kappa' \geq \kappa$.
(ii) If $\Gamma \vdash_{W_\kappa} M$ is not safe then $\Gamma \vdash_{W_{\kappa'}} M$ is not safe for all $\kappa' \geq \kappa$.

Proof. Consider the case (i). The proof is by contraposition. Suppose that $\Gamma \vdash_{W_{\kappa'}} M \sqsubseteq_{\approx} N$ for some $\kappa' \geq \kappa$. Let $R : W_\kappa \longleftrightarrow W_{\kappa'}$ be a total function and injective. By Theorem 2,

$$[\![\Gamma \vdash M]\!]^{comp}_{W_\kappa}\ (Str^{comp}_{[\![\Gamma \vdash T]\!]_{R,W_\kappa,W_{\kappa'}}})\ [\![\Gamma \vdash M]\!]^{comp}_{W_{\kappa'}} \quad (1)$$

Let $t \in [\![\Gamma \vdash M]\!]^{comp}_{W_\kappa}$. By (1), there exists $t' \in [\![\Gamma \vdash M]\!]^{comp}_{W_{\kappa'}}$ such that $t(L_{[\![\Gamma \vdash T]\!]_{R,W_\kappa,W_{\kappa'}}})t'$ (*). By assumption, we have $t' \in [\![\Gamma \vdash N]\!]^{comp}_{W_{\kappa'}}$. Let $\pi : W_{\kappa'} \longleftrightarrow W_\kappa$ be such that $\pi : W_{\kappa'} \backslash Range(R) \to W_\kappa$ is a total function. Then $R^{-1} \cup \pi : W_{\kappa'} \longleftrightarrow W_\kappa$ is a total function and surjective. But $W_\kappa \subseteq W_{\kappa'}$, $(R^{-1} \cup \pi) \upharpoonright W_\kappa$ is injective, and $t' \in Domain(L_{[\![\Gamma \vdash T]\!]_{(R^{-1} \cup \pi) \upharpoonright W_\kappa, W_{\kappa'}, W_\kappa}})$. In a manner similar to the proof of Theorem 2, we can show that there exists $t^\dagger \in [\![\Gamma \vdash N]\!]^{comp}_{W_\kappa}$ such that $t'(L_{[\![\Gamma \vdash T]\!]_{R^{-1} \cup \pi, W_{\kappa'}, W_\kappa}})t^\dagger$ (**). It follows from (*), (**), and the definition of R, that $t = t^\dagger$. Therefore, $\Gamma \vdash_{W_\kappa} M \sqsubseteq N$.

Consider the case (ii). Suppose that $\Gamma \vdash_{W_{\kappa'}} M$ is safe for some $\kappa' \geq \kappa$. Let $R : W_\kappa \longleftrightarrow W_{\kappa'}$ be a total function and injective. By Theorem 2, $[\![\Gamma \vdash M]\!]^{comp}_{W_\kappa} (Str^{comp}_{[\![\Gamma \vdash T]\!]_{R,W_\kappa,W_{\kappa'}}}) [\![\Gamma \vdash M]\!]^{comp}_{W_{\kappa'}}$ (1). Let $t \in [\![\Gamma \vdash M]\!]^{comp}_{W_\kappa}$. By (1), there exists $t' \in [\![\Gamma \vdash M]\!]^{comp}_{W_{\kappa'}}$ such that $t(L_{[\![\Gamma \vdash T]\!]_{R,W_\kappa,W_{\kappa'}}})t'$ (*). But t' is safe by assumption, i.e. it does not contain unsafe moves. So it must be that t is also safe. □

Example 3. The term M_1 from Example 1 with parameter $W_0 = \{0\}$ is abort-safe. By Theorem 3, we can conclude that M_1 is abort-safe for all W. So M_1 where X is replaced by int is also abort-safe.

The term M_2 from Example 2 with parameter $W_1 = \{0, 1\}$ is abort-unsafe. By Theorem 4, it follows that M_2 is abort-unsafe for all $W_{\kappa'}$, $\kappa' \geq 1$ (which also includes int). □

6 Application

From now on, we restrict the programming language to the 2nd-order recursion-free fragment. More precisely, function types are restricted to $T ::= B \mid B \to T$. This restriction is made since the game semantic model for this language fragment is decidable, i.e. the model can be given concrete automata-theoretic representations using the regular languages as in [6] and the CSP process algebra as in [4], and so a range of verification problems such as approximation and safety can be solved algorithmically. We have extended the tool given in [4], which supports only IA terms, such that it automatically converts a predicatively parametrically polymorphic IA term into a parameterized CSP process [15] which represents its game semantics. The resulting CSP process is defined by a script in machine readable CSP which the tool outputs.

