Project Nº: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**  
**Future and Emerging Technologies**

## Deliverable D2.5

**Verification of Behavioral Properties**

Due date of deliverable: (T0+24)

Actual submission date: 1st March 2011

Start date of the project: **1st March 2009**  
Duration: **48 months**

Organisation name of lead contractor for this deliverable: **CTH**

---

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td>✓</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>

---

Final version
Executive Summary:
Verification of Behavioral Properties

This document summarises deliverable D2.5 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

In this deliverable we report on the work performed in Task 2.5 on verification of general behavior properties. We present a program logic for the HATS modeling language ABS. The program logic is based on symbolic execution and used to reason about functional behavior of ABS programs. We present further work on possible optimization concerning compositionality and full abstraction which promise a significantly better automation. We report further on our results concerning incremental verification and introduce a variety of techniques suitable to reduce redundancy and minimize the need for redoing analyses when an ABS program is changed locally. Also an analysis to detect and prove absence of dead-locks is presented.

List of Authors

Frank de Boer (CWI)
Richard Bubel (CTH)
Dilian Gurov (KTH)
Elena Giachino (BOL)
Reiner Hähnle (CTH)
Einar Broch Johnsen (UIO)
Ina Schaefer (CTH)
# Contents

1 Introduction .............................................. 5

2 A Program Logic for ABS ............................... 7
   2.1 Introduction ........................................ 7
   2.2 A Program Logic for ABS .......................... 7
      2.2.1 Dynamic Logic ................................ 7
      2.2.2 Sequent Calculus .............................. 8
      2.2.3 Functional ABS ................................ 9
      2.2.4 Sequential ABS ................................ 9
      2.2.5 Object-Oriented ABS ......................... 10
   2.3 Optimizations ...................................... 12
      2.3.1 4-Event Histories ............................ 13
      2.3.2 Abstract Object Creation in Dynamic Logic . 13

3 Analysis of deadlocks in ABS ......................... 15
   3.1 Examples ........................................... 15
   3.2 Contracts ......................................... 16
   3.3 Deadlock Analysis .................................. 17

4 A Framework and Toolset for Modular Verification .... 19
   4.1 Specialization: Procedure-Modular Verification .... 20
   4.2 Specialization: Verification of Software Product Lines . 22

5 Lazy Behavioral Subtyping .............................. 25

6 Efficient Functional Verification of Software Product Lines 26
   6.1 Introduction ........................................ 26
   6.2 Verification of Software Product Lines with Delta-Oriented Slicing 26
      6.2.1 Delta-Oriented Programming .................. 27
      6.2.2 Delta-Oriented Specification ................ 27
      6.2.3 Delta-Oriented Slicing ....................... 27
      6.2.4 Implementation ................................ 27
      6.2.5 Relation to HATS ABS ....................... 28
      6.2.6 Towards Delta-Oriented Verification ......... 28
   6.3 Interleaving Partial Evaluation and Symbolic Execution .... 28

7 Conclusions ............................................. 30

Bibliography .............................................. 30

Glossary .................................................. 34
A The Rules of ABS DL

A.1 Update Simplification Rules ........................................ 36
A.2 Sequential Constructs .............................................. 37
  A.2.1 Pure Expressions .............................................. 37
  A.2.2 Statements ................................................ 37
A.3 Concurrent Constructs ............................................... 38
  A.3.1 History, Messages & Implicit Attributes ...................... 38
  A.3.2 Asynchronous Method Call .................................. 39
  A.3.3 await ...................................................... 39
  A.3.4 get ........................................................ 39
  A.3.5 new and new cog ........................................... 40
  A.3.6 Synchronous Method Call .................................. 40
  A.3.7 return ..................................................... 40

B Papers on Program logics for ABS ................................. 41

B.1 A verification system for distributed objects with asynchronous method calls ........................................ 41
B.2 A system for compositional verification of asynchronous objects ......................................................... 62
B.3 Abstract Object Creation in Dynamic Logic ..................... 84

C Papers on Incremental verification ................................ 100

C.1 Lazy Behavioral Subtyping .......................................... 100
C.2 Compositional Verification of Temporal Safety Properties .... 158
C.3 Compositional Verification of Software Product Lines ........ 189
C.4 Verification of Software Product Lines with Delta-oriented Slicing .............................................. 205
C.5 Interleaving Symbolic Execution and Partial Evaluation ....... 221
Chapter 1

Introduction

In this deliverable we report on our work about verification of behavioral properties. The presented work involves two major components, namely, development of a program logic for core ABS based on symbolic execution and the investigation of techniques that allow incremental verification. Both topics required basic research and shall provide the ground for later tasks extending and implementing these logics and ideas for full ABS (Task 4.3 “Correctness” and Task 1.4 “Analysis”).

Core ABS being a modeling language which is inherently concurrent and especially suited for distributed systems posed a number of challenges: first, it required to develop a highly modular and compositional program logic. To achieve the required degree of compositionally and modularity, concepts like class and interface invariants and histories (trace of messages) had to be utilized and combined. Second, for subsequent use in other tasks concerned with testing, debugging and visualisation, the program logic had to be based on symbolic execution. This means it should as far as possible realize a symbolic interpreter for ABS. A symbolic interpreter executes programs like a normal interpreter except that instead of concrete values, symbolic values are used and manipulated by the program. We describe our work on a program logic for core ABS in Chapter 2 which summarizes and adapts the work published in [3, 4] (reprinted in Appendix B.1 and Appendix B.2).

The concept of histories is central to achieve a compositional and modular program logic for ABS. An interesting subproblem is to model the history in an efficient way. In Section 2.3.1 we briefly report on intermediate results of our investigations.

Another problematic issue to achieve is a fully abstract program logic. One of the major problems is how to deal with object creation as the domain models used are usually too specific and destroy full abstraction of the logic. Traditional approaches use a constant domain assumption, where all objects, whether created or not, can be accessed on the logic level. Object creation is then modeled by setting a Boolean flag that represents createdness status. This way of modeling object creation has a strong resemblance to object activation. Object activation bears several pitfalls stemming from the need to explicitly model the state of an object. Under certain circumstances it allows to express properties that cannot be observed on the programming language level (e.g., which object had been created before another one). For example, additional system invariants have to ensure that attributes of created objects only refer to other created objects. In addition, one flag is often not enough to encode the lifecycle of an object. But with introducing more flags, their consistency must be ensured and, hence, additional invariants need to be added ensuring a well-formed object state. Consequently, we investigated an alternative approach, which deviates from the constant domain assumption and allows monotonously increasing domains. The approach is sketched in Section 2.3.2 and summarizes the work that is presented in [2] (reprinted Appendix B.3).

In Chapter 3 we describe an approach to detect deadlocks and to ensure that ABS models do not exhibit such undesired behavior. The deadlock analysis is based on contracts and provides input for the tasks concerned with developing the contract language for ABS (Task 1.3 “Analysis”).

The following chapter turns towards incremental verification. Incremental verification is crucial for software systems of a high variability degree like software product lines (SPL). Verification and even more
complex analysis are expensive in terms of time and costs. If it would not be possible to keep local changes
local and to modularize the necessary work, valuable resources are wasted by redundant execution of (sub-
)analyses and each minor change would require to repeat all analyses globally.

The first incremental verification technique to be introduced is a model checking based approach to
modular behavioral verification of procedural languages and software product lines in particular. Chapter 4
presents the framework and toolset CVPP for modular verification. The chapter summarizes the work
published in [28, 44, 43] (reprinted in Appendices C.2 and C.3).

In Chapter 5 we report on our efforts on incremental reasoning related to late binding in object-oriented
class hierarchies. The main motivation for this work is to separate subtyping from code reuse by introducing
two hierarchies: subtyping at the level of interfaces and code reuse at the level of, e.g., class inheritance.
The reported work has been published in [22, 23] (reprinted in Appendix C.1).

Functional verification at the family engineering level, however, will require methodological approaches
that go beyond the verification technologies available so far. This is done in Chapter 6. In Section 6.2 we
declare an incremental verification technique making use of the delta technique used by ABS to model
variability. In the easiest approach a delta represents a feature and describes which classes, methods or
code needs to be added or removed. The idea is to exploit the structural information present in the
deltas to identify the proofs that need to be reproven and also which parts of a proof can be reused as
they are not affected by the delta. The work has been published in [11] (reprinted in Appendix C.4). Section 6.3 summarises a technique that interleaves partial evaluation and symbolic execution steps during
the verification process leading to a considerable speed-up of symbolic execution. This work has been
Chapter 2

A Program Logic for ABS

2.1 Introduction

In this section we report about the work on a program logic for Core ABS. The main technical work has been reported in [3, 4] (Appendix B.1 and Appendix B.2) using a subset of the CREOL [32] language. The chosen subset represents except for some details the core of the distributed concurrent object model of ABS.

We introduce the program logic in Section 2.2 providing an overview of the basic concepts and ideas. For the technical details we refer to the above mentioned papers. The logic is based on Core ABS as reported in [17].

2.2 A Program Logic for ABS

2.2.1 Dynamic Logic

Dynamic logic (DL) [25] is a modal logic. First-order dynamic logic bares several similarities to Hoare logic [26]: programs occur directly as part of DL formulas and not in an encoded form. In contrast to Hoare logic, DL is closed with respect to quantification and allows arbitrary nesting of programs.

To be more precise, CoreABS DL (ABS DL for short) is a sorted first-order logic extended by the two modalities $[·]$ (box) and $⟨·⟩$ (diamond). Let $\phi$ be an arbitrary ABS DL formula, $p$ a program then $[p]\phi$ and $⟨p⟩\phi$ are ABS DL formulas. The intuitive meaning is that $[p]\phi$ is true in a state $s$ if for all terminating runs of $p$ started in $s$ the formula $\phi$ holds in the resulting final states. The formula $⟨p⟩\phi$ is true in a state $s$ if there exists a terminating run such that in its final state $\phi$ holds. For deterministic programs the box modality is equivalent to partial correctness, while the diamond expresses total correctness. The formula

$$\forall x, y; (i = x \land j = y \rightarrow [i = j; i = i-j; j = i+j])$$

"stores" in the logic variables $x, y$ the initial value of the program variables $i, j$ and expresses that in each terminating run of the program inside the box modality, their values have been swapped. We use the opportunity to introduce a few further notions: in ABS DL we distinguish between rigid and non-rigid function/predicate symbols: Rigid symbols do not depend on the state, i.e., their meaning cannot be changed by programs. In the above formula, the logic variables $x, y$ are rigid making their use to store initial values safe as their value cannot be changed by any program. The equality symbol $\equiv$ is an example of a rigid predicate symbol. The arithmetic operators $+, -, /, \ldots$ and comparators $<, >, \geq, \ldots$ whose value should only depend on their arguments, but not on the program state, are typical examples for rigid function symbols.

The values of the program variables $i, j$ should depend on the state in which they are evaluated. Consequently, they are modeled as non-rigid constant/function symbols. In contrast to logic variables they are allowed to occur within programs, but they cannot be bound by quantifiers.

ABS DL utilizes state updates as introduced in [8, 38]. Updates are similar to abstract state machines or generalized substitutions as known from the B method. An elementary update $l := r$ is a pair of a location
(program variable) \( l \) and a term \( r \). Its meaning is equivalent to that of an assignment except for \( r \) being side-effect free. Elementary updates \( u_1, u_2 \) can be composed to parallel updates \( u_1 || u_2 \). A parallel update has the meaning of a parallel assignment. In particular, the updates do not influence each other as the right hand sides are evaluated in the pre-state of the parallel update. In case of a clash where a location \( l \) is updated to different values in \( u_1 \) or \( u_2 \) a last-one-wins semantics is used to resolve the conflict. Updates can be applied on terms or formulas. Let \( u \) be an update and \( \phi \) a formula then \( \{u\} \phi \) is again a formula. Some examples:

- the formula \( \{i := 0\} i \geq 0 \) is valid as \( i \geq 0 \) is evaluated in a state where \( i \) is 0;
- the evaluation of \( \{i := j \mid j := i\} \phi \) in a state \( s \) is equivalent to the evaluation of the formula \( \phi \) in a state \( s' \) which coincides with \( s \) except for \( i \) and \( j \) whose values are swapped;
- the parallel update in \( \{i := 0 \mid i := 1\} \phi \) has a conflict as \( i \) is updated to two different values. As we use a last-one-wins semantics to resolve the conflict, the above formula is equivalent to \( \{i := 0\} \phi \).

### 2.2.2 Sequent Calculus

To reason about ABS programs in ABS DL we use a Gentzen-style sequent calculus. A sequent

\[
\phi_1, \ldots, \phi_m \Rightarrow \psi_1, \ldots, \psi_n
\]

consists of two sets of formulas where \( \phi_1, \ldots, \phi_m \) is called the sequent’s antecedent and \( \psi_1, \ldots, \psi_m \) the sequent’s succedent. Its meaning is that of an implication \( \bigwedge_{i=1}^{m} \phi_i \rightarrow \bigvee_{j=1}^{n} \psi_j \). When formulating rules we use often the short-form \( \Gamma, \phi \Rightarrow \Delta \) where only the formulas of interest are mentioned explicitly (here: \( \phi \)) and all others are collapsed into the schemavariables \( \Gamma \) and \( \Delta \) representing formula sets. A sequent calculus rule is of the general form

\[
\frac{\text{premises}}{\text{name} \Gamma_1 \Rightarrow \Delta_1 \mid \ldots \mid \Gamma_n \Rightarrow \Delta_n \Rightarrow \Delta} \tag{2.1}
\]

and called **sound** if the validity of the conclusion follows from the validity of the conjunction of the premises. The sequent rules are used to construct a sequent proof. The proof object of a sequent calculus is a tree, each of whose nodes is annotated by a sequent. The tree is constructed by successively applying sequent rules to its leaves. A sequent rule is applied to a leaf node \( \text{ln} \) by matching its conclusion to the leaf’s sequent and then adding the instantiated premises as new leaf nodes to node \( \text{ln} \). Figure 2.1 shows a selection of typical first-order rules.

\[
\begin{align*}
\text{andLeft} & \quad \Gamma \Rightarrow \phi, \Delta \\
& \quad \Gamma \Rightarrow \psi, \Delta \\
\Rightarrow & \quad \Gamma \Rightarrow \phi \land \psi, \Delta \\
\text{impRight} & \quad \Gamma, \phi \Rightarrow \psi, \Delta \\
\Rightarrow & \quad \Gamma \Rightarrow \phi \rightarrow \psi, \Delta \\
\text{allRight} & \quad \Gamma \Rightarrow \psi, \Delta \\
& \quad \Gamma \Rightarrow \forall v. \phi, \Delta \\
\end{align*}
\]

Figure 2.1: First-order sequent calculus rules.

Several rules in the sequent calculus rule are rewrite rules, which simply replace a subterm/-formula by different one. To keep the notational overhead small we refer to the same notation as \( \phi' \) using

\[
\frac{\text{name} \phi}{\phi'}
\]

to express a rule that replaces an arbitrary occurrence of \( \phi \) by \( \phi' \). Several rules for reasoning about programs are rewrite rules ans are applicable to both, box and diamond modalities. We use in these cases the symbol \( \langle \cdot \rangle \) which can be instantiated to either the diamond or box modality.
2.2.3 Functional ABS

ABS contains a functional sublanguage which allows to define algebraic data types. The functional language fragment is a pure language, i.e., it is not allowed to have side-effects. Another restriction is that functions are not first-class citizens, i.e., it is not possible to pass functions as arguments.

In ABS DL the functional fragment is integrated as follows: For each data type definition

\[
data \text{name}\{\text{cons}_1, \ldots, \text{cons}_n}\end{align*}
\]

there is a sort of the same name and a function symbol for each constructor with a matching signature. For each function definition

\[
def \text{func}(\text{params}) = t[\text{params}]
\]

there is a rewrite rule

\[
\text{func}_{\text{def}} \quad \frac{t[\text{params}/\bar{x}]}{\text{func}(\bar{x})}
\]

replacing an occurrence of a function application by its definition. The case expression is canonically replaced by a nested conditional term \((\phi \land t_1 : t_2)\) which evaluates to \(t_1\) if \(\phi\) is satisfied otherwise to \(t_2\). The nesting ensures that we respect the first match strategy.

We give a brief example. Assume the following definition of the Gaussian sum:

\[
def \text{gSum nth}(\text{Int} \ n) = \text{case } n \{
0 \Rightarrow 0
_ \Rightarrow n + \text{gSum}(n-1)
\}
\]

The corresponding rewrite rule is then:

\[
g\text{Sum}_{\text{def}} \quad \frac{(x = 0 ? 0 : x + \text{gSum}(x - 1))}{\text{gSum}(x)}
\]

Locally defined data-types or functions are transformed to global definitions. Renaming ensures that no name clashes occur.

Formally, we define a function \(\tau\) which maps pure expressions, i.e., expressions without side-effects, to a term of the logic. For most pure expressions this function is mostly the identity with exception of case and let expressions. Further, as it proves convenient, we define an embedding function \(\iota\) which translates a term of type \(\text{Bool}\) into a formula (basically: let \(t\) be a term of type \(\text{Bool}\) then \(\iota(t) = (t = \text{True})\)). In the following rules we skip \(\tau\) and \(\iota\) to keep the notation clean.

We do not handle parametric types at the moment. There is no principal difficulty with them, but we decided to concentrate for the moment on the most crucial aspect of Core ABS which is concurrency. Parametric types will be added later on.

2.2.4 Sequential ABS

As a start we give the ABS calculus rules for the basic sequential programming constructs. The most basic statement is the assignment rule

\[
\text{assignPureExp} \quad \frac{\{U\} \{x := e_p\} \{\omega\}_\phi}{\{U\} \{x=e_p; \omega\}_\phi}
\]

where \(U\) is an update, \(x\) is a program variable (or field), \(e_p\) stands for a pure expression and \(\omega\) is the remaining program to be symbolically executed. As can be seen, assignments are successively turned into updates which accumulate in front of ABS DL formulas. There are simplification rules that turn sequences of
updates into parallel updates, see Appendix A. The actual computation of the effects of an update is delayed until symbolic execution has finished and a first-order post condition is reached. To increase readability of the rules, we will omit the leading updates $U$ in the following, but assume their presence implicitly.

As a second example we present the sequent rule formalizing the semantics of the conditional statement. On encounter of a conditional statement as first-active statement the rule

$$\text{ifSplit} \quad e_p \rightarrow \{p; \omega\} \phi \quad \land \quad \neg e_p \rightarrow \{q; \omega\} \phi$$

is applied. This sequent rule captures the control flow of the ABS program precisely: If the guard expression $e_p$ (in ABS always a pure expression) of the conditional statement evaluates to True in the current state then execution continues with the conditional’s then branch $p$ followed by the remaining program $\omega$, otherwise $q$ has to be executed also followed by $\omega$. The rule does not split immediately, but postpones the split by connecting both execution paths with a conjunction. This gives the calculus the chance to detect an infeasible path by applying simplification rules to $e_p$ and avoids unnecessary branching.

We give one more example on the rules for sequential ABS, namely for local variable declarations in the version without initializer:

$$\text{varDeclNoInit} \quad \{\hat{x} := \text{null}\} \{\omega[x/\hat{x}]\} \phi \\ \{T_x; \omega\} \phi$$

The type of a program variable $x$ declared by a local variable declaration must be of reference type $T$. On symbolic execution of such a variable declaration the local variable and all its occurrences are renamed (if necessary) to avoid name clashes due to scoping. The new name is denote by $\hat{x}$ in the above rule and $\omega[x/\hat{x}]$ denotes the renaming of the variable occurrences in the rest program. ABS assigns those values then implicitly the value null which is reflected by the elementary update in the premise of the rule.

For all other rules treating sequential constructs like assert, skip etc. see Appendix A. The rules concerning loops and method calls are mentioned later in the following sections focusing on object-orientation and concurrent constructs (see also Appendix A).

### 2.2.5 Object-Oriented ABS

#### History, Interface- and Class Invariants

The program logic presented aims to be fully modular and compositional. Consequently, we consider only one object (called this-object) and one method of that object’s class at any given time. Compositionality ensures us then that the proven properties carry over to all objects of the class and are independent of the environment as long as the interface invariants of the communication partners are not violated. Interface invariants are the only information about the behavior of other objects that we may use for verification. Further, the only information we have about other objects is their interface type—we cannot make any assumptions about their implementing class. Another concept that we need are communication histories as introduced in [31]. A communication history $H$ is a sequence of messages representing the trace of the system. The message types it can contain are method invocation messages and method completion messages as well as object creation messages.

Every interface $I$ specifies an interface invariant $IInv(I)(H)$ which is a formula that (usually) describes the functional behavior of the implementing interfaces with respect to the current history $H$. To be compositional the proof obligations (formulas to be proven) have to restrict the history to those messages that are related to the caller (this) and the callee.

Similar to interfaces, each class $C$ specifies a so-called class invariant $CInv(C)(H, \bar{A})$ which consists of three different components: one relating the value of the attributes $\bar{A}$ to the communication history and the two others conjunctively adding the invariants of the implemented interfaces projected to this as callee and those interface invariants that are invoked by the class $C$. 


Finally, there exist a number of predicates defined on the history that check for certain properties. The most important is the predicate \( Wf(H) \) which checks whether the history is well-formed, e.g., each completion invocation message must have a corresponding invocation message as one of its predecessors, etc. There are further predicates such as \( Comp(H, l) \) which checks whether a certain completion message specified by a unique label (future) is present in the history.

**Concurrency**

ABS distinguishes between two different kinds of method calls, namely asynchronous and synchronous method calls. Synchronous method calls are only allowed within a concurrent object group. In their case control is immediately passed and the current thread waits for the result. In the case of asynchronous calls, control is not passed and the current task continues until the result value of the method call is explicitly queried.

We discuss first the rule for asynchronous method invocations. An asynchronous method invocation spawns a new thread while the current one continues. A so-called future is returned immediately, which allows us to query availability and poll the result at a later point. In the rule

\[
\text{asyncMC} \quad \Gamma \implies o \neq \text{null} \rightarrow \{\text{block}; \omega\} \phi, \Delta \\
\Gamma \implies \neg(o \neq \text{null}) \rightarrow \{r := (\text{this}, o, m, i)\} \{U_{invoc}^H\} (Inv(I)(H)^{\text{this}, o} \land \{\omega\} \phi), \Delta \\
\Gamma \implies \{r = o!m(\text{args}); \omega\} \phi, \Delta
\]

the symbol \( r \) is a variable of type Fut, representing a future object, \( \text{args} \) is a sequence of pure expressions, and \( I \) is the interface of \( o \). The update \( U_{invoc}^H \) updates the global history variable \( H \) such that the new history \( H \) is a continuation of the current one containing the just issued invocation message. Formally, \( U_{invoc}^H \) is defined as \( H := \text{some } H. (H \leq H \land Invoc(H, r, \text{args})) \) where \( \text{some} \), like Hilbert's \( \epsilon \) operator, returns an arbitrary term for which the condition in its scope is true. The formula in the scope of the update ensures that at call time the interface invariants of the callee holds with respect to the program history.

The block statement is not an ABS statement, but is only used internally to keep the presentation of the rules compact. It has no other side-effects than blocking the current thread without releasing control and renders as first active statement each box formula trivially true and each diamond formula false (see Appendix A).

Polling the result of an asynchronous method call is achieved in ABS via the get primitive. In the rule

\[
\text{get} \quad \Gamma \implies \{U_{comp}^{H,x}\} [\omega] \phi, \Delta \\
\Gamma \implies [x = l.get; \omega] \phi, \Delta
\]

the update \( U_{comp}^{H,x} \) is defined as \( H, x := \text{some } H, y.(H \leq H \land Inv(I)(H)^{\text{this}, l, \text{callee} \land Comp(H, l, y)}) \) where \( I \) is the interface of the callee which can be extracted from \( l \).

The rule assigns to the global history variable \( H \) its history continuation and it assigns to the result variable \( x \) the result value as recorded in the completion message of the history.

The ABS statement await \( 1? \) provides the possibility to check on the program level whether a future can be polled safely (i.e., without blocking the execution) or not. The await statement checks for the existence of a result and yields control if the result is not yet available. On the program logic level the following rule axiomatizes this behavior:

\[
\text{awaitComp} \quad \Gamma \implies CIInv(C)(H, \tilde{A}), \Delta \\
\Gamma \implies \{U_{H,\tilde{A}}\} (Comp(H, r) \rightarrow [\omega] \phi), \Delta \\
\Gamma \implies [\text{await } r?; \omega] \phi, \Delta
\]

Here, \( \tilde{A} \) is the sequence of all attributes defined in the class \( C \) of the this object. The update \( U_{H,\tilde{A}} \) stands for the update \( H, \tilde{A} := (\text{some } H, \tilde{y}.CIInv(C)(H, \tilde{y}) \land H \leq H) \) which assigns arbitrary values to the history variable and all attributes of the this object such that the class invariant is satisfied and the new
history is a continuation of the old history. This way we erase all possibly outdated information about the values of the attributes and history, keeping only the information we know to be true.

We turn now towards the rule for synchronous method calls. In contrast to asynchronous method calls, synchronous calls are only allowed within an object group. Further, synchronous calls provide the possibility for re-entrance, so we have to ensure that the class invariant holds and not merely the interface invariants. In addition, we have to eliminate all knowledge of the concrete values of the attributes and only rely on the information provided by the class invariant. The following rule captures this semantics (using the same abbreviations and conventions as above):

\[
\text{syncMC} \quad \Gamma \Rightarrow o \neq \text{null} \rightarrow \nu \{b; \omega\} \phi, \Delta
\]
\[
\Gamma \Rightarrow \neg(o \neq \text{null}) \rightarrow \text{SameCOG}(H, \text{this}, o) \wedge \text{CInv}(C)(H, \bar{A}), \Delta
\]
\[
\Gamma \Rightarrow \neg(o \neq \text{null}) \rightarrow \{l := (\text{this}, o, \text{m}, i)\} \nu \{\text{U}H, \bar{A}\} \nu \{r = l.\text{get}; \omega\} \phi, \Delta
\]

- **SameCOG** is a predicate defined over a history and checks whether two objects belong to the same concurrent object group.
- **l** is a new variable of type \text{Fut}.

Finally, loops are treated using loop invariants \text{inv}. The loop invariant rule is conventional and requires to prove that the invariant is initially valid, is preserved by the loop body, and that \text{inv} is strong enough to prove the property of interest after exiting the loop. The update \text{U}_{H, \text{mod}} is an anonymizing update as defined above for synchronous method calls and the \text{await} statement, but where \text{mod} are the variables and fields whose value is potentially modified inside the loop body.

\[
\text{loop} \quad \Gamma \Rightarrow \text{inv}, \Delta
\]
\[
\Gamma \Rightarrow \nu \{H_{\text{mod}}\} \{e_p \wedge \text{inv}\} \nu \{p\} \text{inv}, \Delta \quad \text{(init. valid)}
\]
\[
\Gamma \Rightarrow \nu \{H_{\text{mod}}\} \{e_p \wedge \text{inv}\} \nu \{p\} \phi, \Phi \quad \text{(use case)}
\]

The rules for object creation and return of a method basically only extend the history by an object creation, respectively, a method completion message. In case of object creation we have to distinguish between two semantics depending on whether the newly created object belongs to the same concurrent object group or to the one of the current object. The difference is merely in the determination of the object group identifier in the creation message which has to be either chosen freshly or taken from the \text{this} object. The details appear in Appendix \ref{appendix:A}.

### 2.3 Optimizations

In the previous sections we presented a program logic for Core ABS as reported in Deliverable D1.1a “Report on the Core ABS Language” \cite{Hats2023}. In this section we discuss possible optimisations in the modeling of central aspects like histories (Section \ref{2.3.1}) and object creation (Section \ref{2.3.2}).

The alternative history model requires less quantification and has therewith the potential to increase significantly the automation of the reasoning process. The alternative modeling of object creation promises benefits with respect to full abstraction and reduced complexity due to simpler system properties such as that only created objects are reachable from created ones.

These optimisations have not yet been incorporated into the ABS program logic as the full consequences of such an integration have not yet sufficiently been explored and the exact modeling of histories is ongoing work and not fully stabilised. In particular, the topic of modeling histories in an efficient manner will be continued as part of Task 4.3 “Correctness” and most likely be integrated into the ABS program logic as one result of that task.
2.3.1 4-Event Histories

As explained previously, the execution of a distributed system can be represented by its communication history or trace; i.e., the sequence of observable communication events between system components [10, 27]. At any point in time the communication history abstractly captures the system state [15, 14]. In fact communication traces are used in semantics for full abstraction results (e.g., [29, 1]). A system may be specified by the finite initial segments of its communication histories. Let the local history of an object reflect the communication between the object and its surroundings. A history invariant is a predicate over the communication history, which holds for all finite sequences in the prefix-closure of the set of possible histories, expressing safety properties [5].

In on-going work, we develop a new partial correctness proof system for the ABS language. A class is specified by a class invariant over the class attributes and the local communication history. Thus, the class invariant directly relates the internal implementation of the class to its observable behavior. The proof system is derived from a standard sequential language by means of a syntactic encoding, extending a transformational technique originally proposed by Olderog and Apt [36] to use non-deterministic assignments to the local history to reflect the activity of other processes at processor release points. This way, the reasoning inside a class is comparable to reasoning about a simple sequential while-language extended with non-deterministic assignment, and amounts to proving that the class invariant is maintained from one release point to another. By hiding the internal state, an external specification of an object is obtained as an invariant over the local history. In order to derive a global specification of a system composed of several objects, one may compose the specifications of different objects. Modularity is achieved since history invariants can be established independently for each object and composed at need.

This work extends and improves previous work by UIO [20, 21] as well as recent work by Ahrendt and Dylla [3, 4]. Technically, we here develop a novel system based on a four-event semantics for asynchronous method calls, which introduces disjoint alphabets for the local histories of different objects. Compared to previous work, this allows us to formulate a much simpler, Hoare-style proof system for object-oriented languages based on concurrent objects with asynchronous method calls, and to reduce the complexity of reasoning about such concurrent programs by significantly simplifying the formulas in terms of the number of needed quantifiers.

2.3.2 Abstract Object Creation in Dynamic Logic

The ABS provides a high-level specification of object creation which abstracts away from the underlying representation of objects and the implementation of the generation of new object identities. At the abstraction level of the ABS, objects are described as instances of their classes, i.e., the classes provide the only operations which can be performed on objects. Moreover, these operations can only be performed on the created objects, the objects not (yet) created do not exist and therefore can also not be referred to by any construct of the ABS. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the ABS language itself. Most formalisations of object-oriented programs, like embeddings into the logic of higher-order theorem provers PVS [9] and Isabelle [33], or dynamic logic as employed in the KeY theorem prover [8], however, use an explicit representation of objects. Object creation is then formalized in terms of the information about which objects are in fact created. Such an explicit representation of objects additionally requires an axiomatization of certain consistency requirements, e.g., the global invariant that the values of the fields of created objects only refer to created objects. These requirements pervade the correctness proofs with the basic case distinction between “to be or not to be created”.

In order to investigate the possibility of a fully abstract logic regarding the creation of new objects, we developed a weakest precondition calculus that employs abstract object creation in dynamic logic, the logic underlying the KeY theorem prover [8]. This representation allows to both specify and verify properties of objects at the abstraction level of the ABS. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties.
The generalization of Hoare logic to dynamic logic is of particular interest because it allows for the specification of properties of dynamic object structures which cannot be expressed in first-order logic, like reachability. In Hoare logic such properties require quantification over (finite) sequences or recursively defined predicates in the specification language which seriously complicates both the weakest precondition calculus and the underlying logic. In dynamic logic we can restrict ourselves to first-order quantification and use the modalities to express, for example, reachability properties.

An interesting consequence of the abstraction level of the specification language is the dynamic scope of the quantification over objects because it is restricted to the created objects and as such is also affected by object creation. However, we show that the standard logic of first-order quantification also applies in the presence of (object) quantifiers with a dynamic scope.

Further, we show how to symbolically execute abstract object creation in KeY. In general, symbolic execution in KeY accumulates the effect of the assignments executed during computation in the form of updates (see Section 2.2.1) in front of modal formulas. Updates are continuously simplified by suitable first-order rewrite rules. However, we cannot simply accumulate abstract object creation expressions, because their effect can only be processed at the time when they are actually applied to the formula in their scope. We show how to solve this problem by the introduction of fresh logical variables which are used as temporary place holders for the newly created objects. The use of these place holders together with the fact that we can always anticipate object creation allows to symbolically execute abstract object creation. Technical details are found in the paper [2], see Appendix B.3.

A denotational semantics of CREOL is presented in [4] and will provide important input for Task 4.3 “Correctness” concerned with global correctness properties.
Chapter 3

Analysis of deadlocks in ABS

The main new feature of the ABS concurrency model are concurrent object groups (COGs): collections of objects that perform collaborative work. The group abstraction is an encapsulation mechanism that is convenient in distributed programming in several circumstances. For example, in order to achieve continuous availability, or load balancing through replication, or for retrieval services of distributed data. In these cases, in order to ensure consistency, the group abstraction must define suitable protocols to synchronize group members. As usual with synchronization protocols, it is possible that object groups may exhibit deadlocks, which are particularly hard to discover in the ABS context because of the two encapsulation levels of the systems (the object and the group level).

In a model with COGs and cooperative scheduling a typical deadlock situation occurs when two active tasks in different COGs are waiting for each other to return a value. We develop a technique for the analysis of deadlocks in programs based on contracts—abstract descriptions of behaviors that contain the information to detect deadlocks. In particular, we define an inference system for associating a contract with every method of the program and to each expression to evaluate. Then we define an algorithm—the dla algorithm—that collects information about group dependencies. The presence of circularities in the result of dla reveals the potential presence of deadlocked computations.

3.1 Examples

We introduce our formal development of deadlock analysis by discussing a couple of expressions that exhibit deadlocks. Let us consider a simple class declaration:

```java
class C {
    C m1(C b, C c) { return (b!m2(c)).get; ;}
    C m2(C c) { return (c!m3).get; ;}
    C m3 { return new C ;}
}
```

Let us first suppose we evaluate the following expression: `(new C)!m1(new cog C, new cog C)`, which results in the creation of a new object, referred at runtime as o1, in the current group, say G1, and of two new objects in two new groups, say o2 in G2 and o3 in G3. The invocation of m1 triggers a new task in G1, performing the invocation of o2!m2(o3) and blocking, by means of the get operation, until the evaluation of m2’s body produces a value. The body of m2 will be evaluated in a new task within the group G2, by executing (o3!m3).get. Figure 3.1 illustrates the stage in the execution when the body of m3 has terminated, and the newly created object o4 is about to be returned to the task in G2 and then to the task in G1 and all the locks are going to be released.

Let us now suppose we evaluate the following expression: `(new C)!m1(new cog C, new C)`. Now we have just the two groups G1 and G2 and the object o3 belongs to the group G1. This causes the invocation of m3 within m2’s body to block indefinitely, because the lock on G1 is kept by the task performing m1. See
Deadlocks may be difficult to discover when they are caused by schedulers’ choices. For example, consider the following class definition:

```java
class D {
    D m1(D b, D c) { return b!m2(c); c!m2(b); }
    D m2(D c) { return (c!m3).get; }
    D m3 { return new D; }
}
```

Let us suppose we evaluate the following expression: `(new D)!m1(new cog D, new D)`. The deadlock in this case is not caused, as before, by a cyclic use of get, but by two subsequent asynchronous calls that trigger two parallel executions of method `m2` in groups `G1` and `G2`, respectively. See Figure 3.2, where we denote with a broken line the running tasks, with broken arrows the asynchronous invocation, and with plain arrows the get dependencies. Notice that a different evolution of this computation may lead to a deadlock-free scenario: for example, if the execution of method `m3` on `o2` completes before the call of the same method on `o3` is performed. Our algorithm discovers also these potential deadlocks and reports them. In this case the (potential) deadlock could be avoided by putting a synchronization point between the two asynchronous calls, such as a get or an await operation.

### 3.2 Contracts

`Contracts γ, γ′, ...` are terms defined by the grammar below:

\[
γ ::= ε | C.m : G(\overline{G}); γ | C.m^γ : G(\overline{G}); γ | C.m^α : G(\overline{G}); γ
\]

The sequence `γ` collects the method invocations inside expressions. In particular, the items of the sequence may be empty, denoted `ε`; or `C.m : G(\overline{G})`, specifying that the method `m` of class `C` is going to be invoked on an object of group `G` and with arguments of group `\overline{G}`; or `C.m^γ : G(\overline{G})`, a method invocation followed by a get operation; or `C.m^α : G(\overline{G})`, a method invocation followed by an await operation.
For example, the contract \texttt{C.m:G(); D.n:G'}(); \texttt{D.p:G''(); E.p:G''();} defines three method invocations on different groups; the second invocation is followed by a \texttt{get} and the third one by an \texttt{await}.

\textit{Method contracts} are of the form \texttt{G(G\{\gamma\} G')}, where \texttt{G}, \texttt{G'} are pairwise different group names—\texttt{G(G)} is the \textit{header}, \texttt{G'} is the \textit{returned group}, and \texttt{\gamma} is a \textit{contract}. A method contract \texttt{G(G\{\gamma\} G')} binds the group of the object \texttt{this} and the group of the arguments of the method invocation in the sequence \texttt{\gamma}. The returned group \texttt{G'} may belong to \texttt{G}, \texttt{G'} or not, that is, it may be a new group created by the method.

For example, let \texttt{\gamma = C.m:G(); D.n:G'(); E.p:G''();} in (i) \texttt{G(G',G''){\gamma} G''} and (ii) \texttt{G(G'){\gamma} G''}. In case (i) every group name in \texttt{\gamma} is \textit{bound} by names in the header. This means that method invocations are bound to either the group name of the caller or to group names of the arguments. This is not the case for (ii), where the third method invocation in its body and the returned group are a group name that is unbound by the header. This means that the method with contract (ii) is creating an object of class \texttt{E} belonging to a new group—called \texttt{G''} in the body—and is performing the third invocation to a method of this object.

### 3.3 Deadlock Analysis

Contracts retain the necessary information about deadlocks of an expression and the analysis may be safely reduced to them, abstracting away from all the other details.

The presence of a circularity in a configuration signals the presence of two threads mutually waiting for the release of the group lock of the other.

Formally, a dependency between a group name \texttt{G} and another one \texttt{G'} is a pair \texttt{(G, G')}. Sets of pairs of group names are noted by \texttt{G}. A set \texttt{G} is \textit{acyclic}, written \texttt{acyclic(G)}, if the transitive closure of \texttt{G} does not contain any pair \texttt{(G, G')}.

Look at the following annotation with contracts of the class \texttt{C} above:

```java
class C {
    G(G',G''){C.m2:G'(G'')} { C m1(C b, C c) { return (b!m2(c)).get ;} }
    F(F'{}C.m3:F';) C m2(C c) { return (c!m3).get ;} }
    E(){} C m3 { return new C ;}
}
```

The algorithm \texttt{dla} associates to the expression \texttt{(new C)!m1(new cog C, new cog C, new C)} the set of pairs \{ (G,G'), (G',G'') \}, which intuitively corresponds to the arrows in Figure 3.1 presents no circularity and, therefore, no deadlock. In contrast, for the expression \texttt{(new C)!m1(new cog C, new cog C, new C)} the algorithm computes \{ (G,G'), (G',G) \} which has a circularity.

Next, consider the class \texttt{D} with the following annotation:
In this case the algorithm yields for the expression `(new D) ! m_1(new cog D, new D)` the set `{ (G,G') , (G',G''), (G,G''), (G'',G') }` which has a circularity in G'.

```java
class D {
    G(G',G'') {D.m_2 : G'(G''); D.m_2 : G''(G'); }
    D m_1(D b, D c) { return b ! m_2(c); c ! m_2(b); }
    F(F') {D.m_3 : F';} D m_2(D c) { return (c ! m_3).get; }
    E() {  }
}
```
Chapter 4

A Framework and Toolset for Modular Verification

In modern computing systems, code changes frequently. Modules (or components) evolve rapidly or exist in multiple versions customized for various users, and in mobile contexts, a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready-made off-the-shelf components, and each component may be dynamically replaced by a new one that provides improved or additional functionality. The static and dynamic variability, especially when software product lines are involved, makes it more important to provide formal correctness guarantees for the behavior of such systems, but at the same time also more difficult. Modularity of verification is a key to providing such guarantees in the presence of variability.

In the next paragraphs, we describe the general framework which has been presented in [28]. Its specialization and application towards modular reasoning about software and to cope with the high variability present in software product lines are described in Section 4.1 and Section 4.2.

Modular Verification

In modular verification, correctness of the software components is specified and verified independently (locally) for each module, while correctness of the whole system is specified through a global property, the correctness of which is verified relative to the local specifications rather than relative to the actual implementations of the modules. It is this relativization on local specifications that enables verification of global properties in the presence of static and dynamic variability. In particular, it allows an independent evolution of the implementations of individual modules, only requiring the re-establishment of their local correctness.

The CVPP Tool Set

The CVPP tool set is designed to tackle exactly this kind of verification problems by supporting an algorithmic technique for compositional verification. Its focus is on control-flow safety properties of programs with (possibly recursive) procedures. Such properties typically describe sets of allowed sequences of method invocations, and are conveniently expressed in temporal logic. The underlying program model is that of flow graphs, abstracting completely from program data. This rather severe restriction on the program model is imposed by the maximal model construction that is the core of our modular verification technique (see [24] for a proof of soundness and completeness for this program model). However, the model can be enhanced with exception information or multi-threading. A high-level overview of CVPP’s architecture is shown in Figure 4.1 where rounded boxes denote data formats, squared boxes tool components, and dashed lines denote external formats or tools.

Temporal Safety Properties

Abstracting away from all data may seem like a severe restriction, but still many useful properties can be expressed, such as: within atomic transactions, there are no calls to non-atomic methods; in a voting system, candidate selection has to be finished, before the vote can be confirmed; a method that changes sensitive data is only called from within a dedicated authentication method, i.e.,
Unauthorized access is not possible; or finally, in a door access control system, the password has to be checked before the door is unlocked, and it can only be changed when the door is unlocked. Extending the technique with data will allow for a significantly wider range of properties and possible applications. However, this needs to be carefully combined with abstraction techniques to control the complexity of verification. Such an extension will be investigated in future work.

**Verification Scenarios** The most general verification scenario supported by our method is for open system verification, where some components are given by an implementation (referred to as concrete components), while others are only given by a specification (abstract components). This can typically happen with dynamically reconfigurable or evolving software, where some components are either not known or simply not statically fixed at verification time. Thus, verification of a global property of an open system has to be relativized on the local specifications of the abstract components. The implementations of the abstract components, once available, are checked against their local specifications.

In the modular software design paradigm the goal is to verify the modules of a software system locally, i.e., independently of each other, and then to combine the local correctness arguments into a global correctness proof of the whole system. In our verification framework, modular verification is simply an instance of the more general case of open system verification described above, with modules as components and where all components are abstract.

One can view the notion of module on different levels of granularity. One (rather extreme) case in procedural programming languages is when every procedure itself is considered a module and is equipped with a specification. In this case we obtain procedure-modular verification, similar to many Hoare logic based verification approaches.

**Adaption to HATS ABS** The tool set currently takes as input annotated programs in Java bytecode. We believe that this is important for HATS. We are currently investigating the adaptation to ABS. The program model we use is a very abstract one and can be used to model the control flow of ABS models. However, the way we capture concurrency in our program model is based on locks (following the Java concurrency model).

Hence, the two main issues to be resolved in order to adapt CVPP to ABS are the extension of our program model to the concurrency model of ABS, and the adaptation of our program model extraction facility to ABS models as input.

### 4.1 Specialization: Procedure-Modular Verification

The technical details to our approach presented in this section can be found in [44], which is an extended version of [43]. Both papers are reprinted in the Appendix C.2.