Let us consider an implementation of the linear search algorithm:

$$
\begin{array}{l}
x[k] : \mathsf{var}X,\ y : \mathsf{exp}X,\ \mathsf{abort} : \mathsf{com} \vdash \\
\quad \mathsf{new}_X\ a[k]\ \mathsf{in}\ \mathsf{new}_{int_{k+1}}\ i := 0\ \mathsf{in} \\
\quad \mathsf{while}\,(i < k)\,\mathsf{do}\,\{a[i] := x[i];\ \ i := i + 1;\ \} \\
\quad \mathsf{new}_X\ z := y\ \mathsf{in}\ \mathsf{new}_{bool}\ present := false\ \mathsf{in} \\
\quad \mathsf{while}\,(i < k)\,\mathsf{do}\,\{\ \mathsf{if}\,(\mathbf{a[i]} = \mathbf{z})\,\mathsf{then}\,present := true;\ \ i := i + 1;\ \} \\
\quad \mathsf{if}\,(\neg\, present)\,\mathsf{then}\,\mathsf{abort}\ : \mathsf{com}
\end{array}
$$

The code includes a meta variable $k > 0$, representing array size, which will be replaced by several different values. The data stored in the arrays x, a, and the

expression y is of type X, and the type of index i is int_{k+1}, i.e. one more than the size of the array. The program first copies the input array x into a local array a, and the input expression y into a local variable z. Then, the local array is searched for an occurrence of the value stored in z. If the search fails, then abort is executed. The equality is the only operation on the data of type X (see the bold in the code), so this term does not satisfy ($\mathbf{NoEq}_X$).

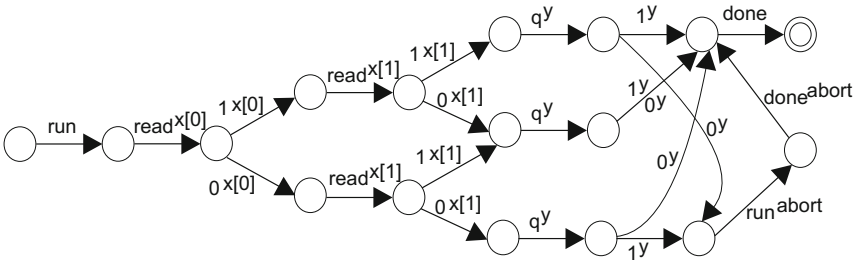

Fig. 3. Model for the linear search with k=2 and $W_1 = \{0, 1\}$.

A model for the term with $k = 2$ and parameter $W_1 = \{0, 1\}$ is shown in Fig. 3. If the value read from y has occurred in x then the term terminates successfully without executing abort; otherwise the term runs abort.

If this term is tested for abort-safety, we obtain the following counter-example:

$$run \;\; read^{x[0]} \;\; 1^{x[0]} \;\; read^{x[1]} \;\; 1^{x[1]} \;\; q^{y} \;\; 0^{y} \;\; run^{abort} \;\; done^{abort} done$$

By Theorem 4, it follows that this term is abort-unsafe for all $W_{\kappa'}$, $\kappa' \geq 1$. So if X is replaced by int, the term is also abort-unsafe. We performed experiments for the linear search term with different sizes of k and $W_\kappa = \{0, \ldots, \kappa\}$, by converting the term into a CSP process and then using FDR model checker [2] to generate its model and test its abort-safety.

Experimental results are shown in Table 1. The execution time is given in seconds, and the size of the final model in number of states. We ran FDR on a Machine AMD Phenom II X4 940 with 4GB RAM. We can see that the model and the time increase very fast as we increase the size of W_κ. Still by using Theorem 4, it only suffices to check the term with parameter W_1 in order to infer its safety for all $W_{\kappa'}$, $\kappa' \geq 1$.

6.1 Integration with Abstraction Refinement

We can combine our parameterized approach with an abstraction refinement procedure (ARP) [3], since both approaches can be applied to terms which contain infinite (integer) types. The former approach will be used to handle all

[2] http://www.fsel.com

Table 1. Verification of linear search

	W_1		W_2		W_3	
k	Time	Model	Time	Model	Time	Model
5	2	36	2	68	2	124
10	7	66	8	138	10	274
15	18	96	20	208	39	424
20	40	126	47	278	160	574

data-independent integer types, which will be treated as parameters, while the rest of infinite types will be handled by the latter approach.