One popular framework for modular specification and verification of software is provided by Hoare logic, where it is natural to take the individual procedures as modules, in order to achieve scalability. While Hoare logic allows the local effect of invoking a given procedure to be specified, temporal logic is better suited for
capturing its interaction with the environment, such as the allowed sequences of procedure invocations. We show that procedure-modular verification is also appropriate for safety temporal logic: for each procedure the local property specifies its legal call sequences, while the system’s global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

```java
// @global_ltl_prop: even -> X ((even && !entry) W odd)
public class EvenOdd {
    /** @local_interface: required odd
     * @local_ltl_prop: G (X (!even || !entry) && (odd -> X G even)) */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }

    /** @local_interface: required even
     * @local_ltl_prop: G (X (!odd || !entry) && (even -> X G odd)) */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

Figure 4.2: A simple annotated Java program.

The ProMoVer Wrapper Tool To support our approach, we have developed a fully automated verification tool, ProMoVer. It takes as input a Java program annotated with global and method-local correctness assertions written in temporal logic, as shown in Figure 4.2 and automatically invokes a number of tools from CVPP, to perform the individual local and global correctness checks. Essentially, ProMoVer is a wrapper that performs a standard verification scenario in the general tool set, which demonstrates that procedure-modular verification of temporal safety properties can be applied automatically. Figure 4.3 gives a schematic overview of how ProMoVer wraps up the individual CVPP tools. ProMoVer is implemented in Python and can be tested online via a web interface [45].
into two independent subtasks: a check that each method implementation satisfies its local property, and a check that the composition of local properties entails the global property. Notice that the second subtask only relies on the local properties and does not require the implementations of the individual methods. Thus, changing a method implementation does not require the global property to be reverified, only the local property. If the second subtask fails, PROMoVER provides a counter example in the form of a program behavior that violates the respective property.

**Optimizations** To make the annotation procedure comparatively lightweight, together with PROMoVER we also provide a facility to extract a candidate local property specifying the legal call sequences by means of static analysis, given a concrete procedure implementation. A user then only has to inspect the extracted specifications and potentially remove superfluous constraints in order to accommodate possible evolution of the code.

A further reduction of effort is achieved by focusing on the public procedures only, the private ones being considered merely as an implementation means. Finally, PROMoVER also provides proof storage and reuse: only the properties that are affected by a change (either in implementation or in specification) are reverified, all other results are reused.

**Practical Evaluation** We evaluated the validity of the approach on some typical Java Card e-commerce applications. Such security-relevant applications are an important target for formal verification techniques. For these applications we verify the absence of calls to non-atomic methods within transactions. Such properties, specifying legal sequences of calls to security-related methods, are an important class of platform-specific security properties. The PROMoVER web interface allows the user to verify similar platform-specific security properties, for which a ready-made formalization is provided.

### 4.2 Specialization: Verification of Software Product Lines

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to adapt to their application context. Software product line engineering aims at developing a set of systems variants with well-defined commonalities and variabilities by managed reuse in order to decrease time to market and to improve quality. During family engineering reusable core assets are developed that are used to realize the actual products during application engineering.

This section gives an overview of our modifications to the modular verification approach to meet the requirements in a software product line setting. The technical details can be found in [41] (reprinted in Appendix C.3).

**Hierarchical Variability Models** A software product line can be described by a hierarchical variability model, as shown in Figure 4.4 illustrating a product line of cash desks. In this model, on each hierarchy level the commonalities of the products are specified in a common core, while the variabilities are represented by explicit variation points. Each variation point is associated with a set of variants that represent choices for realizing the variation points in different products. A variant can itself contain commonalities defined in a common core and variabilities specified by variation points introducing a new level of hierarchy.

**Challenges for Verification** The number of products defined by a hierarchical variability model is exponential in the size of the model. This exponential explosion of the number of products poses serious problems to ensure critical product requirements by static analysis or other formal verification techniques. In general, it is unfeasible to verify all products individually. Formal verification techniques would only scale if their complexity is linear in the size of the hierarchical variability model and not in the number of products, which is exponential in the size of the variability model. In order to achieve this scalability, the verification techniques have to be compositional allowing to relativize the product properties towards properties of variation points.
Our Approach  We propose simple hierarchical variability models (SHVM) as a novel way to specify products consisting of sets of public and private methods by defining common core methods and explicit variation points on different hierarchical levels. Compositionality, and in particular the ability to relativize global SHVM properties on local assumptions for the core methods and the variation points, is achieved by means of maximal flow graphs that are derived algorithmically from the local assumptions. The flow graphs replace the assumptions when verifying global properties. The local specifications of core methods are verified by extracting flow graphs from the method implementations and model checking the induced behaviors against their specification.

Tool Support  We have adapted PROMOVER for verifying temporal safety properties of software product lines described through SHVMs. For this adaptation, we have extended the annotation language to support the definition of core methods, variants and variation points and the associated specifications by designated pragmas. The tool takes as input a source code file in which the SHVM to be analyzed is represented by annotations. The product property, the variation point properties and the specifications of the public core methods are also provided by annotations. Figure 4.5 shows the annotation for the @EnterProd variation point, and the annotations for its Keyboard variant with core method enterProd(). PROMOVER fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

Practical Evaluation  For evaluating our compositional verification approach, we considered the verification of the temporal safety property “the entering of products is finished, before the payment process is started” for different versions of a trading system product line described as SHVMs with different hierarchical depths and different total numbers of modules. For each product line, we compared the time required to verify all induced products individually with the time for compositional verification. The experiments confirm that non-compositional verification of software product lines becomes quickly infeasible while our compositional approach scales linearly in the number of features.
HATS Deliverable D2.5 Verification of Behavioral Properties

Figure 4.5: Annotations for variation point @EnterProducts and its variant Keyboard.
Chapter 5

Lazy Behavioral Subtyping

Within Task 2.5 UIO studied techniques for incremental reasoning related to late binding in object-oriented class hierarchies. The main motivation for this work is to separate subtyping from code reuse by introducing two hierarchies: subtyping at the level of interfaces and code reuse at the level of, e.g., class inheritance. The approach we take is based on combining syntax-driven inference systems like type and effect systems with Hoare logics in order to track behavioral dependencies between classes. The technique has been named lazy behavioral subtyping (LBS) as it separates the specification from the usage of methods. This separation allows redefinitions of methods to only comply with the behavioral constraints introduced by the usage of the methods and not by their specifications. The technique allows incremental reasoning about a larger set of programs than those supported by the standard techniques based on behavioral subtyping, as more liberal method redefinitions in subclasses are supported while we still avoid to violate established proofs from verified superclasses.

The work is presented in the form of two journal papers, one focussing on the basic system in the context of single inheritance class hierarchies [22] (reprinted in Appendix C.1), the second focussing on how the techniques may be extended to more complex code reuse mechanisms based on multiple inheritance [23] (reprinted in Appendix C.1). The latter suggests that the technique of lazy behavioral subtyping may be further extended to provide flexible reasoning mechanisms for other code structuring mechanisms. An extension of the approach to traits has been worked out by UIO in collaboration with Ina Schaefer (CTH) and Ferruccio Damiani, but has not yet been published. The next step would be the extension from traits to delta-oriented systems.

An investigation on how to apply these techniques to systems in which method definitions or requirements in superclasses may change is also under way. This situation typically occurs in program refactoring or in code which evolves at runtime. A CREOL-based system for asynchronous dynamic code evolution at runtime has been reported in Task 3.1 “Evolvable Systems: Modeling and Specification” and has been published as a conference paper [30]. Deliverable D3.1b “Final report on Evolvable Systems” [18] explains how we plan to extend this approach to dynamic delta modules.

As a next step, we intend to apply these techniques to the ABS language. We have already identified two possible application domains for LBS-like techniques; one is related to reasoning about the effect of feature integration and the other to instance-specific user requirements and how these should propagate through the code base of the software product family.
Chapter 6

Efficient Functional Verification of Software Product Lines

6.1 Introduction

In Chapter 2 we concentrated on functional verification of Core ABS, i.e., ABS programs without variability models. For the verification of full ABS models that specify a software product line one has two possibilities: (i) verification at the family engineering level or (ii) verification at the product level. The latter is possible, because after product selection in Full ABS the generated product is in the Core ABS fragment \[19\]. Both approaches have obvious pros and cons:

- Verification at the family engineering level can make efficient reuse of properties that have been established for code that is shared among different products. On the other hand, it suffers from combinatorial explosion already when faced with relatively small feature models.

- Verification on the product level can be done with the technology that has been developed for Core ABS, however, it is bound to redo the same tasks for each newly derived product.

The first point is substantiated by the largely negative results we obtained from a case study reported in \[16, Section 3.4.4\] and \[12\]. We provided a Java implementation of the HATS trading case study \[16, Chapter 3\], using the Java Modeling Language (JML) as specification language and the KeY tool in its version for sequential Java as verification system. We focused on the customization logic of the software responsible for adapting the software system to different deployment scenarios by instantiating feature sets as requested by the customer, for example, Cash, CreditCard, Scanner, etc. Even for a small model such as in this case study, the combinatorial explosion of possible feature combinations proved to be too much. We performed a kind of slicing “by hand”, but, while it helped, the experimental results indicate that this is not always sufficient to counter combinatorial explosion, even in relatively small feature hierarchies.

As a consequence, it is clear that verification at the family engineering level will require methodological approaches that go beyond the technologies available so far. In Chapter 4 we described how behavioral verification can be modularized and hence made efficient for SPLs that follow so-called simple hierarchical variability models, however, this is not sufficient for functional verification involving unbounded data types.

In the remaining chapter we describe two novel suggestions how to make functional verification at the family engineering level more efficient. In Section 6.2 we lift the idea of delta-oriented modeling to the level of specifications while in Section 6.3 we show that partial evaluation techniques can boost the performance of formal verification of programs that exhibit a high degree of variability.

6.2 Verification of Software Product Lines with Delta-Oriented Slicing

A software product line (SPL) \[37\] is a set of software systems (called products) with well-defined commonalities and variabilities. SPLs are often used in domains (e.g., communications, medical, transportation)
where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness by any method.

### 6.2.1 Delta-Oriented Programming

Software product lines can be implemented using *delta-oriented programming* (DOP) [40, 42, 39]. Delta-oriented programming offers an expressive and flexible “programming meta-language” for specifying a set of products. Its aim is to relax the restrictions of currently established SPL description formalisms such as feature-oriented programming (FOP) [6] by adding the explicit possibility to remove parts of a program. In delta-oriented programming, an SPL is implemented as a *core module* together with a set of *delta modules*. The core module contains a complete product implementation for some valid feature configuration, which can be developed by conventional single-application engineering techniques. Delta modules specify changes to be applied to the core module in order to implement other products. Each delta module $d$ contains an *application condition* $\varphi_d$, which is a propositional formula over the feature set $F$. The application conditions specify which delta modules are necessary for which features. For every pair of valid products $P_1, P_2 \in F$, $\Delta(P_1, P_2)$ is the set of delta modules that have to be applied to the product $P_1$ in order to obtain a product $P_2$ with a different feature configuration.

### 6.2.2 Delta-Oriented Specification

Product properties can be specified in the design-by-contract style [35] using method contracts which are expressed by pre- and post-conditions and class invariants. A core module is specified just as a conventional program. When a new product is derived by delta application, in general, both the implementation as well as the specification change. Hence, we extend the delta language with operations to manipulate specifications such that for a new product also the specification can be derived by delta application. The modification operations that can be applied to specifications are the addition of class invariants and method contracts and their removal.

### 6.2.3 Delta-Oriented Slicing

Delta-oriented slicing [11] for efficient verification of software product lines aims at exploiting the structural information available in a delta-oriented SPL model to reuse verification results. It allows determining which specifications of the new product remain valid (i.e., the proofs done for the old product are not affected by the change) and which parts have to be (re-)proven in order to establish the specified properties. Assume that we have verified all specifications of a given product and that we have obtained another product by delta application. The delta-oriented slicing algorithm considers the modification operations of all delta modules that have been applied to obtain the new product and analyses with specifications and corresponding proofs are invalidated by the modifications. Only these specifications have to be re-established for the new product. Even for these modified or otherwise affected product parts, we can save proof effort by applying a previously-developed proof reuse technique [7] based on the assumption of similarity between the two implementation variants. The number of specifications that are invalidated by a single modification in a delta module depends on the modularity of the programming language and the proof system. For instance, in a programming language with inheritance, a modification to the inheritance structure is likely to invalidate all specifications in the affected part of the inheritance hierarchy.

### 6.2.4 Implementation

The technology that we are using in [11] to illustrate the delta-oriented slicing approach is Java for implementing single products, JML [34] for formal specifications and the KeY system [8] for deductive verification. However, we only make the following assumptions about the verification system:

- We concentrate on systems that manipulate an explicit proof object in the proof assistant style.
- We support both ways in which verification systems can treat method calls: using the method contract or inlining the implementation.

- Our method is parametric in how a verification system treats invariants. In the worst case, all methods in the program have to be verified to preserve every invariant, as the invariant vocabulary is (in general) unrestricted. In practice, verification systems use criteria such as visibility, syntax and typing, assignable clauses or ownership to reduce the workload. We simply limit ourselves to requiring that all relevant invariants must be checked.

### 6.2.5 Relation to HATS ABS

Variability of ABS models is expressed by a core ABS model corresponding to a valid feature configuration and a set of model deltas containing modifications to the core model. A model delta is associated to a set of product features. If an ABS model for a different set of features should be obtained, the modification specified in the applicable model deltas have to be carried out in the core model. This variability modeling technique is conceptually the same as the one used in delta-oriented programming [10].

Hence, delta-oriented slicing is also applicable for efficiently verifying a family of ABS models. In order to transfer the delta-oriented slicing technique proposed in [11] to ABS, the delta language for ABS models has to be extended by modification operations for the specifications of ABS models. This allows deriving a new ABS model and its specification from an existing and already specified ABS model. Furthermore, the delta-oriented slicing algorithm has to be adapted to the modification operations used in delta modules for ABS models. For each modification operation on an ABS model or its specification, it has to be analyzed which specifications and corresponding proofs are invalidated by the applied changes. These are the specifications which have to be re-established for the newly generated ABS model. The number of invalidated specifications is influenced by the modularity of the ABS models and the associated proof system.

### 6.2.6 Towards Delta-Oriented Verification

The idea of delta-oriented specification can be pushed even further: instead of re-using proofs whose validity has been shown to be unaffected by a given change, we intend to lift delta-slicing to the level of proof trees. The basis for this will be a delta-oriented extension of the familiar notion of method contracts based on delta-oriented specification (Section 6.2.2). This will make it possible to compute minimal validity-preserving changes to existing proof trees that correspond exactly to the application of a model- or specification-delta. This work, together with the adaption to ABS, will be carried out as part of Task 4.3 “Correctness”.

### 6.3 Interleaving Partial Evaluation and Symbolic Execution

Symbolic execution and partial evaluation both are generalizations of standard interpretation of programs, however, they generalize in different ways: while symbolic execution permits interpretation of a program with symbolic (i.e., unspecified) initial values, the aim of partial evaluation is to transform a program with partially specified input values into a (hopefully, more efficient) program that has only the unspecified arguments as input. For fully specified input arguments the result of both mechanisms is standard program interpretation.

Combining both technologies shows that partial evaluation and symbolic execution are not only compatible with each other, but that there is considerable potential for synergies. This allows to interleave symbolic execution and partial evaluation steps within a uniform (logic-based) framework in a sound way. Intermittent partial evaluation during symbolic execution has the effect that the remaining program that is yet to be executed is continuously simplified relative to the current path conditions and the current symbolic state in each symbolic execution trace.

In [13] (Appendix C.5) we describe the approach in detail and present a prototypical implementation based on the KeY tool [8]. The program logic is based on a dynamic logic similar to the one presented in
Dynamic logic contains programs as first-class citizens similar to Hoare logic. The fact that programs occur in an unencoded form within a formula makes it straightforward to define a partial evaluation operator on the logic level that takes context information of the current proof-situation and specializes the remaining program occurring in the formula. The most immediate effect is that the specialized program is simpler in terms of expression complexity, e.g., variables are replaced by constants where possible. In addition, branching is reduced by simplification or elimination of guards in conditional expressions/statements and loops. Further, branching points are reduced by type narrowing reducing possible branching on the concrete type of an object.

These effects lead to a much smaller proof tree and a considerable performance increase by replacing first-order proof search with computation. We expect that this approach proves even more powerful when dealing with software product lines. The reason is that these programs contain a number of variability points causing significant branching of proofs [12]. If those variability points can be resolved (or simplified) by exploiting proof context information (e.g., system configuration encoded as precondition), the degree and performance of automation should be drastically improved.
Chapter 7

Conclusions

In this deliverable we reported on a program logic based on symbolic execution and a deadlock analysis for Core ABS as well as on a number of incremental verification techniques suited to deal with software exposing a high degree of variability. Prototypical tools are available for several of the developed techniques, although not yet fully adapted for Core ABS.

The presented techniques are used as input to several recently started and future tasks like Task 1.3 “Analysis”, Task 2.3 “Testing, Debugging and Visualization”, Task 2.6 “Refinement and Abstraction” and Task 4.3 “Correctness”. In these tasks we will follow up on them and integrate them deeply into the HATS tool suite, methodology and workflow.
Bibliography


[45] Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. PROMOVer web interface. [http://www.csc.kth.se/~siavashs/ProMoVer](http://www.csc.kth.se/~siavashs/ProMoVer)
Glossary

ABS  Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

COP  Context-Oriented Programming

Compositional Verification  Compositional verification ensures that properties proven locally (e.g., only looking at one object and method at a time) can be generalized to global properties.

Core  In delta modeling, the core is the basic product that may be modified by product deltas. The core may also be seen as a delta modifying the empty product.

DOP  Delta Oriented Programming

Dead-lock  Two (or more) threads block waiting for the other(s) to free required resources.

Delta  A unit of functionality and conflict resolution in delta modeling, able to modify a product using invasive composition of code or other content.

Delta Model  Generally, a means for expressing the semantics of features within product lines. In the formalism, a delta model is defined more specifically as \((D, \prec)\), a partially ordered set of deltas.

Dynamic Logic  A member of the family of modal logics where programs are first-class citizens. Similar to and subsumes Hoare logics.

Feature  Generally, an increment in software functionality. On the level of feature models it is merely a label with no inherent semantic meaning.

Feature Configuration  A subset of available features. It identifies a single product in a product line if it is valid in the feature model.

History  Trace of messages representing the observable behaviour of a system run.

Invariant  A property that has to be kept invariant in any observable state.

MC  Model Checking

Partial Evaluation  Program specialisation technique creating an optimized program by assuming a given subset of input values as constant.

Product  One member of a product line, with well-defined commonalities and variabilities to other products.

Product Line  Generally, a family of products with well-defined commonalities and variabilities. In the delta modeling formalism, a product line is more specifically defined as \((\Phi, c, D, \prec, \gamma)\), a feature model, a core product, a delta model and an application function.

Software Family  See Software Product Line.
**Software Product Line** A family of software systems with well-defined commonalities and variabilities. See also Product Line.

**SPL** Software Product Line

**SVHM** Simple Hierarchical Variability Model
Appendix A

The Rules of ABS DL

It is the subject of future work to design rules that can prove termination of concurrent ABS programs. For this reason, some of the rules that handle the transfer of control are only available in the partial correctness ("box") version. We have also refrained from giving a rule for total correctness of while loops, even though this is, of course, possible for sequential programs. For all statements where it makes no difference we give both, the "box" and "diamond" versions.

A.1 Update Simplification Rules

Please note that the premise of rule updateOnBoxDia is intentionally identical to its conclusion: updates are never applied to program formulas, but delayed until symbolic execution ends.

\[
\begin{align*}
\text{updateOnPVarEq} & \quad \frac{v}{\{ l := v \} } l \\
\text{updateOnPVarNeq} & \quad \frac{x}{\{ l := v \} } \ (x \neq l) \\
\text{updateOnBoxDia} & \quad \frac{\{ u \} \{ p \} \phi}{\{ u \} \{ p \} \phi} \\
\text{updateOnConnective} & \quad \frac{(\{ u \} \phi \circ \{ u \} \psi)}{\{ u \} (\phi \circ \psi)}, \circ \in \{ \land, \lor, \rightarrow, \leftrightarrow \} \\
\text{updateOnNegation} & \quad \frac{\neg\{ u \} \phi}{\{ u \} \neg \phi} \\
\text{updateOnPredFunc} & \quad \frac{s(\{ u \} t_1, \ldots, \{ u \} t_n)}{\{ u \} s(t_1, \ldots, t_n)}, s \text{ rigid (state-independent)} \\
\text{updateOnQuantifier} & \quad \frac{Q x. \{ u \} \phi}{\{ u \} Q x. \phi}, \text{ if } x \text{ not free in } u, \ Q \in \forall, \exists \\
\text{updateOnElementaryUpdate} & \quad \frac{l := \{ u \} v}{\{ u \} l := v} \\
\text{updateMerge} & \quad \frac{\{ u \} l_1 := \{ u \} v_1 \ldots \| l_n := \{ u \} v_n}{\{ u \} \{ l_1 := v_1 \ldots \| l_n := v_n \}} \\
\text{updateLastWin} & \quad \frac{l_1 := v_1 \ldots \| l_{i-1} := v_{i-1} \| l_{i+1} := v_{i+1} \ldots \| l_j := v_j \ldots \| l_n := v_n}{l_i := v_i \ldots \| l_{i-1} := v_{i-1} \| l_i := v_i \| l_{i+1} := v_{i+1} \ldots \| l_j := v_j \ldots \| l_n := v_n}, \text{ with } l_i = l_j, \ j > i \\
\text{updateSwap} & \quad \frac{\ldots \| l_j := v_j \ldots \| l_i := v_i \ldots}{\ldots \| l_j := v_j \ldots \| l_i := v_i \ldots}, \text{ with } j > i, l_i \neq l_j \text{ and } l_k \notin \{ l_i, l_j \}, i < k < j \\
\text{unusedUpdate} & \quad \frac{\{ u_1 \ldots \| u_j-1 \| u_{j+1} \ldots \| u_n \} \phi}{\{ u_1 \ldots \| u_j-1 \| l_j := v_j \| u_{j+1} \ldots \| u_n \} \phi}, \ l_j \notin \phi
\end{align*}
\]
A.2 Sequential Constructs

A.2.1 Pure Expressions

Pure ABS expressions are side effect-free and can be directly translated into first-order logic terms. The function \( \tau : \text{PureExp} \rightarrow \text{Trm} \) implements this translation:

\[
\begin{align*}
\tau(x) &= x, \quad x \text{ is variable or literal} \\
\tau(f(t_1, \ldots, t_n)) &= f(\tau(t_1), \ldots, \tau(t_n)) \\
\tau(\text{let } x : T = t_1 \text{ in } t_2) &= \{\text{subst } T x; \tau(t_1)\} \tau(t_2)
\end{align*}
\]

with \( \{\text{subst } T x; t\} \xi \) being the substitution operator of the logic, binding the logic variable \( x \) to \( t \) in term or formula \( \xi \). In contrast to update expressions, \( x \) is a logic variable and rigid (state-independent).

The translation of the case expression is performed branch-wise. The case expression is translated into a nested conditional expression. To this end, all anonymous variable occurrences are replaced by new free variables. Then all free variables of the case variable \( v \) and pattern \( i \) are existentially bound to values, whenever for the \( i \)-th branch \( p_i \Rightarrow t_i \); it is possible that \( v \neq p_i \) holds. The translated term then evaluates to \( t'_i \) which is obtained from \( t_i \) by replacing the free variables with values satisfying the matching condition. Otherwise, the free variables are not bound and the else branch of the conditional term contains the translation for the default pattern \( p_{i+1} \Rightarrow t_{i+1} \). In the following, we skip \( \tau \) for those expressions where it is the identity.

The function \( \iota : \text{Trm} \rightarrow \text{Fml} \) takes a term \( t \) of type \( \text{Bool} \) and returns a formula \( \iota(t) \) such that \( \iota(t) \) is satisfied if and only if \( t \Rightarrow \text{True} \) is satisfied.

A.2.2 Statements

**Assignment**

\[
\text{assignPureExp} \quad \frac{\{x := \tau(e_p)\}}{\{x = e_p; \omega\} \phi}
\]

**assert**

\[
\text{assertDia} \quad \frac{\Gamma \Rightarrow \iota(\tau(e_p)) \land \{\omega\} \phi, \Delta}{\{\text{assert } e_p; \omega\} \phi, \Delta}
\]

\[
\text{assertBox} \quad \frac{\Gamma \Rightarrow \iota(\tau(e_p)) \rightarrow [\omega] \phi, \Delta}{\{\text{assert } e_p; \omega\} \phi, \Delta}
\]

**block and skip**

The block statement halts execution without releasing control. It is not a Core ABS statement, but introduced in the logic to achieve a more compact formulation of the rules. The skip statement is a statement that does nothing.

\[
\text{skip} \quad \frac{\{\omega\} \phi}{\{\text{skip; } \omega\} \phi}
\]

**Conditional Statement**

\[
\text{ifSplit} \quad \frac{\iota(\tau(e_p)) \rightarrow (p; \omega) \phi \land \neg \iota(\tau(e_p)) \rightarrow (q; \omega) \phi}{\{\text{if } (e_p) \{p\} \text{ else } \{q\} \omega\} \phi}
\]

37
Expression Statement
An expression statement is only executed for its side-effects. To avoid duplication of rules we assign the expression to a newly introduced program variable of the same type. In subsequent steps the variable is removed from the sequent as it does not occur at any other place:

\[ \text{expStmnt} \frac{\{x = e; \omega\}}{\{e; \omega\}} \]

with

- \(x\) is a new program variable of the same type as \(e\)

Loop Statement

\[
\begin{align*}
\Gamma &\Rightarrow inv, \Delta & \text{(init. valid)} \\
\Gamma &\Rightarrow \{U_{H,mod}\} (i(\tau(e_p)) \land inv) \rightarrow [p]inv, \Delta & \text{(preserves inv.)} \\
\Gamma &\Rightarrow \{U_{H,mod}\} (-i(\tau(e_p)) \land inv) \rightarrow [p]\phi, \Delta & \text{(use case)} \\
\Gamma &\Rightarrow [\text{while}(e_p)\{p\}]\phi, \Delta
\end{align*}
\]

Variable Declaration

\[
\begin{align*}
\text{varDeclNoInit} &\frac{\{\hat{x} := \text{null}\} \{\omega[x/\hat{x}]\}}{\{T \ x; \omega\}} \\
\text{varDeclInit} &\frac{\{\hat{x} = e; \omega[x/\hat{x}]\}}{\{T \ x = e; \omega\}}
\end{align*}
\]

with

- \(T\) denoting a reference type
- \(\hat{x}\) new name of program variable \(x\) to avoid name clashes due to scoping
- \(\omega[x/\hat{x}]\) is identical to the rest program \(\omega\) except that all occurrences of \(x\) have been replaced by \(\hat{x}\)

A.3 Concurrent Constructs

A.3.1 History, Messages & Implicit Attributes

The history is a sequence of messages. The following message types are provided:

\[
\begin{align*}
&< \text{invoc}, (\text{caller}, \text{callee}, \text{method}, \text{uniqueId}), \text{args} > \\
&< \text{comp}, (\text{caller}, \text{callee}, \text{method}, \text{uniqueId}), \text{returnValue} > \\
&< \text{new}, (\text{o}, \text{objId}, \text{objGrpId}), \text{args} >
\end{align*}
\]

The first component specifies the message kind: method invocation invoc, method completion comp and object creation new. The tuple \((\text{caller}, \text{callee}, \text{method}, \text{uniqueId})\) represents a future instance \(r\) with \text{uniqueId} ensuring its uniqueness. We access the components of the future \(r\) by \(r\text{.callee}, r\text{.caller}, r\text{.method}\), and \(r\text{.id}\).
The history $H$ is a program variable of type \texttt{History} and represents the communication trace of the system. We assume further the existence of a program variable \texttt{this} referring to the current object and a program variable \texttt{thisGrpId} (implicit attribute) of \texttt{this} denoting the current object group of \texttt{this}.

$H_1 \leq H_2$ expresses that history $H_1$ is a prefix of history $H_2$. The predicate $\text{Invoc}(H, r, \texttt{args})$ is satisfied if history $H$ contains an invocation message belonging to future $r$ and where the passed method arguments are equal to $\texttt{args}$. Similar for predicate $\text{Comp}$ and $\text{New}$. In addition, there is a predicate $\text{SameCOG}(H, o, u)$ which takes a history and two objects as arguments and evaluates to true if both objects belong to the same concurrent object group.

### A.3.2 Asynchronous Method Call

$\text{asyncMC} \triangleq o.\texttt{=} \texttt{null} \rightarrow \{ \texttt{block; } \omega \} \phi, \Delta
\Gamma \Rightarrow \neg(o.\texttt{=} \texttt{null}) \rightarrow \{ r := (\texttt{this}, o, m, i); \omega \} \phi, \Delta$

with

- $U^\text{invoc}_H$ defined as $H := \texttt{some } H. (H \leq H \land \text{Invoc}(H, r, \tau(\texttt{args})))$
- $(\texttt{this}, o, m, i)$ represents the future instance with $i$ being a unique identifier for the future

### A.3.3 \texttt{await}

$\text{awaitComp} \quad \Gamma \Rightarrow \text{ClInv}(C(H, \bar{A}), \Delta)
\Gamma \Rightarrow \{ U^\text{comp}_{H, \bar{A}} \} \{ \text{Comp}(H, r) \rightarrow [\omega] \phi \}, \Delta
\Gamma \Rightarrow \{ \text{await } r?; \omega \} \phi, \Delta$

$\text{awaitCond} \quad \Gamma \Rightarrow \text{ClInv}(C(H, \bar{A}), \Delta)
\Gamma \Rightarrow \{ U^\text{comp}_{H, \bar{A}} \} \{ \iota(\tau(e_p)) \rightarrow [\omega] \phi \}, \Delta
\Gamma \Rightarrow \{ \text{await } e_p?; \omega \} \phi, \Delta$

$suspend \quad \{ \text{await True}; \omega \} \phi
\{ \text{suspend}; \omega \} \phi$

The \texttt{await} statement releases control, therefore we have to establish the invariant of the class of \texttt{this} and when continuing execution with the remaining program, we have to anonymize the knowledge about the attribute values of \texttt{this} using the update $U^\text{comp}_{H, \bar{A}}$ defined as $H, \bar{A} := \texttt{some } H, \bar{y}. (\text{ClInv}(C(H, \bar{y}), \bar{y} \land H \leq H))$. This update anonymizes the history and class attributes as these may have changed. The only knowledge we have about their value when resuming is that the class invariant holds and the new history is a continuation of the old one.

### A.3.4 \texttt{get}

$\text{get} \quad \Gamma \Rightarrow \{ U^\text{comp}_{H, \bar{x}} \} [\omega] \phi, \Delta
\Gamma \Rightarrow [x = l. \texttt{get}; \omega] \phi, \Delta$

the update $U^\text{comp}_{H, \bar{x}}$ is defined as $H, x := \texttt{some } H, y. (H \leq H \land \text{Inv}(I)(H)^{\text{this}, \text{callee}} \land \text{Comp}(H, l, y))$ where $I$ is the interface of the callee which can be extracted from $l$. 

39
A.3.5 new and new cog

\[
\begin{align*}
\{ l := (o, oid, thisGrpId) \} \{ U_H^{new} \} \\
\quad (Inv(l)(H)^{this,o} \land \{ x = o.$init(); \omega \} \phi) \\
\quad \Gamma \Rightarrow \{ x = \text{new } C(\text{args}) ; \omega, \Delta \phi \}
\end{align*}
\]

- \((o, oid, thisGrpId)\) represents a tuple with \(o\) being an object reference to the new object, \(oid\) its unique object identifier and \(thisGrpId\) the object group identifier of the \(this\) object to which the newly created object also belongs.

- \$init\) is a virtual method implicitly available in each class \(C\) implementing its initialiser block. The call to \$init\) is synchronous as objects of same object group are executed by same thread.

- \(U_H^{new}\) is defined as \(H := \text{some } H. (H \leq H \land \text{New}(H, l, t(\text{args})))\).

\[
\begin{align*}
\{ l := (o, oid, objGrpId) \} \{ U_H^{new} \} \\
\quad (Inv(l)(H)^{this,o} \land \{ x = o.$init(); \omega \} \phi) \\
\quad \Gamma \Rightarrow \{ x = \text{new } \text{cog } C(\text{args}) ; \omega, \Delta \phi \}
\end{align*}
\]

- \((o, oid, objGrpId)\) represents a tuple with \(o\) being an object reference to the new object, \(oid\) its unique identifier and \(objGrpId\) a unique object group identifier representing the new concurrent object group.

- \$init\) and \(U_H^{new}\) are defined as above.

A.3.6 Synchronous Method Call

\[
\begin{align*}
\Gamma \Rightarrow o \neq \text{null} & \rightarrow \{ \text{block}; \omega \} \phi, \Delta \\
\Gamma \Rightarrow \neg (o \neq \text{null}) & \rightarrow \text{SameCOG}(H, this, o) \land CInv(C)(H, \bar{A}), \Delta \\
\Gamma \Rightarrow \neg (o \neq \text{null}) & \rightarrow \{ l := (\text{this}, o, m, i) \} \{ U_H^{invoc} \} \\
\quad (Inv(l)(H)^{this,o} \land \{ U_{H, \bar{A}} \} \{ r = l.get(); \omega \} \phi), \Delta \\
\quad \Gamma \Rightarrow \{ r = o.m(\bar{args}); \omega \} \phi, \Delta
\end{align*}
\]

- \(U_H^{invoc}\) defined as \(H := \text{some } H. (H \leq H \land \text{Invoc}(H, l, t(\bar{args})))\)

- \((\text{this}, o, m, i)\) represent the future instance with \(i\) being a unique identifier for the future

- \(\bar{H}, \bar{A} := (\text{some } H, \bar{y}.CInv(C)(H, \bar{y}) \land H \leq H)\) anonymizes the history and class attributes as these may have changed. The only knowledge we have about their value when resuming is that the class invariant holds and the new history is a continuation of the old one.

A.3.7 return

\[
\begin{align*}
\text{return} & \quad \Gamma \Rightarrow \{ U_H^{rel} \} \phi, \Delta \\
\Gamma & \Rightarrow \{ \text{return } e_p; \omega \} \phi, \Delta
\end{align*}
\]

with \(U_H^{rel}\) defined as

\[
H := \text{some } H. (H \leq H \land \text{Comp}(H, (\text{caller}, \text{this}, m_{\text{this}}, futId_{\text{this}}), \tau(e_p)))
\]

The future \((\text{caller}, \text{this}, m_{\text{this}}, futId_{\text{this}}) \) \(m_{\text{this}}\) occurring in the completion predicate consists of

- a reference to the \text{caller} object of the currently executed method \(m_{\text{this}}\),

- \text{this} playing the role of the callee and

- \(futId_{\text{this}}\) being the unique future identifier created when the method had been invoked.
Appendix B

Papers on Program logics for ABS

B.1 A verification system for distributed objects with asynchronous method calls
A Verification System for Distributed Objects with Asynchronous Method Calls *

Wolfgang Ahrendt¹ and Maximilian Dylla¹²

¹ Chalmers University of Technology, Gothenburg, Sweden
² Saarland University, Saarbrücken, Germany

Abstract. We present a verification system for Creol, an object-oriented modeling language for concurrent distributed applications. The system is an instance of KeY, a framework for object-oriented software verification, which has so far been applied foremost to sequential Java. Building on KeY characteristic concepts, like dynamic logic, sequent calculus, explicit substitutions, and the taclet rule language, the system presented in this paper addresses functional correctness of Creol models featuring local cooperative thread parallelism and global communication via asynchronous method calls. The calculus heavily operates on communication histories which describe the interfaces of Creol units. Two example scenarios demonstrate the usage of the system.

1 Introduction

The area of object-oriented program verification made significant progress during the last decade. Systems like Boogie [6], ESC/Java2 [23], KeY [9], and Krakatoa [22] provide a high degree of automation, elaborate user interfaces, extensive tool integration, support for various specification languages, and high coverage of a real world target language (Spec# in case of Boogie, Java in case of the other mentioned tools).

However, this development mostly concerns sequential, free-standing applications. When it comes to verifying functional properties of concurrent and distributed applications, the situation is different. Even if there is a rich literature on the verification of ‘distributed formalisms’ (based for instance on process calculi [35, 27, 36]), there are hardly any systems yet matching the aforementioned characteristics. Moreover, many formalisms lack a connection to the dominating paradigm of today’s software engineering, object-orientation, which is an obstacle for the integration into software development environments and methods.

This work is a contribution towards effective and integrated verification of concurrent, distributed systems. We present a verification system that is built on two foundations: the Creol modeling language for concurrent and distributed

* This work has partially been supported by the EU-project FP7-ICT-2007-3 HATS: Highly Adaptable and Trustworthy Software using Formal Methods, and the EU COST action IC0701: Formal Verification of Object-Oriented Software.
object-oriented systems [32], and the KeY approach and system for the verification of object-oriented programs [9]. By combining KeY’s proving technology with Creol’s novel approach to modular modeling of components, we achieve a system for highly modular verification of concurrent, distributed object-oriented applications. While being a prototype system yet, past experience with the technological and conceptual basis justifies the perspective of future versions to enjoy similar features as state-of-the-art sequential verification systems already do.

Creol is an executable object-oriented modeling language. It features concurrency in two ways. First of all, different objects execute truly in parallel, as if each object had its own processor. Objects have references to each other, but cannot access each other’s internal state. Consequently, there is no remote access to attributes, like ‘o.a’ in other languages. The only way for objects to exchange information is through methods. Calls to methods are asynchronous [31], in the sense that the calling code is able to continue execution even before the callee replies. Mutual information hiding is further strengthened by object variables being typed by interfaces only, not by classes. The loose coupling of objects, their strong information hiding and true parallelism, is what suggest distributed scenarios, with each object being identified with a node. The second type of concurrency is object internal. Each call to a method spawns a separate thread of execution. Within one object, these threads execute interleaved, with only one thread running at a time. Here, the key to modularity is the cooperative nature of the scheduling: a thread is only ever interrupted when it actively releases control, at ‘release points’.

Altogether, Creol allows highly modular verification. Within one class, the various methods can be proved correct in isolation, in spite of the shared memory (the attributes), by guaranteeing and assuming a class invariant at each release point in the code. At the inter-object level, the vehicle to connect the verification of the various classes is the ‘history’ of inter-object communications. Interface specifications are expressed in terms of the history, and class invariants relate the history with the internal state. The fact that each object has only partial knowledge about the global communication history is modeled by projecting the global history onto the individual objects [30].

Our system is based on the KeY framework for verifying object-oriented software. The most elaborate instance of KeY is a verification system for sequential Java [9]. Other target languages of KeY are C [39], ASMs [40], and hybrid systems [42]. All these have in common that they use dynamic logic, explicit substitutions, and a sequent calculus realized by the ‘taclet’ language. These concepts, to be introduced in the course of the paper, have proved to be a solid foundation of a long lasting and far reaching research project and system for verifying functional correctness of Java [9]. Dynamic logic features full source code transparency, like Hoare logic, but is more expressive than that. Explicit (simultaneous) substitutions, called updates, provide a compact representation of the symbolic state, and allow a natural forward style symbolic execution. Apart from verification, updates are also employed for test case generation and symbolic debugging. Sequent calculi are well-suited for the interleaved automated and interactive usage.
And finally, taclets provide a high-level rule language capturing both the logical and the operational meaning of rules. They are well suited both for the base logic and for the axiomatization of application specific operations and predicates. KeY has been used in a number of case studies, like the verification of the Java Card API Reference Implementation [38], the Mondex case study (the most substantial benchmark in the Grand Challenge repository) [44], the Schoor-Waite algorithm [12], and the electronic purse application Demoney [37]. The system is also used for teaching in various courses at Chalmers University and several other universities.

However, the KeY approach has so far almost only been applied to the sequential setting.\(^3\) It is precisely the described modularity of Creol that allowed us to base our verification system on the same framework. The main challenges for adjusting the KeY approach to Creol were the handling of asynchronous method calls, the handling of release points, and, most of all, the extensive usage of the communication history throughout the calculus.

The structure of the paper is as follows. Sect. 2 introduces Creol, and gives examples of its usage. In Sect. 3, we describe the logic and calculus characteristic for KeY, insofar as they are (largely) independent of the particular target language. Thereafter, Sect. 4 presents a KeY style logic and calculus for Creol specifically. Sect. 5 discusses system oriented aspects of KeY for Creol, including a small account on taclets. Sect. 6 then demonstrates the usage of the systems in examples. In Sect. 7, we discuss related work, and draw conclusions.

2 Overview of Creol

In this section, we introduce our slightly adapted version of Creol, using an automated teller machine scenario adapted from [29]. The example will also be used to discuss Creol verification in later sections.

The scenario we consider has three kinds of actors. There are several teller machines (class `ATM`), several users (class `User`), and one server (class `Server`). In the course of a certain session, a teller machine communicates with one user, and with the server, as depicted in Fig. 1.

![Diagram of communication between User, ATM, and Server](attachment:communication_diagram.png)

**Fig. 1.** Communication of the automated teller machine

The picture shows that, while `User` and `Server` implement one interface each (USR resp. S), the class `ATM` implements two interfaces, `ATMU` and `ATMS`, \(^3\) See Sect. 7 for an exception.
dedicated for the communication in either of the directions. The Creol definition
of the interfaces is given in Fig. 2. (We omit ATMS, which is empty.)

```plaintext
interface USR
begin
  with ATMU
  op giveCode(in; out code:Int)
  op withdraw(in; out amount:Int)
  op dispense(in amount:Int; out)
  op returnCard(in; out)
end

interface ATMU
begin
  with USR
  op insert(in cardId:Int; out)
end

interface S
begin
  with ATMS
  op authorize(in cardId:Int, code:Int; out ok:Bool)
  op debit(in cardId:Int, amount:Int; out ok:Bool)
end
```

Fig. 2. The interfaces of the automated teller machine

We can observe that the signature of operations contains (possibly empty)
lists for in- and out-parameters. The operations offered by interfaces appear in
the scope of ‘with cointerface’, with the meaning that those operations can only
be called from instances of classes implementing that cointerface. For instance,
the server cannot call insert on a teller machine, not even if it was in the posses-
sion of an ATMU typed reference. Another consequence of cointerfaces is that
the implementations of operations have a well-typed reference to the caller, without
that reference being passed explicitly as an input parameter.

The class ATM in Fig. 3 is an example for a class definition. Variables are im-
plicitly initialized with false or 0 for primitive types, and null for labels and object
references. Some variables are declared of type Label[...], like var li:Label[Int].
Later, the execution of the call li!caller.giveCode(), for instance, allocates a new
label, and assigns it to li. The label is later used in the reply statement li?(code),
to associate the reply with the respective call. The effect of the reply is that
code is assigned the output of the (li-labeled) call to giveCode, provided that
the corresponding reply message has already arrived. Otherwise, the statement
blocks, without the thread releasing control. (This ‘busy waiting’ can be avoided
by the await statement, see below.) The effect of li?(x) is similar to treating x as
a future variable [15, 5] or promise [34]. In a label type Label[T], the T indicates
the type of the output of the called operation.

Note that the calls to dispense and returnCard are executed before any of the
replies is asked back. This allows the two called methods to execute interleaved
on the processor of the called object. (Note that the calls went to the same
class ATM implements ATMS, ATMU
begin
  var server : S;
  with USR
    op insert(in card:Int; out) ==
      var li:Label[Int]; var lb:Label[Boolean]; var l:Label[];
      var l2:Label[]; var code:Int; var ok:Boolean; var am:Int;
      li!caller.giveCode(); li?(code);
      lb!server.authorize(card,code); lb?(ok);
      if ok
        then li!caller.withdraw(); li?(am);
        lb!server.debit(card,am); lb?(ok);
        if ok
          then li!caller.dispense(am); l2!caller.returnCard(); l?(); l2?()
          else li!caller.returnCard(); l?()
        end
      else li!caller.returnCard(); l?()
    end
  end
end

Fig. 3. The class implementing the teller machine

object.) In general, arbitrary code can be executed in between a call and the corresponding reply. We want to highlight that the implementation of insert extensively uses the caller reference, which is known to be of type USR, for callbacks. This style of coupling communicating objects might clarify the distribution of operations over interfaces in the teller machine scenario (cf. Fig. 2).

We discuss further features of Creol not captured by the above example. New objects are created by \( x := \text{new} \ C(e^*) \), where \( C \) is a class identifier supplied with a list of class parameters. As indicated earlier, \( l? (x^*) \) blocks execution, without releasing control, until the corresponding reply message has arrived. In contrast, the command \( \text{await} \ l? \) releases control if the reply for \( l \) has not yet arrived, such that the scheduler can pass control to another thread of this object. Other release points are \( \text{await} \ b \), releasing control if the Boolean expression \( b \) is false, and the unconditioned \( \text{release} \). The example code above did not contain release points, but see the buffer example in Sect. 6.1 (Fig. 7).

In Creol, expressions have no effect on the state. We model errors, like division by zero, by non-terminating (and non-releasing) blocking. The same holds for a call on the null reference and a reply on the null label.

3 The KeY approach: Logic, Calculus, and System

3.1 Dynamic Logic with Explicit Substitutions

KeY is a deductive verification system for functional correctness. Its core is a theorem prover for formulas in dynamic logic (DL) [25], which, like Hoare logic [26], is transparent with respect to the programs that are subject to verification. DL
is a particular kind of modal logic. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of functions and predicates. The modalities are ‘indexed’ with pieces of program code, describing how to reach one world (state) from the other. DL extends full first-order logic with two additional (mix-fix) operators: $\langle . \rangle$ (diamond) and $[ . ]$ (box). In both cases, the first argument is a program (fragment), whereas the second argument is another DL formula. A formula $\langle p \rangle \phi$ is true in a state $s$ if execution of $p$ terminates when started in $s$ and results in a state where $\phi$ is true. As for the other operator, a formula $[p] \phi$ is true in a state $s$ if execution of $p$, when started in $s$, either does not terminate or results in a state where $\phi$ is true. In other words, the difference between the operators is the one between total and partial correctness.

DL is closed under all logical connectives. For instance, the following formula states equivalence of $p$ and $q$ w.r.t. the “output”, the program variable $x$.

$$\forall v. (\langle p \rangle x \overset{\ast}{=} v \leftrightarrow \langle q \rangle x \overset{\ast}{=} v)$$

A frequent pattern of DL formulas is $\phi \rightarrow \langle p \rangle \psi$, stating that the program $p$, when started from a state satisfying $\phi$, terminates with $\psi$ being true afterwards. The formula $\phi \rightarrow [p] \psi$, on the other hand, does not claim termination, and corresponds to the Hoare triple $\{ \phi \} p \{ \psi \}$.

The main advantage of DL over Hoare logic is increased expressiveness: pre- or postconditions can contain programs themselves, for instance to express that a linked structure is acyclic. Also, the relation of different programs to each other (like the correctness of transformations) can be expressed elegantly.

All major program logics (Hoare logic, wp calculus, DL) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having explicit substitutions in the logic, called ‘updates’. This allows for accumulating and simplifying the effect of a program, in a forward style. Elementary updates have the form $x := e$, where $x$ is a location (in the case of Creol, an attribute or local variable) and $e$ is a (side-effect free) expression. Elementary updates are combined to simultaneous updates, like in $x_1 := e_1 \mid x_2 := e_2$, where $e_1$ and $e_2$ are evaluated in the same state. For instance, $x := y \mid y := x$ stands for exchanging the values of $x$ and $y$. Updates are brought into the logic via the update modality $\{ . \}_U$, connecting arbitrary updates with arbitrary formulas, like in $x \lt y \rightarrow \{ x := y \mid y := x \} y \lt x$. A typical usage of updates during proving is in formulas of the form $\{ U \} \langle p \rangle \phi$, where $U$ is an update, accumulating the effects of program execution up to a certain point, $p$ is the remaining program yet to be executed, and $\phi$ a postcondition. A full account of KeY style DL is found in [11].

---

4 Just as in standard modal logic, the diamond vs. box operators quantify existentially vs. universally over states (reached by the program). In case deterministic programs, however, the only difference between the two is whether termination is claimed or not.
3.2 Sequent Calculus

The heart of KeY, the prover, uses a sequent calculus for reducing proof obligations to axioms. A sequent is a pair of sets of formulas written as $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$. The intuitive meaning is that, if all $\phi_1, \ldots, \phi_m$ hold, at least one of $\psi_1, \ldots, \psi_n$ must hold. Rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises. In Fig. 4 we present a selection of the rules dealing with propositional connectives and quantifiers (see [24] for the full set). $\phi[v/e]$ denotes a formula resulting from replacing $v$ with $e$ in $\phi$.

**Fig. 4.** A selection of first-order rules

When it comes to the rules dealing with programs, many of them are not sensitive to the side of the sequent and can even be applied to subformulas. For instance, $(\text{skip}; \omega)\phi$ can be rewritten to $(\omega)\phi$ regardless of where it occurs. For that we introduce the following syntax

```
[\phi']
[\phi]
```

for a rule stating that the premise sequent $[\phi']$ is constructed by replacing $\phi$ with $\phi'$ anywhere in the conclusion sequent $[\phi]$. In Fig. 5 we present some rules dealing with statements. (assign and if are simplified, see Sect. 4.1.) The schematic modality $\llbracket \cdot \rrbracket$ can be instantiated with both $\llbracket \cdot \rrbracket$ and $\llbracket \cdot \rrbracket$, though consistently within a single rule application. Total correctness formulas of the form $(\text{while } \ldots)\phi$ are proved by combining induction with unwind.

**Fig. 5.** Dynamic logic rules

Because updates are essentially delayed substitutions, they are eventually resolved by application to the succeeding formula, e.g., $\{u := e\}(u > 0)$ leads to $e > 0$. Update application is only defined on formulas not starting with box or diamond. For formulas of the form $\{U\}(s)\phi$ or $\{U\}[s]\phi$, the calculus first applies rules matching the first statement in $s$. This leads to nested updates, which are
in the next step merged into a single simultaneous update. Once the box or diamond modality is completely resolved, the entire update is applied to the postcondition.

4 A Calculus for Creol Dynamic Logic

Building on the logic and the calculus presented in the previous section, we proceed with the sequent rules handling Creol statements. For the full set of rules, see [20].

4.1 Sequential Constructs

We start with assignments. As soon as the right side is simply a variable or literal (summarized as ‘terminal expression’, te) the assignment can be transformed to an update, such that the effect will eventually (not immediately) be applied to the postcondition. The same applies for implicit assignments in variable declarations. We give only the rule for integer variable declaration.

\[
\begin{align*}
\text{assign} & \quad \{ x := \text{te} \} \langle \omega \rangle \phi \\
\text{intDecl} & \quad \{ i := 0 \} \langle \omega \rangle \phi \\
\end{align*}
\]

The same mechanism can be used for operator expressions, as long as all arguments are terminal and errors can be excluded. For instance, a division can be shifted to an update iff the divisor is not zero. Otherwise, execution blocks. This semantics is captured by the following rule.

\[
\text{DivTerminal} \quad \frac{\neg \text{te}_2 \cdot \not= 0 \to \{ x := \text{te}_1/\text{te}_2 \} \langle \omega \rangle \phi \land (\text{te}_2 \cdot \not= 0 \to \langle \text{block;} \langle \omega \rangle \phi \rangle)}{\{ x := \text{te}_1/\text{te}_2; \langle \omega \rangle \phi \}}
\]

An error could occur arbitrarily deep in an expression. Therefore, expressions are unfolded until they consist only of a top level operator applied to terminal expressions. This is exemplified by the following rules (\(x'\) and \(x''\) are new program variables).

\[
\begin{align*}
\{ x' := e_1; x'' := e_2; x := x' + x''; \langle \omega \rangle \phi \} & \quad \{ x := e_1 + e_2; \langle \omega \rangle \phi \} \\
\{ x := e_1 + e_2; \langle \omega \rangle \phi \} & \quad \{ x := \text{te}_1 + \text{te}_2; \langle \omega \rangle \phi \}
\end{align*}
\]

In the left rule \(e_i\) are non-terminal expressions. As all expressions are unfolded, nested divisions will eventually be analyzed by DivTerminal. Other statements using expressions, like if, are unfolded in the same way, until the condition is terminal and the following rule applies:

\[
\begin{align*}
\text{if} \quad \frac{((tb \not= \text{true} \to \langle p; \langle \omega \rangle \phi \rangle) \land (tb \not= \text{false} \to \langle q; \langle \omega \rangle \phi \rangle))}{\{ \text{if } tb \text{ then } p \text{ else } q \text{ end}; \langle \omega \rangle \phi \}}
\end{align*}
\]

Note that application of this rule may lead to proof branching in subsequent steps. As for while, the unwind rule was presented in Sect. 3.2. An alternative
rule using a loop invariant is discussed in section 4.3. That rule, however, only covers the box operator. Finally, the rules for the block statement reflect the fact that a non-terminating program is always partially correct, but never totally correct:

\[
\begin{align*}
\text{blockBox} & \rightarrow \begin{cases} 
\text{true} & \text{block} \omega \phi \\
\text{false} & \text{block} \omega \phi 
\end{cases} \\
\text{blockDia} & \rightarrow \begin{cases} 
\text{false} & \langle \text{block} \omega \rangle \phi \\
\text{false} & \langle \text{block} \omega \rangle \phi 
\end{cases}
\end{align*}
\]

4.2 Interface and Class Invariants

The verification process of Creol programs is completely modular. This means we verify only one method (of one class) at the time and do not consider any other code during this process. Instead, we take into account the other threads of the object by guaranteeing the class invariant at release points and assuming it again when execution proceeds. As for the behavior of other objects, that is represented by using specification of their interfaces. An additional construct in the proof is the communication history, which both the specifications as well as the class invariants talk about. These concepts for reasoning about Creol were introduced in [17, 19].

The communication history can be viewed as a list of messages of method invocations, method completions, and object creations. For modular reasoning we always consider projections of the system wide history \( H \). Every interface is specified by an interface invariant \( \text{inv}_I(H/o/I) \), with \( o \) ranging over objects of type \( I \). The system wide history \( H \) is projected \( (H/o/I) \) to messages concerning \( o \) and talking about methods declared in \( I \). During verification at method calls and replies, \( H/this/I \) is checked against the specification. Continuing the previous example of Fig. 2 the interface USR is equipped with the following invariant:

\[
H/USR \leq (\rightarrow \text{giveCode}[\cdot \rightarrow \text{withdraw}[\cdot \rightarrow \text{dispense}]; \rightarrow \text{returnCard}])^* \]

where \( \cdot \) is appending, \( \rightarrow \) are invocation messages, \( \leftarrow \) are completion messages, brackets are used for optional occurrence, and \( ^* \) is the Kleene star. The parameters and communication partners are omitted for brevity. The invariant expresses that the history of the interface is always a prefix of this regular expression, such that an interaction with the user always begins with requesting PIN code and ends with requesting removal of the card. The interface S is specified by:

\[
H/S \leq (\rightarrow \text{authorize}(cid,.) \cdot (\leftarrow \text{authorize}(false)) \cdot (\leftarrow \text{authorize}(true) \cdot \rightarrow \text{debit}(cid,.) \cdot \leftarrow \text{debit}(.)))^* 
\]

Communication partners are omitted. The dot ‘.’ is used as a wildcard for a parameter. Parameters (including the card id \( cid \)) and communication partners are quantified universally. The meaning of the invariant is that only after authorization can the debit procedure be attempted.

We turn to the class invariant \( \text{inv}_C(H/this, W) \), which forms a contract between all threads of the object. \( W \) is the vector of class attributes. Those might get overwritten by other threads during suspension of this thread, but the invariant expresses properties of \( W \) every thread is respecting. The class invariant
is parametrized by $\mathcal{H}/\this$ which is the projection of the system wide history to the object the invariant belongs to. It contains all messages sent to or by the object this. A class invariant consists of several parts:

$$inv_C(\mathcal{H}/\this, \overline{W}) \triangleq F(\mathcal{H}/\this, \overline{W}) \land Wf(\mathcal{H}/\this) \land \forall\text{obj} \bigwedge_i inv_I(\mathcal{H}/\this/\text{obj}/I)$$

$F(\mathcal{H}/\this, \overline{W})$ relates the state of the ordinary class attributes $\overline{W}$ with the history, reflecting the refinement of the fully abstract interface specification to the local state. $Wf(\mathcal{H}/\this)$ is a predicate being interpreted to true for well-formed histories. A well-formed history starts with the creation message of this, contains invocation messages for all completion messages, and does not include any object references being null. Then, all invariants of all interfaces $I$ invoked or implemented by the class of this put in a conjunction to ensure that all methods respect them. $\text{obj}$ are the objects known by this. Now we can formulate the proof obligation for a method. The precondition is the class invariant, instantiated with a history ending on an invocation of the method. After executing the $\text{body}$ the invariant holds again for the history ending with its completion message.

$$\vdash inv_C(\mathcal{H}/\this, \overline{W}) \rightarrow [\text{body}]inv_C(\mathcal{H}/\this, \overline{W})$$

Let us proceed with an example for a class invariant. For class ATM of Fig. 3, the formula $F$ is:

$$F_{\text{ATM}}(\mathcal{H}/\this, \overline{W}) \triangleq \neg \text{server} \neq \text{null} \land \forall \text{cid}. \sum_{\text{wd}}(\mathcal{H}/\text{cid}) \triangleq \sum_{\text{deb}}(\mathcal{H}/\text{cid})$$

It states that the reference server is never null and the sum of all withdrawn money for all cards $\text{cid}$ equals the sum of the money debited. More detailed, $\sum_{\text{wd}}(h)$ calculates the sum of the money withdrawn in the history $h$. (In the equations, $msg$ is used as the ‘otherwise case’.)

$$\begin{align*}
\sum_{\text{wd}}(\epsilon) &= 0 \\
\sum_{\text{wd}}(h \rightarrow \text{withdraw}(am)) &= \sum_{\text{wd}}(h) + am \\
\sum_{\text{wd}}(h \cdot msg) &= \sum_{\text{wd}}(h)
\end{align*}$$

$\sum_{\text{deb}}(h)$ is the sum of the money debited from the corresponding bank account. Only successful debit calls are counted.

$$\begin{align*}
\sum_{\text{deb}}(\epsilon) &= 0 \\
\sum_{\text{deb}}(h \rightarrow \text{debit}(am, \text{cid}) \cdot \leftarrow \text{debit}(\text{true})) &= \sum_{\text{deb}}(h) + am \\
\sum_{\text{deb}}(h \cdot msg) &= \sum_{\text{deb}}(h)
\end{align*}$$

In the system such equations are realized as taclets (see Sect. 5).

### 4.3 Concurrent Constructs

There are two different levels of communication, namely inter-thread communication within one object via shared memory (the class attributes $\overline{W}$) and inter-object communication via method calls and replies. We start with the rules
concerning the first and focus on the latter further below. In this section we abbreviate $H$/this by $H$.

The simplest form of a release point is release. As mentioned before the class invariant forms a contract between all threads of an object. So the rule for release forces us to show that the class invariant is established in the current state, before releasing the processor. When this thread resumes, the invariant can be assumed before the remaining code $\omega$ is executed.

$$\frac{\Gamma \vdash inv_C(H,WW), \Delta \quad \Gamma \vdash \{U_{H,WW}\}[\omega]\phi, \Delta}{\Gamma \vdash [release; \omega]\phi, \Delta}$$

Here, $U_{H,WW}$ is the update $H,WW := some H,WW.(inv_C(H,WW) \land H \leq H)$. The update $U_{H,WW}$ represents an arbitrary but fixed system state satisfying the class invariant in which execution continues. By $H \leq H$ we denote that the old history $H$ is a prefix of the new one $H$. The update is necessary because values of the class attributes could have been overwritten by other threads, and because $H$ might have grown meanwhile.

Note that this rule, as well as all rules in this section, can also be applied when the modality is preceded by updates, which is the typical scenario. These updates are preserved in the instantiation of the premises (see [11]).

The await $b$ statement is handled by a similar rule, with the additional assumption that the guard $b$ holds when execution resumes. A minor complication is that we also must assume that evaluation of $b$ does not block due to an error.

$$\frac{\Gamma \vdash inv_C(H,WW), \Delta \quad \Gamma \vdash \{U_{H,WW}\}((x := b)x \equiv true \rightarrow [\omega]\phi), \Delta}{\Gamma \vdash [await b; \omega]\phi, \Delta}$$

By replacing $(x := b)x \equiv true$ with $Comp(H,l)$ in the above rule, we get a rule for await $l$?. The predicate $Comp(H,l)$ is valid if a completion message with the label $l$ is contained in the history $H$. The handling of $Comp(H,l)$ in the proof is discussed further below.

Partial correctness of a loop can also be shown with help of a loop invariant $inv_{loop}(H,mod)$, where $mod$ is the modifier set of the loop (all variables assigned in the loop). To be most general, all class attributes could be included in the modifier set. The history could be omitted as a parameter of the loop invariant if there are no method calls, method completions or object creations in the loop body.

$$\frac{\Gamma \vdash (x:=b)true \rightarrow inv_{loop}(H,mod) \land Wf(H), \Delta \quad \Gamma \vdash \{U_{H,mod}\}((x := b)x \equiv true \rightarrow [p]inv_{loop}(H,mod)), \Delta \quad \Gamma \vdash \{U_{H,mod}\}((x := b)x \equiv false \rightarrow [\omega]\phi), \Delta}{\Gamma \vdash [while b do p end; \omega]\phi, \Delta}$$

The update $U_{H,mod}$ is defined as:

$$H,mod := some H,m.(Wf(H) \land H \leq H \land inv_{loop}(H,m))$$
It creates a new history $H$ and a new modifier set, such that the loop invariant holds. If the condition $b$ of the loop contains an exceptions the implication of all branches are true.

Analogous to $\text{Comp}(H, l)$ there are predicates $\text{Invoc}(H, l)$ and $\text{New}(H, o)$ which guarantee the existence of an invocation message with label $l$ and an object creation message with reference $o$ in the history $H$, respectively. During a proof, uncertainty is inherent in the projection of the history to this, as there could be incoming method invocations at any time. When dealing with method calls we only state the existence of a corresponding message in the history. We do not append it to the history. In general all rules of Sect. 4.1 would need to cover potential extensions, using the prefix predicate $\leq$. It is however equivalent to extend the history on access (release points, method calls, etc.).

To exemplify some properties of the predicates dealing with the history we give the following formula which is a tautology.

$$\text{Comp}(H_0, l) \land H_0 \leq H_1 \rightarrow \text{Comp}(H_1, l)$$

Besides $\text{Comp}$, $\text{New}$, as well as $\text{Invoc}$ are monotonous w.r.t. $\leq$. Additionally, the contra-position is used in our proof system.

We turn attention towards method invocation $l \circ \circ \text{mtd}(p \circ \text{in})$. Its execution assigns a unique reference to $l$, and extends the history by the corresponding invocation message:

$$\Gamma \vdash Wf(H) \land inv_{I}(H/o/I), \Delta$$
$$\Gamma \vdash o \equiv \text{null} \rightarrow \{\text{block}; \omega\} \phi, \Delta$$
$$\text{invoc}$$
$$\Gamma \vdash -o \equiv \text{null} \rightarrow \{l := (\text{this}, o, mtd, p \circ \text{in}, i)\} \{U^{\text{invoc}}_{H}\} [\omega] \phi, \Delta$$

If $o$ is null, execution blocks. In the first branch, the invariant of the remote interface $I$ must be shown ($I$ being the type of $o$). The index $i$ is new and assures uniqueness of the label $l$. The abbreviation $U^{\text{invoc}}_{H}$ for the update, is in its full form:

$$H := \text{some } H \circ \circ \{Wf(H) \land H \leq H \land inv_{I}(H/o/I, \Delta) \land \text{Invoc}(H, l) \land \neg Invoc(H, l)\}$$

The new history contains the invocation message $\text{Invoc}(H, l)$. As the label $l$ is unique the invocation message must not be included in the previous history ($\neg \text{Invoc}(H, l)$), which prefixes the new one ($H \leq H$). The new history $H$ is well-formed ($Wf(H)$) and it respects the interface invariant $inv_{I}(H/o/I, \Delta)$ where the in-parameters $p$ are added as they occur in the appended invocation message.

A completion statement $\text{comp}(\overline{p_{out}})$ assigns the return parameters of the method call identified by the label $l$ to $p_{out}$. If the label $l$ is null, execution blocks.

$$\Gamma \vdash \text{Invoc}(H, l) \land Wf(H) \land inv_{I}(H/l.\text{callee}/I), \Delta$$
$$\Gamma \vdash l \equiv \text{null} \rightarrow \{\text{block}; \omega\} \phi, \Delta$$
$$\text{comp}$$
$$\Gamma \vdash -l \equiv \text{null} \rightarrow \{U^{\text{comp}}_{H, \overline{p_{out}}}\}[\omega] \phi, \Delta$$
$$\Gamma \vdash [l(\overline{p_{out}}); \omega]\phi, \Delta$$
As we are extending the history with a completion message, we check the existence of the corresponding invocation message by $Invoc(H,l)$ to ensure well-formedness. The selector $callee$ delivers the reference of the sender of the completion message. $U^comp_{H,pout}$ is analogous to $U^{invoc}_{H}$ where the only difference is that $p_{out}$ is overwritten and $Comp$ is used instead of $Invoc$.

$$H,p_{out} := some\ H,\ p.$$  

$$Wf(H) \land H \leq H \land inv_I(H/l.callee/I,p) \land Comp(H,l) \land \neg Comp(H,l)$$

We omit the rule for object creation, mentioning only that the new reference is constructed by the pair (this, $i$), here $i$ is an object local, successively incremented index. An alternative, fully abstract modeling of object creation in DL is investigated in [4] and can be adapted also here.

Finally, we consider the return statement. It sends the completion message belonging to the method call of the verification process and the thread terminates afterwards. The class invariant is not explicitly mentioned in the following rule as it is contained in $\phi$ (see previous section).

$$\Gamma \vdash Invoc(H,l) \land Wf(H) \land inv_I(H/caller/I),\Delta$$  

$$\Gamma \vdash \{U^return_H\}\phi,\Delta$$

$$\Gamma \vdash \langle return(p_{out}) \rangle \phi,\Delta$$

Here, $l$ is the label of the message which created the thread subject to verification, $I$ the corresponding interface, and $caller$ the corresponding caller. The update $U^return_H$ adds the completion message to the history which must not occur in the previous history.

$$H := some\ H,\ (Wf(H) \land H \leq H \land inv_I(H/caller/I) \land Comp(H,l) \land \neg Comp(H,l))$$

5 A System for Creol Verification

The verification system for Creol is based on KeY[9]. Written in Java and published under the GNU general public license, it is available from the project’s website\(^5\). The current version is a prototype which provides the functionalities presented in this paper. It has a graphical user interface where the proof tree and open proof goals are displayed. Other features are pretty-printing and syntax-highlighting of the subformula/subterm currently pointed at with the mouse pointer. This enables a context sensitive menu offering only the rules applicable to the highlighted subformula/subterm. Apart from the rule name, tool-tips describe the effect of a rule. Besides interactive application of rules, automatic strategies can be configured. A more detailed description of the KeY interface is available in [3].

Problem files, logical rules, and axiomatizations of data types are written in the taclet language [43]. In Fig. 6 the rule impRight from Fig. 4 and the equation Eq. (2) are defined in the taclet language. A find describes the formula the rule

\(^5\) www.key-project.org
is applicable to. replacewith specifies the replacement for the find formula, assumes characterizes further assumptions not subject to replacements, and add causes its argument to be added. The arrow \( \Rightarrow \) indicates on which side of the sequent the formulas are found, replaced or added. Writing a semicolon between two occurrences of replacewith or add causes a branching. Taclets omitting the sequence arrow \( \Rightarrow \) are rewriting rules applicable in all contexts.

\[
\text{impRight} \{ \text{find}(\Rightarrow \phi \rightarrow \psi) \Rightarrow \text{replacewith}(\Rightarrow \psi) \Rightarrow \text{add}(\phi \Rightarrow) \} \\
\text{compMon} \{ \text{find}(\text{Comp}(H_1,L) \Rightarrow) \Rightarrow \text{assumes}(\text{Prefix}(H_1,H_2) \Rightarrow) \Rightarrow \text{add}(\text{Comp}(H_2,L) \Rightarrow) \} 
\]

Fig. 6. Rules in the taclet language

The theory explained in the previous section needed some small extensions to be run in the system. First, the some quantifier was not implemented, but is expressed by another formula. For example, the update formula like \( \{H := \text{some } H. (Wf(H) \land H \leq H) \} \phi \) is rewritten to:

\[
\forall H_0. \{H \equiv H_0 \rightarrow \forall H_1. \{H := H_1\}((Wf(H_1) \land H_0 \leq H_1) \rightarrow \phi)\}
\]

The old value of \( H \) is saved in \( H_0 \), and the new variable \( H_1 \) is assigned to \( H \). The implication assures that \( H_1 \) has the desired properties when evaluating \( \phi \).

Finally, there are different prefix predicates \( \leq_I \) where \( I \) is an interface. Thereby the interface invariant for \( I' \) is monotonous on \( \leq_I \) if \( I' \neq I \). The rules invoc, comp, and return use \( \leq_I \) where \( I \) is the interface the message the rule adds corresponds to. Release points and the loop invariant use a prefix predicate \( \leq_{all} \) which is not monotonous for interface specifications.

The Creol parser is written in about 3900 lines of code using ANTLR as parser generator. The adaptions in the KeY-system took another 5000 lines. Finally, the rules written in the taclet language are about 1700 lines long.

6 Verification Examples

6.1 Unbounded buffer

We give an implementation for an unbounded first-in-first-out (FIFO) buffer. This example is adapted from [18]. The interface contains two methods put and get which can be used to put into and to obtain an element from the buffer.

\[
\text{interface} \text{FifoBuffer} \\
\text{begin with} \text{Any} \\
\text{op} \text{put(in } x: \text{Any; out) } \\
\text{op} \text{get(in; out } x: \text{Any)} \\
\text{end}
\]

The interface invariant expresses that the sequence of elements retrieved from the buffer are a prefix of the elements put into the buffer. This ensures the FIFO
property. Additionally, no element must equal null. We define \( inv_1(H, callee) \) (slightly simplified) as:

\[
\text{out}(H/I, callee) \leq \text{in}(H/I, callee) \land \forall x.(x \in \text{in}(H/I, callee) \rightarrow \neg x \doteq \text{null})
\]

where \( I \) is \( \text{FifoBuffer} \) and \( \text{in}, \text{out} \) are defined as:

\[
\begin{align*}
\text{in}(\epsilon, o) &= \epsilon \\
\text{in}(h \cdot o_2 \leftarrow o.\text{put}(x);), o) &= \text{in}(h, o) \cdot x \\
\text{in}(h \cdot \text{msg}, o) &= \text{in}(h, o)
\end{align*}
\]

\[
\begin{align*}
\text{out}(\epsilon, o) &= \epsilon \\
\text{out}(h \cdot o_2 \leftarrow o.\text{get}(;), o) &= \text{out}(h, o) \cdot x \\
\text{out}(h \cdot \text{msg}, o) &= \text{out}(h, o)
\end{align*}
\]

Note that we do not guarantee that a caller gets the same objects it has put into the buffer. Such a buffer can be used for fair work balancing where a request is put into the buffer and workers take them out again.

The implementation of the buffer, given in Fig. 7, uses a chain of objects where each of them can store one element. The attribute cell is null if the object does not store an element. In next the reference to the following chain of objects is stored. Requests are forwarded to it if the object cannot serve them alone. The variable cnt holds the number of elements stored in cell and all following objects. Calls of get on an empty buffer are suspended until there are elements in the buffer.

```java
class BufferImpl implements FifoBuffer {
    var cell:Any; var cnt:Int; var next:FifoBuffer;

    begin with Any
        op put(in x:Any; out) ===
            if cnt=0 then cell:=x
            else if next=null then next:=new Buffer end;
            var l:Label[ ]; l!next.put(x); l?()
            end;
            cnt:= cnt+1; return()
        op get(in ; out x:Any) ===
            await (cnt>0);
            if cell=null then var l:Label[ ]; l!next.get(); l?(x)
            else x:=cell; cell:=null
            end;
            cnt:=cnt−1; return(x)
    end
}
```

Fig. 7. The class implementing the buffer

For the class invariant we define another term \( buf(o_1, o_2, h) \) which for an object \( o_1 \) and its next object \( o_2 \) reconstructs from the history \( h \) the elements in
cell and all following objects.

\[ \text{buf}(o_1, o_2, h) = \begin{cases} 
\epsilon & \text{if } h = \epsilon \lor o_1 = \text{null} \lor o_2 = \text{null} \\
\text{buf}(o_1, o_2, h') \cdot x & \text{if } h = h' \cdot o_1 \leftarrow o_2.\text{put}(x) \\
\text{rest}(\text{buf}(o_1, o_2, h')) & \text{if } h = h' \cdot o_1 \leftarrow o_2.\text{get}(); x \\
\text{buf}(o_1, o_2, h') & \text{otherwise } h = h' \cdot \text{msg}
\end{cases} \]

\text{rest} removes the first element of a sequence. Let us proceed with the class invariant. The attribute \text{cnt} equals the number of elements in \text{cell} and all following buffer cells. The interface invariant of \text{FifoBuffer} has to hold for both the interface called and implemented by the class. Additionally, we state that the sequence of values put into the current cell equals the sequence of values obtained from the buffer with the \text{cell} and the content of the following buffer appended.

\[ |\text{cell} \cdot \text{buf}(\mathcal{H}/\text{next}, \text{this}, \text{next})| = \text{cnt} \]

\[ \land (\neg \text{next} = \text{null} \rightarrow \text{invI}(\mathcal{H}/\text{next}, \text{next})) \land \text{invI}(\mathcal{H}, \text{this}) \]

\[ \land \text{in}(\mathcal{H}, \text{this}) = \text{out}(\mathcal{H}, \text{this}) \cdot \text{cell} \cdot \text{buf}(\mathcal{H}/\text{next}, \text{this}, \text{next}) \]

In the above formula \(I\), is instantiated by \text{FifoBuffer} and \(\mathcal{H}\) is an abbreviation for \(\mathcal{H}/\text{this}\). If \text{cell} is null it is omitted. The example with the given specifications was proved interactively by the system. The method \text{put} was verified in 1024 proof steps and 80 branches, whereas \text{get} needed 587 proof steps and 43 branches. Great parts of the proof were transformations of the sequences the buffer was specified with. However they went rather smoothly as the problem of the equality of two sequences is human-readable even if the automated strategy gets stuck. It seems that a logical toolbox expressing sets, relations and other well-understood mathematical notions would simplify the process of specifying and verifying other case studies.

6.2 Automated teller machine

The example of the automated teller machine distributed throughout the paper was successfully verified in 2495 steps (27 branches) by the system. As the implementation of the class makes heavy use of asynchronous method calls and \((co)\)interfaces, it has been shown that our system can deal with them. The amount of method calls produces a chain of prefixed histories where the monotonicity of properties has to be used often. This leads to a number of predicates expressing properties of histories on the left-hand-side of the sequent. Hence, the automated strategy must use the monotonicity with care to improve readability if a branch cannot be closed by it. The experiences with specifications in form of regular expressions were promising. They are easy to write down and a automated strategy can deal with them as the number of successor states is usually limited which narrows the search space of the proof.

7 Discussion and Conclusion

Creol’s notion of inter-object communication is inspired by notions from process algebras (CSP [27], CCS [35], \(\pi\)-calculus [36]), which however model syn-
chronous communication mostly. Moreover, Creol differs from those in integrating the notion of processes in the object-oriented setting, using named objects and methods rather than named channels. This also introduces more structure to the message passing (calls, replies, caller references, counterfaces). The message passing paradigm on the inter-object level is combined with the shared memory paradigm on the local inter-thread level. Early approaches to the verification of shared memory concurrency are interference freedom based on proof outlines [41] and the rely/guarantee method [33]. Other approaches use object invariants as a combined assumption/guarantee, both in the sequential setting to achieve modularity [7, 8], and in the concurrent setting [28]. Compared to the last mentioned works, Creol is more restrictive in that it forces shared memory to be entirely object internal. All knowledge of remote data is contained in fully abstract interface specifications talking about the communication history. Communication histories appeared originally both in the CSP as well as the object-oriented setting [14, 27], and were used for specification and verification for instance in [45, 16].

KeY is among the state-of-the-art approaches to the verification of (at first) sequential object-oriented programs, together with systems like Boogie [6], ESC/Java(2) [23], and Krakatoa [22]. In comparison to those, KeY is unique in that it does not merely generate verification conditions for an external off-the-shelf prover, but employs a calculus where symbolic execution of programs is interleaved with first-order theorem proving strategies. This goes together with the nature of first-order DL, which syntactically interleaves modalities and first-order operators. The cornerstone for KeY style symbolic execution, the updates, have similarities to generalized substitutions in formalisms such as the B method [2]. Updates are, however, tailored to symbolic execution rather than modeling (for instance, conflicts are resolved via right-win). The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [21] and for symbolic debugging. The role of updates is largely orthogonal to the target language, allowing us to fully reuse this machinery for Creol.

As for Creol’s thread concurrency model, this differs from many other languages in that it is cooperative, meaning the programmer actively releases control (conditionally). This simplifies reasoning considerably as compared to reasoning about preemptive concurrency, where atomicity has to be enforced by dedicated constructs. There is work on verifying a limited fragment of concurrent Java with KeY [10]. Here, the main idea is to prove the correctness of all permutations of schedulings at once. In [1], concurrent correctness of Java threads is addressed by combining sequential correctness with interference freedom tests and cooperation tests.

Very related to our work is the extension of the Boogie methodology to concurrent programs [28], targeting concurrent Spec#. From the beginning, this work is deeply integrated into an elaborate formal development environment, with all the features mentioned the first paragraph of this paper. The methodology requires users to annotate code with commands in between which an object
is allowed to violate its invariant. This is combined with ownership of objects by threads. Just as in our system, invariants have to be established at specific points, and can be assumed at others. Also similar is the erasing of knowledge, there with the havoc statement, here with the some operator. Differences (apart from the asynchronous method calls) are the purely cooperative nature of our threads, and that our shared memory is object local, which makes ownership trivial. Connected to this is the inherently fully abstract specification of remote object interfaces, employing histories. The Boogie approach can simulate histories as well (see Fig. 1 in [28]), but it lies in the responsibility of the user whether or not the simulated history reflects the real one.

The system presented in this paper is still a prototype. It supports Creol dynamic logic, but the front-end for loading code and generating proof obligations is yet unfinished. This however will not be a real challenge, given the KeY infrastructure. Also, the automated strategies are very rudimentary yet. We currently achieve an automation of 90% (automatic per total proof steps), which is very low by our standards. As we are only at the beginning of the work on automated strategies tailored to Creol, there is great potential here. The true challenge has been the omnipresence of the history, and it is here that future research on verification in this domain will focus on. This concerns various levels: better support for history based specifications, like a library of frequently used queries on histories, or the usage of specification patterns [13], extended and configurable proof support for history based reasoning, and improved presentation on the syntax level and in the user interface.

We consider Creol’s approach to modular object-oriented modeling as a good basis for scaling ‘sequential formal methods’ to the concurrent distributed setting, in particular when targeting functional correctness. The key is a very strong separation of concerns, which however naturally follows ultimate object-oriented principles. KeY has proved to be a good conceptual and technical basis for such an undertaking, which we argue can lead to an efficient and user-friendly environment for the verification of distributed object applications.

Acknowledgments

The authors would like to thank Frank de Boer, Einar Broch Johnsen, Olaf Owe, and Martin Steffen for fruitful discussions on the subject, Richard Bubel and Markus Drescher for their comments on drafts of this paper, Richard Bubel moreover for his guidance concerning implementation issues, and the anonymous reviewers for detailed comments.

References

11. B. Beckert, V. Klebanov, and S. Schlager. Dynamic logic. In Beckert et al. [9], pages 69–177.
37. W. Mostowski. The demoney case study. In Beckert et al. [9], pages 533–568.
B.2 A system for compositional verification of asynchronous objects
A system for compositional verification of asynchronous objects

Wolfgang Ahrendt a,∗, Maximilian Dylla b

a Department of Computer Science and Engineering, Chalmers University of Technology, Sweden
b Saarbrücken Graduate School of Computer Science, Saarland University, Germany

A R T I C L E   I N F O

Article history:
Received 19 April 2010
Received in revised form 2 August 2010
Accepted 5 August 2010
Available online xxxx

Keywords:
Verification
Concurrency
Semantics
Object-orientation

A B S T R A C T

We present a semantics, calculus, and system for compositional verification of Creol, an object-oriented modelling language for concurrent distributed applications. The system is an instance of KeY, a framework for object-oriented software verification, which has so far been applied foremost to sequential Java. Building on KeY characteristic concepts, like dynamic logic, sequent calculus, symbolic execution via explicit substitutions, and the taclet rule language, the presented system addresses functional correctness of Creol models featuring local cooperative thread parallelism and global communication via asynchronous method calls. The calculus heavily operates on communication histories specified by the interfaces of Creol units. Two example scenarios demonstrate the usage of the system. This article extends the conference paper of Ahrendt and Dylla (2009) [5] with a denotational semantics of Creol and an assumption-commitment style semantics of the logic.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

The area of object-oriented program verification made significant progress during the last decade. Systems like Boogie [8], ESC/Java2 [31], KeY [12], and Krakatoa [30] provide a high degree of automation, elaborate user interfaces, extensive tool integration, support for various specification languages, and high coverage of a real world target language (C# in case of Boogie, Java in case of the other mentioned tools).

However, this development mostly concerns sequential, stand-alone applications. When it comes to verifying functional properties of concurrent and distributed applications, the situation is different. Even if there is a range of literature on the verification of ‘distributed formalisms’ (based for instance on process calculi [46,35,47]), there are hardly any systems yet matching the aforementioned characteristics. Moreover, many formalisms lack a connection to the dominating paradigm of today’s software engineering, object-orientation, which is an obstacle for the integration into software development environments and methods.

This work is a contribution towards effective and integrated verification of concurrent, distributed systems. We present a verification system that is built on two foundations: the Creol modeling language for concurrent and distributed object-oriented systems [43], and the KeY approach and system for the verification of object-oriented programs [12]. By combining KeY’s proving technology with Creol’s novel approach to modular modeling of components, which has been successfully applied to industrial scale problems [21], we achieve a system for compositional verification of concurrent, distributed object-oriented applications. While still being a prototype system, past experience with the technological and conceptual basis justifies the perspective of future versions to enjoy similar features as state-of-the-art sequential verification systems already do. The scaling up of verification technology from the sequential to the concurrent/distributed setting would, however, would not be possible without modularity being the central element in the design of Creol, as discussed in the following.

∗ Corresponding author.
E-mail addresses: ahrendt@chalmers.se (W. Ahrendt), mdylla@mpi-inf.mpg.de (M. Dylla).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.08.003

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
Creol is an executable object-oriented modeling language. It features concurrency in two ways. First of all, different objects execute truly in parallel, as if each object had its own processor. Objects have references to each other, but cannot access each other’s internal state. Consequently, there is no remote access to attributes, like ‘o.a’ in other languages. The only way for objects to exchange information is through methods. Calls to methods are asynchronous [42], in the sense that the calling code is able to continue execution even before the callee replies. Mutual information hiding is further strengthened by object variables being typed by interfaces only, not by classes. The loose coupling of objects, their strong information hiding and true parallelism, is what suggests distributed scenarios, with each object being identified with a node. The second type of concurrency is object internal. Each call to a method spawns a separate thread of execution. Within one object, these threads execute interleaved, with only one thread running at a time. Here, the key to modularity is the cooperative nature of the scheduling: a thread is only ever interrupted when it actively releases control at ‘release points’.

Altogether, Creol allows compositional verification. Within one class, the various methods can be proved correct in isolation, in spite of the shared memory (the attributes), by guaranteeing and assuming a class invariant at each release point in the code. At the inter-object level, the vehicle to connect the verification of the various classes is the ‘history’ of inter-object communications. Interface invariants are expressed in terms of the history only, while class invariants relate the history with the internal state. The fact that each object has only partial knowledge about the global communication history is modeled by projecting the global history onto the individual objects [41]. A reader with no prior exposition to Creol may consider the tutorial [38].

Our system is based on the KeY framework for verifying object-oriented software. The most elaborate instance of KeY is a verification system for sequential Java [12]. Other target languages of KeY are C [51], ASMs [52], and hybrid systems [55]. What all these have in common is that they use dynamic logic, explicit substitutions, and a sequent calculus realised by the ‘taclet’ language. These concepts, to be introduced in the course of the paper, have proved to be a solid foundation of a long lasting and far reaching research project and system for verifying functional correctness of Java [12]. Dynamic logic features full source code transparency, like Hoare logic, but is more expressive than that. Explicit (simultaneous) substitutions, called updates, provide a compact representation of the symbolic state, and allow a natural forward style symbolic execution. Apart from verification, updates are also employed for test case generation and symbolic debugging. Sequent calculi are well-suited for the interleaved automated and interactive usage. And finally, taclets provide a high-level rule language capturing both the logical and the operational meaning of rules. They are well suited both for the base logic and for the axiomatization of application specific operations and predicates. KeY has been used in a number of case studies, like the verification of the Java Card API Reference Implementation [50], the Mondex case study (the most substantial benchmark in the Grand Challenge repository) [57], the Schorr-Waite algorithm [16], and the electronic purse application Demoney [49]. The system is also used for teaching in various courses at Chalmers University and several other universities.

However, the KeY approach has so far almost only been applied to the sequential setting.¹ It is precisely the described modularity of Creol that allowed us to base our verification system on the same framework. The main challenges for adjusting the KeY approach to Creol were the handling of asynchronous method calls, the handling of release points, and, most of all, the extensive usage of the communication history throughout the calculus.

This article extends the conference paper [5] with a denotational semantics of Creol (Section 3) and with an assumption-commitment/rely-guarantee style semantics of the logic (Section 5). Moreover, the calculus presented here is simplified with respect to [5]. On the implementation side, new strategies were realised, resulting in an automation degree of more than 98% in the examined case studies, see Section 8.

Ultimately, the presented semantics and calculus should be connected, by (a) proving soundness of the calculus, (b) investigating the degree of completeness, and (c) precisely defining and proving compositionality in this setting. These issues are future work, see also Section 9.

The structure of the paper is as follows. Section 2 introduces Creol, and gives examples of its usage. Section 3 then presents a denotational semantics of Creol. In Section 4, we describe the logic and calculus characteristic for KeY, insofar as it is (largely) independent of the particular target language. Thereafter, Section 5 presents the semantics of the logic, followed by Section 6 containing the calculus for Creol specifically. Section 7 discusses system oriented aspects of KeY for Creol, including a small account on taclets. Section 8 then demonstrates the usage of the systems in examples. In Section 9, we discuss related work, and draw conclusions. Finally, Appendix A defines the syntax of Creol, followed by Appendix B and Appendix C serving as a reference for the semantics and the calculus, respectively.

2. Overview of Creol

In this section, we introduce Creol, using an automated teller machine scenario adapted from [39]. The example will also be used to discuss Creol verification in later sections. The scenario we consider has three kinds of actors. There are several teller machines (class ATM), several users (class User), and one server (class Server). In the course of a certain session, a teller machine communicates with one user, and with the server, as depicted in Fig. 1. The picture shows that, while User and Server implement one interface each (USR resp. S), the class ATM implements two interfaces, ATMU and ATMS, dedicated for the communication in either of the directions. The Creol definition of the interfaces is given in Fig. 2. (We omit ATMS, which is empty.)

¹ See Section 9 for an exception.
Fig. 1. Communication of the automated teller machine, where the dotted arrow represents return messages only, whereas the other arrows stand for both initiated communication and replies.

interface USR
begin
with ATMU
op giveCode(in; out code: Int)
op withdraw(in; out amount: Int)
op dispense(in amount: Int; out)
op returnCard(in; out)
end

interface ATMU
begin
with USR
op insert(in cardId: Int; out)
end

interface S
begin
with ATMS
op authorize(in cardId: Int, code: Int; out ok: Bool)
op debit(in cardId: Int, amount: Int; out ok: Bool)
end

Fig. 2. The interfaces of the automated teller machine.

class ATM implements ATMS, ATMU
begin
var server : S;
with USR
op insert(in cardId: Int; out) ==
var li: Label[Int]; var lb: Label[Bool]; var l: Label[];
var l2: Label[]; var code: Int; var ok: Bool; var am: Int;
li!caller.giveCode(); li?(code);
lb!server.authorize(cardId, code); lb?(ok);
if ok
then li!caller.withdraw(); li?(am);
lb!server.debit(cardId, am); lb?(ok);
else li!caller.returnCard(); li?
else li!caller.returnCard(); li?
end
return ()
end

Fig. 3. The class implementing the teller machine.

We can observe that the signature of operations contain (possibly empty) lists for in- and out-parameters. The operations offered by interfaces appear in the scope of `with cointerface`, with the meaning that those operations can only be called from instances of classes implementing that cointerface. For instance, the server cannot call `insert` on a teller machine, not even if it was in the possession of an ATMU typed reference. Another consequence of cointerfaces is that the implementations of operations have a well-typed reference to the caller, without that reference being passed explicitly as an input parameter.

The class ATM in Fig. 3 is an example for a class definition. Variables are implicitly initialised with `false` or 0 for primitive types, and null for labels and object references. Some variables are declared of type `Label[...], like `var li: Label[Int]. Later, the execution of the call `li!caller.giveCode()`, for instance, allocates a new label, and assigns it to `li`. The label is later used in the reply statement `li?(code)`, to associate the reply with the respective call. The effect of the reply is that `code` is assigned the output of the (`li-labeled`) call to `giveCode`, provided that the according reply message has already arrived. Otherwise, the statement blocks, without the thread releasing control. (This ‘busy waiting’ can be avoided by using the `await` statement, see below.) The effect of `li?(x)` is similar to treating `x` as a future variable [20, 7] or promise [45]. In a label type `Label[T]`, the `T` indicates the type of the output of the called operation.

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
Note that the calls to dispense and returnCard are executed before any of the replies is asked back. This allows the two called methods to execute interleaved on the processor of the called object. (Note that the calls went to the same object.)

In general, arbitrary code can be executed between a call and the corresponding reply. We want to highlight that the implementation of insert extensively uses the caller reference, which is known to be of type USR, for callbacks. This style of coupling communicating objects might clarify the distribution of operations over interfaces in the teller machine scenario (cf. Fig. 2).

We discuss further features of Creol not captured by the above example. New objects are created by \( x := \text{new} \ C(e^*) \), where \( C \) is a class identifier supplied with a list of class parameters. As indicated earlier, \( P(x^*) \) blocks execution, without releasing control, until the corresponding reply message has arrived. In contrast, the command \( \text{await} \ P \) releases control if the reply for \( l \) has not yet arrived, such that the scheduler can pass control to another thread of this object. Other release points are \( \text{await} \ b \), releasing control if the Boolean expression \( b \) is false, and the unconditioned \( \text{release} \). The example code above did not contain release points, but see the buffer example in Section 8.1 (Fig. 8).

In Creol, expressions have no effect on the state. We model errors, like division by zero, by non-terminating (and non-releasing) blocking. The same holds for a call on the null reference and a reply on the null label.

3. Denotational semantics for Creol

Previous work on the semantics of Creol focused on operational semantics [40,15]. In this article, we present a denotational semantics of Creol. It is an intrinsic feature of denotational semantics that they are compositional. This is a very good fit to our goal of compositional verification, because the compositional calculus can relate in a natural way to the semantics.

The comprehensive surveys of de Roever et al. on compositional verification of concurrent programs [22,36] were a great inspiration for this work. One basic principle, invented by Zwiets [59], is to construct histories of process interactions for each process independently, by non-deterministically ‘guessing’ the relevant observations on other processes. Then, in the composition of processes, we merge those histories which ‘agree’ on certain observations. That merge is defined as the inverse of a projection. We drive this ‘guess-and-merge’ principle very far, to cope with dynamic creation of arbitrarily many objects and threads. For instance, we will ‘guess’ the number of times a certain method is called, to then require that the result can only be merged with histories actually providing the right number of calls (among other things).

We proceed in a bottom-up manner, ranging from single statements via method bodies and objects to the semantics of the complete program. As a first ingredient we use a state denoted by \( \sigma \). The state is a partial function pointing from the local variables of a thread and the object attributes to their current values. Sometimes we restrict the preimage of the state to the local variables or the object attributes by writing \( \sigma |_l \) or \( \sigma |_o \), respectively. The other important member of our semantics is the history \( \theta \) being a sequence of messages. Our semantics is defined by a function on programs

\[
M : \text{PROG} \rightarrow (\Sigma \rightarrow 2^{\Sigma \times H})
\]

where \( \text{PROG} \) is the set of programs, \( \Sigma \) denotes the set of all states and \( H \) contains all histories. The function \( M \) associates to a program a function which relates every initial state with a set of pairs containing the possible states and histories the program terminates with. Note that the resulting history only reflects the run of this very program, not any ‘initial’ history. (But see sequential composition below, and the semantics of formulas relative to an initial history, Section 5). Programs that never terminate have the empty set as their semantics. As an example let us look at the block statement, which never terminates.

\[
M(\text{block})(\sigma) = \{\}
\]

\[
M(\text{skip})(\sigma) = \{(\sigma, \{\})\}
\]

The skip statement terminates, but causes no changes, so its set includes the tuple of the input state \( \sigma \) and the empty history \( \{\} \). Proceed with more interesting statements we turn our attention towards the assignment. Clearly, the state has to be modified, because the value of the assigned variable \( x \) might have changed.

\[
M(x := e)(\sigma) = \{(\sigma', \{\}) \mid \exists v. \ e \mathrel{=} e(\sigma), \sigma' = (\sigma : x \leftarrow v)\}
\]

(By \( (\sigma : x \leftarrow v) \) we mean the modification, or extension, of \( \sigma \) at \( x \) with \( v \).) Thus the set contains the state \( \sigma' \) which equals the old state \( \sigma \) up to the value for \( x \). Here and later on, we write \( e(\sigma) \), where evaluation of expression \( e \) with respect to the state \( \sigma \). We chose \( \mathcal{E} \) to be a partial function, not always returning a value. In particular, if the evaluation of \( \mathcal{E} \) encounters a division by zero, \( \mathcal{E} \) does not return a value. As we quantify the value \( v \) existentially, division by zero in \( e \) makes the condition of the set always false, such that \( M(x := e)(\sigma) \) becomes the empty set. This demonstrates that our semantics models ‘abnormal’ termination by non-termination.

For the sequential composition of statements \( S_1 \) and \( S_2 \) (which might be in turn sequences of statements) we give the following semantics

\[
M(S_1;S_2)(\sigma) = \{(\sigma_2, \theta_1^\top \theta_2) \mid \exists \sigma_1. (\sigma_1, \theta_1) \in M(S_1)(\sigma), (\sigma_2, \theta_2) \in M(S_2)(\sigma_1)\}
\]

\[\text{This is equivalent to modelling } \mathcal{E} \text{ as a relation which is functional.}\]
where $\sqcup$ denotes the concatenation of histories. A more involved form of sequential composition is the loop statement. 

$$
\mathcal{M}(\text{while } b \text{ do } S \text{ end})(\sigma_0) = \begin{cases}
(\sigma, \theta) \\
\exists k \in \mathbb{N}, (\sigma_1, \theta_1), \ldots, (\sigma_k, \theta_k) \text{ such that } \\
\sigma = \sigma_1, \theta = \theta_1 \cdots \theta_k, \mathcal{B}(-b)\sigma_k, \\
\text{for } i = 0, \ldots, k-1 : \mathcal{B}(b)\sigma_i, (\sigma_{i+1}, \theta_{i+1}) \in \mathcal{M}(S)(\sigma_i)
\end{cases}
$$

We consider $k - 1$ executions of the loop body $S$, where every time the body is started in an end-state of the previous execution. So the Boolean condition $b$ has to hold before those $k - 1$ repetitions being expressed by $\mathcal{B}(b)\sigma_k$. Finally, when the entry condition $b$ is checked the $k$-th time, we have $\mathcal{B}(-b)\sigma_k$. The history of all those runs is concatenated as in the previous semantics for sequential execution. If the loop never terminates there is no such $k$, leading to an empty set. The remaining sequential statements are given in Appendix B.

We continue with object internal parallelism via shared memory. The release statement allows another thread to run. 

$$
\mathcal{M}(\text{release})(\sigma) = \{ (\sigma', (\text{yield}(\sigma|_o))^{-} (\text{resume}(\sigma'|_a))) | \sigma'|_i = \sigma|_i \}
$$

(1)

The compositional semantics stores a yield-resume pair here, in order to mark the points where later, when merging ‘parallel’ histories, a thread switch is allowed. Locally, we mimic the effect of other threads by allowing all possible values of the object attributes, while the local variables are preserved in the new state $\sigma'$. Later, when we compose the histories of several threads, we will only allow the switch of control from a yield to the resume of some thread such that both store the same attribute values, see Eqs. (7) and (8).

The other form of a releasing statement is the await statement where execution is released, and only continued if a condition $b$ is fulfilled which is the meaning of $\mathcal{B}(b)\sigma'$. 

$$
\mathcal{M}(\text{await } b)(\sigma) = \{ (\sigma', (\text{yield}(\sigma|_o))^{-} (\text{resume}(\sigma'|_a))) | \sigma'|_i = \sigma|_i, \mathcal{B}(b)\sigma' \}
$$

(2)

await can also check for the termination of another thread, using the condition ‘?’. Here, a little complication is added to the semantics. The message $\langle \text{comp}(\ell(b)\sigma, \bar{v}) \rangle$ will later, in the parallel composition of histories, serve as an assertion to the history that the thread serving the call belonging to the label $l$ has completed. (The return values $\bar{v}$ remain unused here in contrast to Eq. (4.).)

$$
\mathcal{M}(\text{await } l?)(\sigma) = \{ (\sigma', (\text{yield}(\sigma|_o))^{-} (\text{resume}(\sigma'|_a))^{-} (\text{comp}(\ell(b)\sigma', \bar{v}))) | \sigma'|_i = \sigma|_i \}
$$

(3)

Whether the callee thread really terminated cannot be checked here, as it would break the compositionality, and is therefore addressed in the composition of several objects to the semantics of a program run (see Eq. (12)).

Next, we turn our attention towards inter-object parallelism using message passing. Even though the communication is asynchronous in the sense that the invocation of a method is separated from the retrieval of its return parameters, the method invocation is modelled as an atomic bidirectional communication event. The caller provides the method name and the corresponding parameters and receives in turn the ID of the thread assigned to accomplish the work. 

$$
\mathcal{M}(\text{inv.(m)}(\bar{x}))(\sigma) = \begin{cases}
(\sigma_1, \theta) \\
\exists \text{ oid. } \bar{v}. \exists \text{ oid. oid = } \mathcal{E}(\text{oid}|_o)\sigma, \bar{v} = \mathcal{E}(\bar{x}\sigma), \\
\sigma_1 = (\sigma : l \rightarrow ((\mathcal{E}(\text{this}|_\sigma) \cdot \mathcal{E}(\text{me}|_\sigma), \text{oid, (m, i)))), \\
\theta = (\text{invoc}(\mathcal{E}(\text{l}|_\sigma), \bar{v})
\end{cases}
$$

By quantifying oid we ensure that the result of $\mathcal{E}(\text{oid}|_o)\sigma$ is not null if the set is non-empty. In the semantics we quantify existentially over the thread ID $i$, which completes the identity $\langle\text{oid, (m, i)}\rangle$ of the callee. During the composition in Eq. (12) we will require the thread ID to match the ID of the actual communication partner. We identify the pair of current object and thread as the caller, where both this and me are special variables where the former is a object attribute keeping the object ID and the latter is a thread-local variable storing its thread id. The pair of the identifier of the caller and the callee is assigned to the label. Because both identifiers consist of the object ID and thread id, they uniquely determine the communication partners. The representation of the communication event in the history is the invocation message containing the label and the values of the method parameters $\mathcal{E}(\bar{x}\sigma)$. The reply statement, which is the counterpart of the invocation, uses the same label to identify the corresponding completion message. 

$$
\mathcal{M}(\text{reply}(\bar{y}))(\sigma) = \{ (\sigma_1, \theta) | \exists \bar{v}. \exists \text{ lv. } \bar{v} = \mathcal{E}(\text{lv}|_\sigma)\sigma, \sigma_1 = (\sigma : \bar{y} \rightarrow \bar{v}), \theta = \langle\text{comp lv, } \bar{v}\rangle\}
$$

(4)

Besides the label, the completion message contains the values $\bar{v}$ of the return parameters of the method call, which are used to update the state as well. Similar to the previous definition we deal with the exception of the label being uninitialised by the existential quantifier $\exists \bar{v}$ It is worthwhile emphasizing that the history we are constructing here is purely thread local and will be composed to the semantics of the complete program hereafter. The last statement to cover is object creation. As reference to the object the pair $(C, i)$ composed of the class $C$ and an integer $i$ is written to the variable o. Uniqueness of $i$ is assured later in the parallel composition.

$$
\mathcal{M}(\text{o := new } C)(\sigma) = \{ (\sigma_1, \theta) | \exists \text{ i. } \sigma_1 = (\sigma : o \rightarrow (C, i)), \theta = \langle\text{new } \mathcal{E}(\text{this}|_\sigma) \cdot \mathcal{E}(\text{o}|_\sigma)\rangle\}
$$

(5)

In addition to the object ID of the current object, the ID of the newly created object is encoded in the new message.

---

3 In other versions of Creol, even the releasing is conditional.

---

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
Having dealt with single statements, we carry on with the semantics of a single thread as an instance of a method $m$. 

\[
\mathcal{M}(\text{op } m(\text{in } \bar{x}; \text{ out } \bar{y}) == \text{body})(\sigma) = \begin{cases} 
\{ \exists \bar{v}. \exists o_1. \exists o_2. \\
(\sigma_1, \theta, \theta') \in \mathcal{M}(\text{body})(o_1), \\
\theta = (\text{call}(\sigma) \circ \theta'), \theta_2 = (\text{end}(o_1, o_2), \bar{y})_\sigma \} 
\end{cases}
\]  

(6)

Some values $\bar{v}$ serve as the input parameters updating the initial state $\sigma$ which in turn acts as an input for the semantics of the method body. The resulting history $\theta$ includes all communication initiated by the method, but not its own invocation and completion message contained in $\theta_1$ and $\theta_2$. To model the asynchronism we write $\text{begin}$ instead of $\text{invoc}$ and $\text{end}$ in place of $\text{comp}$. When composing the histories of the caller and callee in Eq. (12) we will require that the $\text{invoc}$ and $\text{comp}$ messages of the caller frame the $\text{begin}$ and $\text{end}$ messages of the callee. In that way the point in time when a method is invoked is not necessarily the same point in time when the thread starts running. Additionally $\theta_1$ starts with a $\text{resume}$ as in general another thread might run in advance, and $\theta_2$ ends with $\text{yield}$ to allow other threads to work.

Now we are ready to compose the semantics of a number $i$ of threads of the same method. We use another semantic function $\mathcal{M} : \text{METHODS} \times \mathbb{N} \rightarrow \Sigma \rightarrow 2^{\Sigma}$ which gives for a method $m$ and a number of threads $i$ the function relating the initial state to a set of histories. The initial state $\sigma$ only consists of the value for $\text{this}$ as defined in Eq. (10).

\[
\mathcal{M}(m, i)(\sigma) = \begin{cases} 
\theta \downarrow \{(m, j) \mid j \in \{1, \ldots, i\}\} = \theta, \\
\forall j \in \{1, \ldots, i\}, \exists t. \exists \sigma_1. \exists \sigma_2. \exists \sigma''_j. \\
(m, j), \sigma_i = (\sigma : \text{me} \rightarrow t, \text{caller} \rightarrow o, \bar{a} \rightarrow \bar{v}), \\
(\theta_1, \sigma''_j) \in \mathcal{M}(\text{op } m(\text{in } \bar{x}; \text{ out } \bar{y}) == \text{body})(\sigma_i), \\
\theta \downarrow [t] = \theta_1, \text{cond}_m(\theta) 
\end{cases}
\]  

(7)

With “$\forall j \in \{1, \ldots, i\}, \exists t$.”, we create $i$ different thread IDs $t$ and corresponding initial states $\sigma_i$ for each of them. Except for the thread ID saved in $\text{me}$ and the reference to the caller, the states differ in the object attributes $\bar{a}$, as the threads start executing after some other thread yielded, leaving the object attributes with any values. The next line of the conditions restricts the histories $\theta_1$ and states $\sigma''_j$ as a possible result of executing the method $m$ on the state $\sigma_i$. The merging $\theta$ of the histories $\theta_i$ is described as the inverse of projection to $i$, as in [59]. The projection on sets of threads $T$ removes the messages not involving any thread in $T$.

\[
\theta \downarrow \{(m, j) \mid j \in \{1, \ldots, i\}\} = \theta, \\
\forall j \in \{1, \ldots, i\}, \exists t. \exists \sigma_1. \exists \sigma_2. \exists \sigma''_j. \\
(m, j), \sigma_i = (\sigma : \text{me} \rightarrow t, \text{caller} \rightarrow o, \bar{a} \rightarrow \bar{v}), \\
(\theta_1, \sigma''_j) \in \mathcal{M}(\text{op } m(\text{in } \bar{x}; \text{ out } \bar{y}) == \text{body})(\sigma_i), \\
\theta \downarrow [t] = \theta_1, \text{cond}_m(\theta) 
\]  

(8)

With “$\forall j \in \{1, \ldots, i\}, \exists t$.,” we require that all messages of $\theta_i$ are contained in $\theta$ in the correct order. To restrict $\theta$ to messages which are part of any history $\theta_i$, we write $\theta \downarrow \{(m, j) \mid j \in \{1, \ldots, i\}\} = \theta$. In particular, $\mathcal{M}(m, 0)(\sigma) = \{\}$. The notion of $\text{cond}_m(\theta)$ abbreviates the following condition $4$ making sure that a switch between two different histories $\theta_i$ is only possible at the release points of the methods.

\[
\forall m_1, m_2, t_1, t_2, \theta_1, \theta_2. (\{m_1\} \downarrow \{m_2\} \subseteq \theta, m_1 \in \theta_1, m_2 \in \theta_2, t_1 \neq t_2) \Rightarrow [m_1 = \text{yield}(\cdot), m_2 = \text{resume}(\cdot)] 
\]  

We marked the release points in the Eqs. (1)–(3) by the $\text{yield}$–$\text{resume}$ pair. Now, the histories of the threads can be split up in between those messages, and only there, as formally described by the above formula. The next step is to introduce the semantics of a method being the union over all possible numbers of threads.

\[
\mathcal{M}(m)(\sigma) = \bigcup_{i \in \mathbb{N}} \mathcal{M}(m, i)(\sigma) 
\]  

(9)

The step is necessary as the number of threads can depend on the parameters of the program. By considering all methods $m$ of the class $C$ instantiated by the object $o$ we obtain the semantics of an object $o$.

\[
\mathcal{M}(o) = \begin{cases} 
\exists \theta. \theta \downarrow \{m \mid m \text{ is method of } C\} = \theta, \\
\forall \text{methods } m \text{ of } C. \exists \theta_\phi. \exists \theta'. \exists \phi'. \\
\theta_m \in \mathcal{M}(m)(\text{this} \rightarrow o), \theta \downarrow \{m\} = \theta_m, \\
\theta' = (\text{yield}(\mathcal{M}(\text{const}(\phi'))))^{-\phi'}, \text{cond}_o(\theta'), \\
\theta'' = \text{created}(\theta', \phi')^{-\phi'} \downarrow \text{mp} 
\end{cases}
\]  

(10)

The composition essentially works the same way as for threads of the same method (Eq. (7)), explained by the fact that all threads are communicating using the same principle, namely shared memory. So we retrieve a possible history for each method where the state exclusively contains the object identifier $o$ accessible by $\text{this}$. The projections maintain the property that every message of all histories $\theta_m$ is included in $\theta$, but no more. The $\text{yield}$ message, preceding $\theta$, stores the result of the

---

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
constructor of $C$, executing only assignments to object attributes. $\text{cond}_a(\theta')$ furthermore restricts the object attributes at thread switches to match:

$$\text{cond}_a(\theta') \equiv \{ [8] \text{ and } \forall \sigma', \sigma''. \langle \text{yield}(\sigma') \rangle \rightarrow (\text{resume}(\sigma'')) \subseteq \theta' \Rightarrow \sigma' = \sigma'' \}$$

The resulting history $\theta''$ starts with the object creation message created which will be enforced in Eqs. (12), (13) to occur after the corresponding new message of the creator (see Eq. (5)). Finally, the shared memory concurrency is hidden by removing all related messages from $\theta'$ by the following projection (where \( \downarrow_{MP} \) reads as projection to method passing):

$$\theta \downarrow_{MP} = \begin{cases} \theta \downarrow_{MP} & \text{if } m = \text{yield}(\cdot) \text{ or } m = \text{resume}(\cdot) \\ (\theta \downarrow_{MP} \downarrow \{m\} \theta) & \text{otherwise} \end{cases}$$

Analogous to the semantics of methods (Eq. (7)) we define the semantics of a class $C$ with respect to a given number of objects $i$.

$$\mathcal{M}(C, i) = \left\{ \theta \Vert \begin{array}{l} \theta \downarrow \{(C, j) \mid j \in \{1, \ldots, i\} = \theta, \\
\forall j \in \{1, \ldots, i\}. \exists o. \exists \theta_o. o = (C, j), \theta_o \in \mathcal{M}(o), \theta \downarrow \{o\} = \theta_o \end{array} \right\}$$

This is a case of message passing parallelism, which we can describe by using the same projections as before, but this time without any requirement on release points as Eq. (8) for shared memory parallelism. (Note that \text{yield} and \text{resume} are not contained anymore in $\mathcal{M}(o)$.) Therefore, the only condition on $\theta$ is that it contains exactly all messages of the histories $\theta_0$ and the orderings of their messages are preserved, allowing all possible interleavings. We define the semantics of a class as the union over the number of instances.

$$\mathcal{M}(C) = \bigcup_{i \in \mathbb{N}} \mathcal{M}(C, i).$$

What is left to describe is the semantics of a complete program. This is more involved, as it is only here where we require the various communication histories to actually be consistent with each other, in order to be merged to a global one. As a tool we will reason about the set of messages of a certain type contained in a history, to be obtained by the following function.

$$\text{gettype}(\langle \cdot \rangle \theta) = \begin{cases} \text{gettype}(\theta) \cup \{m\} & \text{if } m \text{ has type } \text{type} \\ \text{gettype}(\theta) & \text{otherwise} \end{cases}$$

type will be a name of a message type, like \text{invoc} or \text{end}. Another prerequisite is a Boolean function determining the order of two given messages within the history.

$$m_1 \prec_o m_2 \equiv \exists \theta_1, \theta_2. \theta_1 \prec \theta_2 \Rightarrow \theta_1 \rightarrow \{m_1\} \rightarrow \theta_2 \rightarrow \{m_2\}$$

Finally, we turn to the actual semantics of a program $P$, given by a set of histories.

$$\mathcal{M}(P) = \left\{ \theta' \Vert \begin{array}{l} \exists o. \theta \downarrow \{(C, j) \mid j \in \text{class of } P \} = \theta, \\
\forall \text{ classes } C \text{ of } P. \exists \theta_C. \exists \theta. \theta_C \in \mathcal{M}(C), \theta \downarrow \{C\} = \theta_C, \\
o.\text{class} = \text{Main}, \theta' = \langle \text{new}(\cdot, o) \rangle \rightarrow \langle \text{invoc}(\cdot, (\text{main}, i), \bar{v}) \rangle \theta \rightarrow \theta \\
\forall \text{ type } \bar{v}. \forall \theta_1, \theta_2. \theta' = \theta_1 \Rightarrow \theta_2 \rightarrow \text{gettype}(\theta_1) \cap \text{gettype}(\theta_2) = \{\} \\
\exists \text{ function } f : \text{get}_\text{new}(\theta') \rightarrow \text{get}_\text{created}(\theta'), \text{cond}_f(f) \\
\exists \text{ function } g : \text{get}_\text{invoc}(\theta') \rightarrow \text{get}_\text{begin}(\theta'), \text{cond}_f(g) \\
\exists \text{ function } h : \text{get}_\text{comp}(\theta') \rightarrow \text{get}_\text{end}(\theta'), \text{cond}_f(h) \end{array} \right\}$$

The history $\theta'$ is composed of the histories $\theta_C$ given by all the classes $C$ of the program $P$. Following the same lines as the previous compositions, e.g. the one for message passing in Eq. (11), the merging is described as the inverse of projection. We assume that every program has a class called Main which contains a method called main. An object $o$ is created initially as denoted by the $\langle \text{new}(\cdot, o) \rangle$ message and its method main is invoked afterwards which is the meaning of the message $\langle \text{invoc}(\cdot, (\text{main}, i), \bar{v}) \rangle$. As we are existentially quantifying over the parameters of messages there could be identical messages in the history, which is prohibited by the third line. Now, a valid history $\theta'$ must ensure that every new object was created, which is achieved by the bijection $f$. In terms of messages it must hold that a message (created($o_1$, $o_2$)) of the history of the caller $o_1$ (see Eq. (10)) is always preceded by the creation message (new($o_1$, $o_2$)) of the callee $o_2$ (described in Eq. (5)). Formally, we express this fact, which we abbreviated as $\text{cond}_f(f)$, as:

$$\forall x. \forall y. f(x) = y \Rightarrow x.\text{caller} = y.\text{caller}, x.\text{callee} = y.\text{callee}, x \prec_o y$$

The next function $g$ does a similar job in relating a begin message to every invoc message. Analogous to Eq. (13) we ensure within $\text{cond}_f(g)$ that invoc appears in the history before its related begin message. The last function $h$ connects every comp message with an end message ordering the first before the latter by $\text{cond}_f(h)$. As it is not necessary to ask for the result of a method, $h$ need not be surjective. On the other hand the return values can be checked several times for a single method call, so in general the function $h$ is not injective. In total, for every method call the messages invoc and comp (if it exists) frame the pair of begin, end introduced by Eq. (6). Thus the callee starts executing after the message call and the caller receives its parameters after the callee has terminated, as one would expect. Using the functions $f$, $g$, and $h$ we induce a partial ordering on messages within histories representing asynchronous communication which makes this approach comparable to the classical work of [29].
4. The KeY approach: Logic, calculus, and system

4.1. Dynamic logic with explicit substitutions

KeY is a deductive verification system for functional correctness. Its core is a theorem prover for formulas in dynamic logic (DL) [33], which, like Hoare logic [34], is transparent with respect to the programs that are subject to verification. DL is a particular kind of modal logic. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of functions and predicates. The modalities are ‘indexed’ with pieces of program code, describing how to reach one world (state) from the other. DL extends typed first-order logic with two additional (mix-fix) operators: ⟨, ⟩. ⟨diamond⟩ and ⟨, ⟩. ⟨box⟩. In both cases, the first argument is a program (fragment), whereas the second argument is another DL formula. A formula (p)φ is true in a state s if there is one terminating run of p, started in s, which results in a state where φ is true. As for the other operator, a formula [p]φ is true in a state s if all terminating runs of p, started in s, result in a state where φ is true. Note that (p)φ implies termination of at least one run, whereas [p]φ is trivially true if there is no terminating run. (DL is not able to express termination of all possible runs.) For deterministic program (fragment)s p, the difference between (p)φ and [p]φ is only termination.

DL is closed under all logical connectives. For instance, in a DL formula ⟨p⟩φ, the postcondition φ may be any DL formula again, like in ⟨p⟩⟨q⟩ψ. Also, arbitrary connectives can enclose a box or diamond. For instance, the following formula states equivalence of p and q w.r.t. the “output”, the program variable x.

∀ v. (⟨p⟩x = v ↔ ⟨q⟩ x = v )

(14)

Our version of dynamic logic distinguishes strictly between logical variables and program variables. Logical variables (like v in (14)), can be quantified over. They cannot appear in programs, and their meaning does not vary among different states. Program variables (like x in (14)), on the other hand depend on the state, but cannot be quantified over. Expressions in the logic, outside the box or diamond modality, can contain both types of variables.

A frequent pattern of DL formulas is φ → ⟨p⟩ψ, stating that the program p, when started from a state satisfying φ, has a terminating run with ψ being true afterwards. The formula φ → [p]ψ, on the other hand, does not claim termination, and corresponds to the Hoare triple {φ} p {ψ}.

The main advantage of DL over Hoare logic is increased expressiveness: Pre- or postconditions can contain programs themselves, for instance to express that a linked structure is acyclic. Also, the relation of different programs to each other (like the correctness of transformations) can be expressed elegantly, see Eq. (14).

What all major program logics (Hoare logic, wp calculus,DL) have in common is that the resolution of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having explicit substitutions in the logic, called ‘updates’. This allows for accumulating and simplifying the effect of a program, in a forward style. Elementary updates have the form x := e, where x is a location (in the case of Creol, an attribute or local variable) and e is a (side-effect free) expression. Elementary updates are combined to simultaneous updates, like in x₁ := e₁ | x₂ := e₂, where e₁ and e₂ are evaluated in the same state. For instance, x := y | y := x stands for exchanging the values of x and y. Updates are brought into the logic via the update modality ⟨, ⟩., connecting arbitrary updates with arbitrary formulas, like in x < y → ⟨x := y | y := x⟩ y < x. A typical usage of updates during proving is in formulas of the form the update |ḥ| pφ, where ᦠ is an update, accumulating the effects of program execution up to a certain point, p is the remaining program yet to be executed, and φ a postcondition. A full account of KeY style DL is found in [14].

4.2. Sequent calculus

The heart of KeY, the prover, uses a sequent calculus for reducing proof obligations to axioms. A sequent is a pair of sets of formulas written as φ₁, . . . , φₘ ⊨ ψ₁, . . . , ψᵣ. A sequent is valid if the validity of all φ₁, . . . , φₘ implies validity of at least one of ψ₁, . . . , ψᵣ. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by Γ ⊨ φ → ψ, Δ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one, or more sequents as premises:

\[
\frac{Γ₁ ⊨ Δ₁ \ldots Γₙ ⊨ Δₙ} {Γ ⊨ Δ}
\]

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion (‘top-down’). Operationally, rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises. In Fig. 4 we present a selection of the rules dealing with propositional connectives and quantifiers (see [32] for the full set). φ⁺ denotes a formula resulting from replacing the variables x with expressions φ in φ.

When it comes to the rules dealing with programs, many of them are not sensitive to the side of the sequent and can even be applied to subformulas. For instance, (skip; ω)φ can be rewritten to ⟨ω⟩φ regardless of where it occurs. For that we introduce the following syntax

\[
φ' \vdash φ
\]
where $\phi$ and $\phi'$ are single formulas (i.e., there is no sequent arrow $\vdash$, neither $\Gamma$ or $\Delta$). This denotes a rule where the (only) premise sequent is constructed by replacing $\phi$ with $\phi'$ anywhere in the conclusion sequent $\phi$. In Fig. 5 we present some rules dealing with statements. (assign and if are simplified, see Section 6.1.) The schematic modality $\{\cdot\}$ can be instantiated with both $[\cdot]$ and $\langle \cdot \rangle$, though consistently within a single rule application. Total correctness formulas of the form $([\text{while} \ldots])\phi$ are proved by combining induction with unwind.

Because updates are essentially delayed substitutions, they are eventually resolved by application to the succeeding formula, e.g., $(u := e)(u > 0)$ leads to $e > 0$. Update application is only defined on formulas not starting with box or diamond. For formulas of the form $\{t\} \mid s \mid \phi$ or $\{t\} \mid s \mid \phi$, the calculus first applies rules matching the first statement in $s$. This leads to nested updates, which are in the next step merged into a single simultaneous update. Once the box or diamond modality is completely resolved, the entire update is applied to the postcondition.

5. Semantics of Creol dynamic logic

Syntactically, we arrive at Creol dynamic logic simply by having Creol statements within the modalities box and diamond. However, we need to significantly extend the meaning of formulas, to be able to compositionally verify programs, one method at a time. The various methods of one class rely on each other respecting the class invariant for the shared variable concurrency to function correctly. Also, caller objects rely on the callees’ interface invariant. Correctness of Creol code must therefore include that these invariants are respected. We formalise in the following what this means exactly.

In order to evaluate formulas, we need the following semantic artifacts: a state $\sigma$, i.e., an assignment of object attributes and local variables to values, a (semantic) history $\theta$, and an assignment of logical variables $\gamma$. Further, $\text{IlInv}$ and $\text{ClInv}$ map interfaces and classes to their invariant, respectively. Interface invariants do not ‘talk’ about attributes or local variables. Instead, they talk about the history, for which we use the reserved symbol $\mathcal{H}$, which is interpreted by the given semantic history (typically $\theta$). Moreover, interface invariants have the logical variables $\text{caller}$ and $\text{callee}$, typically as argument of the projection operator, see Section 6.2. Finally, in place of class invariants, the code of a method body is ever only correct relative to its class $C$.

For notational simplicity, we assume each method $m$ to appear in exactly one interface. (This can always be achieved by renaming or qualifying of methods, and if necessary, cloning of the method bodies.) The function $\text{inf}(m)$ returns the interface $m$ belongs to.

The following sets of histories are used in the definition of formula semantics.

\[
\text{commit}_{\text{Inf}, \gamma} = \begin{cases} \theta & \text{if } \theta = \theta_0^\gamma(\text{invoc}((\text{oid}(t), (\text{oid'}(m, i), \bar{v}))) \\
& \text{then } (\theta, \gamma) \models \text{IlInv}(m)_{\text{oid}, \text{oid'}}_{\text{caller}, \text{callee}} \end{cases}
\]

\[
\text{assume}_{\text{Inf}, \gamma} = \begin{cases} \theta & \text{if } \theta = \theta_0^\gamma(\text{comp}((\text{oid}(t), (\text{oid'}(m, i), \bar{v}))) \\
& \text{then } (\theta, \gamma) \models \text{IlInv}(m)_{\text{oid}, \text{oid'}}_{\text{caller}, \text{callee}} \end{cases}
\]

\[
\text{guarantee}_{\text{Inv}, C, \gamma} = \begin{cases} \theta & \text{if } \theta = \theta_0^\gamma(\text{yield}(\sigma)) \\
& \text{then } (\sigma, \theta, \gamma) \models \text{ClInv}(C) \end{cases}
\]

\[
\text{rely}_{\text{Inv}, C, \gamma} = \begin{cases} \theta & \text{if } \theta = \theta_0^\gamma(\text{resume}(\sigma)) \\
& \text{then } (\sigma, \theta, \gamma) \models \text{ClInv}(C) \end{cases}
\]

With the help of these sets, we can define the semantics of (the base case of) dynamic logic formulas, relative to an initial state $\sigma$, an initial history $\theta$, and an assignment $\gamma$ of logical variables to values:

$$\sigma, \theta, \gamma, \text{IlInv}, \text{ClInv}, C \models [S]\phi$$
iff
for all \((\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)\):
if
\[\{\theta' | \theta' \leq \theta \triangleright \theta_1\} \subseteq \text{assume}_{\text{Inv}, \gamma} \cap \text{rely}_{\text{Inv}, \gamma}\]
then
\[\{\theta'' | \theta'' \leq \theta_1\} \subseteq \text{commit}_{\text{Inv}, \gamma} \cap \text{guarantee}_{\text{Inv}, \gamma} \text{ and } (\sigma_1, \theta_2, \gamma, \text{Inv}, \text{Cl}, C) \models \varphi\]

Intuitively, this means that \(S\) (if executed in class \(C\)) has to commit to the invariants of interfaces it calls, and has to guarantee the class invariant of \(C\) at release points, and has to establish \(\varphi\) on termination. For that, \(S\) can assume the invariant of replying interfaces (both before and during \(S\)), and can rely on other threads of this object to establish the invariant of \(C\) when releasing control (both before and during \(S\)). The definition combines assumption–commitment style reasoning [48], adapted to asynchronous method calls, with rely-guarantee style reasoning [44], adapted to object local cooperative parallelism. See the discussion in Section 9. The reader may have noted the slight asymmetry in the above definition, between \(\theta_2 \leftarrow \theta_1\) (in assume \(\cap\) rely) and \(\theta_1\) only (in commit \(\cap\) guarantee). The reason is that, in a compositional setting, each unit \(S\) can assume (rely on) the contracts of other units in both its own trace \(\theta_1\) and its pre-history \(\theta\). But a unit \(S\) can only ever commit (guarantee) anything about its own trace \(\theta_1\), not its pre-history \(\theta\).

Note that \(\mathcal{S}\) does not claim termination of any run, as the set \(\mathcal{M}(S)(\sigma)\) of terminating runs is allowed to be empty. On the other hand, \((\sigma, \theta, \gamma, \text{Inv}, \text{Cl}, C) \models (S)\) is defined by replacing in the above definition “for all \((\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)\)” with “there exists \((\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)\)”. Thereby, we claim, among other things, the existence of a terminating run (but not termination of all runs, cf. Section 4.1).

The semantics of Boolean connectives and quantifiers is defined as in first-order logic. As for the update operator, its semantics is straightforward:
\[
(\sigma, \theta, \gamma, \text{Inv}, \text{Cl}, C) \models [x_1 := e_1 | \ldots | x_n := e_n] \varphi
\]
iff
\[
((\sigma : x_1 \rightarrow \mathcal{E}(e_1)\sigma : \ldots : x_n \rightarrow \mathcal{E}(e_n)\sigma), \theta, \gamma, \text{Inv}, \text{Cl}, C) \models \varphi
\]

Now, validity of formulas, in the context of invariants \(\text{Inv}, \text{Cl}\), and class \(C\), is defined by:
\[
(\text{Inv}, \text{Cl}, C) \models \varphi
\]
iff
for all \(\sigma, \theta, \gamma : (\sigma, \theta, \gamma, \text{Inv}, \text{Cl}, C) \models \varphi\)

6. A calculus for Creol dynamic logic

Building on the logic and the calculus presented in the previous sections, we proceed with the sequent rules handling Creol statements. For the full set of rules, see [27].

6.1. Sequential constructs

We start with assignments. As soon as the right side is simply a variable or literal (summarised as ‘terminal expression’, \(te\)) the assignment can be transformed to an update, such that the effect will eventually (not immediately) be applied to the postcondition. The same applies for implicit assignments in variable declarations. We give only the rule for integer variable declaration.

assign \[
\frac{\{x := te\} \{\omega\} \phi}{\{x := te; \omega\} \phi}
\]
intDecl \[
\frac{\{i := 0\} \{\omega\} \phi}{\{\text{var } i : \text{int}; \omega\} \phi}
\]
The same mechanism can be used for operator expressions, as long as all arguments are terminal and errors can be excluded. For instance, a division can be shifted to an update iff the divisor is not zero. Otherwise, execution blocks. This semantics is captured by the following rule.

\[
\text{DivTerminal} \quad \frac{\langle -te_2 \neq 0 \rightarrow \{x := te_1/te_2\} \{\omega\} \phi \rangle \land \langle te_2 \neq 0 \rightarrow \{\text{block}; \omega\} \phi \rangle}{\{x := te_1/te_2; \omega\} \phi}
\]

An error could occur arbitrary deep in an expression. Therefore, expressions are unfolded until they consist only of a top level operator applied to terminal expressions. This is exemplified by the following rules (\(x'\) and \(x''\) are new program variables).

\[
\frac{\{x' := e_1; x' := e_2; x := x' + x''; \omega\} \phi}{\{x := e_1 + e_2; \omega\} \phi}
\]
\[
\frac{\{x := te_1 + te_2; \omega\} \phi}{\{x := te_1 + te_2; \omega\} \phi}
\]
In the left rule $e_i$ are non-terminal expressions. As all expressions are unfolded every division will eventually be analysed by DivTerminal. Other statements using expressions, like if, are unfolded in the same way as a division by zero could be contained in the expression.

\[
\begin{align*}
\text{ifUnfold} & \quad \{ x := b; \text{if } x \text{ then } p \text{ else } q \text{ end}; \omega \} \phi \\
\text{ifUnfold} & \quad \{ \text{if } b \text{ then } p \text{ else } q \text{ end}; \omega \} \phi
\end{align*}
\]

In the above rule $x$ is a new program variable. Once the condition of the if statement has been analysed, the rule below checks whether both branches have to be symbolically executed depending on the truth value of the terminal Boolean expression $tb$.

\[
\text{if} \quad \begin{cases}
   (tb \equiv \text{true} \rightarrow \{ p; \omega \} \phi) \land (tb \equiv \text{false} \rightarrow \{ q; \omega \} \phi)
\end{cases}
\]

Note that application of this rule may lead to proof branching in subsequent steps. As for while, the unwind rule was presented in Section 4.2. An alternative rule using a loop invariant is discussed in Section 6.3. That rule, however, only covers the box operator. Finally, the rules for the block statement reflect the fact that a non-terminating program is always partially correct, but never totally correct:

\[
\begin{align*}
\text{blockBox} & \quad \text{true} \\
\text{blockDia} & \quad \text{false}
\end{align*}
\]

6.2. Interface and class invariants

The verification process of Creol programs is compositional. This means we verify only one method (of one class) at a time and do not consider any other code during this process. Instead, we take into account the other threads of the object by guaranteeing the class invariant at release points and relying on it again when execution proceeds. As for the behaviour of other objects, it is represented by using the invariants of their interfaces. An additional construct in the proof is the communication history, which both the specifications as well as the class invariants talk about. These concepts for reasoning about Creol were introduced in [24,25].

Every interface is specified by an interface invariant $\text{Inv}(I)(H)$. (Strictly, $\text{Inv}(I)$ is a formula, and ‘($H$)’ only indicates that $H$ appears in that formula.) For the reasoning to be compositional, it is required that the system wide history $H$ appears only projected to the logical variables caller and callee, like in $H/\{\text{caller, callee}\}$, with ‘/’ being the syntactical representation of the semantical projection ‘↓’. This specification is used during verification at method calls and replies.

Continuing the previous example of Fig. 2 the interface USR is equipped with the following invariant:

\[
H/\{\text{caller, callee}\} \rightarrow (\rightarrow \text{giveCode}(\cdot, \cdot \rightarrow \text{withdraw}(\cdot \rightarrow \text{dispense})) \rightarrow \text{returnCard})*
\]

where $/ \rightarrow$ projects on invocation messages, $\cdot$ is concatenation, $\rightarrow$ are invocation messages, $\cdot$ are completion messages and brackets are used for optional occurrence. The parameters and communication partners are omitted for brevity. The invariant expresses that the history of this interface is always a prefix of this regular expression, such that an interaction with the user always begins with requesting PIN code and ends with requesting removal of the card. The interface S is specified by:

\[
H/\{\text{caller, callee}\} \leq \left( \rightarrow \text{authorize}(cid, .) \cdot \left( \leftarrow \text{authorize}(false) \right) \cdot \left( \leftarrow \text{authorize}(true) \cdot \rightarrow \text{debit}(cid, .) \cdot \leftarrow \text{debit}(., .) \right) \right) \right)^*\)
\]

Communication partners are omitted. The dot ‘.’ is used as a wildcard for a parameter. Parameters (including the card ID cid) and communication partners are fixed during one iteration of the Kleene star. The meaning of the invariant is that only after authorisation can the debit procedure be attempted.

We turn to the class invariant $\text{Inv}(C)(H, \overline{W})$, which forms a contract between all threads of the object this. $\overline{W}$ is the vector of object attributes. Those might get overwritten by other threads during a suspension of a thread, but the invariant expresses properties of $\overline{W}$ every thread is supposed to respect. A class invariant consists of several parts:

\[
\text{Inv}(C)(H, \overline{W}) \triangleq F(H, \overline{W}) \wedge \bigwedge_{\text{implementMed}} \text{Inv}(I)(H)_{\text{class}} \wedge \bigwedge_{\text{invokeMed}} \text{Inv}(I)(H)_{\text{call}}
\]

$F(H, \overline{W})$ relates the state of the ordinary object attributes $\overline{W}$ with the history, reflecting the refinement of the fully abstract interface specification to the local state. Then, all invariants of all interfaces I invoked or implemented by the class C are put in a conjunction to ensure that all methods respect them. Now we can formulate the proof obligation for a method. The precondition is the class invariant, instantiated with a history ending on an invocation of the method. After executing the body, the invariant holds again for the history ending with the completion message of the method. For each method
**op m(in \(\overline{x}\), out \(\overline{y}\)) := body** of class \(C\) and interface \(I\), where \(\overline{W}\) are the attributes of class \(C\), we have the following **proof obligation**:

\[
(\text{In}(\overline{C}, \overline{W}) \vdash \text{body}(\overline{C}, \overline{W}) \triangleright \text{Out}(\overline{C}, \overline{W}) \wedge \text{Inv}(\overline{C}, \overline{W}, (\text{caller}, i), (\text{this}, \text{me}), \overline{x}) \wedge \text{Cl}(\overline{C})(\overline{C}, \overline{W}) \rightarrow [\text{body}, \text{return}(\overline{y})](\text{In}(\overline{C}, \overline{W}) \triangleright \text{Out}(\overline{C}, \overline{W}))
\]

\(\text{Wf}(\overline{C})\) holding for well-formed forms. A well-formed history starts with the creation message of this class, contains invocation messages for all completion messages, and does not include any object references being null.

Let us proceed with an example for a class invariant. For class \(\text{ATM}\) of **Fig. 3**, the formula \(F\) is:

\[
F_{\text{ATM}}(\overline{C}, \overline{W}) \equiv \text{server} \equiv \text{null} \land \forall \text{cid}. \text{sum}_{\text{dis}}(\text{cid}) = \text{sum}_{\text{deb}}(\text{cid})
\]

It states that the reference server is never null and the sum of all dispensed money for all cards \(\text{cid}\) equals the sum of the money debited. More detailed, \(\text{sum}_{\text{dis}}(h)\) calculates the sum of the money dispensed in the history \(h\). (In the equations, \(\text{msg}\) is used as the ‘otherwise case’.)

\[
\begin{align*}
\text{sum}_{\text{dis}}(\epsilon) & = 0 \\
\text{sum}_{\text{dis}}(h \cdot \text{dispense}(am)) & = \text{sum}_{\text{dis}}(h) + am \\
\text{sum}_{\text{dis}}(h \cdot \text{msg}) & = \text{sum}_{\text{dis}}(h)
\end{align*}
\]

\(\text{sum}_{\text{deb}}(h)\) is the sum of the money debited from the corresponding bank account. Only successful debit calls are counted.

\[
\begin{align*}
\text{sum}_{\text{deb}}(\epsilon) & = 0 \\
\text{sum}_{\text{deb}}(h \cdot \text{debit}(cid, \text{am}) \cdot \text{false}) & = \text{sum}_{\text{deb}}(h) + am \\
\text{sum}_{\text{deb}}(h \cdot \text{msg}) & = \text{sum}_{\text{deb}}(h)
\end{align*}
\]

In the system such equations are realised as taclets (see Section 7).

### 6.3. Concurrent constructs

There are two different levels of communication, namely inter-thread communication within one object via shared memory (the class attributes \(\overline{W}\)) and inter-object communication via method calls and replies. We start with the rules concerning the first and focus on the latter further below.

The simplest form of a release point is **release**. As mentioned before the class invariant forms a contract between all threads of an object. So the rule for **release** forces us to show that the class invariant is established in the current state (being the commitment of (15)), before releasing the processor. When this rule is used, it can rely on the invariant for the remaining code \(\omega\) to be executed.

\[
\begin{array}{c}
\text{release} \\
\frac{\Gamma \vdash \text{Cl}(\overline{C}, \overline{W}), \Delta}{\Gamma \vdash \text{release}; \omega \phi, \Delta}
\end{array}
\]

We omit the class as an argument of \(\text{Cl}\) for brevity as it can be obtained from this. Here, \(\text{U}_{\overline{C}, \overline{W}}\) is the update \(\overline{C}, \overline{W} := \text{some} \overline{H}, \overline{W}, (\text{Cl}(\overline{H}, \overline{W}) \wedge \overline{H} \leq \overline{H})\). This update represents an arbitrary but fixed system state, satisfying the class invariant (thus relying on it), in which execution continues. This is necessary because values of the object attributes could have been overwritten by other threads. By \(\overline{H} \leq \overline{H}\) we denote that the old history \(\overline{H}\) is a prefix of the new one \(\overline{H}\). Note that this rule, as well as all rules in this section, can also be applied when the modality is preceded by updates, which is the typical scenario. These updates are preserved in the instantiation of the premises (see [14]).

The **await** \(b\) statement is handled by a similar rule, with the additional assumption that the guard \(b\) holds when execution resumes. A minor complication is that we also must assume that the evaluation of \(b\) does not block due to an error. The two assumptions together are expressed via \((x := b)x \equiv \text{true}\).

\[
\begin{array}{c}
\text{awaitExp} \\
\frac{\Gamma \vdash \text{Cl}(\overline{C}, \overline{W}), \Delta}{\Gamma \vdash \text{awaitExp} \cdot \text{await}(b); \omega \phi, \Delta}
\end{array}
\]

By replacing \((x := b)x \equiv \text{true}\) with \(\text{Comp}(\overline{C}, l)\) in the above rule, we get a rule for **await** \(l\). The predicate \(\text{Comp}(\overline{C}, l)\) is valid if a completion message with the label \(l\) is contained in the history \(\overline{H}\). The handling of \(\text{Comp}(\overline{C}, l)\) in the proof is discussed further below.

Partial correctness of a loop can also be shown with the help of a loop invariant \(\text{inv}_{\text{loop}}(\overline{C}, \overline{mod})\), where \(\overline{mod}\) is the modifier set of the loop (all variables assigned in the loop). To be most general, all object attributes could be included in the modifier set. The history could be omitted as a parameter of the loop invariant if there are no method calls, method completions or object creations in the loop body.

\[
\begin{array}{c}
\text{loop} \\
\frac{\Gamma \vdash (x := b)x \equiv \text{true} \rightarrow \text{inv}_{\text{loop}}(\overline{C}, \overline{mod}), \Delta}{\Gamma \vdash \text{while do p end}; \omega \phi, \Delta}
\end{array}
\]

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
The update \( U_{\text{inv}}^\text{loop} \) is defined as:

\[
\mathcal{H}, \text{mod} := \text{some } H, \overline{m}.(\mathcal{H} \leq H \land \text{inv}_{\text{loop}}(H, \overline{m}))
\]

It creates a new history \( H \) and a new modifier set, such that the loop invariant holds. If the condition \( b \) throws an exception, the implication of all branches are true.

Analogous to \( \text{Comp}(\mathcal{H}, l, \overline{y}) \) there are predicates \( \text{Invoc}(\mathcal{H}, l, \overline{x}) \) and \( \text{New}(\mathcal{H}, o) \) which guarantee the existence of an invocation message with label \( l \), parameters \( \overline{x} \) and an object creation message with reference \( o \) in the history \( \mathcal{H} \), respectively.

To exemplify some properties of the predicates dealing with the history we give the following formula which is a tautology.

\[
\text{Comp}(\mathcal{H}_0, l, \overline{y}) \land \mathcal{H}_0 \leq \mathcal{H}_1 \rightarrow \text{Comp}(\mathcal{H}_1, l, \overline{y})
\]  

(18)

Moreover \( \text{Comp}, \text{New} \), as well as \( \text{Invoc} \) are monotonic w.r.t. \( \leq \). Additionally, the contraposition is used in our proof system.

We turn attention towards method invocation \( \text{ll0.mtd}(\overline{x}) \). Its execution assigns a unique reference to \( l \), and extends the history by the corresponding invocation message:

\[
\begin{align*}
\Gamma \vdash o \equiv \text{null} & \rightarrow \{ \text{block}; o \} \phi, \Delta \\
\text{invoc} \quad \Gamma \vdash \neg o \equiv \text{null} & \rightarrow \{ l := ((\text{this}, \overline{m}), (o, i)) \} \{ U_{\mathcal{H}}^\text{invoc} \} (\text{ll0}(l)(\mathcal{H})^\text{this,i}\}_\text{caller,callee} \land \{ o \} \phi, \Delta \\
\Gamma \vdash \{ \text{ll0.mtd}(\overline{x}); o \} \phi, \Delta
\end{align*}
\]

Here, \( i \) is a new constant symbol representing the thread ID. If \( o \) is null, execution blocks. In the first branch, the invariant of the remote interface \( l \) must be shown to fulfill the assumption (16) where \( l \) is the type of \( o \). The abbreviation \( U_{\mathcal{H}}^\text{invoc} \) for the update in its full form is:

\[
\mathcal{H} := \text{some } H.,(\mathcal{H} \leq H \land \text{Invoc}(H, l, \overline{x}))
\]

The new history contains the invocation message \( \text{invoc}(l, \overline{x}) \).