In the ARP (see [3] for details), instead of finite integers we introduce a new data type of abstracted integers int_π. We use the following finitary abstractions π: $[\,] = \{\mathbb{Z}\}$, $[n, m] = \{< n, n, \ldots, 0, \ldots, m-1, m, > m\}$, where Z, $<n = \{n' \mid n' < n\}$, and $>n = \{n' \mid n' > n\}$ are abstract values. Abstractions are refined by splitting abstract values. We check safety of $\Gamma \vdash_W M : T$ (with infinite integer data types) by performing a sequence of iterations. The initial abstracted term $\Gamma_0 \vdash_W M_0 : T_0$ uses the coarsest abstraction $[\,]$ for any integer identifier. In every iteration, the game semantics model of the abstracted term is checked for safety. If no counterexample or a genuine one is found, the procedure terminates. Otherwise, if a spurious counter-example is found, it is used to generate a refined abstracted term, which is passed to the next iteration.

For example, let us reconsider the linear search term. The ARP needs $k + 2$ iterations to automatically adjust the type of the local variable i from the coarsest abstraction with a single abstract value $[\,]$ to the abstraction $[0, k]$. For such abstraction of i ($int_{[0,k]}$), a genuine counter-example is found.

7 Conclusion

The paper presents how we can automatically verify parameterized terms for all instances of the parameter X. We described here the case where there is one data type variable X. If there is more than one such variable, the obtained results can be applied to one at a time. The approach proposed here can be also extended to terms with nondeterminism [5], concurrency [7], and other features.

References

1. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In: O'Hearn, P.W., Tennent, R.D. (eds.) Algol-like languages, Birkhaüser, Boston (1997)
2. Abramsky, S., McCusker, G.: Game Semantics. In: Proceedings of the 1997 Marktoberdorf Summer School: Computational Logic, pp. 1–56. Springer, Heidelberg (1998)
3. Dimovski, A., Ghica, D.R., Lazić, R.: Data-Abstraction Refinement: A Game Semantic Approach. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 102–117. Springer, Heidelberg (2005)

4. Dimovski, A., Lazić, R.: Compositional Software Verification Based on Game Semantics and Process Algebras. Int. Journal on STTT 9(1), 37–51 (2007)
5. Dimovski, A.: A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 121–135. Springer, Heidelberg (2010)
6. Ghica, D.R., McCusker, G.: The Regular-Language Semantics of Second-order Idealized Algol. Theoretical Computer Science 309(1–3), 469–502 (2003)
7. Ghica, D.R., Murawski, A.S., Ong, C.-H.L.: Syntactic control of concurrency. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 683–694. Springer, Heidelberg (2004)
8. Hughes, D.J.D.: Hypergame Semantics: Full Completness for System F. D. Phil. Thesis, Oxford University (1999)
9. Laird, J.: Game Semantics for a Polymorphic Programming Language. In: Proceedings of LICS 2010. IEEE, pp. 41–49. IEEE, Los Alamitos (2010)
10. Lazić, R.: A Semantic Study of Data Independence with Applications to Model Checking. D. Phil. Thesis, Oxford University (1999)
11. Lazić, R., Nowak, D.: A Unifying Approach to Data-Independence. In: Wagner, T.A., Rana, O.F. (eds.) AA-WS 2000. LNCS (LNAI), vol. 1887, pp. 581–595. Springer, Heidelberg (2001)
12. Murawski, A.S., Ouaknine, J.: On Probabilistic Program Equivalence and Refinement. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 156–170. Springer, Heidelberg (2005)
13. Ma, Q., Reynolds, J.C.: Types, Abstraction, and Parametric Polymorphism, Part 2. In: Schmidt, D., Main, M.G., Melton, A.C., Mislove, M.W., Brookes, S.D. (eds.) MFPS 1991. LNCS, vol. 598, pp. 1–40. Springer, Heidelberg (1992)
14. O'Hearn, P.W., Tennent, R.D.: Parametricity and Local Variables. Journal of the ACM 42(3), 658–709 (1995)
15. Roscoe, W.A.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
16. Wadler, P.: Theorems for Free! In: FPCA 1989, pp. 347–379. ACM, New York (1989)

Author Index