A completion statement \( \text{Comp}(\overline{y}) \) assigns the return parameters of the method call identified by the label \( l \) to \( \overline{y} \). If the label \( l \) is null, the execution blocks.

\[
\begin{align*}
\Gamma \vdash l \equiv \text{null} & \rightarrow \{ \text{block}; \overline{y} \} \phi, \Delta \\
\text{comp} \quad \Gamma \vdash \neg l \equiv \text{null} & \rightarrow \{ U_{\mathcal{H}, \overline{y}}^\text{comp} \} \{ \overline{y} \} \phi, \Delta \\
\Gamma \vdash \{ \text{Comp}(\overline{y}); \overline{y} \} \phi, \Delta
\end{align*}
\]

The update \( U_{\mathcal{H}, \overline{y}}^\text{comp} \) is analogous to \( U_{\mathcal{H}}^\text{invoc} \). It overwrites the return parameters \( \overline{y} \), uses \( \text{Comp} \) to denote that the completion messages occurs in the new history, and assumes the interface invariant.

\[
\mathcal{H}, \overline{y} := \text{some } H, \overline{p}.(\mathcal{H} \leq H \land \text{Invoc}(H, l, \overline{x}) \land \text{Comp}(H, l, \overline{p}))
\]

\( I \) is obtainable from the label \( l \) as it contains the method which was called.

We omit the rule for object creation, mentioning only that the new reference is constructed by the pair \((\text{this, } i)\), where \( i \) is an object local, successively incremented index. An alternative, fully abstract modeling of object creation in DL is investigated in [4] and can be adapted also here. Finally, we consider the return statement. It sends the completion message belonging to the method call of the verification process and the thread terminates afterwards. The class invariant is not explicitly mentioned in the following rule as it is contained in \( \phi \) (see previous section).

\[
\begin{align*}
\Gamma \vdash l \equiv \{ \text{return}(\overline{y}) \} \phi, \Delta \\
\Gamma \vdash \{ \text{return}(\overline{y}) \} \phi, \Delta
\end{align*}
\]

The update \( U_{\mathcal{H}}^\text{return} \) adds the completion message to the history which must not occur in the previous history.

\[
\mathcal{H} := \text{some } H.,(\mathcal{H} \leq H \land \text{Comp}(H, (\text{caller, this}), \overline{y})).
\]

7. A system for Creol verification

The verification system for Creol is based on KeY [12]. Written in Java and published under the GNU general public license, it is available from the projects website. 6 The current version is a prototype which provides the functionalities presented in this article. In the following paragraphs of this section we briefly describe selected aspects of the system, namely its graphical user interface, its architecture, the implementation of the calculus, and the proof strategy.

In the graphical user interface the proof tree and open proof goals are displayed. Other features are pretty-printing and syntax-highlighting of the subformula/subterm currently pointed at with the mouse pointer. This enables a context sensitive menu offering only the rules applicable to the highlighted subformula/subterm. Apart from the rule name, tooltips describe the effect of a rule. Besides interactive application of rules, automatic strategies can be configured. A more detailed description of the KeY interface is available in [3].

---

6 www.key-project.org.
We describe the architecture of the prototype by means of Fig. 6. The problem file contains Creol code and its specification. Together with the specification specific rules the problem files of the verified methods were each about 600 lines long. In a first step the file is handed to a parser which passes the code residing in the modalities to the Creol parser. Both parsers use the ANTLR [54] parser generator at which only the Creol parser was created from scratch taking about 3900 lines of code. The output of the parsers is an abstract syntax tree (AST) of logical formulae containing a program AST at each modality. For reading the rules of the calculus into memory the same parsers are used where the Creol specific rules are written in 1400 lines. The applicable rules are determined by a tree matching procedure providing the input for the strategy. Each rule is equipped with a heuristic tag which is used by the strategy together with information about the context of application (e.g. the term to be replaced resides on the right side of the sequence arrow) to rank the applicable rules. Finally the chosen rule is applied to the current sequence by transforming the AST as described by the rule which brings us back to the situation where applicable rules should be identified. Overall, the adaptions in the KeY-system took about 5000 lines.

Problem files, files containing logical rules, or axiomatizations of data types are written in the taclet language [56]. In Fig. 7 the rule impRight from Fig. 4 and the Eq. (18) with only one variable $Y$ are defined in the taclet language. A find describes the formula the rule is applicable to, replacewith specifies the replacement for the find formula, assumes characterises further assumptions not subject to replacements, and add causes its argument to be added. The arrow ==> indicates on which side of the sequent the formula is found, replaced, or added. Writing a semicolon between two occurrences of replacewith or add causes a branching. Taclets omitting the sequence arrow ==> are rewriting rules applicable in all contexts.

The theory explained in the previous section needed some small extensions to be run in the system. First, the some quantifier was not implemented, but is expressed by another formula. For example, the update formula like \( \{ H :\mathcal{H}(H_1 \land H_2 \leq H_1) \} \) \( \phi \) is rewritten to:

\[
\forall H_0. \quad (\mathcal{H} \models H_0 \rightarrow \forall H_1. \{ \mathcal{H} := H_1 \}((\mathcal{Wf}(H_1) \land H_0 \leq H_1) \rightarrow \phi))
\]

The old value of \( \mathcal{H} \) is saved in \( H_0 \), and the new variable \( H_1 \) is assigned to \( \mathcal{H} \). The implication assures that \( H_1 \) has the desired properties when evaluating \( \phi \).

Secondly, there are different prefix predicates \( \leq I \) where \( I \) is an interface. Thereby the interface invariant for \( I \) is monotonous on \( \leq I \) if \( I' \neq I \). The rules invoke, comp, and return use \( \leq I \) where \( I \) is the interface the message added by the rule corresponds to. Release points and the loop invariant use a prefix predicate \( \leq \text{inf} \) which is not monotonous for interface specifications.

To achieve the remarkable high degree of automation, a specialised proof heuristic for the Creol specific layout of the sequents was coined. Typically, the challenging parts of a correctness proof of a Creol method is to verify that a class invariant holds after the symbolic execution of the method, or that the interface invariant holds when asynchronous communication is performed. This instantiation of an invariant has to be related with the assumption of the invariant at the last release point. Therefore the backwards-monotonicity of the prefix predicate is applied to the invariant being the proof obligation until the last release point is reached. While symbolically executing Creol code, a great number of equalities are induced by the implementation of the non-deterministic updates (see Eq. (19)). To avoid the existence of formulae expressing properties of the same history but being stated by different variables, the equalities are applied eagerly. Not surprisingly, by defining a normal form for the terms and formulae expressing lists and prefix relationships among them, the ratio of automated steps was significantly enhanced. Up to the decision whether a loop is unrolled or a (given) invariant is applied, it should be possible in theory to design a fully-automated strategy for the verification of single methods. During the development of the proof heuristics it was highly beneficial to draw the proof tree and mark fully-atomised leaves such that in the next attempt another issue could be tackled. Further research in computer-aid proof tree visualisation following the lines of [19, 11] could greatly simplify this process.
class BufferImpl implements FifoBuffer
var cell:Any; var cnt:Int; var next:FifoBuffer;
begin with Any
  op put(in x:Any; out) ==
    if cnt=0 then cell:=x
      else if next=null then next:=new Buffer end;
    var i:Label[]; !next.put(x); !i()
    end;
    cnt:= cnt+1; return()
  op get(in; out x:Any) ==
    await (cnt>0);
    if cell=null then i:Label[Any]; !next.get(); !i(x)
      else x:=cell; cell:=null
    end;
    cnt:= cnt−1; return(x)
end

3. Verification examples

8.1. Unbounded buffer

We give an implementation for an unbounded first-in-first-out (FIFO) buffer. This example is adapted from [25]. The interface contains two methods put and get which can be used to put into and to obtain an element from the buffer.

interface FifoBuffer
begin with Any
  op put(in x:Any; out)
  op get(in; out x:Any)
end

The interface invariant expresses that the sequence of elements retrieved from the buffer are a prefix of the elements put into the buffer. This ensures the FIFO property. Additionally, all elements must not equal null. We define the interface invariant \( IInv(FifoBuffer)(H / \{ caller, callee \}) \) (we write \( H \) instead of \( H / \{ caller, callee \} \)) as:

\[
\text{out}(H, \text{callee}) \leq \text{in}(H, \text{callee}) \land \forall x. (x \in \text{in}(H, \text{callee}) \rightarrow \neg x = \text{null})
\]

where \( in, out \) are defined as:

\[
\begin{align*}
\text{in}(\epsilon, o) &= \epsilon \\
\text{in}(h \cdot o_2 \leftarrow o\text{.put}(x), o) &= \text{in}(h, o) \cdot x \\
\text{in}(h \cdot \text{msg}, o) &= \text{in}(h, o)
\end{align*}
\]

\[
\begin{align*}
\text{out}(\epsilon, o) &= \epsilon \\
\text{out}(h \cdot o_2 \leftarrow o\text{.get}(\cdot), x, o) &= \text{out}(h, o) \cdot x \\
\text{out}(h \cdot \text{msg}, o) &= \text{out}(h, o)
\end{align*}
\]

Note that we do not guarantee that a caller gets the same objects it has put into the buffer. Such a buffer can be used for fair work balancing where a request is put into the buffer and workers take them out again.

The implementation of the buffer, given in Fig. 8, uses a chain of objects where each of them can store one element. The attribute \( cell \) is null if the object does not store an element. In next the reference to the following chain of objects is stored. Requests are forwarded to it if the object cannot serve them alone. The variable \( cnt \) holds the number of elements stored in \( cell \) and all following objects. Calls of \( get \) on an empty buffer are suspended until there are elements in the buffer.

Let us proceed with the class invariant. The attribute \( cnt \) equals the number of elements in \( cell \) and all following buffer cells. The interface invariant of FifoBuffer has to hold for both the interface called and implemented by the class. Additionally, we state that the sequence of values put into the current cell equals the sequence of values obtained from the buffer with the \( cell \) and the content of the following buffer appended. (Again, we write \( H \) instead of \( H / \{ caller, callee \} \).

\[
\text{cell} \cdot \text{buf}(H, \text{this}, next) \models \text{cnt}
\land (\neg \text{next} \models \text{null} \rightarrow IInv(FifoBuffer)(H, next)) \land IInv(FifoBuffer)(H, \text{this})
\land \text{in}(H, \text{this}) \models \text{out}(H, \text{this}) \cdot \text{cell} \cdot \text{buf}(H, \text{this}, next)
\]

If \( cell \) is null it is omitted. The term \( \text{buf}(o_1, o_2, h) \) in the above formula reconstructs for an object \( o_1 \) and its next object \( o_2 \) from the history \( h \) the elements in \( cell \) and all following objects.

\[
\text{buf}(o_1, o_2, h) = \begin{cases} 
\epsilon & \text{if } h \models e \lor o_1 \models \text{null} \lor o_2 \models \text{null} \\
\text{buf}(o_1, o_2, h') \cdot x & \text{if } h \models h' \lor o_1 \models o_2, \text{put}(x) \\
\text{rest} (\text{buf}(o_1, o_2, h')) & \text{if } h \models h' \lor o_1 \models o_2, \text{get}(\cdot) \\
\text{buf}(o_1, o_2, h') & \text{otherwise } h \models h' \cdot \text{msg}
\end{cases}
\]
rest removes the first element of a sequence. The example with the given specifications was proved interactively by the system. The method put was verified in 2846 steps and 85 branches, whereas get needed 2614 steps and 66 branches. With respect to degree of automation these proofs were very promising, as the system achieved 99.1% of automated steps over the total number of steps for the proof of put and 98.4% when proving get. The proofs were performed by a system implementing an older version of the calculus described in the previous version of this paper [5]. Due to the fact that the calculus was simplified since then, the authors claim that the degree of automation can easily be transferred to the new setting.

8.2. Automated teller machine

The example of the automated teller machine distributed throughout the paper was successfully verified by usage of 45 branches and 7480 steps in total where 98.4% of them were automatic. As the implementation of the class makes heavy use of asynchronous method calls and (co)interfaces, it has been shown that our system can easily deal with them. The experiences with specifications in the form of regular expressions were promising. They are easy to write down and an automated strategy can deal with them as the number of successor states is usually limited which narrows the search space of the proof. A further step in generalising the system could be the introduction of a logical toolbox expressing sets, relations and other well-understood mathematical notions simplifying the process of specifying and verifying other case studies.

9. Discussion and conclusion

Creol’s notion of inter-object communication is inspired by notions from process algebras (CSP [35], CCS [46], \pi-calculus [47]), which however model synchronous communication mostly. Moreover, Creol differs from those in integrating the notion of processes in the object-oriented setting, using named objects and methods rather than named channels. This also introduces more structure to the message passing (calls, replies, caller references, co-interfaces). The message passing paradigm on the inter-object level is combined with the shared memory paradigm on the local inter-thread level.

An early approach to the verification of shared-variable concurrency is ‘interference freedom tests’ [53]. A corresponding method targeting synchronous message passing is ‘cooperation tests’ [6]. Both the above are based on proof outlines of the composed processes, and therefore non-compositional. The first compositional proof methods were proposed by Cliff Jones for shared-variable concurrency, called ‘rely-guarantee’ [44], and by Jay Misra and Mani Chandy for synchronous message passing, called ‘assumption-commitment’ [48]. In both cases, the “key to formulating compositional proof methods for concurrent processes is the realisation that one has to specify not only their initial-final state behaviour, but also their interaction at intermediate points.” [22]. Our definition of the validity of formulas \( S \varphi \), along with our calculus, combines assumption-commitment style reasoning, adapted to asynchronous method calls, with rely-guarantee style reasoning, adapted to object local cooperative parallelism. The notational style we used for semantic definitions is inspired by Hooman, de Roever et al. [36].

Extending on the above principles of compositional verification, object invariants are used as a combined assumption/commitment or rely/guarantee conditions, respectively, both in the sequential setting to achieve modularity [9,10], and in the concurrent setting [37]. Compared to the last mentioned works, Creol is more restrictive in that it forces shared memory to be entirely object internal. All knowledge of remote data is contained in fully abstract interface specifications talking about the communication history. Communication histories appeared originally both in the CSP as well as the object-oriented setting [18,35]. A sound and complete compositional proof system based on history invariants and history projections was presented by Job Zwiers [59]. For other usages of communication histories in specification and verification, see for instance [58,23].

KeY is among the state-of-the-art approaches to the verification of (at first) sequential object-oriented programs, together with systems like Boogie [8], ESC/Java2 [31], and Krakatoa [30]. In comparison to those, KeY is unique in that it does not merely generate verification conditions for an external off-the-shelf prover, but employs a calculus where symbolic execution of programs is interleaved with first-order theorem proving strategies. This goes together with the nature of first-order DL, which syntactically interleaves modalities and first-order operators. The cornerstone for KeY style symbolic execution, the updates, have similarities to generalised substitutions in formalisms such as the B method [2]. Updates are, however, tailored to symbolic execution rather than modeling (for instance, conflicts are resolved via last-win). The KeY tool uses these updates not only for verification, but also for test case generation with high code-based coverage [28] and for symbolic debugging. The role of updates is largely orthogonal to the target language, allowing us to fully reuse this machinery for Creol.

As for Creol’s thread concurrency model, this differs from many other languages in that it is cooperative, meaning the programmer actively releases control (conditionally). This simplifies reasoning considerably as compared to reasoning about preemptive concurrency, where atomicity has to be enforced by dedicated constructs. There is work on verifying a limited fragment of concurrent Java with KeY [13]. Here, the main idea is to prove the correctness of all permutations of schedulings at once. In [1], concurrent correctness of Java threads is addressed by combining sequential correctness with interference freedom tests and cooperation tests.

Related closely to our work is the extension of the Boogie methodology to concurrent programs [37], targeting concurrent Spec#. From the beginning, this work is deeply integrated into an elaborate formal development environment, with all the
features mentioned the first paragraph of this paper. The methodology requires users to annotate code with commands in between which an object is allowed to violate its invariant. This is combined with ownership of objects by threads. Just as in our system, invariants have to be established at specific points, and can be assumed at others. Also similar is the erasing of knowledge, there with the havoc statement, here with the \texttt{some} operator. Differences (apart from the asynchronous method calls) are the purely cooperative nature of our threads, and that our shared memory is object local, which makes ownership trivial. Connected to this is the inherently fully abstract specification of remote object interfaces, employing histories. The Boogie approach can simulate histories as well (see Fig. 1 in [37]), but it lies in the responsibility of the user whether or not the simulated history reflects the real one.

The system presented in this paper is still a prototype. It supports Creol dynamic logic, but the front-end for loading code and generating proof obligations is yet unfinished. This however will not be a real challenge, given the KeY infrastructure. The impressively high degree of automation shows great potential for further applications of the system, which could be eased even more by providing better support for history-based specifications, like a library of frequently used queries on histories, or the usage of specification patterns [17], extended and configurable proof support for history based reasoning, and improved presentation on the syntax level and in the user interface.

Among the next issues on the agenda of this line of research are soundness, completeness, and compositionality. We believe that the calculus presented in Section 6 is ‘nearly’ sound wrt. the semantics from Section 3. We write ‘nearly’ because one should never claim soundness of a calculus for concurrency without having proven it. W.-P. de Roever reported that “every alleged proof method for concurrency, which reached his desk and […] had not been proven sound, turned out to be unsound” [22]. Therefore, a few fixes in details of the calculus or/and the semantics are to be expected. A related item of future work is the investigation of the degree of completeness. Finally, an important step to be taken is the precise formulation of the property of parallel compositionality, together with a proof of it. Compositionality can be formulated as a ‘parallel composition rule’ on the level of the calculus, or alternatively as a meta-level property. The latter could be more appropriate in a setting where the parallel composition (of dynamically created threads in dynamically created objects) is not explicit as a language construct, but rather implicit in the semantics.

We consider Creol’s approach to modular object-oriented modeling as a good basis for scaling ‘sequential formal methods’ to the concurrent distributed setting, in particular when targeting functional correctness. The key is a very strong separation of concerns, which however naturally follows object-oriented principles. KeY has proved to be a good conceptual and technical basis for such an undertaking, which we argue can lead to an efficient and user-friendly environment for the verification of distributed object applications.

Acknowledgements

The authors would like to thank Richard Bubel, Jasmin Christian Blanchette, Frank de Boer, Einar Broch Johnsen, Olaf Owe, Martin Steffen, and Ilham Kurnia for fruitful discussions on the subject and feedback on earlier versions of the paper. We thank Richard Bubel moreover for his guidance concerning implementation issues. Finally, we thank the anonymous reviewers for many insightful comments which led to many improvements in the final version of the paper.

The work of first author has partially been supported by the EU-project FP7-ICT-2007-3 HATS: Highly Adaptable and Trustworthy Software using Formal Methods. The work of second author has partially been supported by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments, and by the EU COST action IC0701: Formal Verification of Object-Oriented Software.

Appendix A. Syntax

This part of the appendix contains the grammars describing the syntax of Creol code used throughout this article. In comparison to other publications about Creol, some features which we do not cover are excluded from the syntax definition.

We start with a grammar for interfaces and classes where the statements will be given by another grammar. Braces are used as part of the grammar and do not occur in programs. By \( \ast \) we denote the Kleene star, \( + \) stands for multiple occurrences but at least once and \( ? \) is used for optional occurrence.

\[
\begin{align*}
  f & ::= \{ c \mid i \}\ast & \text{file} \\
  c & ::= \text{class } C \begin{cases} \text{begin} & \{ \text{implements } \{ l \}\ast \} \\
  \{ \text{var } x : T \}\ast \ \\
  \{ \text{with } l \{ \text{op } mtd(\text{in } \{ d \}\ast ; \text{out } \{ d \}\ast ) \Rightarrow s \; \text{return}(e\ast)\}\ast \} \end{cases} & \text{class} \\
  \{ \text{object attributes} \} \ast & \text{methods} \\
  i & ::= \text{interface } I \begin{cases} \text{begin} & \{ \text{with } l \{ \text{op } mtd(\text{in } \{ d \}\ast ; \text{out } \{ d \}\ast ) \}\ast \} \ast \ \\
  \ast \} \ast & \text{methods} \\
  d & ::= x : T \mid x : T \ast & \text{parameters} \\
  T & ::= \text{Int } \mid \text{Bool } \mid I & \text{types}
\end{align*}
\]

A source code file contains a number of classes and interfaces. An interface is similar to a class but contains only method declarations. A class can contain object attributes indicated by variable declarations \texttt{var }x : T as in the previous grammar.

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
Those are followed by the definition of a cointerface $I$ indicated by \textbf{with} $I$. For a cointerface several methods can be defined by \textbf{op mtd}($in\ [d]^{\gamma}; out\ [d]^{\gamma} \Rightarrow s$ where \textbf{mtd} is the identifier, $d$ are optional lists of declarations of input and output parameters, and $s$ are statements of the next grammar. The next to last line of the grammar describes the syntax of a list of method parameter declarations with variable identifiers $x$ which are typed by \textbf{Int}, \textbf{Bool}, or an interface $I$. We continue with the grammar defining statements.

$$s ::= \textbf{skip} \mid s ; s \mid \textbf{var}\ x : D \mid x ::= e$$

$$e ::= \textbf{b} \mid \textbf{i} \mid e = e \mid \textbf{null} \mid \textbf{this} \mid \textbf{me} \mid \textbf{caller}$$

Besides the \textbf{skip} statement, a statement can be the sequential concatenation of two statements expressed by a semicolon. Variables are declared by use of the keyword \textbf{var} followed by the a variable identifier $x$ a colon and a type. The assignment uses := as the single equal sign is reserved for equality. A special assignment is the object creation where the expression $e$ of the usual assignment is replaced by \textbf{new} $C$ with $C$ being a class identifier. At the well-known if construct the \textbf{else} branch is optional. The \textbf{while} loop is conditioned by the Boolean expression $b$ and contains another statement $s$ in its body. A method invocation uses $l$ as a label and furthermore contains a variable $x$, a method $\textbf{mtd}$ with a possibly empty list of parameters. The method completion makes use of the label $l$ and has a possible empty list of variables as parameters. The statement \textbf{await} can be followed either by a Boolean expression $b$ or a label $l$ and a question mark. Expressions can either be an integer expression, a Boolean expression, equality of two expressions or the special keywords \textbf{null}, \textbf{this}, \textbf{me}, and \textbf{caller}. Boolean expressions $b$ and integer expressions $i$ follow the principles of most programming languages. Types are either \textbf{Bool}, \textbf{Int}, an interface $I$ or a \textbf{Label} which contains a list of types not containing \textbf{Label} in the brackets.

\section*{Appendix B. Semantics}

In this section we repeat all formulae of the semantics presented in Section 3 as they are distributed over the text. Two additional formulae omitted in the main part of the paper concerning variable declarations and the if-statement are given. The function defining the semantics of programs:

$$\mathcal{M} : \text{PROG} \rightarrow (\Sigma \rightarrow 2^{\Sigma \times \mathbb{N}})$$

The \textbf{block} statement:

$$\mathcal{M}(\textbf{block})(\sigma) = \emptyset$$

The \textbf{skip} statement:

$$\mathcal{M}(\textbf{skip})(\sigma) = \{(\sigma, \langle \rangle)\}$$

The declaration of a variable $i$ adds a new mapping to the partial function $\sigma$ representing the state:

$$\mathcal{M}(\textbf{var}\ i : \textbf{Int})(\sigma) = \{(\sigma', \langle \rangle) \mid \exists v . \sigma' = (\sigma : i \rightarrow v)\}$$

We note that $i$ is a local variable such that it holds $\sigma|_a = \sigma'|_a$. The assignment statement:

$$\mathcal{M}(x ::= e)(\sigma) = \{(\sigma', \langle \rangle) \mid \exists v . e(\sigma, \sigma' = (\sigma : x \rightarrow v)\}$$

Sequential composition of statements:

$$\mathcal{M}(\textbf{S}_1; \textbf{S}_2)(\sigma) = \{(\sigma_2, \theta_1, \theta_2) \mid \exists \theta_1 . (\sigma_1, \theta_1) \in \mathcal{M}(\textbf{S}_1)(\sigma), (\sigma_2, \theta_2) \in \mathcal{M}(\textbf{S}_2)(\sigma)\}$$

The branching \textbf{if}-statement not given in the main part of the paper is presented here:

$$\mathcal{M}(\textbf{if}\ b\ \textbf{then}\ \textbf{S}_1\ \textbf{else}\ \textbf{S}_2\ \textbf{end})(\sigma) = \{(\sigma_1, \theta) \mid \exists \theta. \sigma = \theta_1 \ldots \theta_k, \mathcal{M}(\textbf{S}_1)(\sigma) \cup \{(\sigma_1, \theta) \mid \mathcal{M}(\textbf{S}_2)(\sigma)\}$$

Depending on the evaluation of $b$ either the first or the second set of the union is selected which delivers the semantics of the corresponding branch. The semantics of a \textbf{while}-loop:

$$\mathcal{M}(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{end})(\sigma_0) = \{ \sigma, \theta \mid \exists k \in \mathbb{N}, \{\sigma, \theta_1, \ldots, \sigma_k, \theta_k\} \text{ such that }$$

$$\sigma = \sigma_k, \theta = \theta_1^\ldots \theta_k, \mathcal{M}(\textbf{S} \sigma_k), \text{ for } i = 0, \ldots, k - 1 : \mathcal{M}(\textbf{S} \sigma_i), (\sigma_{i+1}, \theta_{i+1}) \in \mathcal{M}(\textbf{S})(\sigma_i)\}$$

Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
Having dealt with all sequential statements we turn to concurrent statements communicating via shared memory. The first statement is the \texttt{release}-statement:

\[ \mathcal{M} \text{(release)}(\sigma) = \{ (\sigma' , \langle \text{yield}(\sigma[I]) \rangle \rightarrow \langle \text{resume}(\sigma[I]) \rangle) \mid \sigma' |_1 = \sigma |_1 \} \]

The conditional \texttt{await}-statement occurs in two different versions, where the first deals with a Boolean condition \( b \) whereas the second refers to a label \( l \):

\[ \mathcal{M} \text{(await} b \text{)}(\sigma) = \{ (\sigma', \langle \text{yield}(\sigma[I]) \rangle \rightarrow \langle \text{resume}(\sigma[I]) \rangle) \mid \sigma' |_1 = \sigma |_1, \mathcal{B}(b) \sigma' \} \]

\[ \mathcal{M} \text{(await} l \text{)}(\sigma) = \{ (\sigma', \langle \text{yield}(\sigma[I]) \rangle \rightarrow \langle \text{resume}(\sigma[I]) \rangle \rightarrow \langle \text{comp}(E(\ell)(\sigma'), \bar{v}) \rangle) \mid \sigma' |_1 = \sigma |_1 \} \]

We proceed with concurrency by means of message passing using the invocation-statement and the completion-statement:

\[ \mathcal{M} \text{(invoc} l \text{)(}\ell\text{)}(\sigma) = \{ (\sigma, \theta) \mid \exists! \bar{i} l. \ell(\bar{i}) \sigma, \theta = \langle \text{comp}(\text{lv}(\bar{i}), \bar{v}) \rangle \} \]

Now, the only statement missing is the object creation:

\[ \mathcal{M} \text{(new} C\text{)}(\sigma) = \{ (\sigma, \theta) \mid \exists! \bar{i} C. \theta = \langle \text{new}(E(\ell)(\sigma), E(\ell)(\sigma)) \rangle \} \]

Given the semantics of all statements the semantics of a method looks as follows:

\[ \mathcal{M} \text{(op} m\text{)(in} \bar{x}\text{; out} \bar{y}\text{)} = \text{body}(\sigma) = \{ (\sigma, \theta) \mid \exists! \bar{i} C. \theta = \langle \text{new}(E(\ell)(\sigma), E(\ell)(\sigma)) \rangle \} \]

For \( i \) threads of method \( m \) the semantics are:

\[ \mathcal{M}(m,i)(\sigma) = \{ \theta \mid \exists! \bar{i} C. \theta = \langle \text{body}(\sigma) \rangle \} \]

By the union over all numbers of threads we obtain the semantics of a method \( m \):

\[ \mathcal{M}(m)(\sigma) = \bigcup_{i \in \mathbb{N}} \mathcal{M}(m,i)(\sigma) \]

The semantics of an object combine the semantics of all methods of its class:

\[ \mathcal{M}(\theta) = \{ \theta' \mid \exists! \bar{i} C. \theta' = \langle \text{body}(\sigma) \rangle \} \]

As for methods we first give the semantics of a class having \( i \) instances (objects):

\[ \mathcal{M}(C,i) = \{ \theta \mid \exists! \bar{i} C. \theta = \langle \text{body}(\sigma) \rangle \} \]

Finally, the semantics of a program \( P \) are:

\[ \mathcal{M}(P) = \{ \theta' \mid \exists! \bar{i} C. \theta' = \langle \text{body}(\sigma) \rangle \} \]
Appendix C. Calculus

Most of the rules in this section were explained in the main sections of this article and therefore we do not give further details on them here.

$$\begin{align*}
\text{skip} & \quad \frac{}{\{\omega \}}
\text{assign} & \quad \frac{x := te}{\{\omega \}}
\end{align*}$$

There are four different variable declaration and each of them comes with its own implicit initialisation which is 0 for integers, false for Booleans, and null for object references and labels.

$$\begin{align*}
\text{intDecl} & \quad \frac{i := 0}{\{\omega \}}
\text{boolDecl} & \quad \frac{b := \text{false}}{\{\omega \}}
\text{objDecl} & \quad \frac{0 := \text{null}}{\{\omega \}}
\text{labelDecl} & \quad \frac{l := \text{null}}{\{\omega \}}
\end{align*}$$

A division by zero could occur arbitrarily deep in an expression. Thus those are disassembled leading to four rules per Boolean and integer operator which follow the scheme below. They introduce new variables $x$ and $x''$ which save the arguments of the expressions. By $te$ a terminal expression is denoted meaning that $te$ cannot be further taken apart. Instances of terminal expressions are variables and constants. In contrast $e$ stands for expressions which contain an operator at their top level.

$$\begin{align*}
\text{DivTerminal} & \quad \frac{\{xe_2 = 0 \rightarrow \{xe_1/te_2\} \{\omega\}\} \land (te_2 \neq 0 \rightarrow \{\text{block}; \omega\})}{\{xe_1 + te_2; \omega\}}
\end{align*}$$

Only one rule for the division differs which checks for divisions by zero.

$$\begin{align*}
\text{ifUnfold} & \quad \frac{x := b; \text{if } x \text{ then } p \text{ else } q \text{ end}; \omega\}}{\{if \ b \ then \ p \ else \ q \ end; \omega\}}
\end{align*}$$

A similar rule exists for the while loop, but it is omitted here. Once, the condition of the if statement has been analysed the rule below checks whether both branches have to be symbolically executed depending on the truth value of the terminal Boolean expression $tb$.

$$\begin{align*}
\text{blockBox} & \quad \frac{\text{true}}{\{\text{block}; \omega\}}
\text{blockDia} & \quad \frac{\text{false}}{\{\text{block}; \omega\}}
\end{align*}$$

The rule for object creation involves the history, but does not involve any invariants.

$$\begin{align*}
\text{new} & \quad \frac{\Gamma \vdash o := (\text{this}, (c, l)) \{U^\text{new}_H\} \{\omega\}; \Delta}{\Gamma \vdash \{\text{new } c; \omega\}; \Delta}
\end{align*}$$

In the upper rule $i$ is a new constant symbol with respect to the history. To the variable $o$ the reference being a pair of caller (the object subject to verification) and new object ID is assigned. The update reads as follows $U^\text{new}_H = H := \text{some } H.(H \subseteq H \land \text{New}(H, o))$.

References


Please cite this article in press as: W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.08.003
B.3  Abstract Object Creation in Dynamic Logic
Abstract Object Creation in Dynamic Logic *

To Be or Not To Be Created

Wolfgang Ahrendt¹, Frank S. de Boer², and Immo Grabe³,²

¹ Chalmers University, Göteborg, Sweden
² CWI, Amsterdam, The Netherlands
³ Christian-Albrechts-University Kiel, Germany

Abstract. In this paper we present the integration of a weakest precondition calculus for abstract object creation in dynamic logic and the underlying theorem prover of the KeY tool. This integration allows to both specify and verify properties of objects at the abstraction level of the (object-oriented) programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties. Further, we show how to symbolically execute abstract object creation.

1 Introduction

In object-oriented programming languages like Java objects can be dynamically created by the constructor methods provided by their class. This high-level way of object creation abstracts from the underlying representation of objects and the implementation of object creation. At the abstraction level of the programming language objects are described as instances of their classes, i.e., the classes provide the only operations which can be performed on objects. Moreover, these operations can only be performed on the created objects, the objects not (yet) created do not exist and therefore can also not be referred to by any programming construct.

For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. Specification languages like the Java Modeling Language (JML) [⁸] and the Object Constraint Language (OCL) [¹⁰] abstract from the underlying representation of objects. In [⁴] a Hoare logic is presented for verifying properties of an object-oriented programming language at the abstraction level of the programming language itself. This Hoare logic is based on a weakest precondition calculus for object creation which abstracts from the implementation of object creation.

In this paper we investigate the integration of a weakest precondition calculus for abstract object creation in dynamic logic and the underlying theorem prover of the KeY tool [¹]. This integration allows to both specify and verify properties

* This work has been supported by the EU-project IST-33826 Credo: Modelling and analysis of evolutionary structures for distributed services. For more information, see http://credo.cwi.nl.
of objects at the abstraction level of the programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties.

The generalization of Hoare logic to dynamic logic is of particular interest because it allows for the specification of properties of dynamic object structures which cannot be expressed in first-order logic, like reachability. In Hoare logic such properties require quantification over (finite) sequences or recursively defined predicates in the specification language which seriously complicates both the weakest precondition calculus and the underlying logic. In dynamic logic we can restrict to first-order quantification and use the modalities to express for example reachability properties.

An interesting consequence of the abstraction level of the specification language studied in this paper is the *dynamic scope* of the quantification over objects because it is restricted to the created objects and as such is also affected by object creation. However, we show that the standard logic of first-order quantification also applies in the presence of (object) quantifiers with a dynamic scope.

Further, we show how to symbolically execute abstract object creation in KeY. In general, symbolic execution in KeY accumulates in a simultaneous substitution the assignments generated by a computation. This accumulation involves a pre-processing of the substitution which in general simplifies its actual application. However, we can not simply accumulate abstract object creation because its side-effects can only be processed by the actual application of the corresponding substitution. We show how to solve this problem by the introduction of fresh logical variables which are used as temporary place holders for the newly created objects. The use of these place holders together with the fact that we can always anticipate object creation allows to symbolically execute abstract object creation.

**Related work**

To reason about object-oriented programs most of the existing theorem provers like PVS [12], Isabelle [7] and the KeY tool, use an explicit representation of objects. Object creation is then formalized in terms of the information about which objects are in fact created. Such an explicit representation of objects additionally requires an axiomatization of certain consistency requirements, e.g., the global invariant that the values of the fields of created objects only refer to created objects. These requirements pervade the correctness proofs with the basic case distinction between "to be or not to be created" and adds considerably to the length of the proofs, as we will illustrate in this paper.

The contribution of this paper is the formalization of object creation in dynamic logic which abstracts from an explicit representation of objects and the corresponding implementation of object creation. Proofs in this formalization only refer to created objects and as such are not pervaded by irrelevant implementation details.
Outline

In section 2 we introduce a dynamic logic for a simple WHILE-language with object creation. This language allow us to focus on object creation. We present the axiomatization of the language in terms of the sequent calculus given in section 3. It should be observed that this calculus can be extended to other programming constructs of existing object-oriented languages like Java as described in [3]. With the calculus at hand symbolic execution of programs is described in section 4. After a discussion of the state of the art in symbolic execution with respect to object creation and a look into the expressiveness of our approach in section 5 we conclude with section 6.

2 Dynamic Logic

We introduce a very simple WHILE-language as our object-oriented programming language in order to focus on object creation. In [3] Becker and Platzer present a similar dynamic logic for Java Card called ODL. ODL covers the type system of Java. Besides the type system, dynamic dispatch, side-effects of expressions, and exception handling are presented in terms of program transformations. However ODL models object creation in terms of an explicit representation of objects. To obtain a logic covering Java Card that follows our theory of abstract object creation this representation can be replaced by our theory or our theory can be extended analogous to [3]. This justifies our focus on object creation.

2.1 Syntax

We assume the set \( F \) of fields and \( GVar \) of global variables to be given. Fields are the instance variables of objects. We assume a partitioning of \( GVar \) into a set \( PVar \) of program variables and a set \( LVar \) of logical variables. Logical variables do not change during program execution, i.e. there are no assignments to logical variables. They are used to express invariant properties and for (first-order) quantification. We restrict to the types Object, Integer and Boolean. All fields and variables are typed. We have the following grammar for statements and expressions.

\[
\begin{align*}
  s &::= \text{while } e \text{ do } s \text{ od } | \text{if } e_1 \text{ then } s_2 \text{ else } s_3 \text{ fi } | s_1 ; s_2 | \text{skip} \\
  u &::= \text{new } | e_1 . x := e_2 | u := e \\
  e &::= u | e . x | \text{null } | e_1 = e_2 | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 | f(e_1, \ldots, e_n)
\end{align*}
\]

The statement \( \text{while } e \text{ do } s \text{ od } \) denotes the usual looping. Conditional branching is denoted by \( \text{if } \text{-then-else} \). The condition for both looping and branching is given by a Boolean expression. A semicolon denotes sequential composition. By \( \text{skip} \) we denote the empty statement. Object creation is denoted by \( u := \text{new} \), where \( u \) is a program variable. An assignment to a program variable is denoted by \( u := e \). A dot denotes dereferencing, i.e., \( e_1 . x := e_2 \) denotes an assignment to the field \( x \) of the object referenced by \( e_1 \). For technical convenience only we do
not have assignments $e.x \leftarrow \text{new}$. In order to separate object creation from the aliasing problem we reason about such assignments in terms of the statement $u \leftarrow \text{new}; e.x \leftarrow u$, where $u$ is a fresh program variable. Note that program variables do not have aliases. Possible aliases are expressions $e_1.x$ and $e_2.x$ in case $e_1$ and $e_2$ refer to the same object.

The expression $\text{null}$ of type Object denotes the undefined reference. The Boolean expression $e_1 = e_2$ denotes the test for equality between the values of the expressions $e_1$ and $e_2$. A conditional expression is denoted by $\text{if } \ldots \text{then } \ldots \text{else }$. The function $f(e_1,\ldots,e_n)$ denotes an arithmetic or Boolean operation of arity $n$. We assume every statement and expression to be well-typed. It is important to note that object expressions, i.e., expressions of type Object, can only be compared for equality, dereferenced or appear as argument of a conditional expression.

Formulas. The formulas of first-order dynamic logic are built from Boolean expressions using the connectives $\wedge, \lor, \rightarrow, \neg$ and the (first-order) quantifiers $\forall, \exists$. In addition, if $\phi$ is a formula and $s$ a program, then $(s)\phi$, $[s]\phi$ and $\{U\}\phi$ are formulas. An update $U$ is denoted by $\{U\}$. An update is of the form $u := \text{new}$, $e_1.x := e_2$, or $u := e$. In section 4 multiple updates are merged to simultaneous updates. For the symbolic execution we use updates to delay substitution. Merging the updates gets more complex in the presence of object creation (see section 4). As an example the following formula $\forall l.(u := \text{new})(u = l)$ states that every new object indeed is new because the logical variable $l$ ranges over all the objects that exist before the object creation $u := \text{new}$. Consequently, after the execution of $u := \text{new}$ we have that the new object is not equal to any object that already existed before, i.e., $\neg(u = l)$, when $l$ refers to an "old" object. Note that the formula $(u := \text{new})\forall l.(u = l)$ is clearly false and thus has a completely different meaning. This is explained in more detail below.

2.2 Semantics

We assume given an arbitrary (infinite) set $O$ of object identities, with typical element $o$. We define null itself to be an element of $O$, i.e., the value of the expression null is null itself. The domain of values of type $T$ we denote by $\text{dom}(T)$, e.g., $\text{dom}(\text{Object}) = O$.

States. A state $\Sigma = (\sigma, \tau)$ is a pair consisting of a heap $\sigma$ and an environment $\tau$. The heap $\sigma$ is a partial function such that $\sigma(o)$ for every $o \in O$, if defined, denotes the internal state of object $o$. That is, the value of a field $x$ of an object $o$, for which $\sigma(o)$ is defined, is given by $\sigma(o)(x) \in \text{dom}(T)$. The domain $\text{dom}(\sigma)$ of objects that exist in a heap $\sigma$ is given by the set of objects $o$ for which $\sigma(o)$ is defined. In order to describe unbounded object creation we require the domain of a heap to be finite.

The environment $\tau$ assigns values to the global variables. The value of a variable $v$ is given by $\tau(v)$.

We require every state $\Sigma = (\sigma, \tau)$ to be consistent, i.e.,
null ∈ dom(σ),
– σ(o)(x) ∈ dom(σ) for every o ∈ dom(σ) and field x of type Object,
– τ(v) ∈ dom(σ) for every global variable v of type Object.

In words, null is an existing object, the fields of type Object of existing objects refer to existing objects and all global variables of type Object refer to existing objects.

Semantics of Expressions and Statements. The semantics of an expression e of type T is a partial function $\llbracket e \rrbracket : \Sigma \mapsto \text{dom}(T)$. As an example, if $\llbracket e \rrbracket$ is defined and does not evaluate to null then

$$\llbracket e.x \rrbracket(\sigma, \tau)(x) = \sigma(\llbracket e \rrbracket(\sigma, \tau))(x),$$

otherwise $\llbracket e.x \rrbracket$ is undefined. For a general treatment of failures we assume given a predicate $\text{def}(e)$ which defines the conditions under which the expression e is defined. For example, we have that $\text{def}(u.x) \equiv \neg(u = \text{null}).$

The semantics of a statement s is a partial function $\llbracket s \rrbracket : \Sigma \mapsto \Sigma$. We focus on the semantics of object creation. In order to formally describe the initialization of newly created objects, we first introduce for each type T an initial value of type T, i.e., $\text{init}_{\text{Object}} = \text{null}$, $\text{init}_{\text{Integer}} = 0$ and $\text{init}_{\text{Boolean}} = \text{true}$. We define init to be the initial state, i.e., the state that assigns to each field x of type T its initial value $\text{init}_T$. For the selection of a new object we use a choice function $\nu$ on heaps to get a fresh object, i.e., $\nu(\sigma) \not\in \text{dom}(\sigma)$.

We now define

$$\llbracket u := \text{new} \rrbracket(\sigma, \tau) = (\sigma[o := \text{init}], \tau[u := o]),$$

where $o = \nu(\sigma)$. The update $\sigma[o := \text{init}]$ assigns the local state init to the new object o and the update $\tau[u := o]$ assigns this object to the program variable u.

Semantics of Formulas. The definition of the semantics $\sigma, \tau \models \phi$ of a formula $\phi$ is restricted to consistent states. A formula in dynamic logic is valid if $\sigma, \tau \models \phi$ holds for every consistent state. For a logical variable l, we have the following standard semantics of universal quantification

$$(\sigma, \tau) \models \forall l. p \iff \text{for all } o \in O : (\sigma, \tau[l := o]) \models p,$$

where the consistency of $(\sigma, \tau[l := o])$ implies that the object o exists in $\sigma$. Consequently, quantification is restricted to the existing objects. Note that null is always included in the scope of the quantification (in other words, the scope of the quantification is non-empty).

Returning to the above example, we have

$$(\sigma, \tau) \models \forall l. (u := \text{new}) \neg(u = l)$$

iff

$$(\sigma, \tau[l := o]) \models (u := \text{new}) \neg(u = l)$$
for every $o \in \text{dom}(\sigma)$. Let $o' = \nu(\sigma)$. By the semantics of the diamond modality of dynamic logic and the above semantics of object creation we conclude that

\[
(\sigma, \tau[l := o]) \models (u := \text{new}) \neg(u = l)
\]

iff

\[
(\sigma[o' := \text{init}], \tau[l := o]) \models \neg(u = l)
\]

iff

\[ o \neq o' \]

### 3 Axiomatization

In this section, we introduce a proof system for dynamic logic with abstract object creation. It is characteristic for dynamic logic, in contrast to Hoare logic or weakest precondition calculus, that program reasoning is fully interleaved with first-order logic reasoning, because diamond, box or update modalities can appear both above and below the logical connectives and quantifiers. It is therefore important to realise that in the following proof rules, $\phi$, $\psi$ and alike, match any formula of our logic, possibly containing programs or updates.

#### 3.1 Sequent Calculus

We follow [3, 1] in presenting the proof system for dynamic logic as a sequent calculus. A sequent is a pair of sets of formulas (each formula closed for logical variables) written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. The intuitive meaning is that, given all of $\phi_1, ..., \phi_m$ hold, at least one of $\psi_1, ..., \psi_n$ must hold. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \rightarrow \psi, \Delta$ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one or many sequents as premises:

\[
\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}
\]

Semantically, a rule states that the validity of all $n$ premises implies the validity of the conclusion (‘top-down’). Operationally, rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises, starting from the initial sequent to be proved. Rules with no premise close the current proof branch. In fig. 1 we present some of the rules dealing with propositional connectives and quantifiers (see [6] for the full set). We omit the rules for the left-hand side, the rules to deal with negation and the rule to cover conditional expressions. $\phi[l/e]$ denotes standard substitution of $l$ with $e$ in $\phi$.

When it comes to the rules dealing with programs, most of them are not sensitive to the side of the sequent and can moreover be applied to subformulas even. For instance, $\langle s_1; s_2 \rangle \phi$ can be split up into $\langle s_1 \rangle \langle s_2 \rangle \phi$ regardless of where it occurs. For that we introduce the following syntax

\[
\frac{\{ \phi' \}}{\{ \phi \}}
\]
for a schema rule which states that any sequent $\lfloor \phi \rfloor$ with occurrences of the formula $\phi$ can be derived from the sequent $\lfloor \phi' \rfloor$ obtained by replacing all occurrences of $\phi'$ in the sequent $\lfloor \phi' \rfloor$ by $\phi$.

In fig. 2 we present the rules dealing with statements. The schematic modality $\langle \cdot \rangle$ can be instantiated with both $[\cdot]$ and $\langle \cdot \rangle$, though consistently within a single rule application. The extension of these rules with the predicate $\text{def}(e)$ for reasoning about failures is standard and therefore omitted.
3.2 Application of General Updates

Updates are essentially delayed substitutions. Consequently, they are finally resolved by ‘applying’ them to the formula they are preceding, such that, for instance \( \{ u := e \} \{ u > 0 \} \) leads to \( e > 0 \). Update application is only allowed on formulas not starting with either of the diamond, box or update modalities. (The last restriction will be dropped for symbolic execution, see section 4.)

We now define update application on formulas in terms of a rewrite relation \( \{ U \} \phi \leadsto \phi' \) on formulas. As a technical vehicle, we extend the update operator to expressions, such that \( \{ U \} e \) is an expression, for all updates \( U \) and expressions \( e \).

Accordingly, the rewrite relation \( \leadsto \) carries over to such expressions: \( \{ U \} e \leadsto e' \).

\[
\begin{align*}
\{ U \} \phi_1 \ast \{ U \} \phi_2 & \leadsto \phi' \\
\{ U \} (\neg \phi) & \leadsto \phi' \\
\{ U_{nc} \} (Q_l. \phi) & \leadsto \phi' \\
\{ U \} \alpha & \leadsto \alpha \\
\{ U_{nc} \} e_1 = \{ U_{nc} \} e_2 & \leadsto e' \\
\{ U \} f(e_1, \ldots, e_n) & \leadsto e' \\
\{ u := e_1 \} e_2.x & \leadsto e' \\
\{ e.x := e_1 \} e_2.y & \leadsto e' \\
\{ e.x := e_1 \} \{ e2.y \} & \leadsto e' \\
\{ u := e \} u_2 & \leadsto u_2 \\
\{ u := e \} u & \leadsto e \\
\{ e := e_1 \} e_2.x & \leadsto e' \\
\{ e.x := e_1 \} (e_2.x) & \leadsto e' \\
\{ e.x := e_1 \} \{ e2.x \} & \leadsto e'
\end{align*}
\]

Fig. 3. Update Application, with \( U_{nc} \) matching non-creating updates only

Fig. 3 defines \( \leadsto \) for all standard cases (see also [11, 1]). The aliasing analysis performed by the last rule is the very motivation for having conditional expressions in our language in the first place. Object creating updates of the form \( \{ u := \text{new} \} \) are only covered as far as they behave like standard updates. The cases where object creation makes a difference are excluded (note that \( U_{nc} \) only matches non-creating updates), to be discussed separately in section 3.3. The relation \( \leadsto \) is defined in a big-step manner, such that updates are resolved completely in a single \( \leadsto \) step.

Note that \( \leadsto \) is not defined for formulas of the form \( \{ U \} \langle s \rangle \phi \), \( \{ U \} [ s ] \phi \) or \( \{ U \} \{ U' \} \phi \), meaning they are not subject to update application. We will however return to \( \{ U \} \{ U' \} \phi \) in section 4.

\textsuperscript{4} The real benefit of the delay will only get apparent in the context of symbolic execution, see Sect. 4.
The following rule links the rewrite relation \( \rightsquigarrow \) with the sequent calculus:

\[
\text{applyUpd} \quad \frac{[\phi']}{[\{U\} \phi]}\quad \text{with } [U] \phi \rightsquigarrow \phi'
\]

3.3 Contextual Application of Object Creating Updates

We want to define the update \( \{u := \text{new}\} e \) but we cannot do so by simple substitution, i.e., replacing \( u \) in \( e \) by some expression, because we cannot refer to the newly created object in the state prior to its creation. However, since object expressions can only be compared for equality or dereferenced and do not appear as an argument of any other function, we can define the update \( \{u := \text{new}\} e \) by a contextual analysis of the occurrences of \( u \) in \( e \). We formalize this analysis in terms of a rewrite relation

\[
\{u := \text{new}\} e \rightsquigarrow e'
\]

which is defined inductively by the following rules.

If \( e \) equals null or any global variable different from \( u \) then

\[
\{u := \text{new}\} e \rightsquigarrow e
\]

Since the fields of a newly created object are initialized we have

\[
\{u := \text{new}\} u.x \rightsquigarrow \text{init}_T
\]

where \( T \) is the type of \( x \).

If \( e \not\equiv u \) and \( e \) is not a conditional expression then

\[
\frac{\{(u := \text{new}) e\}.x \rightsquigarrow e'}{\{u := \text{new}\} (e.x) \rightsquigarrow e'}
\]

Otherwise, if \( e \) is a conditional expression then

\[
\frac{\text{if } \{u := \text{new}\} b \text{ then } \{u := \text{new}\} (e_1.x) \text{ else } \{u := \text{new}\} (e_2.x) \text{ fi } \rightsquigarrow e'}{\{u := \text{new}\} (\text{if } b \text{ then } e_1 \text{ else } e_2 \text{ fi} .x) \rightsquigarrow e'}
\]

Note that we use here the valid equation:

\[
\text{if } b \text{ then } e_1 \text{ else } e_2 \text{ fi} .x = \text{if } b \text{ then } e_1 .x \text{ else } e_2 .x \text{ fi}.
\]

The only other possible context of \( u \) is that of an equality \( e = e' \). We distinguish the following cases.

If neither \( e \) nor \( e' \) is \( u \) or a conditional expression then they cannot refer to the newly created object and we define

\[
\{(u := \text{new}) e \} = \{(u := \text{new}) e' \} \rightsquigarrow e''
\]

\[
\{u := \text{new}\} (e = e') \rightsquigarrow e''
\]

If \( e \) is \( u \) and \( e' \) is neither \( u \) nor a conditional expression (or vice versa) then after \( u := \text{new} \) the expressions \( e \) and \( e' \) cannot denote the same object (because
one of them refers to the newly created object and the other one refers to an already existing object) and so we define

$$\{u := new\}(e = e') \rightsquigarrow false$$

On the other hand if both the expressions $e$ and $e'$ equal $u$ we obviously have

$$\{u := new\}(e = e') \rightsquigarrow true$$

If $e$ is a conditional expression of the form if $b$ then $e_1$ else $e_2$ fi then

$$\text{if } \{u := new\} b \text{ then } \{u := new\} (e_1 = e') \text{ else } \{u := new\} (e_2 = e') \text{ fi } \rightsquigarrow e''$$

And similar for $e' = e$. Note that we use here valid equation: (if $b$ then $e_1$ else $e_2$ fi = $e'$) = if $b$ then $e_1 = e'$ else $e_2 = e'$.

Since object expressions can only be compared for equality, dereferenced or appear as argument of a conditional expression, it is easy to see that for every boolean expression $e$ there exists a boolean expression $e'$ such that $\{u := new\} e \rightsquigarrow e'$.

The following lemma states the semantic correctness of the rewrite relation $\{u := new\} e \rightsquigarrow e'$: The value of $e'$ in the state before the assignment $u := new$ equals the value of $e$ after the assignment.

**Lemma 1.** If $\{u := new\} e \rightsquigarrow e'$ and $[u := new](\Sigma) = \Sigma'$ then $[e'](\Sigma) = [e](\Sigma')$.

Now we generalize the rewrite relation $\rightsquigarrow$ for the computation of $\{u := new\} \phi$, where $\phi$ is a first order formula in predicate logic (which does not contain modalities). We treat the following main case

$$\frac{(\{u := new\} \phi[l/u]) \land \forall l. (\{u := new\} \phi) \rightsquigarrow \psi}{\{u := new\} \forall l. \phi \rightsquigarrow \psi}$$

where $l$ is a logical variable. This rewrite rule takes care of the *changing scope* of the quantified variable $l$ by distinguishing the following cases: $p$ holds for the new object is expressed by the first conjunct $\{u := new\} \phi[l/u]$ which is obtained by applying the update to $\phi[l/u]$ and $p$ holds for all 'old' objects is expressed by the second conjunct $\forall l. (\{u := new\} \phi)$.

As an example, we derive $\{u := new\} \forall l. \neg(u = l) \rightsquigarrow \neg(true) \land \forall l. \neg false$:

$$\frac{\{u := new\} (u = u) \rightsquigarrow true}{\{u := new\} \neg (u = u) \rightsquigarrow \neg(true)} \quad \frac{\{u := new\} (u = l) \rightsquigarrow false}{\forall l. \{u := new\} \neg (u = l) \rightsquigarrow \forall l. \neg false}$$

$$\frac{\{u := new\} \neg (u = u) \rightsquigarrow \neg(true) \land \forall l. \{u := new\} \neg (u = l) \rightsquigarrow \forall l. \neg false}{\{u := new\} \forall l. \neg (u = l) \rightsquigarrow \neg(true) \land \forall l. \neg false}$$
The resulting formula is equivalent to false. This is indeed correct because \( (u := \text{new}) \forall l. \neg (u = l) \) is invalid. In fact we have the following derivation in dynamic logic for \( \neg (u := \text{new}) \forall l. \neg (u = l) \).

\[
\begin{align*}
\text{closeTrue} & \quad \forall l. \neg \text{false} \vdash \text{true} \\
\text{notLeft} & \quad \neg (\text{true}) \land \forall l. \neg \text{false} \vdash \\
\text{andLeft} & \quad \{u := \text{new}\} \forall l. \neg (u = l) \vdash \\
\text{applyUpd} & \quad (u := \text{new}) \forall l. \neg (u = l) \vdash \\
\text{assignVar} & \quad (u := \text{new}) \forall l. \neg (u = l) \vdash \\
\text{notRight} & \quad \vdash \neg (u := \text{new}) \forall l. \neg (u = l)
\end{align*}
\]

On the other hand, we have the following derivation of

\[\forall l. (u := \text{new}) \neg (u = l)\]

which expresses in an abstract and natural way that \( u \) indeed is a new object different from all the ‘old’ objects.

\[
\begin{align*}
\text{closeFalse} & \quad \text{false} \vdash \\
\text{notRight} & \quad \vdash \neg \text{false} \\
\text{applyUpd} & \quad \vdash \{u := \text{new}\} \neg (u = c) \\
\text{assignVar} & \quad \vdash (u := \text{new}) \neg (u = c) \\
\text{allRight} & \quad \vdash \forall l. (\{u := \text{new}\} \neg (u = l))
\end{align*}
\]

This latter example shows that the standard rules for quantification apply to the quantification over the existing objects.

4 Symbolic Execution

4.1 Simultaneous Updates for Symbolic State Representation

The proof system presented so far allows for classical backwards reasoning, in a weakest precondition manner. We now generalise the notion of updates, to allow for accumulating simultaneous substitutions, thereby delaying their application. In particular, this can be done in a forward manner, giving the proofs a symbolic execution nature. We illustrate this principle by example, in Fig. 4.

The first application of the update rule \( \text{mergeUpd} \) introduces what is called the simultaneous update \( w := u | u := v \). After the second application of \( \text{mergeUpd} \), the \( w \) from the inner update was turned into a \( u \) in the simultaneous update, by applying the outer update to the inner one. For a full account on simultaneous updates, see [11].

The idea to use simultaneous updates for symbolic execution developed in the course of the KeY project [1], and turned out a powerful concept for verification of real world (Java) programs. A simultaneous update acts as a representation...
of the symbolic state which is reached by ‘executing’ the program in the proof up to the current proof node. The program is ‘executed’ in a forward manner, avoiding the backwards execution of (pure) weakest precondition calculi, thereby achieving better readability of proofs. This delayed, accumulated substitution is only applied to the post-condition as a final, single step. However, the intermediate updates are also applied earlier, namely to the branching conditions of a program (via the rules if and applyUpd). In effect, this computes, for each branch, the condition on the initial state under which this branch is executed. The KeY tool uses these artifacts not only for verification, but also for test case generation with high code based coverage [5] and for symbolic debugging.

4.2 Symbolic Execution and Abstract Object Creation

One motivation for choosing the setting of dynamic logic with updates is to allow for abstract object creation in symbolic execution style verification. For that, we have to answer the question on how symbolic execution and abstract object creation can be combined. The problem is that there is no natural way of merging an update \{u := \text{new}\} with other updates. Consider, for instance, the following formulas, only the first of which is valid.

\[
\langle u := \text{new}; v := u \rangle (u = v) \quad \langle u := \text{new}; v := \text{new} \rangle (u = v)
\]

Symbolic execution would first generate the following formulas:

\[
\{ u := \text{new} \} \{ v := u \} (u = v) \quad \{ u := \text{new} \} \{ v := \text{new} \} (u = v)
\]

Merging the updates naively would in both cases result in:

\[
\{ u := \text{new} \mid v := \text{new} \} (u = v \land \neg (v = l))
\]

Whichever semantics one might give to a simultaneous update with two object creations, the formula cannot be both valid and invalid.

The proposed solution is twofold: not to merge an object creating update with other updates at all, but create second reference to the new object, to be
used for merging. For this, we introduce a fresh auxiliary variable to store the newly created object, and generate two updates according to the following rule:

\[
\text{createObj} \quad \frac{\{a := \text{new}\} \{u := a\} \phi}{\{u = \text{new}\} \phi}
\]

with \(a\) a fresh program variable.

The inner update \(\{u := a\}\) can be merged normally with other updates resulting from the analysis of \(\phi\). The next point to address is the ‘disruption’ of the symbolic state, caused by creating updates unable to merge with their ‘neighbours’, thereby strictly separating state changes happening before and after object creation. The key idea for overcoming this is to gradually move all object creations to the very front (as if all objects were allocated up front) and perform standard symbolic execution on the remaining updates. We achieve this by the following rule:

\[
\text{creationFirst} \quad \frac{\{u := \text{new}\} U_{nc} \phi}{\{u := \text{new}\} \phi}
\]

with \(U_{nc}\) non-creating, and \(u\) not appearing anywhere in \(U_{nc}\).

We illustrate symbolic execution with abstract object creation by an example.

\[
\begin{align*}
\text{notRight, closeFalse} & \quad \vdash \neg \text{false} \\
\text{applyUpd} & \quad \vdash \{a := \text{new}\} \{v := a\} \neg (w = v) \\
\text{mergeUpd} & \quad \vdash \{a := \text{new}\} \{u := a; v := a\} \neg (w = v) \\
\text{mergeUpd, assignVar} & \quad \vdash \{a := \text{new}\} \{u := a; v := a; w := a\} \neg (w = v) \\
\text{creationFirst} & \quad \vdash \{u := v\} \{a := \text{new}\} \{v := a; w := a\} \neg (w = v) \\
\text{split, createObj} & \quad \vdash \{u := v\} \{v := \text{new}; w := u\} \neg (w = v) \\
\text{split, assignVar} & \quad \vdash \{u := v\} \{v := \text{new}; w := u\} \neg (w = v)
\end{align*}
\]

5 Discussion

5.1 Object Creation vs. Object Activation

Proof systems for object oriented languages usually achieve the uniqueness of objects via an injective mapping (let us call that \(\text{obj}\)) from the natural numbers to object identities. Only the object identities \(\text{obj}(i)\) up to a maximum index \(i\) are considered to stand for actually created objects. In each state, the successor of this maximum index is stored in a ‘ghost’ variable, which we call \(\text{next}\) here. (In case of Java, \(\text{next}\) would be a static field, for each class). Object creation advances the value or \(\text{next}\), which conceptually is more an ‘activation’ than a creation process. Quantifiers cover the entire co-domain of \(\text{obj}\), including ‘not yet created’ objects. In order to restrict a certain property \(\phi\) the ‘created’ objects, the following pattern is used: \(\forall l. (\psi \rightarrow \phi)\), where \(\psi\) achieves the restriction.
to the created objects. ψ can have the form ∃n. (n < next ∧ obj(n) = l), which would be the approach to be taken in ODL [3]. Alternatively, to avoid the extra quantifier, one can introduce a ‘ghost’ instance variable of boolean type, created, to indicate for each object its createdness status [2]. Then, one needs the additional assumption (1): ∀n. (obj(n).created ↔ n < next). In either case, further assumptions need to state that fields of created objects always refer to created objects.

To contrast this with our approach, we refer again to the five step derivation of ∃l. ⟨u := new⟩¬(u = l) in Sect. 3. In a usual approach, this is not a valid formula. Instead, one would state: ∀l. (l.created → ⟨u := new⟩¬(u = l)). An object activation proof of this is given in Fig. 5 (abbreviating created by cr). Many steps in this proof concern the particular details of the explicit representation of objects and the implementation of object creation.

5.2 Expressiveness

Many interesting properties of dynamic object structures, like reachability in dynamic linked data structures, cannot be expressed in first-order predicate logic. There are approaches to simulate reachability by an overapproximation of the reachable states [9]. In first-order dynamic logic however we can use the modalities to express such properties. For example, if a linked list is given in terms of a field next and the data is stored in a field data then the following formula in dynamic logic states that the object denoted by v is reachable from the object denoted by u:

⟨while u ≠ v do u := u.next od⟩(true)
6 Conclusion

In this paper we presented the integration of a weakest precondition calculus for abstract object creation in dynamic logic and the underlying theorem prover of the KeY tool. Abstract object creation is formalized in terms of an inductively defined rewrite relation. The standard sequent calculus for dynamic logic is extended with a schema rule which allows to substitute formulas in sequents and thus provides a general mechanism to import for example specific rewrite relations. The resulting logic abstracts from an explicit representation of objects and the corresponding implementation of object creation. As such it abstracts from irrelevant implementation details which in general complicate proofs. Moreover, it treats the dynamic scope of quantified object variables in a standard manner. Finally, we have shown how to symbolically execute abstract object creation in KeY.

Currently, we are implementing and extending the toy language to other programming constructs of object-oriented languages like Java.

References

2. B. Beckert, V. Klebanov, and S. Schlager. Dynamic logic. In Beckert et al. [1], pages 69–177.
Appendix C

Papers on Incremental verification

C.1 Lazy Behavioral Subtyping
Lazy behavioral subtyping

Johan Dovland *, Einar Broch Johnsen, Olaf Owe, Martin Steffen

Institutt for informatikk, Universitetet i Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway

ARTICLE INFO

Article history:
Available online 15 July 2010

Keywords:
Object orientation
Inheritance
Code reuse
Late binding
Proof systems
Method redefinition
Incremental reasoning
Behavioral subtyping

ABSTRACT

Inheritance combined with late binding allows flexible code reuse but complicates formal reasoning significantly, as a method call’s receiver class is not statically known. This is especially true when programs are incrementally developed by extending class hierarchies. This paper develops a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification of method specifications is avoided without restricting method overriding to fully behavior-preserving redefinition. The approach ensures that when analyzing the methods of a class, it suffices to consider that class and its superclasses. Thus, the full class hierarchy is not needed, and incremental reasoning is supported. We formalize this approach as a calculus which lazily imposes context-dependent subtyping constraints on method definitions. The calculus ensures that all method specifications required by late bound calls remain satisfied when new classes extend a class hierarchy. The calculus does not depend on a specific program logic, but the examples in the paper use a Hoare style proof system. We show soundness of the analysis method. The paper finally demonstrates how lazy behavioral subtyping can be combined with interface specifications to produce an incremental and modular reasoning system for object-oriented class hierarchies.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Inheritance and late binding of method calls are central features of object-oriented languages and contribute to flexible code reuse. A class may extend its superclasses with new methods, possibly overriding existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the behavior of a method call depends on the selected code and the binding of a call to code cannot be statically determined; i.e., the binding at run-time depends on the actual class of the called object. In addition, object-oriented programs are often designed under an open world assumption: Class hierarchies are extended over time as subclasses are gradually developed. Class extensions will lead to new potential bindings for overridden methods. Thus, inherited methods may change behavior due to internal calls.

To control this flexibility, existing reasoning and verification strategies impose restrictions on inheritance and redefinition. One strategy is to ignore openness and assume a closed world; i.e., the proof rules assume that the complete inheritance tree is available at reasoning time (e.g., [43]). This severely restricts the applicability of the proof strategy; for example, libraries are designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, intended to support incremental development and analysis. If the reasoning relies on the world being closed, extending the class hierarchy requires a costly reverification.

** This work was done in the context of the EU FP6 IST project IST-33826 Credo: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl/) and FP7 ICT project 231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu/).
* Corresponding author.
E-mail addresses: johand@ifi.uio.no (J. Dovland), einarj@ifi.uio.no (E.B. Johnsen), olaf@ifi.uio.no (O. Owe), msteffen@ifi.uio.no (M. Steffen).

1567-8326/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2010.07.008
An alternative strategy is to reflect in the verification system that the world is open, but to constrain how methods may be redefined. The general idea is that in order to avoid reverification, any redefinition of a method through overriding must preserve certain properties of the method being redefined. An important part of the properties to be preserved is the method’s contract; i.e., the pre- and postconditions for its body. The contract can be seen as a description of the promised behavior of all implementations of the method as part of its interface description, the method’s specification. Best known as behavioral subtyping (e.g., [3,4,35,37,38,45]), this strategy achieves incremental reasoning by limiting the possibilities for method overriding, and thereby code reuse. Once a specification is given for a method, this specification must be respected by later redefinitions. However, behavioral subtyping has been criticized for being overly restrictive and often violated in practice [46].

A main difficulty with behavioral subtyping is that a strong class specification limits method overriding in subclasses, while a weak class specification limits reasoning. Thus, when writing a class specification one should think of all future code reuse in subclasses. This conflicts with the open world assumption. Another problem is that when reusing a class which only has a weak specification, one must look at the actual code to find out what the class does. The strategy of lazy behavioral subtyping, introduced in this paper, relaxes the restriction to property preservation which applies in behavioral subtyping, while embracing the open world assumption of incremental program development. A class may well be given a strong specification, while the properties to be preserved by subclasses are in general weaker, ensuring that internal calls are correct. The strong specification reduces the need for code inspection. The central idea is as follows: given a method \( m \) specified by a precondition \( p \) and a postcondition \( q \), there is no need to restrict the behavior of methods overriding \( m \) and require that these adhere to that specification. Instead it suffices to preserve the “part” of \( p \) and \( q \) that is actually used to verify the program at the current stage. Specifically, if \( m \) is used in the program in the form of an internal method call \( \{r\} m\ldots\{s\} \), the pre- and postconditions \( r \) and \( s \) at that call-site constitute \( m \)'s required behavior. Observe that the requirements are weaker than the specifications, and it is in fact these weaker requirements that need to be preserved by subclass overrides in order to avoid reverification. We therefore call the corresponding analysis strategy lazy behavioral subtyping.

**Example 1.** Consider the following two classes:

```java
class Account {
    int bal;
    void deposit(nat x) {update(x)}
    void withdraw(nat x) {update(-x)}
    void update(int x) {bal := bal + x}
}

class FeeAccount extends Account {
    int fee;
    void withdraw(nat x) {update(-x+fee)}
}
```

In this example, class `Account` implements ideal bank accounts for which the `withdraw` method satisfies the pre- and post-condition pair \((bal = bal_0, \text{bal} = bal_0 - x)\), where `bal_0` is a logical variable used to capture the initial value of `bal`. The subclass `FeeAccount` redefines the `withdraw` method, charging an additional `fee` for each withdrawal. Thus, class `FeeAccount` is not a behavioral subtype of class `Account`. However, the example illustrates that it might be fruitful to implement `FeeAccount` as an extension of `Account` since much of the existing code can be reused by the subclass. In this paper we focus on incremental reasoning in this setting: Subclasses may reuse and override superclass code in a flexible manner such that superclass specifications need not be respected.

The paper formalizes the lazy behavioral subtyping analysis strategy using an object-oriented kernel language, based on Featherweight Java [28], and using Hoare style proof outlines. Formalized as a syntax-driven inference system, class analysis is done in the context of a proof environment constructed during the analysis. The environment keeps track of the context-dependent requirements on method definitions, derived from late bound internal calls in the known class hierarchy. The strategy is incremental; for the analysis of a class \( C \), only knowledge of \( C \) and its superclasses is needed. We first present a simple form of the calculus, previously published in [20], in order to focus on the mechanics of the lazy behavioral subtyping inference system. In the present paper, the soundness proofs are given for this calculus. Although this system ensures that old proofs are never violated, external calls may result in additional proof obligations in a class which has already been analyzed. As a consequence, it may be necessary to revisit classes at a later stage in the program analysis. To improve this situation, we extend [20] by considering a refined version of the calculus which introduces behavioral interfaces to encapsulate objects. This refined calculus is, in our opinion, more practical for real program analysis and a better candidate for implementation. The behavioral constraints of the interface implemented by a class become proof obligations for that class, and external calls are verified against the behavioral constraints of the interface. As a result, the refined calculus is both incremental and modular: Each class is analyzed once, after its superclasses, ensuring that verified properties of superclasses are not violated, and external calls are analyzed based on interface constraints. Inherited code is analyzed in the context of the subclass only when new properties are needed. A subclass need not implement the interface of a superclass, thereby allowing code to be
reused freely by the subclass without satisfying the behavioral constraints of the superclass. The lazy behavioral subtyping strategy may serve as a blueprint for integrating a flexible system for program verification of late bound method calls into environments for object-oriented program development and analysis tools (e.g., [7, 8, 10]).

**Paper overview.** Section 2 introduces the problem of reasoning about late binding, Section 3 presents the lazy behavioral subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Late bound method calls

2.1. Syntax for an object-oriented kernel language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

**Fig. 1. Syntax for the language OOL.** Here $C$ and $m$ are class and method names (of types Cid and Mid, respectively). Assignable program variables $v$ include field names $f$ and the reserved variable $return$ for return values. The expression $op(e)$ denotes operations over integer and Boolean expressions $(b)$. The notation $[A | B]$ denotes a choice between $A$ and $B$, and $[A]^T$ denotes that $A$ is optional.

**Syntax for an object-oriented kernel language OOL.**

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Late bound method calls

2.1. Syntax for an object-oriented kernel language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

**Fig. 1. Syntax for the language OOL.** Here $C$ and $m$ are class and method names (of types Cid and Mid, respectively). Assignable program variables $v$ include field names $f$ and the reserved variable $return$ for return values. The expression $op(e)$ denotes operations over integer and Boolean expressions $(b)$. The notation $[A | B]$ denotes a choice between $A$ and $B$, and $[A]^T$ denotes that $A$ is optional.

**Syntax for an object-oriented kernel language OOL.**

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Late bound method calls

2.1. Syntax for an object-oriented kernel language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Late bound method calls

2.1. Syntax for an object-oriented kernel language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [28] and Creol [32]). The language is named OOL, and the syntax is given in Fig. 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $OOL$. Named subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.
Late binding (or dynamic dispatch), already present in Simula [14], is a central concept of object-oriented programming. A method call is late bound if the method body is selected at run-time, depending on the callee’s actual class. Late bound calls are bound to the first implementation found above the actual class. For a class \( \text{class } C \text{ extends } B \) \((FMM)\) we recursively define the partial function \( \text{bind} \) for binding late bound calls, by

\[
\text{bind}(C, m) \triangleq \begin{cases} 
\text{if } m \in M & \text{then } C \\
\text{else } \text{bind}(B, m) & \text{fi}.
\end{cases}
\]

where \( m \in M \) denotes that an implementation of \( m \) is found in \( M \). Thus, \( \text{bind}(C, m) \) returns the first class above \( C \) where a definition of \( m \) is found. Assuming type safety, this function is always well-defined. We say that an implementation of \( m \) in \( A \) is visible from \( C \) if \( \text{bind}(C, m) = A \). In this case, a late bound call to \( m \) on an instance of \( C \) will bind to the definition in \( A \).

Late binding is illustrated in Figure 2, in which a class \( C \) and two independent subclasses \( B_1 \) and \( B_2 \) are defined: an object of class \( B_1 \) executes an inherited method \( b \) defined in its superclass \( A \) and this method issues a call to method \( a \) defined in both classes \( A \) and \( B_1 \). With late binding, the code selected for execution is associated with the first matching \( a \) above \( B_1 \); i.e., as the calling object is an instance of \( B_1 \), the method \( a \) of \( B_1 \) is selected and not the one of \( A \). If, however, method \( b \) were executed in an instance of \( A \), the late bound invocation of \( a \) would be bound to the definition in \( A \). Late binding is central to object-oriented programming and especially underlies many of the well-known object-oriented design patterns [23].

For a late bound internal call to \( m \), made by a method defined in class \( C \), we say that a definition of \( m \) in class \( D \) is reachable if the definition in \( D \) is visible from \( C \), or if \( D \) is a subclass of \( C \). As \( m \) may be overridden by any subclass of \( C \), there may be several reachable definitions for a late bound call statement. For the calls to \( a \) in class \( A \) in Fig. 2, the definitions of \( a \) in \( A \), \( B_1 \), and \( B_2 \) are all reachable. At run-time, one of the reachable definitions is selected based on the actual class of the called object. Correspondingly, for an external call \( e \cdot m() \) where \( e : E \), a definition of \( m \) in class \( D \) is reachable if the definition is visible from \( E \) or if \( D \) is a subclass of \( E \).

2.3. Proof outlines

Apart from the treatment of late bound method calls, our initial reasoning system follows standard proof rules [5, 6] for partial correctness, adapted to the object-oriented setting; in particular, de Boer’s technique using sequences in the assertion language addresses the issue of object creation [15]. We present the proof system using Hoare triples \( \{ p \} t \{ q \} \) [24], where \( p \) is the precondition and \( q \) is the postcondition to the statement \( t \). Triples \( \{ p \} t \{ q \} \) have a standard partial correctness semantics: if \( t \) is executed in a state where \( p \) holds and the execution terminates, then \( q \) holds after \( t \) has terminated. The derivation of triples can be done in any suitable program logic. Let \( PL \) be such a program logic and let \( \vdash_{PL} \{ p \} t \{ q \} \) denote that \( \{ p \} t \{ q \} \) is derivable in \( PL \). A proof outline [41] for a method definition \( T \cdot m(T \times)\{t\} \) is the method body decorated with assertions. For the purpose of this paper, we are mainly interested in method calls decorated with pre- and postconditions.

Let the notation \( \{ r \} n() \{ s \} \) in \( O \) be requirements. Thus, for a decorated call \( \{ r \} n() \{ s \} \) in \( O \), \( (r, s) \) is a requirement for \( n \). In order to ensure that this requirement is satisfied, every reachable definition of \( n \) must be analyzed (including definitions which may appear in future subclasses).
Fig. 3. Closed world proof rules. Let \( p[e/v] \) denote the substitution of all occurrences of \( v \) in \( p \) by \( e \) [24], extended for object creation by introducing unused symbols \( \text{fresh} \) following [43]. The function \( \text{reachable}(E, m) \) returns the set of classes statically reachable for the call \( e.m \) (as explained above). For closed world systems this covers all possible definitions of \( m \) chosen by late binding. The body of \( m \) for class \( i \) is given by \( \text{body}^i_m(\pi) \) where \( \pi \) is the formal parameter list. The logical variable list \( \pi \) (assumed disjoint with other variables) is used to formalize that parameters are read-only.

2.4. Reasoning about late bound calls in closed systems

If the proof system assumes a closed world, all classes must be defined before the analysis can begin because the requirement to a method call is derived from the specifications of all reachable implementations of that method. To simplify the presentation in this paper, we omit further details of the assertion language and the proof system (e.g., ignoring the requirement to a method call is derived from the specifications of all reachable implementations of that method). To simplify reasoning about late bound calls and is well-suited for program development, being less restrictive than behavioral subtyping. Behavioral subtyping supports incremental reasoning about extensible class hierarchies. The approach is oriented towards

Example 2. Consider the class hierarchy of Fig. 2, where the methods are decorated with proof outlines. Let \( r_1, s_1 \) and \( r_2, s_2 \) be the requirements for method \( a \) be the requirements for method \( a \) by the proof outlines for the given specifications \( (p_b, q_b) \) and \( (p_c, q_c) \) of methods \( b \) and \( c \), respectively. Assume \( O_1 \vdash p_1 t_1 : (p_1, q_1), O_2 \vdash p_2 t_2 : (p_2, q_2), \) and \( O_3 \vdash p_3 t_3 : (p_3, q_3) \) for the definitions of \( a \) in classes \( A, B_1, \) and \( B_2, \) respectively. Consider initially the class hierarchy consisting of \( A \) and \( B_1 \) and ignore \( B_2 \) for the moment. The proof system of Fig. 3 gives the Hoare triple \( (p_1 \land p_2) a() (q_1 \lor q_2) \) for each call to \( a \), i.e., for the calls in the bodies of methods \( b \) and \( c \) in class \( A \). In order to apply \( \text{Adapt} \), we get the proof obligations: \( r_1 \Rightarrow p_1 \land p_2 \) and \( q_1 \lor q_2 \Rightarrow s_1 \) for \( b \), and \( r_2 \Rightarrow p_1 \land p_2 \), and \( q_1 \lor q_2 \Rightarrow s_2 \) for \( c \). If the class hierarchy is now extended with \( B_2 \), the closed world assumption breaks and the methods \( b \) and \( c \) need to be reverified. With the new Hoare triple \( (p_1 \land p_2 \land p_3) a() (q_1 \lor q_2 \lor q_3) \) at every call-site, the proof obligations given above for applying \( \text{Adapt} \) no longer apply.

3. A lazy approach to incremental reasoning

This section informally presents the approach of lazy behavioral subtyping. Based on an open world assumption, lazy behavioral subtyping supports incremental reasoning about extensible class hierarchies. The approach is oriented towards reasoning about late bound calls and is well-suited for program development, being less restrictive than behavioral subtyping. A formal presentation of lazy behavioral subtyping is given in Section 4.

To illustrate the approach, first reconsider class \( A \) in Fig. 2. The analysis for methods \( b \) and \( c \) requires that \( \{r_1\} a() \{s_1\} \) and \( \{r_2\} a() \{s_2\} \) hold for the internal calls to \( a \) in the bodies of \( b \) and \( c \), respectively. The assertion pairs \( \{r_1, s_1\} \) and \( \{r_2, s_2\} \) may be seen as requirements to all reachable definitions of \( a \). Consequently, for \( a \)’s definition in \( A \), both \( \{r_1\} t_1 \{s_1\} \) and \( \{r_2\} t_1 \{s_2\} \) must hold. Compared to Example 2, the proof obligations for method calls have shifted from the call to the definition site, which allows incremental reasoning. During the verification of a class only the class and its subclasses need to be considered, subclasses are ignored. If we later analyze subclass \( B_1 \) or \( B_2 \), the same requirements apply to their definition of \( a \). Thus, no re verification of the bodies of \( b \) and \( c \) is needed when new subclasses are analyzed.
Although A is analyzed independently of B1 and B2, its requirements must be considered during the analysis of the subclasses. For this purpose, a proof environment is constructed and maintained during the analysis. While analyzing A, it is recorded in the proof environment that A requires both \((r_1, s_1)\) and \((r_2, s_2)\) from a. Subclasses are analyzed in the context of this proof environment, and may in turn extend the proof environment with new requirements, tracking the scope of each requirement. For two independent subclasses, the requirements made by one subclass should not affect the other since internal calls in one subclass cannot bind to method definitions in the other. Hence, the order of subclass analysis does not influence the assertions to be verified in each class. To avoid reverification, the proof environment also tracks the specifications established for each method definition. The analysis of a requirement to a method definition succeeds directly if the requirement follows from the previously established specifications of that method. Otherwise, the requirement may make a new proof outline for the method necessary.

3.1. Assertions and assertion entailment

Consider an assertion language with expressions \(e\) defined by

\[
e := \text{this} \mid \text{return} \mid f \mid x \mid z \mid \text{op}(\bar{e})
\]

In the assertion language, \(f\) is a program field, \(x\) a formal parameter, \(z\) a logical variable, and \(\text{op}\) an operation on data types. An assertion pair (of type \(A\text{Pair}\)) is a pair \((p, q)\) of Boolean expressions. Let \(p'\) denote the expression \(p\) with all occurrences of program variables \(f\) substituted by the corresponding primed variables \(\bar{f}\), avoiding name capture. Since we deal with sets of assertion pairs, the standard adaptation rule of Hoare Logic given in Fig. 3 is insufficient. We need an entailment relation which allows us to combine information from several assertion pairs.

**Definition 1 (Entailment).** Let \((p, q)\) and \((r, s)\) be assertion pairs and let \(\mathcal{U}\) and \(\mathcal{V}\) denote the sets \(\{(p_i, q_i)\mid 1 \leq i \leq n\}\) and \(\{(r_i, s_i)\mid 1 \leq i \leq m\}\). Entailment is defined over assertion pairs and sets of assertion pairs by

1. \((p, q) \rightarrow (r, s) \triangleq (\forall z_1 . \ p \Rightarrow q') \Rightarrow (\forall z_2 . \ r \Rightarrow s')\), where \(z_1\) and \(z_2\) are the logical variables in \((p, q)\) and \((r, s)\), respectively.
2. \(\mathcal{U} \rightarrow (r, s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall z_1 . \ p_i \Rightarrow q'_i)) \Rightarrow (\forall z_2 . \ r \Rightarrow s')\).
3. \(\mathcal{U} \rightarrow \mathcal{V} \triangleq (\bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrow (r_i, s_i))\).

The relation \(\mathcal{U} \rightarrow (r, s)\) corresponds to classic Hoare style reasoning, proving \(\{r\} t \{s\}\) from \(\{p_i\} t \{q_i\}\) for all \(1 \leq i \leq n\), by means of the adaptation and conjunction rules [5]. Note that when proving entailment, program variables (primed and unprimed) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and \(\mathcal{V} \subseteq \mathcal{U}\) implies \(\mathcal{U} \rightarrow \mathcal{V}\).

**Example 3.** Let \(x\) and \(y\) be fields, and \(z_1\) and \(z_2\) be logical variables. The assertion pair \((x = y = z_1, x = y = z_1 + 1)\) entails \((x = y, x = y)\), but it does not entail \((x = z_2, x = z_2 + 1)\), since the implication

\[
(\forall z_1 . \ x = y = z_1 \Rightarrow x' = y' = z_1 + 1) \Rightarrow (\forall z_2 . \ x = z_2 \Rightarrow x' = z_2 + 1)
\]

do not hold. To see that, we take the assertion pairs as pre- and postconditions for the program \(t \triangleq y := y + 1; x := y\). The Hoare triple \(\{x = y = z_1\} t \{x = y = z_1 + 1\}\) is valid, whereas \(\{x = z_2\} t \{x = z_2 + 1\}\) is not valid.

**Example 4.** This example demonstrates entailment for sets of assertion pairs: The two assertion pairs \((x \neq null, x \neq null)\) and \((y = z_1, z_1 = null \lor z_1 = y)\) entail \((x \neq null \lor y \neq null, x \neq null \lor y \neq null)\). This kind of reasoning is relevant for reasoning about class invariants without behavioral subtyping: when defining a subclass with a different class invariant than the superclass, the established knowledge of inherited methods may be used to prove the class invariant of the subclass.

3.2. Class analysis with a proof environment

The role of the proof environment during the class analysis will now be illustrated through a series of examples. Classes are analyzed after their respective superclasses, and each class is analyzed without knowledge of its possible subclasses. The proof environment collects the method specifications and requirements in two mappings \(S\) and \(R\). Given the names of a class and a method, these mappings return a set of assertion pairs. The analysis of a class both uses and extends the proof environment. In particular, \(S(C, m)\) is the set of specifications established for the (possibly inherited) definition of \(m\) in class \(C\), and \(R(C, m)\) is the set of assertion pairs that must be respected by any redefinition of \(m\) below \(C\), as required by the analysis so far. By the analysis of class \(C\), the user given specifications are included in the \(S\) mapping. The analysis of proof outlines for these specifications may in turn impose requirements on internally called methods. These requirements are included in the \(R\) mapping as explained below. The \(S\) and \(R\) mappings accumulate the results of the analysis so far, and form the basis of a mechanizable reasoning system for open class hierarchies (excluding generation of proof outlines). Intuitively, the mapping
S reflects the definition of methods; each lookup $S(C, m)$ returns a set of specifications for a particular implementation of $m$. In contrast, the mapping $R$ reflects the use of methods and may impose requirements on several implementations.

**Propagation of requirements.** If the proof outline $O \vdash_{PL} t : (p, q)$ for a method $T \uparrow m(T x)(t)$ is derived while analyzing a class $C$, we extend $S(C, m)$ with $(p, q)$. The requirements on called methods which are encountered during the analysis of $O$ are verified for the known definitions of these methods that are visible from $C$, and imposed on future subclasses. Thus, for each $\{ r \} n(s) \{ s \}$ in $O$, the following two steps are taken:

1. The requirement $(r, s)$ is analyzed with regard to the definition of $n$ that is visible from $C$.
2. $R(C, n)$ is extended with $(r, s)$.

The analysis in Step 1 ensures that the requirement can be relied on when the call is executed on an instance of class $C$. The inclusion of $(r, s)$ in $R(C, n)$ in Step 2 acts as a restriction on future subclasses of $C$. Whenever $n$ is overridden by a subclass of $C$, requirements $R(C, n)$ are verified for the new definition of $n$. Thereby, the requirement $(r, s)$ can also be relied on when the call in the body of $m$ is executed on an instance of a subclass of $C$. Consequently, the specification $(p, q)$ of $m$ can be relied on when the method is executed on a subclass instance. For a static call $\{ r \} n(A(s)) \{ s \}$ in $O$, the assertion pair $(r, s)$ must follow from $S(A, n)$, the specification of $n$ in $A$. There is no need to impose this assertion pairs on subclass overrides since the call is bound at compile time, i.e., the assertion pair is not included in the set $R(C, n)$.

**Example 5.** Consider the analysis of class $A$ in Fig. 2. The specification $(p_1, q_1)$ is analyzed for the definition of $a$, and included in the mapping $S(A, a)$. For method $b$, the specification $(p_b, q_b)$ is analyzed and included in $S(A, b)$. In the body of $b$, there is a call to $a$ with requirement $(r_1, s_1)$. This requirement is analyzed for $a$ in $A$ (by Step 1), and included in $R(A, a)$ (by Step 2). The analysis of method $c$ follows the same strategy, and leads to the inclusion of $(r_2, s_2)$ in $R(A, a)$. By Step 1, both requirements must be verified for the definition of $a$ in $A$, since this is the definition of $a$ that is visible from $A$. Consequently, for each $(r_1, s_1)$, $S(A, a) \rightarrow (r_1, s_1)$ must hold. This relation holds directly, assuming $(p_1, q_1) \rightarrow (r_1, s_1)$. To summarize, the following assertion pairs are thereby included in the different specification and requirement sets:

$$S(A, a) = \{ (p_1, q_1) \} \quad R(A, a) = \{ (r_1, s_1), (r_2, s_2) \}$$
$$S(A, b) = \{ (p_b, q_b) \}$$
$$S(A, c) = \{ (p_c, q_c) \}$$

In Example 5, it was assumed that the requirements made by $b$ and $c$ followed from the established specification of $a$. Generally, the requirements need not follow from the previously shown specifications. In such a case, it is necessary to provide a new proof outline for the method.

**Example 6.** If $(r_1, s_1)$ does not follow from $(p_1, q_1)$ in Example 5 (i.e., the relation $(p_1, q_1) \rightarrow (r_1, s_1)$ does not hold), a new proof outline $O \vdash_{PL} t_1 : (r_1, s_1)$ must be analyzed similarly to the proof outlines in $A$. The mapping $S(A, a)$ is extended by $(r_1, s_1)$, ensuring the desired relation $S(A, a) \rightarrow (r_1, s_1)$.

The analysis strategy ensures that once a specification $(p, q)$ is included in $S(C, m)$, it will always hold when the definition of method $m$ in $C$ is executed in an instance of any (future) subclass of $C$, without reverifying $m$. Consequently, when a method $n$ called by $m$ is overridden, the requirements made by $C$ must hold for the new definition of $n$.

**Example 7.** Consider the class $B_1$ in Fig. 2, which redefines $a$. By analysis of the proof outline $O_2 \vdash_{PL} t_2 : (p_2, q_2)$, the specification $(p_2, q_2)$ is included in $S(B_1, a)$. In addition, the superclass requirements $R(A, a)$ must hold for the new definition of $a$ in order to ensure that the specifications $S(A, b)$ and $S(A, c)$ of methods $b$ and $c$, respectively, apply for instances of $B_1$. Hence, $S(B_1, a) \rightarrow (r_1, s_1)$ must be ensured for each $(r_1, s_1) \in R(A, a)$, similar to $S(A, a) \rightarrow (r_1, s_1)$ in Example 5.

When a method $m$ is (re)defined in a class $C$, all invocations of $m$ from methods in superclasses will bind to the new definition for instances of $C$. The new definition must therefore support the requirements from all superclasses. Let $R^\uparrow(C, m)$ denote the union of $R(B, m)$ for all $B \leq C$. For each method $m$ defined in $C$, it is necessary to ensure the following property:

$$S(C, m) \rightarrow R^\uparrow(C, m)$$

(1)

It follows that $m$ must support the requirements from $C$ itself; i.e., the formula $S(C, m) \rightarrow R(C, m)$ must hold.

**Context-dependent properties of inherited methods.** Consider now methods that are inherited but not redefined. Assume that a method $m$ is inherited from a superclass of a class $C$. In this case, late bound calls to $m$ from instances of $C$ are bound to the first definition of $m$ above $C$. However, late bound calls made by $m$ are bound in the context of $C$, as $C$ may redefine
methods invoked by \( m \). Furthermore, \( C \) may impose new requirements on \( m \) which were not proved during the analysis of the superclass, resulting in new proof outlines for \( m \). In the analysis of the new proof outlines, we know that late bound calls are bound from \( C \). It would be unsound to extend the specification mapping of the superclass, since the new specifications are only part of the subclass context. Instead, we use \( S(C, m) \) and \( R(C, m) \) for local specification and requirement extensions. These new specifications and requirements only apply in the context of \( C \) and not in the context of its superclasses.

**Example 8.** Assume that the class hierarchy in Fig. 2 is extended by a class \( B_3 \) as indicated in Fig. 4. Class \( B_3 \) inherits the superclass implementation of \( a \). The specification \( (p_d, q_d) \) is included in \( S(B_3, d) \) and the analysis of a proof outline for this specification yields \( \{r_3\}\{a\}\{s_3\} \) as requirement, which is included in \( R(B_3, a) \) and verified for the inherited implementation of \( a \). The verification succeeds if \( S(A, a) \rightarrow (r_3, s_3) \). Otherwise, a new proof outline \( O \vdash_{PL} t_1 : (r_3, s_3) \) is analyzed under the assumption that late bound calls are bound in the context of \( B_3 \). When analyzed, \( (r_3, s_3) \) becomes a specification of \( a \) and it is included in \( S(B_3, a) \). This mapping acts as a local extension of \( S(A, a) \) and contains specifications of \( a \) that hold in the subclass context.

When analyzing a requirement \( \{r\}\{m\}\{s\} \) in \( C \), type safety guarantees that there exists a class \( A \) above \( C \) such that \( m \) is defined in \( A \) and that this method definition is visible from \( C \). For the requirement in Step 1, we can then rely on \( S(A, m) \) and the local extensions of this set for all classes between \( A \) and \( C \). Let the function \( S^\uparrow \) be recursively defined as follows: \( S^\uparrow (C, m) \triangleq S(C, m) \) if \( m \) is defined in \( C \) and \( S^\uparrow (C, m) \triangleq S(C, m) \cup S^\uparrow (B, m) \) otherwise, where \( B \) is the immediate superclass of \( C \). Eq. 1 can now be revised to account for inherited methods:

\[
S^\uparrow (C, m) \rightarrow R^\uparrow (C, m)
\]

Thus, each requirement in \( R(B, m) \), for some class \( B \) above \( C \), must follow from the established specifications of \( m \) in context \( C \). Especially, for each \( (r, s) \in R(C, m) \), \( (r, s) \) must either follow from the superclass specifications or from the local extension \( S(C, m) \). If \( (r, s) \) follows from the local extension \( S(C, m) \), we are in the case when a new proof outline has been analyzed in the context of \( C \). Note that Eq. 2 reduces to Eq. 1 if \( m \) is defined in \( C \).

**Analysis of class hierarchies.** A class hierarchy is analyzed in a top-down manner, starting with \( \text{Object} \) and an empty proof environment. Methods are specified in terms of assertion pairs \( (p, q) \). For each method \( T \vdash m(FX)\{t\} \) defined in a class \( C \), we analyze each \( (p, q) \) occurring either as a specification of \( m \), or as an inherited requirement in \( R^\uparrow (C, m) \). If \( S(C, m) \rightarrow (p, q) \), no further analysis of \( (p, q) \) is needed. Otherwise a proof outline \( O \vdash_{PL} t : (p, q) \), after which \( S(C, m) \) is extended with \( (p, q) \). During the analysis of a proof outline, decorated late bound internal calls \( \{r\} v := n(\overline{v}) \{s\} \) yield requirements \( (r, s) \) on reachable implementations of \( n \). The \( R(C, n) \) mapping is therefore extended with \( (r, s) \) to ensure that future redefinitions of \( n \) will support the requirement. In addition, \( (r, s) \) is analyzed with respect to the implementation of \( n \) that is visible from \( C \); i.e., the first implementation of \( n \) above \( C \), which means that the proof obligation \( S^\uparrow (C, n) \rightarrow (r, s) \) must hold. This obligation will hold directly if the already verified specifications of \( n \) entail \( (r, s) \). Otherwise, a new proof outline \( O' \vdash_{PL} \text{body}(C, n) : (r, s) \) is needed, where method calls in \( O' \) are analyzed in the same manner as for \( O \). The set \( S(C, n) \) is then extended with \( (r, s) \), ensuring the proof obligation \( S^\uparrow (C, n) \rightarrow (r, s) \). For static calls \( \{r\} n@A(\overline{v}) \{s\} \) in \( O \), which are bound to the first implementation of \( n \) above \( A \), the assertion pair \( (r, s) \) must follow by entailment from \( S^\uparrow (A, n) \). For external calls \( \{r\} e.n(\overline{v}) \{s\} \) in \( O \), with \( e : E \), consider first the case that \( (r, s) \) follows by entailment from the requirements \( R^\uparrow (E, n) \) of \( n \) in \( E \). By Eq. (2) we then know that the requirement holds for \( \text{body}(E, n) \). Since \( R^\uparrow (E, n) \) must be respected.
by overridings of \( n \) below \( E \), the assertion pair \( (r, s) \) holds even if \( e \) refers to an instance of a subclass of \( E \) at run-time. In the opposite case, one may extend \( R(E, n) \) upon the need of each external call, but this would lead to re-verification of all subclasses of \( E \). The analysis of external calls is further discussed in Section 5, where behavioral interfaces are used to achieve a modular reasoning system.

Lazy behavioral subtyping. Behavioral subtyping in the traditional sense does not follow from the analysis method outlined above. Behavioral subtyping enforces the property that whenever a method \( m \) is redefined in a class \( C \), its new definition must implement all superclass specifications for \( m \); i.e., the method would have to satisfy \( S(B, m) \) for all \( B \) above \( C \). For example, behavioral subtyping would imply that \( a \) in both \( B_1 \) and \( B_2 \) in Fig. 2 must satisfy \( (p_1, q_1) \). Instead, the \( R \) mapping identifies the requirements imposed by late bound internal calls. Only these assertion pairs must be supported by overriding methods to ensure that the execution of code from its superclasses does not have unexpected results. Thus, only the behavior assumed by the late bound internal call statements is ensured at the subclass level. In this way, requirements are inherited by need, resulting in a lazy form of behavioral subtyping.

Example 9. The following class \( A \) defines two methods \( m \) and \( n \), equipped with method specifications:

```java
class A {
  int n(int y) : (true, return = 5*y) {return := 5*y}
  int m(int x) : (x ≥ 0, return ≥ 2*x) {return := n(x)}
}
```

For method \( n \), analysis of the given specification leads to an inclusion of \((true, return = 5*y)\) in \( S(A, n) \). No requirements are imposed by the analysis of this specification since the method body contains no call statements.

The analysis of method \( m \) leads to an inclusion of the given specification, \((x ≥ 0, return ≥ 2*x)\), in \( S(A, m) \). As the method body contains an internal late bound call, the analysis of a proof outline for this specification leads to a requirement towards the called method \( n \). Let \((y ≥ 0, return ≥ 2*y)\) be the requirement imposed on the internal call to \( n \) in the proof outline for \( m \). During analysis of \( A \), two steps are taken for this requirement. The requirement is verified with regard to the implementation of \( n \) that is visible from \( A \), and it is recorded in \( R(A, n) \) in order to be imposed on future overridings below \( A \). As the definition of \( n \) in \( A \) is the one that is visible from \( A \), the first step is ensured by establishing \( S(A, n) \rightarrow R(A, n) \), which follows directly by the definition of entailment, i.e.,

\[
(true, return = 5*y) \rightarrow (y ≥ 0, return ≥ 2*y)
\]

Next we consider the following extension of \( A \):

```java
class B_1 extends A {
  int n(int y) : (true, return = 2*y) {return := 2*y}
  int m(int x) : (true, return = 2*x)
}
```

The method \( n \) is overridden by \( B_1 \). Method \( m \) is inherited without redefinition, but an additional specification of the inherited method is given. By analyzing \( n \), the given specification \((true, return = 2*y)\) is included in \( S(B_1, n) \) and is verified with regard to the method body. There are no method calls in this method body, and verification of the specification succeeds by standard Hoare reasoning. Additionally, the inherited requirement contained in \( R(A, n) \) must be verified with regard to the subclass implementation of \( n \), i.e., we need to ensure \( S(B_1, n) \rightarrow R(A, n) \). This analysis succeeds by entailment:

\[
(true, return = 2*y) \rightarrow (y ≥ 0, return ≥ 2*y)
\]

Note that behavioral subtyping does not apply to the overriding implementation of \( n \), as the specification \( S(A, n) \) cannot be proved for the new implementation. Even though the overriding does not support behavioral subtyping, the verified specification \( S(A, m) \) of method \( m \) still holds at the subclass level because the requirement imposed by the call to \( n \) in the proof of this specification is satisfied by the overriding method.

Consider also the new specification \((true, return = 2*x)\) of \( m \). As the specification is given by the subclass, it is included in \( S(B_1, m) \). Analysis of this specification leads to the requirement \((true, return = 2*x) \in R(B_1, n) \). This requirement is analyzed with regard to the visible implementation of \( n \) in \( B_1 \) which follows directly by entailment: \( S(B_1, n) \rightarrow R(B_1, n) \). Note that \( S(B_1, m) \) gives a local extension of the inherited specification of \( m \). Especially, the extension relies on the fact that the internal call to \( n \) is bound in the context of class \( B_1 \); the requirement \( R(B_1, n) \) imposed by the call could not be proven with regard to the implementation of \( n \) in \( A \). Combined, the verified specifications of \( m \) in the context of \( B_1 \) are contained in \( S(B_1, m) \):

\[
S(B_1, m) = S(B_1, m) \cup S(A, m) = \{(true, return = 2*x), (x ≥ 0, return ≥ 2*x)\}
\]

which can be reduced to \{\((true, return = 2*x)\)\} by removing the redundant specification.
The following example extends Example 9 to illustrate how subclasses may have conflicting specifications of a superclass method.

**Example 10.** Consider the classes $A$ and $B_1$ given in Example 9 and let $B_2$ be the following extension of $A$:

```java
class $B_2$ extends $A$
{
    int n(int y) : (true, return = $2 \times y + 1$) {return := $2 \times y + 1$}
    int m(int x) : (true, return = $2 \times x + 1$)
}
```

The analysis of $B_2$ is similar to the analysis of $B_1$, and $(true, return = 2 \times y + 1)$ is added to $R(B_2, n)$. Notice that the $m$ specifications of the two subclasses $B_1$ and $B_2$ are conflicting in the sense that their conjunction gives $(true, false)$. However, with lazy behavioral subtyping this does not cause a conflict since the two specifications are kept separate, extending $S(B_1, m)$ and $S(B_2, m)$, respectively. The verified specifications of $m$ in the context of $B_2$ are contained in $S^\uparrow(B_2, m)$:

$$S^\uparrow(B_2, m) = S(B_2, m) \cup S(A, m) =
\{(true, return = 2 \times x + 1), (x \geq 0, return \geq 2 \times x)\}.$$  

4. An assertion calculus for program analysis

The incremental strategy outlined in Section 3 is now formalized as a calculus $LBS(PL)$ which tracks specifications and requirements for method implementations in an extensible class hierarchy, given a sound program logic $PL$. Given a program, the calculus builds an environment which reflects the class hierarchy and captures method specifications and requirements. This environment forms the context for the analysis of new classes, possibly inheriting previously analyzed ones. To emphasize program analysis, we assume that programs are type-safe and hereafter ignore the types of fields and methods in the discussion. The proof environment is formally defined in Section 4.1, and $LBS(PL)$ is given as a set of inference rules in Section 4.2. The soundness of $LBS(PL)$ is established in Section 4.3.

4.1. The proof environment of $LBS(PL)$

A class is represented by a unique name and a tuple $\langle B, \bar{f}, \bar{\mathcal{M}} \rangle$ of type $Class$ from which the superclass name $B$, the fields $\bar{f}$, and the methods $\bar{\mathcal{M}}$ are accessible by observer functions $\text{inh}$, $\text{atts}$, and $\text{mtds}$, respectively. Method names are assumed to be unique within a class. Note that the method specifications $\bar{\mathcal{M}}$ in class definitions are not included in class tuples. Specifications are instead collected in the specification mapping $S$ of proof environments:

**Definition 2 (Proof environments).** A proof environment $E$ of type $Env$ is a tuple $\langle L_E, S_E, R_E \rangle$, where $L_E : Cid \rightarrow Class$ is a partial mapping and $S_E, R_E : Cid \times Mid \rightarrow \text{Set}[APair]$ are total mappings.

In a proof environment $E$, the mapping $L_E$ reflects the class hierarchy, the set $S_E(C, m)$ contains the specifications for $m$ in $C$, and the set $R_E(C, m)$ contains the requirements to $m$ from $C$. For the empty environment $E_{\emptyset}$, $L_{E\emptyset}(C)$ is undefined and $S_{E\emptyset}(C, m) = R_{E\emptyset}(C, m) = \emptyset$ for all $C : Cid$ and $m : Mid$.

Some auxiliary functions on proof environments $E$ are now defined. Assuming that $\text{nil}$ is not a valid $Cid$, these functions range over the type $Cid_n$, where $Cid_n$ equals $Cid$ extended with $\text{nil}$, assuming $\text{Object.inh} = \text{nil}$. Let $M.body = t$ for a

- $\text{bind}_E(\text{nil}, m) \triangleq \perp$
- $\text{bind}_E(C, m) \triangleq \text{if } m \in L_E(C).\text{mtds} \text{ then } C \text{ else } \text{bind}_E(L_E(C).\text{inh}, m)$
- $S^E_{\text{nil}}(\text{nil}, m) \triangleq \emptyset$
- $S^E_{\text{C}}(C, m) \triangleq \text{if } m \in L_E(C).\text{mtds} \text{ then } S_E(C, m) \text{ else } S_E(C, m) \cup S^E_{\text{L}}(L_E(C).\text{inh}, m)$
- $R^E_{\text{nil}}(\text{nil}, m) \triangleq \emptyset$
- $R^E_{\text{C}}(C, m) \triangleq R_E(C, m) \cup R^E_{\text{L}}(L_E(C).\text{inh}, m)$
- $body_E(C, m) \triangleq L_E(\text{bind}_E(C, m)).\text{mtds}(m).\text{body}$
- $\text{nil} \leq_E B \triangleq \perp$
- $C \leq_E B \triangleq C = B \lor L_E(C).\text{inh} \leq_E B$

![Fig. 5. Auxiliary function definitions, where $C, B : Cid$ and $m : Mid$.](image-url)
method definition \( M = m(x)\{t\}. \) Denote by \( \overline{M}(m) \) the definition of method with name \( m \) in \( \overline{M} \), by \( m \in \overline{M} \) that \( m \) is defined in \( \overline{M} \), by \( t' \in t \) that the statement \( t' \) occurs in the statement \( t \), and by \( C \in \mathcal{E} \) that \( L_C(C) \) is defined. The partial function \( \text{bind}_C(C, m) : \text{Cid}_n \times \text{Mid} \rightarrow \text{Cid} \) returns the first class above \( C \) in which the method \( m \) is defined. This function is well-defined since programs are well-typed by assumption. Let the recursively defined functions \( S \uparrow_C(C, m) \) and \( R \uparrow_C(C, m) : \text{Cid}_n \times \text{Mid} \rightarrow \text{Set}[\text{APair}] \) return all specifications of \( m \) above \( C \) and below \( \text{bind}_C(C, m) \), and all requirements to \( m \) that are made by all classes above \( C \) in the proof environment \( \mathcal{E} \), respectively. Finally, \( \text{body}_C(C, m) : \text{Cid} \times \text{Mid} \rightarrow \text{Stmt} \) returns the implementation of \( m \) in \( \text{bind}_C(C, m) \). Let \( \leq_C : \text{Cid}_n \times \text{Cid} \rightarrow \text{Bool} \) be the reflexive and transitive subclass relation on \( \mathcal{E} \). The definitions of these functions are given in Fig. 5.

**A sound environment** reflects that the analyzed classes are correct. If an assertion pair \((p, q)\) appears in \( S_C(C, m) \), there must be a verified proof outline \( O \) in \( \text{PL} \) for the corresponding method body, i.e., \( O \vdash_{\text{PL}} \text{body}_C(C, m) : (p, q) \). Let \( n \) be a method called by \( m \), and let \( \pi \) be the formal parameters of \( n \). For all late bound internal calls \( \{r\} v := n(\pi) \{s\} \) in the proof outline \( O \), the requirement \((r', s')\) is included in \( R_C(C, n) \), where \( r' = (r \land \pi = \pi) \), and \( s' = s[\text{return}/v] \), assuming that the variables \( \pi \), \( \text{return} \) do not occur in \( r, s, \pi \) (otherwise renaming is needed; in the special case where \( v = \text{return} \), \( s' \) is simply \( s \)). Here, \( r' \) accounts for the assignment of actual parameter values to the formal parameters, and \( s' \) accounts for the assignment of the returned value to \( v \). Thus, all requirements made by internal late bound calls in the proof outline are in the \( R \) mapping.

For static internal calls \( \{r\} v := n(A(\pi)) \{s\} \) in \( O \), the assertion pair \((r', s')\) must follow from the specifications \( S \uparrow_C(A, n) \). For external calls \( \{r\} v := e.n(\pi) \{s\} \) in \( O \), with \( e : E \), the requirement \((r', s')\) must follow from \( R \uparrow_C(E, n) \). Note that \( E \) may be independent of the analyzed class \( C \); i.e., the classes need not be related by inheritance. Finally, method specifications must entail the requirements (see Eq. 2 of Section 3.2). Sound environments are defined as follows:

**Definition 3 (Sound environments).** A sound environment \( \mathcal{E} \) of type \( \text{Env} \) satisfies the following conditions for all \( C : \text{Cid} \) and \( m : \text{Mid} \):

1. \( \forall (p, q) \in S_C(C, m) \Rightarrow \exists O . O \vdash_{\text{PL}} \text{body}_C(C, m) : (p, q) \land \forall \{r\} v := n(\pi) \{s\} \in O \Rightarrow R \uparrow_C(C, \pi) \rightarrow (r', s') \land \forall \{r\} v := e.n(\pi) \{s\} \in O \Rightarrow R \uparrow_C(E, \pi) \rightarrow (r', s') \land \forall B . B \notin E \Rightarrow S_{E}(B, m) = R_{E}(B, m) = \emptyset \).

where \( r' = (r \land \pi = \pi' \land \pi) \), \( s' = s[\text{return}/v] \), and \( \pi \) are the formal parameters of \( n \) (assuming that \( \pi \), \( \text{return} \) do not occur in \( r, s, \pi \) or \( \pi \)).

Note that in Condition 1 of Definition 3, the method implementation \( \text{body}_C(C, m) \) need not be in \( C \) itself; the proof outline \( O \) may be given for an inherited method definition.

Let \( \models_C \{p\} t \{q\} \) denote \( \models \{p\} t \{q\} \) under the assumption that internal calls in \( t \) are bound in the context of \( C \), and that each external call in \( t \) is bound in the context of the actual class of the called object. Let \( \models_C m(\pi) : (p, q) \{t\} \) be given by \( \models_C \{p\} t \{q\} \). If there are no method calls in \( t \) and \( \vdash_{\text{PL}} \{p\} t \{q\} \), then \( \models \{p\} t \{q\} \) follows by the soundness of \( \text{PL} \). The following property holds for sound environments:

**Lemma 1.** Given a sound environment \( \mathcal{E} \) and a sound program logic \( \text{PL} \). For all classes \( C : \text{Cid} \), methods \( m : \text{Mid} \), and assertion pairs \((p, q) : \text{APair} \) such that \( C \in \mathcal{E} \) and \((p, q) \in S_{C}(C, m) \), we have \( \models_D m(\pi) : (p, q) \{\text{body}_C(C, m)\} \) for each \( D \leq_C C \).

**Proof.** By induction on the number of calls in \( m \). Since \((p, q) \in S \uparrow_C(C, m) \), it follows from the definition of \( S \uparrow_C \) in Fig. 5 that there exist some class \( B \) such that \( C \leq_C B \), \( \text{bind}_C(C, m) = \text{bind}_B(B, m) \), and \((p, q) \in S_{E}(B, m) \). For such a class \( B \), \( \text{body}_B(B, m) = \text{body}_E(B, m) \). Since \((p, q) \in S_{E}(B, m) \), there must, by Definition 3, Condition 1, exist some proof outline \( O \) such that \( O \vdash_{\text{PL}} \text{body}_B(B, m) : (p, q) \).

In this proof outline, each method call is decorated with pre- and postconditions; i.e., the outline is of the form

\[
(p) t_0[r_1] \{\text{call}_1(s_1)t_1(r_2) \{\text{call}_2(s_2) \ldots t_n[\text{call}_n(s_n)] \{t_n[q]\}
\]

assuming no method calls in the statements \( t_0 \ldots t_n \). For the different \( t_i \), soundness of \( \text{PL} \) then gives \( \models_D \{p\} t_0 \{r_1\} \models_D \{s_1\} t_1 \{r_{i+1}\} \), and \( \models_D \{s_n\} t_n \{q\} \), for \( 1 \leq i < n \). Each call statement is of the form \( v := n(\pi) \), \( v := n@A(\pi) \), or \( v := e.n(\pi) \).

**Base case:** The execution of \( \text{body}_C(C, m) \) does not lead to any method calls. Then \( \models_D m(\pi) : (p, q) \{\text{body}_C(C, m)\} \) follows by the soundness of \( \text{PL} \).

**Induction step:** For each call to that method which is made by some method \( n \) in the body of \( m \), bound to an implementation \( \text{body}_F(F, n) \) in context \( F' \leq_C F \), we have \( \models_{F'} n(\pi) : (g, h) \{\text{body}_F(F, n)\} \) for each \( (g, h) \in S_{F'}(F, n) \). The different call statements are considered separately.

Consider a method call \( \{r\} v := n(\pi) \{s\} \) in \( O \), and let \( r' \) and \( s' \) be as in Definition 3. By the assumptions of the Lemma, the call is bound to \( \text{body}_C(D, n) \) in the context of class \( D \leq_C C \). For all \( (g, h) \in S_{F'}(D, n) \), we have by the induction
hypothesis that \(\models_D n(\overline{x}) : \{g, h\} \{\text{body}_C(D, n)\}\). By Definition 3, Condition 1, we have \(R_C(B, n) \rightarrow (r', s')\). Then the desired conclusion \(\models_D \{r'\} \text{body}_C(D, n) \{s'\}\) follows since \(S \models_D(D, n) \rightarrow R \models_D(D, n)\) by Definition 3, Condition 2, which especially means \(S \models_D(D, n) \rightarrow R_C(B, n)\) since \(D \subseteq E\) and \(R_C(B, n) \subseteq R \models_D(D, n)\).

Consider a method call \(\{r\} v := n(\overline{x})\) \(\{s\}\) in \(O\) and let \(r', s'\) be as in Definition 3. The call is bound to \(\text{body}_C(A, n)\) in the context \(D\). By the induction hypothesis, we have \(\models_D n(\overline{x}) : \{g, h\} \{\text{body}_C(A, n)\}\) for all \((g, h) \in S \models_D(A, n)\). Then the conclusion \(\models_D \{r'\} \text{body}_C(A, n) \{s'\}\) follows since \(S \models_D(A, n) \rightarrow (r', s')\) by Definition 3, Condition 2.

Consider a method call \(\{r\} v := e.n(\overline{x})\) \(\{s\}\) in \(O\) and let \(r', s'\) be as in Definition 3. From Definition 3, Condition 1, we have \(R \models_D(E, n) \rightarrow (r', s')\). The call can be bound in the context of any class \(E'\) below \(E\). By the definition of \(R \models_D\) in Fig. 5, we have \(R \models_D(E, n) \subseteq R \models_D(E', n)\) for \(E' \subseteq E\). Since \(R \models_D(E, n) \rightarrow (r', s')\), this gives \(R \models_D(E', n) \rightarrow (r', s')\). By Definition 3, Condition 2, we have \(S \models_D(E', n) \rightarrow R \models_D(E', n)\), which especially means \(S \models_D(E', n) \rightarrow (r', s')\). The conclusion \(\models_{E'} \{r'\} \text{body}_C(E', n) \{s'\}\) then follows by the induction hypothesis. □

In a minimal environment \(E\), the mapping \(R_C\) only contains requirements that are caused by some proof outline; i.e., there are no superfluous requirements. Minimal environments are defined as follows:

**Definition 4 (Minimal Environments).** A proof environment \(E\) is minimal iff for all \(C : \text{Cid} \) and \(n : \text{Mid}\) with formal parameters \(\overline{x}\):

\[
(\forall (r', s') \in R_C(C, n) . \exists p, q, r, s, m, O \. (p, q) \in S_C(C, m) \land O \vdash_{PL} \text{body}_C(C, m) : (p, q) \land \{r\} v := n(\overline{x}) \{s\} \in O \land r' = (r \land \overline{x} = v) \land s' = s[\text{return} / v]
\]

4.2. The inference rules of \(LBS(PL)\)

An open program may be extended with new classes, and there may be mutual dependencies between the new classes. For example, a method in a new class \(C\) can call a method in another new class \(E\), and a method in \(E\) can call a method in \(C\). In such cases, a complete analysis of one class cannot be carried out without consideration of mutually dependent classes. We therefore choose modules as the granularity of program analysis, where a module consists of a set of classes. Such a module is self-contained with respect to an environment \(E\) if all method calls inside the module can be successfully bound inside that module or to classes represented in \(E\).

In the calculus, judgments have the form \(E \vdash M\), where \(E\) is the proof environment and \(M\) is a list of analysis operations on the class hierarchy. The syntax for analysis operations is outlined in Fig. 6, and the different operations are explained below. Let \(LBS(PL)\) denote the reasoning system for lazy behavioral subtyping based on a (sound) program logic \(PL\), which uses a proof environment \(E : \text{Env}\) and the inference rules given in Figs. 7 and 8.

There are three different environment updates; the loading of a new class \(L\) into the environment and the extension of the specification and requirement mappings with an assertion pair \((p, q)\) for a given method \(m\) and class \(C\). These are denoted \(\text{extl}(C, B, \overline{f}, \overline{M})\), \(\text{extS}(C, m, (p, q))\) and \(\text{extR}(C, m, (p, q))\), respectively. The inference system below ensures that the same class is never loaded twice into the environment. Environment updates are represented by the operator \(\oplus : \text{Env} \times \text{Update} \rightarrow \text{Env}\), where the first argument is the current proof environment and the second argument is the environment update, defined as follows:

\[
\begin{align*}
E \oplus \text{extl}(C, B, \overline{f}, \overline{M}) & \triangleq (L_C[C \mapsto \{B, \overline{f}, \overline{M}\}], S_C, R_C) \\
E \oplus \text{extS}(C, m, (p, q)) & \triangleq (L_C, S_C[(C, m) \mapsto (S_C(C, m) \cup \{(p, q)\})], R_C) \\
E \oplus \text{extR}(C, m, (p, q)) & \triangleq (L_C, S_C, R_C[(C, m) \mapsto (R_C(C, m) \cup \{(p, q)\})])
\end{align*}
\]

The main inference rules of the assertion calculus are given in Fig. 7. In addition, there are lifting rules concerned with the analysis of set and list structures, and trivial cases, which are given in Fig. 8. Note that \(M\) represents a list of module operations which will be analyzed later and which may be empty. Rule (NewModule) initiates the analysis of a new module \(\text{module}(L)\). The analysis continues by manipulation of the \([e : L]\) operation that is generated by this rule. For notational convenience, we let \(\hat{L}\) denote both a set and list of classes. During the analysis of a module, the proof environment is extended in order to keep track of the currently analyzed class hierarchy and the associated method specifications and requirements.

\[
\begin{align*}
O & ::= e \mid \text{anReq}(\overline{M}) \mid \text{anSpec}(\overline{MS}) \mid \text{verify}(m, \overline{R}) \mid \text{anCalls}(t) \mid O \cdot O \\
L & ::= \emptyset \mid [L \mid \text{require}(C, m, (r, s))] \mid L \cup L \\
M & ::= \text{module}(\hat{L}) \mid [(C : O) \mid L] \mid [e : L] \mid M \cdot \text{module}(\hat{L})
\end{align*}
\]

Fig. 6. Syntax for the analysis operations. Here, \(M, MS, S\) and \(T\) are as in Fig. 1, \(S\) is a set of assertion pairs, and \(t\) is a statement decorated with pre- and post conditions to method calls.
Fig. 7. The inference system, where $\mathcal{M}$ is a (possibly empty) list of analysis operations. In the call rules, we have $r' = (r \land r = \top)$ and $s' = s[\text{return} / v]$, where $\top$ are the formal parameters of $n$ (assuming $r$ and $\text{return}$ do not occur in $r$, $s$, or $\top$).

Rule (NewClass) selects a new class $\text{class } C \text{ extends } B \mid M \bar{M}$ from the current module, and initiates the analysis of the class in the current proof environment. Note that at this point in the analysis, class $C$ has no subclasses in the proof environment. Classes are assumed to be syntactically well-formed and well-typed. The premises of (NewClass) ensure that a class cannot be introduced twice and that the superclass has already been analyzed. The rule generates an operation of the form $[[C : O] \mid \mathcal{L}]$. The syntax for analysis operations $O$ is given in Fig. 6.

By application of (NewClass), the class hierarchy is extended with $C$, and $\mathcal{O}$ consists initially of two operations: $\text{anSpec}(\bar{M})$ and $\text{anReq}(\bar{M})$. These operations may be flattened by the rules (DecompSpec) and (DecompMtds) of Fig. 8, respectively. For each specification $m(\bar{x}) : (p, q)$ in $\bar{M}$, Rule (NewSpec) initiates analysis of $(p, q)$ with regard to the visible implementation of the method by means of a verify$(m, (p, q))$ operation. Furthermore, for each method definition $m(\bar{x}) \{ t \}$ in $\bar{M}$, Rule (NewMtd) generates an operation verify$(m, R^\uparrow(B, m))$. Here, $R^\uparrow(B, m)$ contains the requirements towards $m$ that are imposed by the superclasses of $C$. The requirement set of a verify operation may be decomposed by rule (DecompReq) of Fig. 8. (Fig. 8 also contains rules (NoReq), (NoMtds), and (NoSpec) to discard empty verify, anReq, and anSpec operations, respectively.) The two analysis operations $\text{anSpec}(\bar{M})$ and $\text{anReq}(\bar{M})$ thereby ensure that:
Fig. 8. The inference system: Lifting rules decomposing list-like structures and handling trivial cases. Here $\mathcal{M}$ is a (possibly empty) list of analysis operations.

- For each specification $m(\overline{x}) : (p, q)$ in $\overline{\mathcal{M}}$, an operation $\text{verify}(m, (p, q))$ is generated.
- If $m$ is a method defined in $C$, and some superclass of $C$ imposes the requirement $(r, s)$ on $m$, i.e., $m$ overrides a superclass definition, an operation $\text{verify}(m, (r, s))$ is generated.

The generated $\text{verify}$ operations are analyzed either by Rule (ReqDer) or by Rule (ReqNotDer). For each method $m$, the set $S(C, m)$ is initially empty, and this set is only extended by Rule (ReqNotDer). If an assertion pair $(p, q)$ is included in $S(C, m)$, it might be the case that $(p, q)$ follows by entailment from the already verified assertion pairs.

Now, consider the analysis of some operation $\text{verify}(m, (p_1, q_1))$. Since the set $S(C, m)$ is incrementally extended during analysis of $C$, it might be the case that $(p_1, q_1)$ follows by entailment from the already verified assertion pairs, i.e., $S(C, m) \rightarrow (p_1, q_1)$. In this case, no further analysis of $(p_1, q_1)$ is needed, and the $\text{verify}(m, (p_1, q_1))$ operation is discarded by Rule (ReqDer). Otherwise, a proof outline for $(p_1, q_1)$ is needed. The operation is then verified by Rule (ReqNotDer) as described above. In general, the definition of method $m$ can be inherited by $C$ without redefinition, which means that $(p_1, q_1)$ may follow from already verified assertion pairs in the superclass. In Rule (ReqDer), this is captured by the relation $S^\uparrow(C, m) \rightarrow (p_1, q_1)$. Remember that $S^\uparrow(C, m)$ reduces to $S(C, m)$ if $m$ is defined in $C$ (cf. Fig. 5).
Next we consider the analysis of \texttt{anCalls}($O$) operations generated by \texttt{(ReqNotDER)}, where $O$ is a proof outline for some method body where call statements are decorated with pre- and postconditions. The proof outline is decomposed by the rules \texttt{(DecompSeq)} and \texttt{(DecompIE)} in Fig. 8. Rule \texttt{(Skip)} applies to statements which are irrelevant to the \texttt{anCalls} analysis. Rule \texttt{(IntrCall)} (c.f. Fig. 7) analyzes the requirement of an internal call in the proof outline. The rule extends the $R$ mapping and generates a \texttt{verify} operation which analyzes the requirement with respect to the implementation bound from the current class $C$. The extension of the $R$ mapping ensures that future redefinitions must respect the new requirement; i.e., the requirement is imposed whenever redefinitions are considered by \texttt{(NewMtd)}. Rule \texttt{(StaticCall)} handles external calls by ensuring that the required assertion pair follows from the specification of the called method. Note that this rule does not extend the $R$ mapping since the call is bound at compile time. Rule \texttt{(ExtCall)} handles external calls of the form $v \leftarrow e$.$n$(\texttt{E}). The requirement to the external method is removed from the context of the current class and propagated as a \texttt{require} operation in the module operations $L$. The type of the callee is found by static type analysis of $e$, expressed by the premise $e : E$. Rule \texttt{(ExtReq)} can first be applied after the analysis of the callee class is completed, and the requirement must then follow from the requirements of this class. For simplicity we have here omitted formalization of verification, since this will complicate the system further, and since the next section gives a solution without verification.

By the successful analysis of class $C$, an operation on the form $[\langle C : e \rangle ; L]$ is reached, and by application of Rule \texttt{(EMPClass)}, this yields the operation \texttt{$[e ; L]$}. Another class in $L$ can then be enabled for analysis. The analysis of a module is completed by the rule \texttt{(EMPModule)}. Thus, the analysis of a module is completed after the analysis of all the module classes and external requirements made by these classes have succeeded. Note that a proof of $E \vdash module(L)$ has exactly one leaf node $e' \vdash [e ; \emptyset]$; we call $e'$ the environment resulting from the analysis of $module(L)$.

Program analysis is initiated by the judgment $\varepsilon_0 \vdash module(L)$, where $L$ is a module that is self-contained in the empty environment. Subsequent modules are analyzed in sequential order, such that each module is self-contained with respect to the environment resulting from the analysis of previous modules. When the analysis of a module is completed, the resulting environment represents a verified class hierarchy. New modules may introduce subclasses of classes which have been analyzed in previous modules. The calculus is based on an open world assumption in the sense that a module is analyzed in the context of previously analyzed modules, but it is independent of subsequent modules.

**Example 11.** As an illustration of a derivation in $LBS(PL)$, we consider the analysis of class $B_1$ from Example 9. Thus, we assume that class $A$ has already been analyzed, resulting in the environment $\varepsilon_0$ where

\[
L_{E_0}(A) = (\{nil, \emptyset, n(y)\{\text{return} : 5 \ast y\}\{\text{return} : n(x)\})
\]

and $L_{E_0}$ is undefined for all other classes. For simplicity we here ignore class $\textbf{Object}$ and take $A.inh = \text{nil}$. As explained in Example 9, the following $S$ and $R$ sets are non-empty:

\[
S_{E_0}(A, n) = \{(true, \text{return} = 5 \ast y)\}
\]

\[
S_{E_0}(A, m) = \{(x \geq 0, \text{return} \geq 2 \ast x)\}
\]

\[
R_{E_0}(A, n) = \{(y \geq 0, \text{return} \geq 2 \ast x)\}
\]

The analysis of class $B_1$ is initiated by the judgment

\[
\varepsilon_0 \vdash module(\text{class } B_1 \text{ extends } A \{M,M\})
\]

where

\[
M = n(y)\{\text{return} : 2 \ast y\} \quad Sn = n(y) : (true, \text{return} = 2 \ast y)
\]

\[
M\bar{S} = \{Sn Sm\} \quad Sm = m(x) : (true, \text{return} = 2 \ast x)
\]

The successful derivation of this judgment is given in Fig. 9, leading to the resulting environment $\varepsilon_4$ shown in the figure.

### 4.3. Properties of $LBS(PL)$

Although the individual rules of the inference system do not preserve soundness of the proof environment, the soundness of the proof environment is preserved by the successful analysis of a module. This allows us to prove that the proof system is sound for module analysis.

**Lemma 2.** Let $\varepsilon$ be an environment such that for all class names $B$ and method names $m$, the following holds: if $B \in \varepsilon$ then $S^*_e(B, m) \rightarrow R^*_e(B, m)$ and $L_e(B).inh \neq \text{nil}$ $\Rightarrow$ $L_e(B).inh \in \varepsilon$. Otherwise, if $B \notin \varepsilon$, then $S(B, m) = R(B, m) = \emptyset$.
Let $L \triangleq \text{class } C \text{ extends } A \{\overline{f} \overline{M} \overline{MS}\}$ be a class definition such that $C \notin \mathcal{E}$. Let $\mathcal{E} \vdash [\varepsilon : L \cup C]$ be the judgment under evaluation by LBS(PL). Assume that $C$ is loaded for analysis and that the analysis of $C$ succeeds, leading to the judgment $\mathcal{E}' \vdash [(C : \varepsilon) : \mathcal{L}']$. Then the following properties hold for $\mathcal{E}'$ and $\mathcal{L}'$:

1. $L_{\mathcal{E}'} = L_{\mathcal{E}}[\overline{C} \mapsto (A, \overline{f}, \overline{M})].$
2. $A \neq \text{nil} \Rightarrow A \in \mathcal{E}.$
3. For all $B \in \mathcal{E}$ and method name $m$, we have $S_{\mathcal{E}'}(B, m) = S_{\mathcal{E}}(B, m)$ and $R_{\mathcal{E}'}(B, m) = R_{\mathcal{E}}(B, m)$. For any class $B$ such that $B \notin \mathcal{E}'$, we have $S_{\mathcal{E}'}(B, m) = R_{\mathcal{E}'}(B, m) = \emptyset$ for all method names $m$.
4. $\mathcal{L} \subseteq \mathcal{L}'$.
5. For each $(p, q) \in S_{\mathcal{E}'}(C, m)$ for some method name $m$, there is a proof outline $O$ such that $O \vdash_{PL} \text{body}_{\mathcal{E}'}(C, m) : (p, q)$.
6. $S'_{\mathcal{E}'}(C, m) = R'_{\mathcal{E}'}(C, m)$ for all $m$.

Proof. By rule (NewClass), the judgment under consideration leads to $\mathcal{E} \oplus \text{extL}(C, A, \overline{f}, \overline{M}) \vdash [(C : \text{anspec(MS)} \cdot \text{anReq(M)}) : \mathcal{L}].$ The inference rules manipulate this judgment until $\mathcal{E}' \vdash [(C : \varepsilon) : \mathcal{L}']$ is reached. Note that none of the rules (NewModule), (NewClass), (ExtReq), (EmpClass), and (EmpModule) can be applied during this manipulation.

Condition 1. The only rule that extends the $L$ mapping is (NewClass). This rule is applied when the class is loaded, and the condition follows from the premise of this rule.

Condition 2. Again, this follows from the premise of (NewClass).

Condition 3. This is proved by induction over the inference rules. None of the rules remove information from the environment, and the only sets that are extended are the $S$ and $R$ sets of class $C$. Thus, for all $B \neq C$ and methods $m$, we have $S_{\mathcal{E}'}(B, m) = S_{\mathcal{E}}(B, m)$ and $R_{\mathcal{E}'}(B, m) = R_{\mathcal{E}}(B, m)$. Especially, this holds for all classes defined in $\mathcal{E}$ as required by the first part of Condition 3.

Furthermore, if $B \notin \mathcal{E}'$, we know from Condition 1 that $B \neq C$ and $B \notin \mathcal{E}$. The conclusion $S_{\mathcal{E}'}(B, m) = R_{\mathcal{E}'}(B, m) = \emptyset$ then follows by the above paragraph and the assumption $S_{\mathcal{E}}(B, m) = R_{\mathcal{E}}(B, m) = \emptyset$ of the lemma.

Condition 4. By induction over the inference rules. During analysis of $C$, no elements are removed from $\mathcal{L}$.

Condition 5. By induction over the inference rules. Initially, the mapping $S_{\mathcal{E}}(C, m)$ is empty, thus for each $(p, q) \in S_{\mathcal{E}}(C, m)$, rule (ReqNotDer) must have been applied. This rule ensures the existence of $O \vdash_{PL} \text{body}_{\mathcal{E}'}(C, m) : (p, q)$. For each $\{r\} \vdash
n(\mathcal{E}) \{s\} in O, Rule (\text{IntCall}) is applied, ensuring \text{R}_{c}(C, n)\rightarrow(r', s'). For each \{r\} v := n@G(\mathcal{E}) \{s\} in O, Rule (\text{StaticCall}) ensures \text{S}_{c}(G, n)\rightarrow(r', s'). For each \{r\} v := e.n(\mathcal{E}) \{s\} in O, Rule (\text{ExtCall}) ensures require(D, n, (r', s')) \in (L' \setminus L) for e : D.

Condition 6. The mapping \text{R}_{c}(C, m) is initially empty. Thus, if (r, s) \in \text{R}_{c}(C, m), rule (\text{IntCall}) must have been applied during the analysis of C. For each such (r, s), this rule leads to an operation \text{verify}(m, (r, s)). This operation either succeeds by Rule (\text{ReqDir}) or (\text{ReqNotDir}). If (\text{ReqDir}) is applied \text{S}_{c}(C, m)\rightarrow(r', s') must hold. Otherwise, Rule (\text{ReqNotDir}) ensures \text{S}_{c}(C, m)\rightarrow(r, s). Combined, this gives \text{S}_{c}(C, m)\rightarrow(r, s) for each (r, s) \in \text{R}_{c}(C, m), i.e., \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(C, m).

Consider next the requirements inherited by C. If A = nil (where A = L_{c}(C).\text{inlh}, the desired \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(C, m) follows directly by the definition of R \uparrow in \mathcal{E} in Fig. 5. Otherwise, if A \neq nil, we must prove that \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(C, m). By A \neq nil, we know that A \in \mathcal{E} and that \text{S}_{c}(A, m)\rightarrow\text{R}_{c}(A, m) by Conditions 2 and 3 above. If m \notin L_{c}(C).\text{mdts} then \text{S}_{c}(A, m)\subseteq \text{S}_{c}(C, m), which gives \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(A, m). Otherwise, if m \in L_{c}(C).\text{mdts}, the method is analyzed by Rule (\text{NewMiss}), leading to a verify operation on each requirement in \text{R}_{c}(A, m). The analysis of these verify operations ensures \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(A, m). Consequently, we have \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(A, m) also in this case since \text{S}_{c}(C, m) = \text{S}_{c}(C, m).

\begin{theorem}
Let \mathcal{E} be a sound environment and \mathcal{T} a set of class declarations. If a proof of \mathcal{E} \vdash \text{module}(\mathcal{T}) in LBS(PL) has \mathcal{E}' as its resulting proof environment, then \mathcal{E}' is also sound.
\end{theorem}

\begin{proof}
By Rule (\text{NewModule}), the judgment \mathcal{E} \vdash \text{module}(\mathcal{T}) evaluates to \mathcal{E} \vdash [\mathcal{E} ; \mathcal{T}]. The analysis continues by manipulation of this judgment until the judgment \mathcal{E} \vdash \mathcal{E}' is reached.

Consider some class C already defined in the initial environment \mathcal{E}. Since \mathcal{E} is sound, Condition 3 of Lemma 2 ensures that the analysis performed for C remains sound during analysis of the classes in \mathcal{T}.

Let \mathcal{E}'' be the environment in which some class C in \mathcal{T} is analyzed, i.e., the judgment \mathcal{E}'' \vdash \{\text{class C extends A [\mathcal{MM}]}} \bigcup \mathcal{E}''' is manipulated in LBS(PL), and let \mathcal{E}'''' be the environment immediately after analysis of this class, i.e., the judgment \mathcal{E}'''' \vdash [(\mathcal{C} : \mathcal{E}) \in \mathcal{L}].

From Lemma 2, we know that if \text{S}_{c}(B, m)\rightarrow\text{R}_{c}(B, m) for all classes B where B \neq C, then \text{S}_{c}(B, m)\rightarrow\text{R}_{c}(B, m), and that \text{S}_{c}(C, m)\rightarrow\text{R}_{c}(C, m) is established. Thus, if Condition 2 of Definition 3 holds in \mathcal{E}'', then it also holds in \mathcal{E}'''.

As none of the rules (\text{ExrReq}) and (\text{EmrClass}) extends the environment, we may conclude that Condition 2 of Definition 3 holds in the resulting environment \mathcal{E}'''.

Consider next Condition 1 of Definition 3 for class C defined in \mathcal{T}. By Lemma 2, Condition 5, we know that for each (p, q) \in \text{S}_{c}(C, m) there is a proof outline O such that O \vdash_{PL} \text{body}_{c}(C, m) and that for each \{r\} v := n(\mathcal{E}) \{s\} in O, we have \text{R}_{c}(C, n)\rightarrow(r', s'). For each \{r\} v := n@G(\mathcal{E}) \{s\} in O, we have \text{S}_{c}(G, n)\rightarrow(r', s'). For each \{r\} v := e.n(\mathcal{E}) \{s\} in O, we have require(D, n, (r', s')) \in (L'' \setminus L'') for e : D. Thus, Condition 1 of Definition 3 is ensured for \mathcal{E}'''' except that external require operations have not been verified. By Lemma 2, Condition 3, we have (p, q) \in \text{S}_{c}(C, m) and \text{R}_{c}(C, n)\rightarrow(r', s') for each \{r\} v := n(\mathcal{E}) \{s\} in O also for the resulting environment \mathcal{E}'. Furthermore, since the analysis of each require(D, n, (r', s')) operation succeed, Rule (\text{ExrReq}) is applied in some environment \mathcal{F}, where \mathcal{F} either occurs between analysis of two classes, or \mathcal{F} is the resulting environment \mathcal{E}'. In either case, Rule (\text{ExrReq}) ensures D \in \mathcal{F} and \text{R}_{c}(D, m)\rightarrow(r', s') for each \{r\} v := e.n(\mathcal{E}) \{s\}, e : D, in O. If subsequent classes are analyzed, we have \text{R}_{c}(D, m)\rightarrow(r', s') by Lemma 2, Condition 3. Thereby, Condition 1 of Definition 3 is established for \mathcal{E}'.

Since the initial environment \mathcal{E} is sound, Condition 3 of Definition 3 holds in \mathcal{E}. By Condition 2 and 3 of Lemma 2, this proof condition for sound environments is maintained by analysis of each class in \mathcal{T}, which means that also the last condition for sound environments holds for the resulting environment \mathcal{E}'.

\begin{theorem}[Soundness]
If PL is a sound program logic, then LBS(PL) constitutes a sound proof system.
\end{theorem}

\begin{proof}
It follows directly from the definition of sound environments that the empty environment is sound. Theorem 1 and Lemma 1 guarantee that the environment remains sound during the analysis of class modules.

Furthermore, the inference system preserves minimality of proof environments; i.e., only requirements needed by some proof outline are recorded in the \text{R}_{c} mapping.

\begin{lemma}
If \mathcal{E} is a minimal environment and \mathcal{T} is a set of class declarations such that a proof of \mathcal{E} \vdash \text{module}(\mathcal{T}) leads to the resulting environment \mathcal{E}'', then \mathcal{E}' is also minimal.
\end{lemma}

\begin{proof}
By induction over the inference rules. For a class C and method m, the rule (\text{IntCall}) is the only rule that extends \text{R}_{c}(C, m). In order for the rule to be applied, an operation anCalls(\{r\} v := m(\mathcal{E}) \{s\}) must be analyzed in the context of C for some requirement (r, s) to m. This operation can only have been generated by an application of (\text{ReqNotDir}), which guarantees that the requirement is needed by some analyzed proof outline.

Finally we show that the proof system supports verification reuse in the sense that specifications are remembered.
Lemma 4. Let $E$ be an environment and $\mathcal{L}$ a list of class declarations. Whenever a proof outline $O$ such that $O \vdash_{PL} \text{body}_{\mathcal{L}}(C, m) : (p, q)$ is verified during analysis of some class $C$ in $\mathcal{L}$, the specification $(p, q)$ is included in $S_E(C, m)$.

Proof. By induction over the inference rules. The only rule requiring the verification of a proof outline is $(\text{ReqNotDer})$, so it suffices to consider this rule. From the premises of $(\text{ReqNotDer})$ it follows that $S_E(C, m)$ is extended with $(p, q)$ whenever $O \vdash_{PL} \text{body}_{\mathcal{L}}(C, m) : (p, q)$ is verified in $PL$. □

5. External specification by interfaces

In the approach presented so far, each class $C$ provides some specifications of the available methods, inherited or defined, in the form of assertion pairs. These are kept in the $S$ part of the proof environments. Their verification generates $R$ requirements for the late bound internal calls occurring in the class, which are imposed on subclass redefinitions of the called methods. In a subclass, redefined methods are allowed to violate the $S$ specifications of a superclass, but not the $R$ requirements.

A weakness of $LBS(PL)$ concerns the treatment of external calls (as opposed to internal calls): When reasoning about $e.m(\tau)$ with $e : E$, the pre/post assertion of the call must follow from the $R$ requirements to $m$ that have been established for class $E$, or otherwise by adding the corresponding requirement to $E$ and verifying that it holds for that class and any subclasses. Thus, class $E$ and any subclasses may need to be analyzed again with respect to the new requirement. As $R$ requirements generated from internal calls may not in general provide suitable external properties, as illustrated by the next example, reverification will be needed.

Example 12. Reconsider the class $A$ from Example 9:

```java
class A {
  int n(int y) : (true, return = 5 * y) { return := 5*y }
  int m(int x) : (x ≥ 0, return ≥ 2 * x) { return := n(x) }
}
```

As explained in Example 9, the specification and requirement sets are built as follows during the analysis of $A$:

\[
S(A, n) = \{(true, return = 5 \ast y)\}
\]

\[
S(A, m) = \{(x ≥ 0, return ≥ 2 \ast x)\}
\]

\[
R(A, n) = \{(y ≥ 0, return ≥ 2 \ast y)\}
\]

Note that the internal call to method $n$ gave a requirement in $R(A, n)$, whereas no requirements are recorded for method $m$ since it is not called internally. Consider next the following client code:

```java
class Client {
  A a := new A;
  int d1() : (true, return ≥ 10) { return := a.n(5) }
  int d2() : (true, return ≥ 10) { return := a.m(5) }
}
```

The analysis of the specification of method $d1$ leads to a requirement on the call to $a.n$: $(y = 5, return ≥ 10)$. By Rule $(\text{ExtCall})$, this requirement generates an operation $\text{require}(A, n, (y = 5, return ≥ 10))$, since $A$ is the type of $a$. Correspondingly, analysis of the external call in $d2$ gives an operation $\text{require}(A, m, (x = 5, return ≥ 10))$. In this manner, the analysis of $Client$ generates two require operations. Rule $(\text{ExtReq})$ is the only rule that applies to these operations.

For the operation $\text{require}(A, n, (y = 5, return ≥ 10))$, Rule $(\text{ExtReq})$ requires that the relation $R(A, n) \rightarrow (y = 5, return ≥ 10)$ must hold, which follows by the definition of entailment since the following implication holds:

\[
(y = 5 \land (y ≥ 0 \Rightarrow return ≥ 2 \ast y)) \Rightarrow return ≥ 10
\]

Since the requirement follows from $R(A, n)$, it is guaranteed to hold also if the call binds to an instance of a subclass of $A$, such as class $B_1$ in Example 9.

For the operation $\text{require}(A, m, (x = 5, return ≥ 10))$ however, the relation $R(A, m) \rightarrow (x = 5, return ≥ 10)$ does not hold, since $R(A, m)$ is empty. Therefore, the analysis of the specification for $d2$ requires reverification of $A$ and of any subclasses of $A$.

The situation illustrated in Example 12 is not desirable since previously analyzed classes must be analyzed again.

In this section we use behavioral interfaces as a means to specify and reason about requirements on external method calls [13,51]. A behavioral interface describes the visible methods of a class and their specifications, and inheritance may be used to form new interfaces from old ones. These behavioral interfaces are used to type object variables (references), and subtyping follows the interface inheritance hierarchy. A class definition explicitly declares which interface it implements.

Fig. 10. Syntax for the language IOOL, extending the syntactic category $I$ of OOL (see Fig. 1) with interfaces. Types now range over interface names instead of class names. The other syntactic categories of Fig. 1 remain unchanged. Here, $I$ denotes interface names of type $Iid$.

(For simplicity we consider at most one interface per class.) This allows the inheritance hierarchies of interfaces and classes to be kept distinct. Static type checking of an assignment $v ::= e$ must then ensure that the expression $e$ denotes an object supporting the declared interface of the object variable $v$. In this setting, the substitution principle for objects can be reformulated as follows: For an object variable $v$ with declared interface $I$, the actual object referred to by $v$ at run-time will satisfy the behavioral specification $I$. Reasoning about an external call $e.m(E)$ can then be done by relying on the behavioral interface of the object expression $e$, simplifying the (ExtCall) rule presented above to simply check interface requirements. In this way, require operations are no longer needed in the proof system.

In Section 5.1, we define the programming language IOOL, which extends OOL with interfaces. In Section 5.2 we define proof environments of type $IEnv$ where interface information is accounted for, and in Section 5.3 we define the calculus $LBSI(PL)$ for reasoning about IOOL programs.

5.1. Behavioral interfaces

Let the programming language IOOL extend OOL with behavioral interfaces. In the syntax for IOOL, given in Fig. 10, classes are modified such that a class implements a single interface. Note that the types of variables and methods no longer range over class names, as object references are typed by interface names. A behavioral interface $I$ may extend a list $L$ of superinterfaces, and consists of a set $MS$ of method names with signatures and semantic constraints on the use of these methods. The constraints are given as (pre, post) specifications for the methods. An interface may declare signatures of new methods not found in its superinterfaces, and it may declare additional specifications of methods declared in the superinterfaces. The superinterface relationship between interfaces is restricted to a form of behavioral subtyping. Consequently, an interface may not declare method specifications that are in conflict with the specifications declared by the superinterfaces. In the sequel, it is assumed that the interface hierarchy conforms to these requirements. The interfaces thus form a type hierarchy: if $I'$ extends $I$, then $I'$ is a subtype of $I$ and $I$ is a supertype of $I'$. Let $\leq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces under the assumption above. Thus, $I' \leq I$ if $I'$ equals $I$ or if $I'$ (directly or indirectly) extends $I$. An interface $I$ exports the methods declared in $I$ or in the superinterfaces of $I$, with the associated constraints (or requirements) on method use.

A class $C$ implements $I$ if it has an $I$ in the class definition and all methods exported by $I$ are defined, satisfying the constraints of $I$. The analysis of the class must ensure that this requirement holds. Observe that only the methods exported by $I$ are available for external invocations on references typed by $I$. The class may implement additional auxiliary methods for internal use. Inside a class the type of $\text{this}$ is the interface implemented by the class. By type safety, external calls which bind to $\text{this}$ can be assumed to be safe also when the calls are executed on an instance of a subclass of $C$. An instance of $C$ is said to support $I$ and all superinterfaces of $I$; thus ensuring that the object provides the methods exported by $I$ and adheres to the specifications imposed by $I$ on these methods. Objects of different classes may support the same interface, corresponding to different implementations of the interface behavior. If an object supports $I$ (or a subtype of $I$) then the object may be referenced by a variable typed by $I$. The separation of class and inheritance hierarchies means that a subclass $D$ of $C$ need not implement (a subtype of) the interface $I$ implemented by $C$ [3,30]; if $D$ implements $J$, then $J$ need not be a subtype of $I$. In this case, the subclass may freely reuse and redefine superclass methods without adhering to the behavioral constraints imposed by $I$, and instances of $D$ will not behave as subtypes of $I$.

Example 13. Let $A$ be a class implementing an interface $I$ as depicted in Fig. 11, thus instances of $A$ support $I$. A variable $x$ declared with type $I$ (i.e., $x : I$) may refer to an instance of $A$. A subclass $B$ of $A$ may reuse the code of $A$ without implementing the interface $I$; i.e., $B$ may extend $A$ but implement a different interface $J$ where $J$ is not a subtype if $I$. In that case, the substitution principle will ensure that $x$ never refers to an instance of $B$. Given a third class $C$ implementing $I'$, where $I'$ is a subinterface of $I$, the variable $x$ may refer to an instance of $C$ since $C$ implements the subinterface $I'$ of $I$. Assuming that these are the only classes and interfaces in the system, an assignment $x ::= e$ is type safe if $e$ denotes an expression of either type $I$ or type $I'$. When reasoning about an external method call $x.m()$, we can rely on the behavioral constraints of $m$ given by the type $I$ of $x$.

A variable $y : J$ may refer to an instance of $B$, but the substitution principle prohibits $y$ from referring to an instance of $A$ or $C$; e.g., the assignment $y ::= x$ is not type safe. Correspondingly, also the assignment $x ::= y$ is not type safe. Thus, the substitution principle applies to the supported interface of an object, ensuring that a subclass instance cannot be accessed through the type of a superclass unless the subclass explicitly implements the superclass type.
5.2. A proof environment with interfaces

As before, classes are analyzed in the context of a proof environment. Let $\text{Interface}$ denote interface tuples $(I, \mathcal{MS})$, and $\text{IClass}$ denote class tuples $(B, I, f, M)$. Assuming type safety, we ignore the types of fields and methods. The list of superinterfaces $I$ and method specifications $\mathcal{MS}$ of an interface tuple are accessible by the observer functions $\text{inh}$ and $\text{mtds}$, respectively. The supported interface of a class is accessible by the observer function $\text{impl}$. Environments of type $\text{IEnv}$ are defined as follows.

**Definition 5 (Proof environments with interfaces).** A proof environment $\mathcal{E}$ of type $\text{IEnv}$ is a tuple $\langle L_\mathcal{E}, K_\mathcal{E}, S_\mathcal{E}, R_\mathcal{E} \rangle$ where $L_\mathcal{E} : \text{Cid} \rightarrow \text{IClass}$, $K_\mathcal{E} : \text{Iid} \rightarrow \text{Interface}$ are partial mappings and $S_\mathcal{E}, R_\mathcal{E} : \text{Cid} \times \text{Mid} \rightarrow \text{Set}\![\text{APair}]$ are total mappings.

For an interface $I$, let $I \in \mathcal{E}$ denote that $K_\mathcal{E}(I)$ is defined, and let $\text{public}(I)$ denote the set of method names exported by $I$; thus, $m \in \text{public}(I)$ if $m$ is declared by $I$ or by a supertype of $I$. A subtype cannot remove methods declared by a supertype, so $\text{public}(I) \subseteq \text{public}(I')$ if $I' \preceq I$. If $m \in \text{public}(I)$, the function $\text{spec}(I, m)$ returns a set of type $\text{Set}\![\text{APair}]$ with the behavioral constraints imposed on $m$ by $I$, as declared in $I$ or in a supertype of $I$. The function $\text{spec}$ returns a set as a subinterface may provide additional specifications of methods inherited from superinterfaces; if $m \in \text{public}(I)$ and $I' \preceq I$, then $\text{spec}(I, m) \subseteq \text{spec}(I', m)$. These functions are defined in Fig. 12. The superinterface name may be $\text{nil}$, representing no interface.

The definition of sound environments is revised to account for interfaces. In Condition 1, the requirement to an external call must now follow from the interface specification of the called object. Consider a requirement stemming from the analysis of an external call $e.m(\overline{x})$ in some proof outline, where $e : I$. As the interface hides the actual class of the object referenced

\[
\begin{align*}
\text{mids}(\emptyset) & \triangleq \emptyset \\
\text{mids}(m(\overline{x})) & \triangleq \{m\} \cup \text{mids}(\overline{\mathcal{MS}}) \\
\text{public}_\mathcal{E}(\text{nil}) & \triangleq \emptyset \\
\text{public}_\mathcal{E}(I) & \triangleq \text{mids}(\text{K}_\mathcal{E}(I).\text{mtds}) \cup \text{public}_\mathcal{E}(\text{K}_\mathcal{E}(I).\text{inh}) \\
\text{public}_\mathcal{E}(I \setminus I') & \triangleq \text{public}_\mathcal{E}(I) \cup \text{public}_\mathcal{E}(I') \\
\text{spec}(\emptyset, m) & \triangleq \emptyset \\
\text{spec}(n(\overline{x}))(p, q) & \triangleq \text{if } n = m \text{ then } \{(p, q)\} \cup \text{spec}(\overline{\mathcal{MS}}, m) \text{ else } \text{spec}(\overline{\mathcal{MS}}, m) \text{ fi} \\
\text{spec}_\mathcal{E}(\text{nil}, m) & \triangleq \emptyset \\
\text{spec}_\mathcal{E}(I, m) & \triangleq \text{spec}(\text{K}_\mathcal{E}(I).\text{mtds}, m) \cup \text{spec}_\mathcal{E}(\text{K}_\mathcal{E}(I).\text{inh}, m) \\
\text{spec}_\mathcal{E}(I \setminus I', m) & \triangleq \text{spec}_\mathcal{E}(I, m) \cup \text{spec}_\mathcal{E}(I', m) \\
\text{nil} \preceq_\mathcal{E} I & \triangleq \text{false} \\
I \preceq_\mathcal{E} I' & \triangleq I = I \setminus \text{K}_\mathcal{E}(I).\text{inh} \preceq_\mathcal{E} I \\
(I \setminus I') \preceq_\mathcal{E} I & \triangleq I \preceq_\mathcal{E} I \setminus I' \preceq_\mathcal{E} I
\end{align*}
\]

Fig. 12. Auxiliary function definitions, using space as the list separator.
by e, the call is analyzed based on the interface specification of m. A requirement (r, s) must follow from the specification of m given by type I, expressed by spec(I, m) → (r, s). Furthermore, a new condition of sound environment is introduced, expressing that a class satisfies the specifications of the implemented interface. If C implements an interface I, the class defines (or inherits) an implementation of each m ∈ public(I). For each such method, the behavioral specification declared by I must follow from the method specification in the class; i.e., S^I[C, m] → spec(I, m).

**Definition 6 (Sound environments).** A proof environment E of type IEnv is sound if it satisfies the following conditions for each C : Cid and m : Mid.

1. ∀(p, q) ∈ S^C(C, m). ∃O. O ⊢_R body^C(C, m) : (p, q)
   \( \land \forall[r] v := n(\tau) \{ s \} \in O. R^C(C, n) \rightarrow (r', s') \)
   \( \land \forall[r] v := n@A(\tau) \{ s \} \in O. S^A(n) \rightarrow (r', s') \)
   \( \land \forall[r] v := e.\text{spec}(C, n) \{ s \} \in O. e : I \Rightarrow \text{spec}(I, n) \rightarrow (r', s') \)

2. S^I[E(C, m)] → R^I[E(C, m)]

3. ∀n ∈ public^E(I). S^E(C, n) → spec^E(I, n), where I = L_C(C).impl

4. L_C(C).inh ≠ nil ⇒ L_C(C).inh ∈ E
   \( \land \forall B. B \notin E \cdot S_B(B, m) = R_B(B, m) = \emptyset. \)

where r’ = (r ∧ x = ℰ), s’ = s[return/v]. x are the formal parameters of n, and I : lid.

**Lemma 1.** is adapted to the setting of interfaces as follows:

**Lemma 5.** Given a sound environment E : IEnv and a sound program logic PL. For all classes C : Cid, methods m : Mid, and assertion pairs (p, q) : APair such that C ∈ E and (p, q) ∈ S^C(C, m), we have |=_D m(X) : (p, q) (body^C(C, m)) for each D ≤_E C.

**Proof.** The proof is similar to the proof for Lemma 1, except for the treatment of external calls in the induction step. Let O be a proof outline such that O ⊢_R body^E(B, m) : (p, q), where C ≤_E B, bind^E(C, m) = bind^E(B, m), and (p, q) ∈ S_B(B, m). Assume as the induction hypothesis that for any external call to n in O, possibly bound in context E and for all (g, h) ∈ S^E[E(E, n), that |=_E n(X) : (g, h) (body^E(E, n)).

Consider a method call [r] v := e.\text{spec}(C, n) \{ s \} in O. Let r’, s’ be as in Definition 6, and e : I. From Definition 6, Condition 1, we have spec^E(I, n) → (r’, s’). Consider some class E where L_C(E).impl = I. From Definition 6, Condition 3, we have S^E[E(E, n) → spec^E(I, n)]. If the call to n can bind in context E, then type safety ensures j ≤_E I, giving spec^E(I, n) ≤ spec^E(E, n). We then have S^E[E(E, n) → spec^E(I, n) → spec^E(I, n) → (r’, s’)]. By the induction hypothesis, we then arrive at |=_E [r’] body^E(E, n) \{ s’ \}. □

We define an operation to update a proof environment with a new interface, and redefine the operation for loading a new class:

\[
E \oplus \text{extL}(C, B, I, \bar{I}, \bar{M}) \triangleq \langle L_E[C \mapsto \langle B, I, \bar{I}, \bar{M} \rangle], K_E, S_E, R_E \rangle
\]

\[
E \oplus \text{extK}(I, \bar{I}, \bar{M}) \triangleq \langle L_E, K_E[I \mapsto \langle \bar{I}, \bar{M} \rangle], S_E, R_E \rangle
\]

### 5.3. The calculus LBSI(PL) for lazy behavioral subtyping with interfaces

In the calculus for lazy behavioral subtyping with interfaces, judgments have the form E ⊢ M, where E is the proof environment and M is a sequence of interfaces and classes. As before, we assume that superclasses appear before subclasses. This ordering ensures that requirements imposed by superclasses are verified in an incremental manner on subclass overriding. Furthermore, we assume that an interface appears before it is used. More precisely we assume that whenever a class is analyzed, the supported interface is already part of the environment, and for each external call statement v := e.\text{spec}(C, n) in the class where e : I, the interface I is in the environment. These assumptions ensure that the analysis of a class will not be blocked due to a missing superclass or interface.

As the requirements of external calls are now verified against the interface specifications of the called methods, a complete analysis of a class C can be performed based on the knowledge of its superclasses only; other classes need not be considered in order to analyze C. For the revised calculus, it therefore suffices to consider individual classes and interfaces as the granularity of program analysis. The module layer of Section 4 is therefore omitted. The syntax for analysis operations is given by:

\[
M ::= \mathcal{P} | \langle C : \mathcal{O} \rangle \cdot \mathcal{P} \\
\mathcal{O} ::= \epsilon | \text{anReq}(\bar{M}) | \text{anSpec}(\bar{M}) | \text{anCalls}(t) \\
\mathcal{P} ::= K | L | \mathcal{P} \cdot \mathcal{P} | \text{verify}(m, \bar{R}) | \text{intSpec}(\bar{M}) | \mathcal{O} \cdot \mathcal{O}
\]
Fig. 13. The extension of the inference system, where $\mathcal{P}$ is a (possibly empty) sequence of classes and interfaces. Rules (NewClass$'$) and (ExtCall$'$) replace (NewClass) and (ExtCall). The three other rules, concerning interfaces, are new. In Rule (ExtCall$'$), we have $r' = (r \land \pi = \pi)$ and $s' = \{\text{return} / v\}$, where $\pi$ are the formal parameters of $m$.

The new operation $\text{intSpec}(M)$ is used to analyze the interface specifications of methods $M$ with regard to implementations found in the considered class.

For IIOOL, we define a calculus $\text{LBSI}(\mathcal{P}L)$, consisting of a (sound) program logic $\mathcal{P}L$, a proof environment $\mathcal{E}:IEnv$, and the inference rules listed in Fig. 13. In addition to the rules in Fig. 13, $\text{LBSI}(\mathcal{P}L)$ contains the rules in Fig. 7 and Fig. 8, except the rules (NewClass), (ExtCall), (NewModule), (EmpModule) and (ExtReq). Rules (NewClass) and (ExtCall) are renewed as shown in Fig. 13, and rule (ExtReq) is superfluous as the requirements from external calls are analyzed in terms of interface specifications. Rules (NewModule) and (EmpModule) are not needed as modules are removed. For the remaining rules in Fig. 7 and Fig. 8, we assume that module operations are removed as illustrated by (NewClass$'$) and (ExtCall$'$).

Focusing on the changes from Fig. 7 and Fig. 8, the calculus rules are outlined in Fig. 13. Rule (NewINT) extends the environment with a new interface. No analysis of the interface is needed at this point, the specifications of the interface will later be analyzed with regard to each class that implements the interface. (Recall that interfaces are assumed to appear in the sequence $\mathcal{P}$ before they are used.) The rule (NewClass$'$) is similar to the rule from $\text{LBSI}(\mathcal{P}L)$, except that an operation intSpec is introduced which is used to analyze the specifications of the implemented interface. Rule (ExtCall$'$) handles the analysis of external calls; here, the requirement of the call is analyzed with regard to the interface specification of the callee. Rule (IntSpec) is used to verify interface specifications, and rule (DecompInt) is used to flatten the argument of intSpec operations.

In $\text{LBSI}(\mathcal{P}L)$, the different method specifications play a more active role when analyzing classes. Method specifications are used to establish interface properties, which again are used during the analysis of external calls. Thus, requirements to external calls are no longer analyzed based on knowledge from the $R$ mapping of the callee. The $R$ mapping is only used during the analysis of internal calls.

Soundness. For soundness of $\text{LBSI}(\mathcal{P}L)$, Theorem 1 is modified as follows.

**Theorem 3.** Let $\mathcal{P}L$ be a sound program logic, $\mathcal{E}:IEnv$ a sound environment, and $KL$ be an interface or a class definition. If a proof of $\mathcal{E} \vdash KL$ in $\text{LBSI}(\mathcal{P}L)$ has $\mathcal{E}'$ as its resulting proof environment, then $\mathcal{E}'$ is also sound.

**Proof.** The analysis of a new interface maintains soundness as interfaces are assumed to be loaded in the environment before they are used. Consider analysis of the judgment

$$\mathcal{E} \vdash (\text{class } C \text{ extends } A \text{ implements } I \{J \overline{M} \overline{MS}\})$$

The analysis of $C$ succeeds, leading to the judgment $\mathcal{E}' \vdash (C : \epsilon)$ where the operation $(C : \epsilon)$ is discarded by Rule (EmpClass), yielding the resulting environment $\mathcal{E}'$. 
Since ε′ is the environment resulting by analyzing C in the initial environment ε, we have L_C = L_C[C \mapsto \{A, I, \bar{f}, \bar{M}\}], and for all B ∈ E and methods m that S_C(B, m) = S_C(B, m) and R_C(B, m) = R_C(B, m). Especially, for B ∈ E and for each (p, q) ∈ S_C(B, m), we know that Condition 1 of Definition 6 holds also for ε′. Furthermore, since the analysis of C does not modify the K mapping, we have public_C(I) = public_C(I).

For the analysis of a class C, we consider each condition of Definition 6 by itself.

Condition 1 of Definition 6 applies to each element (p, q) of S_C(C, m). The proof is by induction over the inference rules, and it suffices to consider rule (ReqNotDer) which is the only rule that extends the S mapping. If (p, q) ∈ S_C(C, m), this rule ensures the existence of a proof outline O such that O ⊢_{\text{pre}} body_C(C, m) : (p, q). The analysis then continues with an \text{anCalls}(O) operation. For each decorated late bound internal call \{r\} v := n@G\{s\} in O, rule (IntCall) ensures R_C(C, n) \rightarrow TRIANGLE \{r′, s′\} as required by Definition 6. For each static call \{r\} v := e.n\{s\} in O where e : I, rule (ExtCall′) ensures spec_C(I, n) \rightarrow TRIANGLE \{r′, s′\} as required by Definition 6. For each external call \{r\} v := e.n\{s\} in O where e : I, rule (ExtCall′) ensures spec_C(I, n) \rightarrow TRIANGLE \{r′, s′\} as required by Definition 6.

Consider next Condition 2 of Definition 6. Since ε′ is sound, we may assume S↑_{\text{exec}}(B, m) \rightarrow R↑_{\text{exec}}(B, m) for any B ∈ E and method m. By the above discussion, we then have S↑_{\text{exec}}(B, m) \rightarrow R↑_{\text{exec}}(B, m). Consider first the requirements in R_C(C, m) for some method m. For each (r, s) ∈ R_C(C, m), Rule (IntCall) must have been applied during the analysis of C, generating an operation verify(m, (r, s)) which is analyzed in the context of C. Analysis of these verify operations succeed either by Rule (ReqNotDer) of (ReqNotDer), ensuring S↑_{\text{exec}}(C, m) \rightarrow R↑_{\text{exec}}(C, m). If C has no superclass (i.e., L_C(C).inh = nil), the relation S↑_{\text{exec}}(C, m) \rightarrow R↑_{\text{exec}}(C, m) then follows directly. Otherwise, we have A = L_C(C).inh and A \in E, i.e., S↑_{\text{exec}}(A, m) \rightarrow R↑_{\text{exec}}(A, m) holds. For class C, we have R↑_{\text{exec}}(C, m) = R↑_{\text{exec}}(A, m) \cup R_C(C, m). Since S↑_{\text{exec}}(C, m) \rightarrow R↑_{\text{exec}}(A, m). We consider two cases, m \notin L_C(C).mtds and m \notin L_C(C).mtds. If m \notin L_C(C).mtds, we have S↑_{\text{exec}}(C, m) = S↑_{\text{exec}}(A, m) \cup S_C(C, m). The relation S↑_{\text{exec}}(C, m) \rightarrow R↑_{\text{exec}}(A, m) thereby holds by the assumption S↑_{\text{exec}}(A, m) \rightarrow R↑_{\text{exec}}(A, m). If m is defined in C (i.e., m \in L_C(C).mtds), Rule (NewMtd) will lead to an operation verify(m, R↑_{\text{exec}}(A, m)) which is analyzed in the context of class C. For each (r, s) ∈ R↑_{\text{exec}}(A, m), either Rule (ReqDer) or Rule (ReqNotDer) applies, ensuring S↑_{\text{exec}}(C, m) \rightarrow (r, s). The relation S↑_{\text{exec}}(C, m) \rightarrow R↑_{\text{exec}}(A, m) is thereby ensured by the analysis of C.

Condition 3 of Definition 6 concerns the interface I implemented by C, i.e., L_C(C).impl = I. Given the initial judgment (3), application of Rule (NewClass′) gives the judgment

\[ \varepsilon \oplus \text{extl}(C, A, I, \bar{f}, \bar{M}) \vdash \langle C : \text{anSpec}(\bar{M}) \cdot \text{anReq}(\bar{M}) \cdot \text{intSpec}(\bar{M}) \rangle \]

where \(\bar{M} = \text{public}_C(I)\). Since the analysis of C succeeds, the analysis of each of these operations must succeed. Let ε″ be the environment after analysis of the first two operations, i.e., the judgment ε″ \vdash \langle C : \text{intSpec}(\bar{M}) \rangle is reached. The intSpec operation is analyzed by rules (IntSpec) and (DecompInt). None of these rules extend the environment, and for the successful analysis of intSpec(\bar{M}) we therefore have ε″ = ε′. For each m ∈ \bar{M}, rule (IntSpec) ensures S↑_{\text{exec}}(C, m) \rightarrow spec_C(I, m) as required by Condition 3 in Definition 6.

Condition 4 in Definition 6 follows from the soundness of ε′, the premises of Rule (NewClass′), and the property that the analysis of C only extends S(C, m) and R(C, m) for different methods m where C ∈ ε′. □

We conclude this section with an example, extending Example 12 with interfaces.

**Example 14.** Consider the following interface declaration:

```java
interface IA {
  int n(int y) : (y ≥ 0, return ≥ 2 * y)
  int m(int x) : (x ≥ 0, return ≥ 2 * x)
}
```

For this interface, we have spec(IA, n) = (y ≥ 0, return ≥ 2 * y) for the specification of n and spec(IA, m) = (x ≥ 0, return ≥ 2 * x) for that of m.

Let the class A be as in Example 12, except that A is now defined to implement the interface IA:

```java
class A implements IA {
  int n(int y) : (true, return = 5 * y) { return := 5 * y }
  int m(int x) : (x ≥ 0, return ≥ 2 * x) { return := n(x) }
}
```

The internal analysis of A is as above, i.e., the S and R mappings are as in Example 12, but we now need to ensure that A implements IA. By Rule (IntSpec), this means that the following two relations must hold:

\[ S(A, n) \rightarrow spec(IA, n) \quad \text{and} \quad S(A, m) \rightarrow spec(IA, m) \]

These hold given the specifications in Example 12. Consider next the client code, where field a is now typed by interface IA (we omit the declaration of the interface I implemented by Client as it plays no role in establishing the specifications of the class):
class Client implements J {
   IA a := new A;
   int d1() : (true, return >= 10) {return := a.n(5)}
   int d2() : (true, return >= 10) {return := a.n(5)}
}

The verification of \(d1\) leads to the requirement \((y = 5, \text{return} \geq 10)\) towards the call to \(n\). This requirement is now verified against the interface specification \(\text{spec}(IA, n)\), and follows by entailment:

\[(y \geq 0, \text{return} \geq 2 \cdot y) \rightarrow (y = 5, \text{return} \geq 10)\]

In contrast to the situation in Example 12, the verification of method \(d2\) now also succeeds. The call to \(m\) leads to the requirement \((x = 5, \text{return} \geq 10)\) which follows from \(\text{spec}(IA, m)\) by entailment:

\[(x \geq 0, \text{return} \geq 2 \cdot x) \rightarrow (x = 5, \text{return} \geq 10)\]

### 6. Example

In this section we illustrate our approach by a small bank account system implemented by a class \(\text{PosAccount}\) and its subclass \(\text{FeeAccount}\). The example illustrates how interface encapsulation and the separation of class inheritance and subtyping facilitate code reuse. Class \(\text{FeeAccount}\) reuses the implementation of \(\text{PosAccount}\), but the type of \(\text{PosAccount}\) is not supported by \(\text{FeeAccount}\). Thus, \(\text{FeeAccount}\) does not represent a behavioral subtype of \(\text{PosAccount}\).

A system of communicating components can be specified in terms of the observable interaction between the different components \([9, 13, 26, 47]\). In an object-oriented setting with interface encapsulation, the observable interaction of an object may be described by the communication history, which is a sequence of invocation and completion messages of the methods declared by the interface (ignoring outgoing calls). At any point in time, the communication history abstractly captures the system state. Previous work \([19]\) illustrates how the observable interaction and the internal implementation of an object declared by the interface (ignoring outgoing calls). At any point in time, the communication history abstractly captures the system state. Previous work \([19]\) illustrates how the observable interaction and the internal implementation of an object can be connected. Expressing pre- and postconditions to methods declared by an interface in terms of the communication history allows abstract specifications of objects supporting the interface. For this purpose, we assume an auxiliary variable \(h\) of type \(\text{Seq}[\text{Msg}]\), where \(\text{Msg}\) ranges over invocation and completion (return) messages to the methods declared by the interface. However, for the example below it suffices to consider only completion messages, so a history \(h\) will be constructed as a sequence of completion messages by the empty \((\epsilon)\) and right append \((\cdot)\) constructor. In \([19]\), the communication messages are sent between two named objects, the caller and the callee. However, for the purposes of this example, it suffices to record only the name of the completed method and its parameters, where \(\text{this}\) is implicitly taken as the callee. Furthermore, the considered specifications are independent of the actual callers. We may therefore represent completion messages by \((m(\bar{\pi}, r))\), where \(m\) is a method name, \(\bar{\pi}\) are the actual parameter values for this method call, and \(r\) is the returned value. For reasoning purposes, such a completion message is implicitly appended to the history as a side effect of each method termination, and the postcondition of the method must hold after the history extension. As the history accumulates information about method executions, it allows abstract specification of objects in terms of previously executed method calls.

#### 6.1. Class \(\text{PosAccount}\)

Let an interface \(\text{IPosAccount}\) support three methods \(\text{deposit}, \text{withdraw},\) and \(\text{getBalance}\). The \(\text{deposit}\) method deposits an amount on the bank account as specified by the parameter value and returns the current balance after the deposit. The \(\text{getBalance}\) method returns the current balance. The \(\text{withdraw}\) method returns \(\text{true}\) if the withdrawal succeeded, and \(\text{false}\) otherwise. A withdrawal succeeds only if it leads to a non-negative balance. The current balance of the account is abstractly captured by the function \(\text{Val}(h)\) defined by induction over the local communication history as follows:

\[
\begin{align*}
\text{Val}(\epsilon) & \triangleq 0 \\
\text{Val}(h \cdot (\text{deposit}(x, r))) & \triangleq \text{Val}(h) + x \\
\text{Val}(h \cdot (\text{withdraw}(x, r))) & \triangleq \text{if } r \text{ then } \text{Val}(h) - x \text{ else } \text{Val}(h) \text{ fi} \\
\text{Val}(h \cdot \text{others}) & \triangleq \text{Val}(h)
\end{align*}
\]

In this definition, \(\text{others}\) matches all completion messages that are not captured by any of the above cases. In the interface, the three methods are required to maintain \(\text{Val}(h) \geq 0\).

interface \(\text{IPosAccount}\) {
   int deposit(nat x) : (\text{Val}(h) \geq 0, \text{return} = \text{Val}(h) \land \text{return} \geq 0)
   bool withdraw(nat x) : (\text{Val}(h) \geq 0 \land h = h_0, \text{return} = (\text{Val}(h_0) \geq x) \land \text{Val}(h) \geq 0)
   int getBalance() : (\text{Val}(h) \geq 0, \text{return} = \text{Val}(h) \land \text{return} \geq 0)
}
As before, \( h_0, b_0, \ldots \) denote logical variables. The interface IPosAccount is implemented by a class PosAccount, given below. To make method specifications more compact, we have used the notation \( \text{inv} I \) as an abbreviation for the pre/post specification \((I, I)\) for each public method in the class. In this sense, \( I \) becomes a class invariant. The analysis of \( \text{inv} I \) is captured by our reasoning system since the systems allows a method to have more than one specification. In PosAccount, the balance is maintained by a variable \( bal \), and the invariant expresses that the balance equals \( \text{Val}(h) \) and remains non-negative. The expression \( bal = \text{Val}(h) \) relates the internal state of PosAccount objects and the abstract value \( \text{Val}(h) \), and is used in order to ensure the postconditions declared in the interface.

```java
class PosAccount implements IPosAccount {
    int bal = 0;
    int deposit(nat x): (true, return = bal) {
        update(x); return := bal
    }
    bool withdraw(nat x): (bal = b_0, return = (b_0 \geq x)) {
        if (bal \geq x) then update(-x); return := true
        else return := false fi
    }
    int getBalance(): (true, return = bal) {return := bal}
    void update(int v): (bal = b_0 \land h = h_0, bal = b_0 + v \land h = h_0) {
        bal := bal + v
    }
    inv bal = Val(h) \land bal \geq 0
}
```

Notice that the update method is hidden by the interface. This means that the method is only available for internal invocation; i.e. by method calls on the form \( z := \text{update}(e) \). The following simple definition of withdraw maintains the invariant of the class as it preserves \( bal = \text{Val}(h) \) and does not modify the balance:

```java
bool withdraw(int x) {return := false}
```

However, this implementation is not suitable as it fails to meet the pre/post specification of withdraw, which requires that the method must return true if the withdrawal can be performed without resulting in a non-negative balance. Next we consider the verification of the PosAccount class.

**Pre- and postconditions.** The pre- and postconditions given in the class lead to the inclusion of the following specifications in the \( S \) mapping:

\[
(\text{true, return} = \text{bal}) \in S(\text{PosAccount, deposit}) \tag{4}
\]
\[
(bal = b_0, return = (b_0 \geq x)) \in S(\text{PosAccount, withdraw}) \tag{5}
\]
\[
(\text{true, return} = \text{bal}) \in S(\text{PosAccount, getBalance}) \tag{6}
\]
\[
(bal = b_0 \land h = h_0, bal = b_0 + v \land h = h_0) \in S(\text{PosAccount, update}) \tag{7}
\]

These specifications are easily verified for the bodies of their respective methods. For deposit and withdraw, these specifications do not lead to any requirements on update. The method update is verified by the following proof outline:

\[
\{ bal = b_0 \land h = h_0 \} \ {bal := bal + v \{ bal = b_0 + v \land h = h_0 \}
\]

Since the method is not public, a completion message is not appended to the history by method termination. Furthermore, since there are no calls to public methods in the body of update, the relation \( h = h_0 \) is preserved by the method.

**Invariant analysis.** The class invariant is analyzed as a pre/post specification for each public method, i.e., for the methods deposit, withdraw, and getBalance. As a result, the \( S \) mapping is extended such that

\[
(bal = \text{Val}(h) \land bal \geq 0, bal = \text{Val}(h) \land bal \geq 0) \in S(\text{PosAccount, m}), \tag{8}
\]

for \( m \in \{\text{deposit, withdraw, getBalance}\} \). The two methods deposit and withdraw make internal calls to update, which result in the following two requirements:

\[
R(\text{PosAccount, update}) =
\{ (bal = \text{Val}(h) \land bal \geq 0 \land x \geq 0 \land v = x, \\
bal = \text{Val}(h) + x \land bal \geq 0), \\
(bal = \text{Val}(h) \land bal \geq 0 \land bal \geq x \land x \geq 0 \land v = -x, \\
bal = \text{Val}(h) - x \land bal \geq 0) \}
\tag{9}
\]

These requirements follow by entailment from Specification (7).
**Interface specifications.** The implementation of each method exported by interface IPosAccount must satisfy the corresponding interface specification, according to rule (InstrSpec). For getBalance, it can be proved that the method specification, as given by Specifications (6) and (8), entails the interface specification

\[
(Val(h) \geq 0, \text{return } = Val(h) \land \text{return } \geq 0).
\]

The verification of the other two methods follows the same outline, which concludes the verification of class PosAccount.

### 6.2. Class FeeAccount

The interface IFeeAccount resembles IPosAccount, as the same methods are supported. However, IFeeAccount takes an additional fee for each successful withdrawal, and the balance is no longer guaranteed to be non-negative. For simplicity we take fee as a (read-only) parameter of the interface and of the class (which means that it can be used directly in the definition of \textit{Fval} below). As before, the assertion pairs of the methods are expressed in terms of functions on the local history. Define the allowed overdrafts predicate \textit{AO}(h) by means of a function \textit{Fval}(h) over local histories \textit{h} as follows:

\[
\begin{align*}
AO(h) & \triangleq Fval(h) \geq -fee \\
Fval(\epsilon) & \triangleq 0 \\
Fval(h \cdot (\text{deposit}(x, r))) & \triangleq Fval(h) + x \\
Fval(h \cdot (\text{withdraw}(x, r))) & \triangleq \text{if } r \text{ then } Fval(h) - x - fee \text{ else } Fval(h) \text{ fi} \\
Fval(h \cdot \text{others}) & \triangleq Fval(h)
\end{align*}
\]

The interface IFeeAccount is declared by

```plaintext
interface IFeeAccount (nat fee) {
    int deposit(nat x): (AO(h), return = Fval(h) ∧ AO(h))
    bool withdraw(nat x):
        (AO(h) ∧ h = ho, return = (Fval(ho) ≥ x) ∧ AO(h))
    int getBalance(): (AO(h), return = Fval(h) ∧ AO(h))
}
```

Note that IFeeAccount is not a behavioral subtype of IPosAccount: a class that implements IFeeAccount will not implement IPosAccount. Informally, this can be seen from the postcondition of \textit{withdraw}. For both interfaces, \textit{withdraw} returns true if the parameter value is less or equal to the current balance, but IFeeAccount charges an additional fee in this case, as reflected by the \textit{withdraw} case of the \textit{Fval} definition. As an example, consider the following sequence of method calls executed on a newly created object \textit{o}:

\[
o.\text{deposit}(5); o.\text{withdraw}(4); o.\text{withdraw}(1)
\]

When executed on an instance of IPosAccount, the last withdrawal will return \textit{true}: After \textit{o.\text{deposit}(5)} we have \textit{Val(h) = 5}, and after \textit{o.\text{withdraw}(4)} we have \textit{Val(h) = 1}. Since \textit{Val(h) ≥ 1} when \textit{o.\text{withdraw}(1)} is called, the invocation will return \textit{true} and we have \textit{Val(h) = 0} after the three calls. However, if the calls are executed on an instance of IFeeAccount, the last withdrawal may return \textit{false}. Assume that \textit{fee = 2}. After \textit{o.\text{deposit}(5)}, we have \textit{Val(h) = 5}, but after the first withdrawal we have \textit{Val(h) = −1}. Since \textit{−(Fval(h) ≥ 1)} when \textit{o.\text{withdraw}(1)} is called, the last invocation of \textit{withdraw} will return \textit{false}. Thus, this example illustrates that an instance of IFeeAccount is not a behavioral subtype of an IPosAccount instance. Especially, an instance of IFeeAccount cannot be used whenever an IPosAccount instance is expected.

Given that the implementation provided by the PosAccount class is available, it might be desirable to reuse the code from this class when implementing IFeeAccount. In fact, only the \textit{withdraw} method needs reimplementation. The class FeeAccount below implements IFeeAccount and extends the implementation of PosAccount.

```plaintext
class FeeAccount (int fee) extends PosAccount implements IFeeAccount {
    bool withdraw(nat x): (bal = b0, return = (b0 ≥ x)) {
        if (bal := x) then update(-(x+fee)); return := true
        else return := false fi
    }
    inv bal = Fval(h) ∧ bal ≥ −fee
}
```

Note that the interface supported by the superclass is not supported by the subclass. Typing restrictions prohibit that methods on an instance of FeeAccount are called through the superclass interface IPosAccount.
Pre- and postconditions. As the methods \textit{deposit} and \textit{getBalance} are inherited without redefinition, the specifications of these methods still hold in the context of the subclass. Especially, Specifications (4), (6), and (7) above remain valid. For \textit{withdraw}, the declared specification can be proved:

\[
(bal = b_0, \text{ return } = (b_0 \geq x)) \in S(\text{FeeAccount}, \text{withdraw})
\]  

(10)

Invariant analysis. Again, we take \textit{inv} \textit{I} as an abbreviation of a pre/post specification \( (I, I) \) of each public method in the class. The subclass invariant can be proved for the inherited methods \textit{deposit} and \textit{getBalance} as well as for the new definition of the \textit{withdraw} method. From the proof outline for \textit{deposit}, the following requirement on \textit{update} is included in the requirement mapping:

\[
(bal = Fval(h) \land bal \geq -fee \land x \geq 0 \land v = x, \\
bal = Fval(h) + v \land bal \geq -fee) \in R(\text{FeeAccount}, \text{update})
\]

This requirement follows from Specification (7) of \textit{update}. The analysis of \textit{withdraw} gives the following requirement on \textit{update}, which also follows from Specification (7):

\[
(bal = Fval(h) \land bal \geq -fee \land x \geq 0 \land bal \geq x \land v = -(x + fee), \\
bal = Fval(h) - x - fee \land bal \geq -fee) \in R(\text{FeeAccount}, \text{update})
\]

The invariant analysis leads to the inclusion of the invariant as a pre/post specification in the sets \( S(\text{FeeAccount}, \text{deposit}) \), \( S(\text{FeeAccount}, \text{withdraw}) \), and \( S(\text{FeeAccount}, \text{getBalance}) \), similar to Specification (8).

Interface specification. Now reconsider the method \textit{getBalance}. After the above analysis, the specification set for this method is given by:

\[
S^\dagger(\text{FeeAccount}, \text{getBalance}) = S(\text{FeeAccount}, \text{getBalance}) \cup S(\text{PosAccount}, \text{getBalance}) = \{ (bal = Fval(h) \land bal \geq -fee, bal = Fval(h) \land bal \geq -fee) \} \cup \{ (bal = Val(h) \land bal \geq 0, bal = Val(h) \land bal \geq 0), (true, \text{ return } = bal) \}
\]  

(11)

The interface specification of \textit{getBalance} given by \textit{IFeeAccount} is:

\[
(AO(h), \text{ return } = Fval(h) \land AO(h))
\]  

(12)

Interface specification (12) follows by entailment from Specification (11), using \textit{(InvSpec)}. Note that the superclass invariant is not established by the precondition of Specification (12), which means that the superclass invariant cannot be assumed when establishing the postcondition of Specification (12). However, the other superclass specification is needed, expressing that \textit{return} equals \textit{bal}. The verification of the interface specifications for \textit{deposit} and \textit{withdraw} follows the same outline.

7. Related and future work

Object-orientation poses several challenges to program logics; e.g., inheritance, late binding, recursive and re-entrant method calls, aliasing, and object creation. In the last years, several programming logics have been proposed, addressing various of these challenges. For example, \textit{java} creation has been addressed by means of specialized allocation predicates [1] or by encoding heap information into sequences [15]. Numerous proof methods, verification condition generators, and validation environments for object-oriented languages have been developed, including [1,2,8,22,27,29,48,49]. \textit{Java} in particular has attracted much interest, with advances being made for different, mostly sequential, aspects and sublanguages of that language. In particular, most such formalizations concentrate on closed systems.

Class inheritance is a central feature of object orientation which allows subclasses to be designed by reusing and redefining the code of superclasses with a flexibility which goes beyond behavioral subtypism [46]. However, proof systems usually restrict code reuse to behavioral subtyping. For example, a recent survey of challenges and results for the verification of sequential object-oriented programs [36] relies on behavioral subtyping when reasoning about late binding and inheritance. In contrast, proof systems studying late bound methods without relying on behavioral subtyping have been shown to be sound and complete by Pierik and de Boer [43], assuming a closed world. See also [44] for a discussion of (relative) completeness.
in connection with behavioral subtyping. While proof-theoretically satisfactory, the closed world assumption is unrealistic in practice and necessitates costly reverification when the class hierarchy is extended (as discussed in Section 1). Our work on lazy behavioral subtyping is situated between these two approaches.

In order to better support object-oriented design, proof systems should be constructed for incremental (or modular [18]) reasoning. Most prominent in that context are different variations of behavioral subtyping [35, 38, 46]. The underlying idea is quite simple: subtyping in general is intended to capture “specialization” and in object-oriented languages, this may be interpreted such that instances of a subclass can be used where instances of a superclass are expected. To generalize this subsumption property from types (such as method signatures) to behavioral properties is the step from standard to behavioral subtyping. The notion of behavioral subtyping dates back to America [3] and Liskov and Wing [37, 38], and is also sometimes referred to as Liskov’s substitutability principle. The general idea has been explored from various angles. For instance, behavioral subtyping has been characterized model-theoretically [17, 34] and proof-theoretically [4, 38].

Specification inheritance is used to enforce behavioral subtyping in [18], where subtypes inherit specifications from their supertypes (see also [51] which describes specification inheritance for the language Fresco). Virtual methods [45] similarly allow incremental reasoning by committing to certain abstract properties about a method, which must hold for all its implementations. Although sound, the approach does not generally provide complete program logics, as these abstract properties would, in non-trivial cases, be too weak to obtain completeness without over-restricting method redefinition from the point of view of the programmer. Such specifications of virtual methods furthermore force the developer to commit to specific abstract specifications of method behavior early in the design process. This seems overly restrictive and lead to less flexibility in subclass design than the approach as such suggests. In particular, the verification platforms for SpecC [7] and JML [10] rely on versions of behavioral subtyping. Wehrheim [50] investigates behavioral subtyping not in a sequential setting but for active objects. Dynamic binding in a general sense, namely that the code executed is not statically known, does not only arise in object-oriented programs. Ideas from behavioral subtyping have been used to support modular reasoning in the context of aspect-oriented programs [12, 33].

The fragile base class problem emerges when seemingly harmless superclass updates lead to unexpected behavior of subclass instances [40]. Many variations of the problem relate to imprecise specifications and assumptions made in super- or subclasses. By making method requirements and assumptions explicit, our calculus provides an approach to dealing with the fragile base class problem. Subclasses can only rely on requirements made explicit in the requirement property set of the class. Updates in the superclass must respect these assumptions.

Recently incremental reasoning, both for single and multiple inheritance, has been considered in the context of separation logic [11, 39, 42]. These approaches distinguish “static” specifications, given for each method implementation, from “dynamic” specifications used to verify late-bound calls. The dynamic specifications are given at the declaration site, in contrast to our work on lazy behavioral subtyping in which late-bound calls are verified based on call-site requirements. As in lazy behavioral subtyping, the goal is “modularity”; i.e., the goal is to avoid reverification when incrementally developing a program. Complementing the results presented in this paper, we have shown how lazy behavioral subtyping can be used in the setting of multiple inheritance in [21], in which strategies for method binding in multiple inheritance class hierarchies are related to lazy behavioral subtyping.

We currently integrate lazy behavioral subtyping in a program logic for Creol [16, 31], a language for dynamically reprogrammable active objects, developed in the context of the European project Credo. This integration requires a generalization of the analysis to multiple inheritance and concurrent objects, as well as to Creol’s mechanism for class upgrades. Creol’s type system is purely based on interfaces. Interface types provide a clear distinction between internal and external calls. As shown in this paper, the separation of interface level subtyping from class level inheritance allows class inheritance to exploit code reuse quite freely based on lazy behavioral subtyping, while still supporting incremental reasoning techniques. Classes in Creol may implement several interfaces, slightly extending the approach presented in this paper. It is also possible to let interfaces influence the reasoning for internal calls in a more fine-grained manner, with the aim of obtaining even weaker requirements to redefinitions. We are currently investigating the combination of lazy behavioral subtyping with class upgrades. This combination allows class hierarchies to not only evolve by subclass extensions, but also by restructuring the previously analyzed class hierarchy in ways which control the need for reverification.

8. Conclusion

This paper presents lazy behavioral subtyping, a novel strategy for reasoning about late bound internal method calls. The strategy is designed to support incremental reasoning and avoid reverification of method specifications in an open setting, where class hierarchies can be extended by inheritance. To focus the presentation, we have abstracted from many features of object-oriented languages and presented lazy behavioral subtyping for an object-oriented kernel language based on single inheritance. This reflects the mainstream object-oriented languages today, such as Java and C{}

Behavioral subtyping has the advantage of providing incremental and modular reasoning for open object-oriented systems, but severely restricts code reuse compared to programming practice due to behavioral constraints on method overriding. Lazy behavioral subtyping also provides incremental reasoning, but supports significantly more flexible reuse of code. In addition lazy behavioral subtyping offers modularity when combined with interfaces, separating the interface and class
hierarchies to support both subtyping and flexible code reuse. This paper presents both systems with soundness proofs. An example of code reuse in the banking domain demonstrates how incremental reasoning is achieved by lazy behavioral subtyping in a setting where behavioral subtyping does not apply. Lazy behavioral subtyping appears as a promising framework for controlling a range of desirable changes in the development of object-oriented class hierarchies.

Acknowledgment
The authors gratefully recognize the very thorough efforts of the anonymous reviewers of this paper.

References

Incremental reasoning with lazy behavioral subtyping for multiple inheritance✩

Johan Dovland⁎, Einar Broch Johnsen, Olaf Owe, Martin Steffen

Department of Informatics, University of Oslo, Norway

Abstract

Object-orientation supports code reuse and incremental programming. Multiple inheritance increases the possibilities for code reuse, but complicates the binding of method calls and thereby program analysis. Behavioral subtyping allows program analysis under an open world assumption; i.e., under the assumption that class hierarchies are extensible. However, method redefinition is severely restricted by behavioral subtyping, and multiple inheritance may lead to conflicting restrictions from independently designed superclasses. This paper presents a more liberal approach to incremental reasoning for multiple inheritance under an open world assumption. The approach, based on lazy behavioral subtyping, is well-suited for multiple inheritance, as it incrementally imposes context-dependent behavioral constraints on new subclasses. We first present the approach for a simple language and show how incremental reasoning can be combined with flexible code reuse. Then this language is extended with a hierarchy of interface types which is independent of the class hierarchy. In this setting, flexible code reuse can be combined with modular reasoning about external calls in the sense that each class is analyzed only once. We formalize the approach as a calculus and show soundness for both languages.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

Object-orientation supports code reuse and incremental programming through inheritance. Class hierarchies are extended over time as subclasses are developed and added. A class may reuse code from its superclasses but it may also specialize and adapt this code by providing additional method definitions, possibly overriding definitions in superclasses. This way, the class hierarchy allows programs to be represented in a compact and succinct way, significantly reducing the need for code duplication. Late binding is the underlying mechanism for this incremental programming style; the binding of a method call at runtime depends on the actual class of the called object. Consequently, the code to be executed depends on information which is not statically available. Although late binding is an important feature of object-oriented programming, this loss of control severely complicates reasoning about object-oriented programs.

Behavioral subtyping is the most prominent solution to regain static control of late bound method calls (see, e.g., [27,1,25]) with an open world assumption; i.e., where class hierarchies are extensible. This approach achieves incremental reasoning in the sense that a subclass may be analyzed in the context of previously defined classes, such that previously proved properties are ensured by additional verification conditions. However, the approach restricts how methods may be redefined in subclasses. To avoid reverification, any method redefinition must preserve certain properties of the method.

✩ This work was partly supported by the EU projects IST-33826 CREDO: Modeling and Analysis of Evolutionary Structures for Distributed Services (http://credo.cwi.nl) and FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

⁎ Corresponding author.

E-mail addresses: johand@ifi.uio.no (J. Dovland), einarj@ifi.uio.no (E.B. Johnsen), olaf@ifi.uio.no (O. Owe), msteffen@ifi.uio.no (M. Steffen).
which is redefined. In particular, this applies to the method’s contract; i.e., the pre- and postcondition for its body. This contract can be seen as a description of the promised behavior of all implementations of the method. Unfortunately, this restriction limits code reuse and is often violated in practice [37]; for example, it is not respected by the standard Java library definitions.

Multiple inheritance offers greater flexibility than single inheritance, as several class hierarchies can be combined in a subclass. However, it also complicates language design and is often explained in terms of complex run-time data structures such as virtual pointer tables [38], which are hard to understand. Formal treatments are scarce (e.g., [36,9,5,18,40]), but help clarify intricacies, thus facilitating design and reasoning for programs using multiple inheritance. Multiple inheritance also complicates behavioral reasoning, as name conflicts may occur between methods which were independently defined in different branches of the class hierarchy.

Work on behavioral reasoning about object-oriented programs has mostly focused on languages with single inheritance (see, e.g., [34,5,8]). It is an open problem how to design an incremental proof system for multiple inheritance under an open world assumption, without severely restricting code reuse. In this paper we propose a solution to this problem. The approach extends lazy behavioral subtyping, which was developed for single inheritance systems [15,17] to allow more flexible code reuse than reasoning systems based on behavioral subtyping. Our approach applies to a wide class of object-oriented systems, relying on the assumption of a healthy binding strategy, which is needed for incremental reasoning. Healthiness may easily be imposed on non-healthy binding strategies. The approach is formalized as a syntax-driven inference system, for which we show soundness. The inference system combines deductive style program logic with incremental program development, and is well-suited for program development environments [16].

Although this system ensures that old proofs are never violated, an external call \(x.m(\varnothing)\), where \(x\) is an object variable, may result in additional proof obligations for the declared class of \(x\), and that class may already have been verified. As a consequence, it may be necessary to revisit previously verified classes at a later stage in the program analysis. To improve this situation, we extend [16] by considering a refined version of the calculus which introduces behavioral interfaces to encapsulate objects. The behavioral constraints of the interface implemented by a class become proof obligations for that class. As a result, the refined calculus is both incremental and modular: it is no longer necessary to revisit a class due to requirements on calls which occur later during the analysis of unrelated classes. In the refined system, subtyping applies to the inheritance relationship on interfaces, whereas code may be reused more freely in the class hierarchy.

**Paper overview.** Section 2 discusses late binding and multiple inheritance. Section 3 introduces proof environments for behavioral reasoning, and Section 4 presents the inference system for incremental reasoning. Section 5 extends the inference system with interface encapsulation. Section 6 discusses related work and Section 7 concludes the paper.

2. Late binding and multiple inheritance

2.1. Syntax for an object-oriented Kernel language MI

To succinctly explain late binding and our analysis strategy, we consider an object-oriented kernel language called MI with a standard operational semantics, e.g., similar to that of Featherweight Java [21]. The syntax of MI is given in Fig. 1. Vector notation denotes lists; e.g., a list of expressions is written \(\vec{e}\). A program \(P\) consists of a list \(\vec{L}\) of class definitions, followed by a method body. For simplicity, we let expressions \(e\) (other than method calls and object creation) be without side-effects and assume that methods with the same name have the same signature (i.e., no method overloading), class names are unique, programs are well-typed, and we ignore the types of fields and methods. Two notable differences to Featherweight Java are multiple inheritance and a corresponding form of static method calls. These are explained below.

For classes \(C_1\) and \(C_2\), we let \(C_1 \leq C_2\) denote the reflexive and transitive subclass relation derived from class inheritance. We say that \(C_1\) is below \(C_2\) if \(C_1 \leq C_2\). Thus, \(C_1 \leq C_2\) if \(C_1\) and \(C_2\) are the same class or if \(C_1\) extends a class that is below \(C_2\). Furthermore, \(C_2\) is above \(C_1\) if \(C_1\) is below \(C_2\). A subclass is below a superclass. The two classes are related, written \(C_2 \Rightarrow C_1\), if one is below the other.

A class \(C\) extends a list \(\vec{C}\) of superclass names with fields \(\vec{f}\), methods \(\vec{M}\), and method specifications \(\vec{MS}\). The list \(\vec{C}\) is assumed to consist of unique class names, and the names of methods \(\vec{M}\) and fields \(\vec{f}\) are assumed to be unique within the
class B {
  nat x:=0, y := 0;
  n() {y := y + x}
  m():(true, y>= x) {x := x + 1; n()}
}

class C extends B {
  n1() {(y:=x)}
  n2() {(y:=x + 1)}
  m@B():(true, x = y)
}

class D extends B {
  n() {y := y + x}
  m@B():(true, y > x)
}

Fig. 2. Small example of a class hierarchy with subclass specification of inherited methods.

class. We say that C defines a method m if M contains an implementation of m. Let a partial function body(C, m) return this implementation (so body(C, m) is undefined if m is not in M). For a method m defined in some superclass B of C, but not defined in any subclass of B above C, we say that C inherits m from B if m is not defined in C, otherwise m is overridden in C. Since C may extend more than one class, it is possible for C to inherit m from more than one superclass.

A method M takes formal parameters $X$ and contains a statement $t$ as its method body where $X$ and $t$ are read-only. The method may also declare a list $Z$ of local variables. The sequential composition of statements $t_1$ and $t_2$ is written $t_1; t_2$. The statement $v := \text{new} C$ creates a new object of class C with fields instantiated to default values, and assigns the new reference to v. (In MI, a possible constructor method in the class must be called explicitly.) There are standard statements for skip, conditionals if $b$ then $t_1$ else $t_2$ fi, and assignments $v := e$. We use if $b$ then $t$ fi as an abbreviation for if $b$ then $t$ else skip fi.

We syntactically distinguish internal late bound calls, internal static calls, and external calls. For an internal late bound call $m(E)$, the method $m$ is executed on this with the actual parameters $E$. The call is bound at run time depending on the actual class of the object. The symbol @ is used for static binding. An internal static call $m@C(E)$ may occur in a class below C, and the call is bound above C at compile time. This statement generalizes the call to the superclass found in languages with single inheritance; C may here be any class above the current class as long as an implementation of $m$ can be found in a class above C. In an external method call $e.m(E)$, the object $e$ (which may be self) receives a call to the method $m$ with the actual parameters $E$. All external calls are late bound. The statements $v := m(E)$, $v := m@C(E)$, and $v := e.m(E)$ assign the value of the method activation’s return variable to $v$. If $m$ does not return a value, or if the returned value is of no concern, we sometimes use $e.m(E)$, $m@C(E)$, or $m(E)$ directly as statements for simplicity. Note that the list $E$ of actual parameter values may be empty. Similarly to static calls, f@C binds a field $f$ above C.

User given method specifications may occur in class definitions, and are of the form $m@B(X) := (p, q)$. We say that a specification is given in the context of a class C if the specification occurs syntactically in the definition of C. Here, B may be any class above C, as long as an implementation of $m$ can be found above B. Remark that B and C may be the same class. The assertion pair $(p, q)$ defines a pre/post specification for the definition of $m@B(X)$. As the specification is given in the context of C, it must be satisfied when this method definition is executed on instances of classes below C, but it need not be satisfied when $m$ is executed on instances of B (unless B and C are the same class). Assertions may range over the available fields and method parameters, this, return, and logical variables. For convenience, we let $m(X) := (p, q)\{t\}$, defined in class C, abbreviate the combination of the definition $m(X)\{t\}$ and the specification $m@C(X) := (p, q)$.

2.2. Context dependent specifications

Specifications of a method m may be given in the class where m is defined or in a subclass. If m is not overridden by a subclass, the subclass may still provide a specification of m. This is feasible in the presence of late binding, as some method $n$ called by m may be overridden by the subclass. A subclass specification of m may then account for the behavior of m when taking the overriding version of n into account. This is illustrated by the following example.

Example 1. Consider the classes B and C in Fig. 2. Class B defines two methods m and n, where there is a call to n in the body of m. The method n is overridden by the subclass C, and C gives a specification of the inherited method m. When executed on an instance of C, execution of n will lead to a state where x equals y. Note that the specification is given in the context of subclass C; it is not guaranteed to hold when m is executed on an instance of B. The specification of m given in class B holds when m is executed on an instance of B or C.

Observe that different subclasses may override methods in different manners. As illustrated by the next example, this means that different subclasses may have conflicting specifications of some method m, since internal calls in the body of m may bind differently due to late binding.

Example 2. Consider the class D in Fig. 2. As in the above example, the subclass overrides method n and gives a specification of method m. This specification may hold when m is executed on instances of subclass D. There is a conflict between the two specifications of m given in C and D, in the sense that the conjunction of the postconditions equals the unwanted postcondition false. However, this apparent conflict does not lead to any reasoning problems as the specifications are given in the context of unrelated classes. The specification of m given in class B also holds when m is executed on an instance of D.

Inheritance relates classes in a class hierarchy. For single inheritance this hierarchy forms a tree, whereas for multiple inheritance, the hierarchy forms a directed, acyclic graph. In the single inheritance tree, vertical name conflicts occur when a subclass overrides a method from a superclass. The binding strategy for method calls must resolve such conflicts. Late binding or dynamic dispatch selects the method body to be executed at run-time, depending on the callee’s run-time class: the selected body is found by the first matching definition of the method above the actual class of the object. In class hierarchies with multiple inheritance, there are also horizontal name conflicts. These occur when different definitions of the same method are found above a given class, depending on the chosen path through the hierarchy. More elaborate binding strategies are needed to resolve horizontal conflicts. Some binding strategies are infeasible, as they contradict incremental program development. This is illustrated by the following example.

### Example 3

We consider a class hierarchy for a bank account system, given in Fig. 3. Potential problems with horizontal name conflicts are illustrated by the classes in Fig. 4, sketching an implementation of the classes Account, Auth, and AuthAccount. (Implementations of the remaining classes is considered in Example 5.) Class Account implements basic facilities for depositing and withdrawing money. The actual manipulation of the balance is implemented by a method update. Assuming that account owners are identified by integers, the owner of the account is the only client allowed to make withdrawals, as checked by the method validate. Class Auth, developed independently of Account, provides functionality for storing two client identities. These two classes are inherited by the subclass AuthAccount. By overriding the method validate in the subclass, the fields declared by Auth are now allowed to hold clients that are allowed to perform withdrawals. For AuthAccount, inheritance of Account and Auth gives a horizontal name conflict for method update. The behavior of the two versions of update is completely different, which means that the behavior assumed by the add method in Auth will not hold in the subclass if the internal call to update is bound to the implementation in Account. Thus, in order to support incremental design, the internal call in Auth should bind to the definition in Auth, and correspondingly, the internal calls in Account should bind to the definition in Account.

One solution to resolve horizontal name conflicts is by explicit resolution of the names of the superclass’ methods, specified as part of the inheritance list: e.g., by qualification or renaming of methods as in C++ [38], Eiffel [29], and POOL [2]. However, it might be undesirable to force the programmer to modify method names, making programs more difficult to understand and maintain. We generalize this approach and decorate each internal call with a binding clause restricting the binding space. Such a clause may represent a specific name resolution strategy, or be explicitly provided by the programmer. This way, the
approach of this paper is applicable to several resolution strategies. Binding clauses allow us to consider horizontal name conflicts as a natural feature of multiple inheritance. In particular when using libraries, the programmer cannot be expected to know (or resolve) potential name conflicts of, e.g., auxiliary methods in the libraries. To support incremental program development and reasoning, we impose the following healthiness condition on the binding of a method call:

- an internal call made by a method defined in C must bind to a class related to C, and
- an external call x.m, where x has C as declared class, must bind to a class related to C.

It is assumed that vertical name conflicts are resolved as explained above. We say that a binding strategy is healthy if all calls are guaranteed to be healthy. These healthiness restrictions resolve the binding problems illustrated in Example 3. Explicit resolution of horizontal name conflicts may ensure healthiness. It is easy to see that healthiness removes accidental overriding of methods, due to unfortunate binding. In particular, if an empty subclass C extends two unrelated classes A and B, then C does not cause unexpected behavior due to possible horizontal name conflicts in A and B.

Let C#m denote a call to m where the binding is restricted to classes related to C. In Example 3, if the call to update in Auth is replaced by Auth#update, the call becomes healthy. When executed in an instance of AuthAccount, the call will bind in a class related to Auth. For the rest of this paper, we use the convention that an internal call to m made by a method defined in C is understood as C#m. Similarly, an external call x.m with C as the declared class of x, is understood as x.C#m. Note that these translations can be done mechanically and integrated with static type checking. As static calls are inherently healthy, this ensures healthy binding.

We remark that the notation C#m might also be used by the programmer to distinguish between definitions of m that are inherited from different superclasses. In general, several superclasses may restrict the binding of methods, using the notation C1#C2#...#m to restrict m to definitions in classes related to every C. However, in order to keep the presentation simple, we will not consider multiple restrictions, and therefore limit the presentation to calls restricted by the conventions above.

### 2.4. The binding of method calls and fields

For the reasoning system, we need an explicit definition of a healthy resolution strategy. In this paper, we formalize such a strategy by a function bind defined below. Other definitions of bind are possible and would lead to variations of the calculus.

We say that a call to a method m is bound with respect to a search class D; i.e., bind(D, m) denotes the search for a definition of m which starts in D. In this case, the call must bind to a definition of m in a class above D, such that no other definition of m is found by the search below this class. Following \cite{10,13,22}, ambiguities are solved by fixing the order in which inherited classes are searched, e.g., from left to right. Let Cid and Mtd denote class and method names. To make the representation of class hierarchies compact, a class name is bound to a tuple (C, f, M) of type Class, where the declared superclasses C, the fields f, and the method definitions M are accessible by observer functions inh, fields, and mtds, respectively. This binding strategy can be defined by a partial function bind : List[Cid] × Mtd → Cid as follows:

\[
\begin{align*}
\text{bind}(&\text{nil}, m) \triangleq \bot \\
\text{bind}(D \bar{D}, m) &\triangleq D & \text{if } m \in D.mtds \\
\text{bind}(D \bar{D}, m) &\triangleq \text{bind}(D.\text{inh} \bar{D}, m) & \text{otherwise},
\end{align*}
\]

where \(D.\text{inh} \bar{D}\) reduces to \(\bar{D}\) when \(D.\text{inh}\) is empty. Observe that this strategy is not healthy, since an internal call would be bound independently of where the call-site occurs in the class hierarchy, i.e., the class in which the call textually occurs. For internal late bound calls, a healthy strategy can be obtained by restricting the binding to classes related to the call-site. We denote by \(\text{bind}(D, C#m)\) the binding of a call \(C#m\) for search class D. The search is restricted by C; the returned class must be either above or below C. This ensures the healthiness condition described above. By type-safety, there is a definition of m above C; thus \(\text{bind}(D, C#m)\) is well-defined for D below C. A healthy binding strategy may then be defined by the following function:

**Definition 1.** Define \(\text{bind}(\_\_, \_\_\_\_\_) : \text{List[Cid]} \times \text{Cid} \times \text{Mid} \rightarrow \text{Cid}\) by:

\[
\begin{align*}
\text{bind}(\text{nil}, C#m) &\triangleq \bot \\
\text{bind}(D \bar{D}, C#m) &\triangleq D & \text{if } C \Rightarrow D \land m \in D.mtds \\
\text{bind}(D \bar{D}, C#m) &\triangleq \text{bind}(D.\text{inh} \bar{D}, C#m) & \text{if } C \Rightarrow D \land m \notin D.mtds \\
\text{bind}(D \bar{D}, C#m) &\triangleq \text{bind}(\bar{D}, C#m) & \text{otherwise},
\end{align*}
\]

For simplicity we here ignore typing of parameters. With type information, one should in addition to the \(m \in D.mtds\) check in line two, check that the types of the actual parameters are subtypes of the corresponding formal parameters. See \cite{23} for more details.

For external calls, healthiness is ensured by binding the call \(x.m\) by \(\text{bind}(D, C#m)\) where C is the declared class of x and D the actual class of x. A statically bound method call \(m@C\) is bound above C by \(\text{bind}(C, C#m)\). For a method specification \(m@C(x) : (p, q)\) given in the context of some class D, we take \((p, q)\) as a specification of the first implementation of m above C as found by \(\text{bind}(C, C#m)\).

Similar binding functions may be used to define the binding of fields: An occurrence of \(f@B\) is allowed inside a class definition \(C\) if B is above C, and is bound above B; and an unqualified occurrence of \(f\) inside C is understood as \(f@C\).
3. Lazy behavioral subtyping

Lazy behavioral subtyping supports incremental reasoning for extensible class hierarchies; each class is analyzed based on the analysis of its superclasses, but independently of (future) subclasses. Lazy behavioral subtyping was presented for single inheritance in [15,17]. We here present an extension for multiple inheritance and horizontal name conflicts based on the language MI and the healthy binding strategy defined in Section 2.4. With healthy binding, an internal late bound method call binds to a class related to the call-site. Behavioral constraints may therefore be propagated down the class hierarchy, which allows incremental reasoning. The proof method has two parts, a conventional program logic (e.g., [34,19,432]) and, on top of that, a proof environment which incrementally tracks method specifications and requirements.

3.1. Proof outlines

Apart from the treatment of late bound method calls, our initial reasoning system follows standard proof rules [3,4] for partial correctness, adapted to the object-oriented setting; in particular, de Boer's technique using sequences in the assertion language addresses the issue of object creation [12]. We present the proof system using Hoare triples \([p] t [q]\) [19], for assertions \(p\) and \(q\), and statement sequence \(t\). Here, \(p\) is the precondition and \(q\) is the postcondition to \(t\). The meaning \(\models [p] t [q]\) of a triple \([p] t [q]\) is standard: if \(t\) is executed in a state where \(p\) holds and the execution terminates, then \(q\) holds after \(t\) has terminated. The derivation of triples can be done in any suitable program logic. Let PL be such a program logic and let \(\Gamma_{\text{PL}}(p) t [q]\) denote that \([p] t [q]\) is derivable in PL. A proof outline [32] for a triple \([p] t [q]\) is the statement sequence decorated with assertions. The main idea is to decorate different points in the program \(t\) with assertions such that the analysis between the different program points can be done mechanically, ensuring \(\Gamma_{\text{PL}}(p) t [q]\). A classical example is to decorate loops in the program with loop invariants. For the purposes of this paper, we are mainly interested in decoration of method calls with pre- and postconditions.

Let the notation \(O \Gamma_{\text{PL}} t : (p, q)\) mean that \(O\) is a proof outline proving that the specification \((p, q)\) holds for a body \(t\); i.e., \(\Gamma_{\text{PL}}(p) O [q]\) holds when assuming that the pre- and postconditions provided in \(O\) for the method calls contained in \(t\) are correct. The pre- and postconditions for the internal late bound method calls are called requirements. Thus, for a decorated call \(r(n) s\) in \(O\), \((r, s)\) is a requirement for \(n\). Examples of proof outlines can be found in Example 5.

3.2. Method specifications and requirements

The verification technique distinguishes between a method's declared specification (its contract) and its requirement. Roughly, the first captures its announced behavior as declared in the specification list \(S\) of class definitions. For a method \(m\) defined in class \(B\) and a user given specification \(m@B(X) : (p, q)\) given in the context of class \(C\), the assertion pair \((p, q)\) is remembered as a specification of the implementation body \((B, m)\). In contrast, the requirements stem from the analysis of internal late bound method calls and represent properties needed to verify the call-site of a method, namely to satisfy the specification of the call-site. Inside a class hierarchy, a method with a given name may be available in more than one class due to inheritance, and can be called internally from different call-sites. Consequently, the properties related to a method definition are considered for each class and its position in the class hierarchy. If, furthermore, the class hierarchy is incrementally extended, new specifications and requirements may be added. This bookkeeping of properties is done in a proof environment, by means of two mappings \(S\) and \(R\). Method specifications and requirements are written as assertion pairs \((p, q)\) of type APair.

**Definition 2 (Proof Environments).** A proof environment \(E\) of type Env is a tuple \((L_E, S_E, R_E)\), where \(L_E : \text{Cid} \rightarrow \text{Class}\) is a partial mapping and \(S_E, R_E : \text{Cid} \times \text{Cid} \times \text{Mid} \rightarrow \text{Set}[\text{APair}]\) are total mappings.

In a proof environment \(E\), the mapping \(L_E\) reflects the class hierarchy and the two mappings \(S_E\) and \(R_E\) organize the properties collected so far during analysis. Assume that \(m\) is defined in \(B\) and that the user gives a specification \(m@B(X) : (p, q)\) in the context of class \(C\). During the analysis of class \(C\), the specification \((p, q)\) is included in the specification set \(S(C, B, m)\). We use the notation \(S(C, B, m)\) to emphasize that \(m\) is defined in \(B\). A proof outline for the method body must then be supplied by the developer, where method calls are decorated with pre- and postconditions. For each internal late bound call \(r(n) s\) in this outline, the requirement \((r, s)\) is included in the requirement set \(R(C, B, n)\) by the analysis of the outline. Here, \(C\) denotes the class that imposes the requirement, as the original specification \((p, q)\) is given by \(C\) and \(B\) is the call-site. We use the notation \(R(C, B, s)\) for \(R(C, B, n)\) to emphasize that \(B\) is the call-site. Remark that during the analysis of \(C\), the requirement \((r, s)\) is verified for the method that the call will bind to in the context of class \(C\). The inclusion of \((r, s)\) in \(R(C, B, s)\) acts a restriction to future subclasses of \(C\): the requirements made by late bound calls in the proof outline for the body of \(m\) are imposed on subclasses of \(C\). Especially, if the method \(n\) is overridden in a subclass \(D\), the requirements contained in \(R(C, B, s)\) are verified for the new definition of \(n\) when \(D\) is analyzed. For the call \(r(n) s\), the requirement \((r, s)\) then holds for all method implementations that the call can bind to in the context of any class below \(C\). This means that we may rely on the specification \((p, q)\) of body \((B, m)\) also when the method is executed on a subclass instance, i.e., where the internal late bound calls in \(m\) are bound in context of the subclass.
If \((p, q) \in S(C, B.m)\), we may assume this assertion pair when reasoning about calls that can bind to \(body(B, m)\). Especially, we may assume \((p, q)\) when reasoning about static calls \(m@B(r)\) and when reasoning about internal late bound calls that can bind to \(body(B, m)\). As an important property of our reasoning system, we remark that **overriding implementations** of \(m\) in subclasses may satisfy different contracts than the definition in the superclass. Especially, if \(m\) is overridden in a subclass \(D\) of \(C\), the specification \(S(C, B.m)\) is not inherited, i.e., it is not imposed on the new version of \(m\). However, requirements made by superclasses, e.g., \(R(C, B#m)\), are inherited and imposed on the overriding version of \(m\) in \(D\).

In general, if the set \(S(C, B.m)\) is non-empty, the set was extended during the analysis of \(C\), and \(C \leq B\). Likewise, if \(R(C, B#m)\) is non-empty, the set was extended during the analysis of \(C\), and \(C \leq B\). Let \(C \in \mathcal{S}\) denote that \(\leq(C)\) is defined, and \(C \in \mathcal{S}\) denote \(C \in \mathcal{S}\) for each \(C \in \mathcal{C}\). For the empty environment \(\emptyset\), \(\leq(\emptyset)\) is undefined and \(S_{\emptyset}(C, B.m) = R_{\emptyset}(C, B#m) = \emptyset\) for all \(C, B : \text{Cid}\) and \(m : \text{Mid}\). In the following, we let functions indexed by \(\mathcal{S}\), e.g., \(\text{bind}_e(C, B#m)\), denote that the functions are evaluated over the classes defined in \(\mathcal{S}\).

Consider a method \(m\) defined in a class \(B\). By the analysis of \(B\) declared specification of \(m\) given in \(B\) will be included in the set \(S(B, B.m)\). Subclasses of \(B\) may give additional specifications \(s\) for the implementation of \(m\). For example, if a method \(n\) is overridden by a subclass \(C\) of \(B\), and \(m\) calls \(n\), a specification of \(body(B, m)\) given in \(C\) may account for \(m\)'s behavior with the overriding version of \(n\). Hence, the specification of \(m\) as given in the context of class \(C\) is included in the set \(S(C, B.m)\).

**Example 4.** We consider the method \texttt{withdraw}, as implemented in class \texttt{Account} in Fig. 4. By the analysis of that class, the following specification is included in the set \(S(\text{Account}, \text{Account.withdraw})\):

\[
(bal = b_0 \land \text{owner} = \text{id}) \land \text{bal} = b_0 - x)
\]

The method \texttt{validate} is overridden in subclass \texttt{AuthAccount}, which means that withdrawals will succeed if the client \text{id} is contained in the attributes of the class \text{Auth}. The following additional specification of method \texttt{withdraw} can then be given in the context of the subclass:

\[
(bal = b_0 \land (\text{id} = a1 \lor \text{id} = a2)) \lor \text{bal} = b_0 - x)
\]

By the analysis of \text{AuthAccount}, this specification is recorded in the set \(S(\text{AuthAccount}, \text{Account.withdraw})\).

### 3.3 Entailment

Since we deal with sets of assertion pairs, the standard consequence rule of Hoare Logic [3] is insufficient. We need an entailment relation which allows us to combine information from several assertion pairs. Let \(p^o\) denote an expression \(p\) with all occurrences of program fields \(f\) substituted by the corresponding variables \(f^o\), avoiding name capture. The assertion pair \((p, q)\) is understood as an input/output relation \(\forall \mathcal{Z} . p \Rightarrow q^o\), where \(f\) and \(f^o\) denote the input and output values of \(f\), respectively, and \(\mathcal{Z}\) are the logical variables in \(p\) and \(q\). The entailment relation is defined for assertion pairs and for sets of assertion pairs as follows:

**Definition 3 (Entailment).** Let \((p, q)\) and \((r, s)\) be assertion pairs and let \(\mathcal{U}\) and \(\mathcal{V}\) denote the sets \(\{(p_i, q_i) | 1 \leq i \leq n\}\) and \(\{(r_i, s_i) | 1 \leq i \leq m\}\). Entailment is defined by

1. \((p, q) \rightarrow (r, s) \triangleq (\forall \mathcal{Z}_1 . p \Rightarrow q^o) \Rightarrow (\forall \mathcal{Z}_2 . r \Rightarrow s^o)\),
   where \(\mathcal{Z}_1\) and \(\mathcal{Z}_2\) are the logical variables in \((p, q)\) and \((r, s)\), respectively.
2. \(\mathcal{U} \vdash (r, s) \triangleq (\bigwedge_{1 \leq i \leq m} (\forall \mathcal{Z}_1 . p_i \Rightarrow q_i)) \Rightarrow (\forall \mathcal{Z}_2 . r \Rightarrow s^o)\).
3. \(\mathcal{U} \rightarrow \mathcal{V} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrow (r_i, s_i)\).

The relation \(\mathcal{U} \rightarrow (r, s)\) corresponds to classic Hoare style reasoning, proving \(\{r\} t \{s\}\) from \(\{p_i\} t \{q_i\}\) for all \(1 \leq i \leq n\), by means of the adaptation and conjunction rules [3]. Note that when proving entailment, program fields (primed and unprimed) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and \(\mathcal{V} \subseteq \mathcal{U}\) implies \(\mathcal{U} \rightarrow \mathcal{V}\).

### 3.4 Soundness

In order to preserve the validity of previous proofs, it is crucial for incremental reasoning to preserve the declared specifications for inherited methods: for a specification \((p, q)\) included in \(S(C, B.m)\) it is safe to rely on \((p, q)\) when \(body(B, m)\) is executed on an instance of \(C\) or a subclass of \(C\). With the open world assumption the subclasses of \(C\) are unknown when \(C\) is analyzed, so soundness is ensured by tracking the requirements that \((p, q)\) imposes on late bound calls in \(body(B, m)\). If a method \(n\) is overridden in a class \(D\) below \(C\), all requirements towards \(n\) made by classes above \(D\) must be satisfied by \(body(D, n)\). This is expressed by \(S(D, D.n) \rightarrow R(D, n)\), where \(R(D, n)\) is defined as the union of all requirements towards \(n\) made above \(D\); i.e., the union of \(R(C, B#n)\) for all \(D \leq C \leq B\).

In general soundness means that if \(body(B, m)\) is executed on an instance of \(D\), it must be safe to rely on the specifications in the set \(S(D, B.m)\), which is defined as the union of \(S(C, B.m)\) for all classes \(C\) where \(D \leq C \leq B\). Soundness is formalized by the following definition of sound proof environments and **Lemma 1.** Let \(e : C.m\) denote the external call \(e.m\) where \(C\) is the type of \(e\).
Definition 4 (Sound environments). Let $B, C, D : Cid$ and $m, n : Mid$. A sound environment $\mathcal{E} : Env$ satisfies the following two conditions for all $B, C \in \mathcal{E}$ and $m$:

1. $\forall (p, q) \in S_\mathcal{E}(C, B, m). \exists \Omega. B \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q) \land \text{Internal}_{\mathcal{E}}(C, B, O) \land \text{External}_{\mathcal{E}}(O) \land \text{Static}_{\mathcal{E}}(C, O)$

2. $m \in \text{C.mtds} \Rightarrow S_\mathcal{E}(C, C, m) \rightarrow R_\mathcal{E} \uparrow(C, m)$

where $\text{Internal}_{\mathcal{E}}(C, B, O) \triangleq \forall ((r) \in : n(\mathcal{E}) \{|s\}|) \in O. \forall D \in \mathcal{E}. S_\mathcal{E}(D, bind_{\mathcal{E}}(D, B\#n).n) \rightarrow (r', s')$

$\text{External}_{\mathcal{E}}(O) \triangleq \forall ((r) \in : e : D.n(\mathcal{E}) \{|s\}|) \in O. S_\mathcal{E}(D, bind_{\mathcal{E}}(D, D\#n).n) \rightarrow (r', s') \land R_\mathcal{E}(D, n) \rightarrow (r', s')$

$\text{Static}_{\mathcal{E}}(C, O) \triangleq \forall ((r) \in : n(\mathcal{E}) \{|s\}|) \in O. S_\mathcal{E}(C, bind_{\mathcal{E}}(B, B\#n).n) \rightarrow (r', s')$

where $r' = (r[Z_\mathcal{E}/Z] \land \lambda = \lambda[Z_\mathcal{E}/Z])$, and $s' = s[\text{return}/v][Z_\mathcal{E}/Z]$, and $\lambda$ are the formal parameters of $n$.\footnote{Remark that in the case of external calls where $r, s$ range over this, we also need to replace this with a fresh name in $r'$ and $s'$.}

Remark that in the case of external calls where $r, s$ range over this, we also need to replace this with a fresh name in $r'$ and $s'$.

The soundness of a proof environment can be explained informally as follows: Assume that $(p, q) \in S_\mathcal{E}(C, B, m)$ and that there is a proof outline $O$ of $\text{body}(B, m)$ for $(p, q)$. For each internal call $(r) n \{s\}$ in $O$ and for each subclass $D$ of $C$, the requirement $(r', s')$ must follow from the specifications of the method definition to which a call is bound for search class $D$. The assertion pair $(r', s')$ is derived from $(r, s)$ as in Definition 4. For each external call $(r) e : E.n \{s\}$ in $O$, the assertion pair $(r', s')$ must follow from the specification of the method provided by $E$, and it must be imposed on redefinitions below this type. (Remember that by the healthy binding strategy, the external call will bind to $n$ as found above $E$ or to a redefinition below $E$.) For each static call $(r) \in \mathcal{A} \{s\}$ in $O$, the assertion pair $(r', s')$ must follow from the specification of the method implementation to which the call will bind. The assertion pair is not imposed on method overriding since the call is bound at compile time.

Let $\models_\mathcal{E} (p) t [q]$ denote $\models (p) t [q]$ provided that late bound internal calls in $t$ are bound for search class $C$, and let $\models_\mathcal{E} m(\lambda) : (p)(q)t$ be given by $\models (p)(q)t [q]$. If $t$ is without calls and $\vdash_{PL} (p) t [q]$, then $\models (p) t [q]$ follows by the soundness of PL. Lemma 1 states that if $(p, q) \in S_\mathcal{E}(C, B, m)$ and $\text{body}(B, m)$ is executed in an instance of a subclass $D$ of $C$, a sound environment guarantees that $(p, q)$ is a valid specification. The proof of this lemma can be found in Appendix A3.

Lemma 1. Assume a sound environment $\mathcal{E} : Env$ and a sound program logic PL. Let $B, D : Cid$, $m : Mid$, and $(p, q) : \text{APair}$ such that $B, D \in \mathcal{E}$ and $(p, q) \in S_\mathcal{E}(D, B, m)$. Then $\models_{D, m}(\lambda) : (p)(q)\{\text{body}(B, m)\}$.

4. The inference system for incremental reasoning

In this section, the incremental reasoning strategy outlined above is formalized as a calculus $\text{LBS(PL)}$. Given a sound program logic PL, the calculus builds a proof environment which reflects the class hierarchy and captures method specifications and requirements. Initially, the environment is empty and the class hierarchy is analyzed in a top-down manner, starting with classes without superclasses. During class analysis, the proof environment is extended in order to keep track of the currently analyzed class hierarchy and the associated method specifications and requirements. Each class is analyzed after all of its subclasses, based on the environment resulting from the analysis of previous classes. Establishing a proof outline for one method body at a given stage of the overall analysis gives rise to (further) proof-obligations, which are tracked by the proof system. The system itself is formalized as a set of derivation rules (cf. Section 4.1), whose traversal through the class hierarchy is driven by a list of analysis operations.

4.1. The inference system LBS(PL)

An open program may be extended with new classes, and there may be mutual dependencies between these classes. For example, a method in a new class $C$ can call a method in another new class $E$, and a method in $E$ can call a method in $C$. In such cases, a complete analysis of one class cannot be carried out without consideration of mutually dependent classes. We therefore choose modules as the granularity of program analysis, where a module consists of a set of classes. Such a module is self-contained with respect to an environment $\mathcal{E}$ if all method calls inside the module can be successfully bound inside that module or to classes represented in $\mathcal{E}$.

In the calculus, the judgments are of the form $\mathcal{E} \vdash A$, where $\mathcal{E} : Env$ is a proof environment and $A$ is a list of analysis operations. The syntax for analysis operations is summarized in Fig. 5, and the different operations are explained below. For convenience, we let $\mathcal{E}$ denote both a list and set of classes. Let $\text{LBS(PL)}$ denote the reasoning system for lazy behavioral subtyping based on a sound program logic PL, which uses a proof environment $\mathcal{E} : Env$ and the inference rules given in Figs. 6 and 8. $\text{LBS(PL)}$ contains in addition structural rules for flattening $\mathcal{O}$ operations on their last argument and for discarding operations with empty arguments. These rules can be found in Appendix A2. The rules are read from bottom to top, e.g., application of rule NewModule in Fig. 6 on the judgement $\mathcal{E} \vdash \text{module}(\mathcal{E})$ leads to the judgement $\mathcal{E} \vdash [e ; \mathcal{E}]$.\footnote{Please cite this article in press as: J. Dovland et al., Incremental reasoning with lazy behavioral subtyping for multiple inheritance, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.09.006}
\[ E \vdash [C : \varnothing] \cdot M \]
\[ E \vdash [C : \text{anCalls}(D, O) \cdot \theta] : [\mathcal{L}] \cdot M \]
\[ E \vdash [\text{verify}(D, m, (p, q)) \cdot \theta] : [\mathcal{L}] \cdot M \]
\[ E \vdash \text{bind}_x(C, D, m) \rightarrow (p, q) \]
\[ E \vdash [\text{verify}(E, m, (r', s')) \cdot \theta] : [\mathcal{L}] \cdot M \]
\[ E \vdash [\text{bind}_y(C, D, m) \rightarrow (p, q)] \]
\[ E \vdash [\text{bind}_z(C, D, m) \rightarrow (p, q)] \]
\[ E \vdash [\text{bind}_w(C, D, m) \rightarrow (p, q)] \]

**Fig. 5.** Syntax for the analysis operations. Here, \( M, M_0, L \) are as in Fig. 5. \( R \) is a set of assertion pairs, and \( O \) is a statement decorated with pre- and post conditions to method calls.

**Fig. 6.** The inference system. In the call rules, we have as before that \( r' = (r|\exists_0|\exists) \land S = \exists(x|S_0|\exists), \) and \( s' = s[\text{return}/v]|\exists_0|\exists, \) where \( \exists \) are the formal parameters of \( n, \) and \( x \) are the local variables of the calling method.

Environment updates are formalized by the operator \( \_ \oplus _\_ : \text{Env} \times \text{Update} \rightarrow \text{Env}, \) where the second argument represents the update. There are three different environment updates; extending the environment with a new class and extending the specifcations or the requirements of a method in a class. The updates are defined as follows, where the notation \( M[A \mapsto B] \) means the mapping \( M \) where \( A \) maps to \( B : \)

\[ E \oplus \text{ext}(C, D, f, M) = (L \mid C \mapsto (D, f, M), S_0, R_0) \]

A new module \( \text{module}(\bar{L}) \) is initiated for analysis by the rule \( \text{NewModule} \) given in Fig. 6. This rule generates an operation of the form \( [\epsilon : \mathcal{L}], \) where \( \mathcal{L} \) is initialized to \( \bar{L}. \) The module is analyzed by analyzing the classes in \( \bar{L} \) one by one; whenever a class is selected for analysis, the analysis of this class is completed before a new class from \( \bar{L} \) is selected. The set \( \mathcal{L} \) may contain class definitions and require operations as indicated by the production for \( \mathcal{L} \) in Fig. 5. During module analysis, the set \( \mathcal{L} \) contains the unanalyzed classes of \( \bar{L} \) and require operation that originate from the analysis of external calls in the analyzed classes, as explained below. The rules of \( \text{LBS(PL)} \) ensure that the analysis of this module is completed before the remaining modules are considered.
The rule \texttt{NewClass} selects a class \texttt{C extends D \{\}} \texttt{M} from the current module, and removes this class from the set of unanalyzed classes. The premise \texttt{C \not \in E} ensures that the class has not been previously analyzed, and the premise \texttt{D \in E} ensures that \texttt{C} is analyzed after all of its superclasses.

This rule generates an operation of the form \texttt{\{C : \Theta\} : \mathcal{L}}), where the operations \texttt{\Theta} are analyzed in the context of class \texttt{C} before other classes in the module are analyzed. The selection of a class \texttt{C} initiates three analysis operations in the context of \texttt{C}. The operation \texttt{anSpec(\overline{M})} initiates the analysis of the different method specifications given in \texttt{C}; the operation \texttt{anReq(\overline{M})} initiates the analysis of the requirements imposed by the superclasses of \texttt{C} on method definitions in \texttt{C}. The operation \texttt{supCls(\overline{E})} initiates the analysis of delayed requirements, as described below.

For each method definition \texttt{m(\overline{x})(t)(\overline{m}) in \overline{M}} flattening of \texttt{anReq(\overline{M})} leads to an operation \texttt{anReq(m(\overline{x})(t))}. This operation is analyzed by rule \texttt{NewMDT}, leading to an operation \texttt{verify(\texttt{C, m, }R_t^{\uparrow \{C.\text{in}\text{h}, \text{m}\})}}, where \texttt{R_t^{\uparrow \{C.\text{in}\text{h}, \text{m}\}}} contains the requirements towards \texttt{m} that are imposed by superclasses of \texttt{C}. For each method specification \texttt{mID(x) : (p, q)} contained in \texttt{\overline{M}}, application of rule \texttt{NewSpec} leads to an operation \texttt{verify(bind(D, D\#m), m, (p, q))}. These verify operations are analyzed in the context of class \texttt{C}. In general, an operation \texttt{verify(B, n, (p, q))}, for some class \texttt{B \geq C} and method \texttt{n} defined in \texttt{B}, means that the assertion pair \texttt{(p, q)} will be analyzed for the implementation of \texttt{n} as found in class \texttt{B}.

The two operations \texttt{anReq(\overline{M})} and \texttt{anSpec(\overline{M})} generated by \texttt{NewClass} thereby ensures that: (1) If method \texttt{m} is defined in \texttt{C} and some superclass of \texttt{C} imposes some requirement \texttt{(r, s)} towards \texttt{m}, i.e., \texttt{m} overrides a superclass definition, then an operation \texttt{verify(C, m, (r, s))} is generated, and (2) for each method specification \texttt{mID(x) : (p, q)} given in the context of \texttt{C}, an operation \texttt{verify(bind(D, D\#m), m, (p, q))} is generated.

The generated verify operations are analyzed by either rule \texttt{ReqDER} or rule \texttt{ReqNOTDER}. For a method \texttt{m} defined in a class \texttt{B} above \texttt{C}, the set \texttt{\overline{\texttt{S}}(C, B.m)} is initially empty. This set can only be extended during the analysis of class \texttt{C}, by rule \texttt{ReqNOTDER}. This rule requires that if \texttt{\overline{\texttt{S}}(C, B.m)} is extended with \texttt{(p, q)}, then a new proof outline \texttt{O} must be provided for the body of \texttt{m} such that \texttt{O \vdash_p body(B, m)} : \texttt{(p, q)}. The analysis then continues by considering the decorated method body by means of an \texttt{anCalls(\overline{B}, O)} operation as described below. The assertion pair \texttt{(p, q)} then becomes a new specification of \texttt{B.m} in the context of \texttt{C}, and \texttt{(p, q)} is itself assumed when analyzing the method body. This captures the standard approach to reasoning about recursive method calls [20].

Consider next the analysis of some operation \texttt{verify(B, m, (p, q))}. The set \texttt{\overline{\texttt{S}}(C, B.m)} is incrementally extended during the analysis of \texttt{C}, and it might therefore be the case that \texttt{(p, q)} follows by entailment from the already verified assertion pairs for this method, i.e., \texttt{\overline{\texttt{S}}(C, B.m) \rightarrow (p, q)}. In this case, no further analysis of \texttt{(p, q)} is needed, and the operation is then discarded by rule \texttt{ReqDER}. Otherwise, a proof outline must be provided for the method body, and the operation is verified by rule \texttt{ReqNOTDER} as described above. In general, \texttt{B} may be a superclass of \texttt{C}, which means that \texttt{(p, q)} may follow from already verified assertion pairs in superclasses. In rule \texttt{ReqDER}, this is captured by the relation \texttt{\overline{\texttt{S}}(C, B.m) \rightarrow (p, q)}. The definition of \texttt{\overline{\texttt{S}}(C, B.m)} can be found in Appendix A.1, note that \texttt{\overline{\texttt{S}}(C, B.m)} reduces to \texttt{\overline{\texttt{S}}(C, B.m)} when \texttt{B = C}.

Next we consider an operation \texttt{anCalls(\overline{B}, O)} generated by \texttt{ReqNOTDER}. Here, \texttt{O} is a proof outline for the body of some method in class \texttt{B}, where method call statements are decorated with pre- and postconditions. By the structural rules of Appendix A.2, we assume that \texttt{\overline{\texttt{anCalls(\overline{C}, t_1; t_2)}}} is decomposed to \texttt{\overline{\texttt{anCalls(\overline{C}, t_1), anCalls(\overline{C}, t_2)}}}, and that \texttt{anCalls(C, t)} is discarded if there are no call statements in \texttt{t}. The remaining statements are analyzed by rules \texttt{Internal}, \texttt{Static}, and \texttt{External}.

An internal late bound call \texttt{r} \texttt{v := n(\overline{\theta}) (s)} is handled by rule \texttt{Internal}. Two steps are taken for the analysis of the requirement to the call, where \texttt{(r', s')} is given by \texttt{(r, s)} as defined in Fig. 6:

1. An operation \texttt{verify(E, n, (r', s'))} is generated, where \texttt{E} the class that the call will bind to for search class \texttt{C}, i.e., \texttt{E = bind(C, B\#n)}.
2. \texttt{(r', s')} is included in \texttt{R(C, B\#n)}.

For the generated verify operation, \texttt{(r', s')} either follows from the already established assertion pairs \texttt{\overline{\texttt{S}}(C, E.n)}, which means that the requirement can be discarded by rule \texttt{ReqDER}. Otherwise, the operation is analyzed by rule \texttt{ReqNOTDER}, requiring that a proof outline \texttt{O'} is provided such that \texttt{O' \vdash_p body(E, n) : (r', s'))}. This proof outline is analyzed in the same manner as the original proof outline for \texttt{m}. This adds \texttt{(r', s')} to \texttt{S(C, E.n)}, trivializing the proof of \texttt{\overline{\texttt{S}}(C, E.n) \rightarrow (r', s')}.

The extension of the \texttt{R} mapping in step 2 ensures that future redefinitions of \texttt{n} must respect the new requirement \texttt{(r', s')}, i.e., the requirement is imposed whenever redefinitions are considered by \texttt{NewMDT}.

A static call \texttt{r} \texttt{v := n@B(\overline{\theta}) (s)} is handled by rule \texttt{Static}. The verify operation generated by this rule ensures that \texttt{(r', s')} follows from the specification of the method that the call will bind to. Note that this rule does not extend the \texttt{R} mapping, since the call is bound at compile time.

External calls \texttt{r} \texttt{v := e.n(\overline{\theta}) (s)}, with \texttt{e : E}, are analyzed by rule \texttt{External}. The assertion pair \texttt{(r', s')} is lifted outside the context of the analyzed class by this operation, and shifted into the set \texttt{\mathcal{L}} of analysis operations as a \texttt{require} operation. Rule \texttt{extReq} analyzes a \texttt{require} operation, and can be applied after the initial analysis of the declared class of the callee. Rule \texttt{extReq} ensures that \texttt{(r', s')} follows from the specification of \texttt{n} given by the static type \texttt{E} of \texttt{e}, and it is imposed on redefinitions of \texttt{n} below \texttt{E}.

The \texttt{supCls(\overline{E})} operation that is generated by rule \texttt{NewClass} is used to take special care of the case where the new class \texttt{C} introduces a \texttt{diamond} in the class hierarchy. An example of a small class hierarchy with diamond inheritance can be found in Fig. 7. Here, class \texttt{C} extends two different classes \texttt{B_1} and \texttt{B_2}, and both these classes inherit class \texttt{E}. We then say that \texttt{E}
is a common superclass of C. In general, we say that C introduces a diamond with common superclass E if there exist two different classes in C.inh and E is above both these classes. There may be more than one common superclass of C, and we let their union be denoted by commSup(C). A formal definition of commSup can be found in Appendix A.1.

The purpose of the supCls(E) operation, where E is commSup(C) as given by rule NEWCLASS, is explained by considering the class hierarchy in Fig. 7. Consider first the situation before introduction of class C. Assume that the class E implements two methods m and n, and that there is an internal late bound call to n in the body of m. Assume furthermore that none of these methods are overridden in B₂, and that some specification (p, q) of m is given in the context of this class. As explained above, a proof outline for body(E, m) is then analyzed in the context of class B₂ which leads to an inclusion of (p, q) in S(B₂, E, m) by rule ReqNotDer. Due to the internal late bound call to n in the body of m, some requirement (r, s) is recorded in R(B₂, E, n).

We assume that the analysis of this requirement succeeds for B₂, i.e., S₁(B₂, E, n) ⊇ (r, s). Assume next that n is overridden in class B₁. As B₁ and B₂ are two unrelated subclasses of E, they can be analyzed in any order, and the requirements imposed by one of them are not imposed on the other. Especially, requirements contained in R(B₂, E, n) are not imposed on n in E as a method call made on an instance of B₂ cannot be bound to a definition in class E. Consider next the analysis of class C, assuming that the method n is not overridden by C. When m is executed on an instance of C, the internal late bound call to n is bound to the definition in B₁, i.e., bind(C, E, n) = B₁. In order to rely on the already verified specification S(B₂, E, m) of m, the analysis of C must ensure that (r, s) follows from the specification for B₁, n. In this manner, the verification of R(B₂, E, n) is delayed until a method call with these requirements can actually be bound to B₁, n. We then refer to R(B₂, E, n) as a set of delayed requirements. In a more generalized setting, the same argument applies to all such requirements made by any class between C and the common superclass E. For a common superclass E of C and method n called internally in E, the function delReq(C, E, n) returns the union of the delayed requirements. The definition of this function can be found in Appendix A.1. The rules for analyzing the supCls(E) operation are displayed in Fig. 8. If C does not introduce any diamonds, this operation is discarded as the argument will be empty. Otherwise, supMtd generates a supMtd operation for each common superclass. For each method called by a common superclass and not overridden by C, supReq generates a verify operation for the delayed requirements of these calls. The delayed requirements are thereby verified with regard to the implementations that the call will bind to in the context of C.

By the successful analysis of class C, an operation on the form [(C : e) : L] is reached, and by application of rule EmpClass, this yields the operation [e : L]. Another class in L can then be enabled for analysis. The analysis of a module is completed by rule EmpModule. Thus, the analysis of a module is completed after the analysis of the module classes and the require operations generated by the analysis of external calls in these classes have succeeded. Note that a successful analysis of E ⊢ module(Ī) has exactly one leaf node E′ ⊢ [e : Ω], and we call E′ the environment resulting from the analysis of module(Ī).

The analysis of a program is initiated by the judgment E₀ ⊢ module(Ī), where Ī is a set of classes that are self-contained with respect to the empty environment. Subsequent modules are analyzed in sequential order, such that each module is self-contained with respect to the environment resulting from the analysis of previous modules. When the analysis of a module is completed, the resulting environment represents a verified class hierarchy. New modules may introduce subclasses of classes which have been analyzed in previous modules. The calculus is based on an open world assumption in the sense that a module is analyzed in the context of previously analyzed modules, independently of subsequent modules.

With lazy behavioral subtyping, a programmer typically provides S specifications for each class B. Their verification generates R requirements for the internal late bound calls occurring in B, which are imposed on subclass redefinitions of the called methods. In a subclass C, a redefined method m can violate the S specifications of a superclass, but not the R requirements. Note that behavioral subtyping is not implied by this approach. Still, lazy behavioral subtyping supports incremental reasoning under an open world assumption. Class C may provide additional specifications for inherited methods, resulting in additional verification of such methods, which may generate additional R requirements for the future subclasses of C. This means that unrelated subclasses of B may have different R requirements to the same method.
4.2. Soundness of LBS(PL)

The following theorem establishes the soundness of the inference rules, the proof can be found in Appendix A.4.

**Theorem 1.** Let $\mathcal{E} : \text{Env}$ be a sound environment and $\mathcal{L}$ a set of class definitions. If a proof of $\mathcal{E} \vdash \text{module}(\mathcal{L})$ in LBS(PL) has $\mathcal{E}'$ as its resulting environment, then $\mathcal{E}'$ is also sound.

By Lemma 1 and Theorem 1 above, we conclude this section with the following soundness theorem:

**Theorem 2 (Soundness).** If PL is a sound program logic, then LBS(PL) constitutes a sound proof system, in the sense that the environment resulting from the analysis of a program is sound.

**Proof.** In LBS(PL), a program is analyzed as a sequence of module operations. It follows directly from the definition of sound environments that the empty environment is sound. Theorem 1 and Lemma 1 guarantee that the environment remains sound during the analysis of class modules. $\square$

**Example 5.** We consider the analysis of the classes in Fig. 3. To keep the notation below compact, we sometimes use $A$, $AA$, $FA$, and $WA$ as abbreviations for $\text{Account}$, $\text{AuthAccount}$, $\text{FeeAccount}$, and $\text{MyAccount}$ respectively. Let $\mathcal{L}$ be the classes in Fig. 4, and assume that the analysis of these classes starts with an empty proof environment, i.e., the judgement $\mathcal{E}_0 \vdash \text{module}(\mathcal{L})$ is analyzed. Application of rule $\text{NewModule}$ then leads to $\mathcal{E}_0 \vdash [\epsilon ; \mathcal{L}]$. Let $\text{Auth}$ be the first class that is selected for analysis by rule $\text{NewClass}$.

**Analysis of $\text{Auth}$.** When applying rule $\text{NewClass}$, we arrive at the judgement

$$\mathcal{E}_1 \vdash [(\text{Auth} : \text{anSpec}(\text{validate}@\text{Auth}(a) : (\text{true, return} = (a = a1 \lor a = a2)))) ; \mathcal{L}]$$

where $\mathcal{E}_1$ is $\mathcal{E}_0$ extended with the declaration of $\text{Auth}$, and $\mathcal{L}$ are the remaining classes in Fig. 4. For brevity, we here focus on the specification of method $\text{validate}$, as declared in Fig. 4. We ignore the $\text{anReq}$ and $\text{supCls}$ operations generated by $\text{NewClass}$ since $\text{Auth}$ is without superclasses. For the generated $\text{anSpec}$ operation, rule $\text{NewSpec}$ leads to an operation $\text{verify}(\text{Auth}, \text{validate}, (\text{true, return} = (a = a1 \lor a = a2)))$. This operation is analyzed by rule $\text{ReqNotDer}$, leading to the judgement $\mathcal{E}_2 \vdash [(\text{Auth} : \epsilon) ; \mathcal{L}]$, where

$$\mathcal{E}_2 = \mathcal{E}_1 \oplus \text{extS}(\text{Auth, Auth, validate}, (\text{true, return} = (a = a1 \lor a = a2)))$$

Thus, the specification is included in $\mathcal{S}_2(\text{Auth, Auth, validate})$. The proof outline for the method body is trivial, and we ignore the generated $\text{anCalls}$ operation since there are no method calls in the method body. By application of rule $\text{EmpClass}$, we then arrive at $\mathcal{E}_3 \vdash [\epsilon ; \mathcal{L}]$. The only rule that can be applied now is $\text{NewClass}$, and this rule will select class $\text{Account}$ for analysis.

**Analysis of $\text{Account}$.** As for $\text{Auth}$, we ignore the $\text{anReq}$ and $\text{supCls}$ operations generated by $\text{NewClass}$. The rule $\text{NewClass}$ thereby leads to the judgement

$$\mathcal{E}_3 \vdash [(\text{A : \text{anSpec}(\text{\overline{A}})}) ; \mathcal{L}]$$

where $\overline{A}$ is the specification of $\text{update, validate, and withdraw}$ as given in the context of class $\text{Account}$. $\mathcal{E}_3$ extends the class mapping of $\mathcal{E}_2$ with $\text{Account}$, and $\mathcal{L}$ is the remaining class $\text{AuthAccount}$ of the original module. Remember that $\overline{A}$ is used as shorthand for $\text{Account}$.

The specifications of methods $\text{update}$ and $\text{validate}$ are analyzed as described above for the $\text{validate}$ method of $\text{Auth}$. After analysis of these two methods, we arrive at the following judgement:

$$\mathcal{E}_4 \vdash [(\text{A : \text{anSpec}(\text{withdraw}@\text{A}(id, x) : (\text{bal} = b0 \land \text{owner} = id, \text{bal} = b0 - x)}) ; \mathcal{L}]$$

where

- $\mathcal{E}_4 = \mathcal{E}_3 \oplus \text{extS}(\text{A, A, update}, (\text{bal} = b0, \text{bal} = b0 + y))$
- $\oplus \text{extS}(\text{A, A, validate}, (\text{true, return} = (\text{owner} = id)))$

The specification of $\text{withdraw}$ can be verified for the method body by the following proof outline:

$$\text{if } v \text{ then } \{ \text{bal} = b0, \text{v} = \text{true} \} \{ \text{bal} = b0 - x \} \text{ fi}$$

By application of $\text{NewSpec}$ and $\text{ReqNotDer}$ followed by decomposition of the proof outline, we thereby reach the following judgement:

$$\mathcal{E}_5 \vdash [(\text{A : \text{anCalls}(\text{A, } \{ \text{bal} = b0 \land \text{owner} = id \} \text{v} : = \text{validate}(\text{id}) \{ \text{bal} = b0 \land v = \text{true} \}) \text{ anCalls}(\text{A, } \{ \text{bal} = b0 \} \text{update}(\text{-x}) \{ \text{bal} = b0 - x \}) ; \mathcal{L}]$$

where

- $\mathcal{E}_5 = \mathcal{E}_4 \oplus \text{extS}(\text{A, A, withdraw}, (\text{bal} = b0 \land \text{owner} = id, \text{bal} = b0 - x))$

The two $\text{anCalls}$ operations are analyzed by rule $\text{Internal}$. In each case, the generated $\text{verify}$ operation follows by entailment from the specification of the called method, and is discarded by rule $\text{ReqDer}$. (For brevity we ignore the trivial specification of $\text{validate}$, expressing that $\text{bal}$ is not changed by the method.) After application of rule $\text{EmpClass}$, we then arrive at the
\[ S_6(\text{Auth, Auth} \text{. validate}) = (\text{true, return} = (a = a1 \lor a = a2)) \]

\[ S_6(\text{A, A} \text{. update}) = (\text{bal} = b_0, \text{bal} = b_0 + y) \]

\[ S_6(\text{A, A} \text{. validate}) = (\text{true, return} = (\text{owner} = \text{id})) \]

\[ R_6(\text{A, A} \# \text{update}) = (\text{bal} = b_0 \land y = -x, \text{bal} = b_0 - x) \]

\[ R_6(\text{A, A} \# \text{validate}) = (\text{bal} = b_0 \land \text{owner} = \text{id, bal} = b_0 \land \text{return} = \text{true}) \]

Fig. 9. The non-empty S and R sets of proof environment \( \varepsilon_6 \). In addition, the L mapping of \( \varepsilon_6 \) contains the implementation of the two classes Auth and Account.

following judgement:

\[ \varepsilon_6 \vdash [\varepsilon ; L] \quad (1) \]

where

\[ \varepsilon_6 = \varepsilon_5 \oplus \text{extR}(\.A, \text{update, (bal} = b_0 \land y = -x, \text{bal} = b_0 - x)) \]

\[ \oplus \text{extR}(\.A, \text{. validate, (bal} = b_0 \land \text{owner} = \text{id, bal} = b_0 \land \text{return} = \text{true}) \]

Then non-empty S and R sets of environment \( \varepsilon_6 \) are displayed in Fig. 9.

Analysis of AuthAccount. By application of NewClass to Eq. (1), we arrive at the judgement

\[ \varepsilon_7 \vdash ([\.AA: \text{anSpec}(\overline{\varepsilon_6}) \cdot \text{anReq}(\overline{\varepsilon_6}) \cdot \text{supCl}(\emptyset)); \emptyset] \]

where \( \varepsilon_7 \) extends \( \varepsilon_6 \) with the implementation of AuthAccount. \( \overline{\varepsilon_6} \) contains the class specifications, and \( \overline{\varepsilon_6} \) contains the methods defined by the class. Since there are no common superclasses, we hereafter ignore the supCl operation. For method validate, the user given specification leads to the operation

\[ \text{verify}(\.AA, \text{validate, (true, return} = (\text{owner} = \text{id} \lor \text{a1} = \text{id} \lor \text{a2} = \text{id})) \]

which is verified by rule ReqNotDer. The method body can be analyzed by the following proof outline:

\[
\begin{align*}
\{\text{true}\} \ & \ r := \text{validate\.Account(id)} \ \{r = (\text{owner} = \text{id})\} \\
\text{if} \ (r) \ & \ \text{then} \ \{\text{owner} \neq \text{id}\} \ \\
\text{validate\.Account(id)} \ & \ \{\text{owner} \neq \text{id} \land r = (\text{a1} = \text{id} \lor \text{a2} = \text{id})\} \\
\text{fi} \ & \ \{r = (\text{owner} = \text{id} \lor \text{a1} = \text{id} \lor \text{a2} = \text{id})\}; \text{return} \ r
\end{align*}
\]

The pre- and postconditions for the two static calls follow by entailment of the corresponding superclass specification of validate (assuming the trivial specification that validate in Auth do not modify owner). Since the calls are static, none of them leads to an extension of the R mapping. By application of rule ReqNotDer followed by analysis of the proof outline, we thereby arrive at the judgement

\[ \varepsilon_8 \vdash ([\.AA: \text{anSpec}(\overline{\varepsilon_6}) \cdot \text{anReq}(\overline{\varepsilon_6})]; \emptyset] \]

where \( \overline{\varepsilon_6} \) is the user given specification of withdraw, \( \overline{\varepsilon_6} \) is as above, and

\[ \varepsilon_8 = \varepsilon_7 \oplus \text{extS}(\.AA, \text{. validate, (true, return} = (\text{owner} = \text{id} \lor \text{a1} = \text{id} \lor \text{a2} = \text{id})) \]

For method withdraw, the given specification leads to the following operation, which is verified by rule ReqNotDer:

\[ \text{verify}(\.A, \text{withdraw, (bal} = b_0 \land (\text{id} = \text{a1} \lor \text{id} = \text{a2}), \text{bal} = b_0 - x)) \]

The specification can be verified by the following proof outline:

\[
\begin{align*}
\{\text{bal} = b_0 \land (\text{a1} = \text{id} \lor \text{a2} = \text{id})\} \\
\text{v := validate(id)} \ & \ \{\text{bal} = b_0 \land \text{v} = \text{true}\} \\
\text{if} \ v \ & \ \{\text{bal} = b_0\} \text{update(-x)} \{\text{bal} = b_0 - x\} \ \text{fi}
\end{align*}
\]

The internal late bound call to validate leads to the requirement \( (\text{bal} = b_0 \land (\text{id} = \text{a1} \lor \text{id} = \text{a2}) , \text{bal} = b_0 \land \text{return} = \text{true}) \) which is included in \( R(\.AA, \text{. validate}) \). The requirement follows from \( S(\.AA, \text{. validate}) \) by entailment. Correspondingly, the internal late bound call to update leads to the requirement \( (\text{bal} = b_0 \land y = -x, \text{bal} = b_0 - x) \) which is included in \( R(\.AA, \text{. update}) \). For class AuthAccount, this call binds to the inherited implementation of update, and the requirement follows from the superclass specification of this method. The successful analysis of the verify operation above thereby leads to the judgement:

\[ \varepsilon_9 \vdash ([\.AA: \text{anReq}(\overline{\varepsilon_6})]; \emptyset] \]

where

\[ \varepsilon_9 = \varepsilon_8 \oplus \text{extS}(\.AA, \text{. withdraw, (bal} = b_0 \land (\text{id} = \text{a1} \lor \text{id} = \text{a2}), \text{bal} = b_0 - x)) \]

\[ \oplus \text{extR}(\.AA, \text{. validate, (bal} = b_0 \land (\text{a1} = \text{id} \lor \text{a2} = \text{id}), \text{bal} = b_0 \land \text{return} = \text{true}) \]

\[ \oplus \text{extR}(\.AA, \text{. update, (bal} = b_0 \land y = -x, \text{bal} = b_0 - x)) \]

It then remains to analyze the anReq operation. Since validate is the only method defined in AuthAccount, it suffices to consider this method by rule NewMtd. This rule generates a verify operation for the inherited requirements towards validate. The only requirement imposed by superclasses of AuthAccount is \( R(\.AA, \text{. validate}) \) as displayed in Fig. 9, which means


that we arrive at the judgement:
\[ E_9 \vdash \{ \text{AA: verify} (\text{AA, validate}, R_{E_9} (\text{AA}, \# \text{validate})) \} : \emptyset \]
The verification of this operation succeeds by \textit{ReqDer}, since the requirement follows by entailment from the specification of \textit{validate} in \textit{AuthAccount}. The analysis of \textit{AuthAccount} is then completed by rule \textit{EmpClass}. Since this was the last class of the original module operation, analysis of the module is completed by rule \textit{EmpModule}. Environment \( E_9 \) is thereby the environment resulting from the analysis of the module. The non-empty specification and requirement sets of \( E_9 \) are summarized in Fig. 10. As a further illustration of the proof system, we consider the implementation in Fig. 11. This figure provides an implementation of the two remaining classes \textit{FeeAccount} and \textit{MyAccount} of Fig. 3. Assume that a module operation consisting of these two classes are analyzed based on the environment resulting from the analysis of the previous classes, i.e., the judgement
\[ E_9 \vdash \text{module}(I_1) \]
is analyzed, where \( I_1 \) denotes the classes in Fig. 11. Class \textit{FeeAccount} is analyzed before \textit{MyAccount}.

\textbf{Analysis of FeeAccount.} By application of \textit{NewModule} and \textit{NewClass}, we arrive at the judgement
\[ E_{10} \vdash \{ \text{FA: anSpec} (M_S) \cdot \text{anReq}(M) \} ; L_1 \]
where \( M_S \) is the specification of \textit{withdraw}, \( M \) is method \textit{update}, and \( L_1 \) is the remaining class \textit{MyAccount}. We ignore operation \textit{supCls} since \textit{FeeAccount} has no common superclasses, and the \textit{anReq} operation generated for method \textit{monthly} since there are no superclass requirements towards this method. By rule \textit{NewSpec}, the \textit{anSpec} operation leads to the operation
\[ \text{verify} (\text{A, withdraw}, (x > 0 \land \text{accFee} = a_0 \land \text{id} = \text{owner}, \text{accFee} = a_0 + \text{fee})) \]
which is verified by \textit{ReqNotDer}. We assume a proof outline for \textit{withdraw} with the requirement \((x > 0 \land \text{accFee} = a_0 \land \text{id} = \text{owner}, x > 0 \land \text{accFee} = a_0 \land \text{return} = \text{true})\) towards \textit{validate} and \((\text{accFee} = a_0 \land y < 0, \text{accFee} = a_0 + \text{fee})\) towards \textit{update}. The requirement towards \textit{validate} follows by entailment from the superclass specification of the method (assuming the trivial specification that \textit{accFee} is not modified), whereas the requirement towards \textit{update} needs verification since \textit{update} is overridden by \textit{FeeAccount}, and the new implementation of \textit{update} is left unspecified by the developer. This means that the specification set \( S(\text{FA, FA}, \text{update}) \) is extended with \((\text{accFee} = a_0 \land y < 0, \text{accFee} = a_0 + \text{fee})\) which is trivially verified over the method body. The successful analysis of the \textit{verify} operation thereby leads to the following judgement:
\[ E_{11} \vdash \{ \text{FA: anReq}(M) \} ; L_1 \]
where
\[ E_{11} = E_{10} \oplus \text{extS(FA, A, withdraw}, (x > 0 \land \text{accFee} = a_0 \land \text{id} = \text{owner}, \text{accFee} = a_0 + \text{fee})) \]
\[ \oplus \text{extR(FA, A, validate}, (x > 0 \land \text{accFee} = a_0 \land \text{id} = \text{owner}, x > 0 \land \text{accFee} = a_0 \land \text{return} = \text{true})) \]
\[ \oplus \text{extR(FA, A, update}, (\text{accFee} = a_0 \land y < 0, \text{accFee} = a_0 + \text{fee})) \]
\[ \oplus \text{extS(FA, A, update}, (\text{accFee} = a_0 \land y < 0, \text{accFee} = a_0 + \text{fee})) \]
For the \textit{anReq} operation, we need to consider requirement \( R(\text{A, A}, \# \text{update}) \) in Fig. 9, which is analyzed by a \textit{verify} operation in the context of class \textit{FeeAccount}. At this point in the analysis, the requirement does not follow from the specification of the overriding \textit{update} in \textit{FeeAccount}, and the \textit{verify} operation is analyzed by rule \textit{ReqNotDer}. In a proof outline for the method body, we may use the requirement itself as a pre/post specification for the static call, and the analysis of the proof outline.
Fig. 12. Syntax for the language \textit{MI}. The syntactic category \textit{P} of \textit{MI} (see Fig. 1) is extended with interfaces, and class definitions are extended with an \texttt{implements} clause. The other syntactic categories of the Fig. 1 remain unchanged. Here, \textit{I} denotes interface names of type \texttt{lid}.

thereby succeeds by rule \texttt{STATIC}. Since the requirement imposed by the superclass can be verified also for the subclass, we can rely on the original superclass specification of \texttt{withdraw} also when the method is executed on an instance of \texttt{FeeAccount}. The analysis of \texttt{FeeAccount} then leads to the following judgement:

\[ \mathcal{E}_{12} \vdash [\varepsilon ; L_1] \]

where

\[ \mathcal{E}_{12} = \mathcal{E}_{11} \otimes \text{extS}(\text{FA}, \text{FA}, \text{update}, (bal = b_0 \land y = -x, bal = b_0 - x)) \]

\textbf{Analysis of MyAccount}. In class \texttt{MyAccount}, the method \texttt{validate} is overridden in order to direct the internal late bound call in \texttt{Account} to the version found in \texttt{AuthAccount}. By selecting \texttt{MyAccount} for analysis, we arrive at the judgement:

\[ \mathcal{E}_{13} \vdash [(\text{MA} : \text{anReq}(V) \cdot \text{supCls}(\lambda)) : \emptyset] \]

where \textit{V} is the method \texttt{validate}. By rule \texttt{NewMtd}, this operation leads to a \texttt{verify} operation for each requirement imposed by the superclasses. As \texttt{validate} is implemented by a static call to \texttt{validate} in \texttt{AuthAccount}, the analysis of all these requirements succeed, and they are included in the set \texttt{S(MA, MA, validate)}. Let \( \mathcal{E}_{14} \) be the environment after this extension of the \texttt{S} mapping. Next we consider the \texttt{supCls} operation. By rule \texttt{SupMtd}, this operation leads to a \texttt{supMtd} operation for each method called internally by \texttt{Account}, but not overridden by \texttt{MyAccount}. The methods called by \texttt{Account} are \texttt{update} and \texttt{validate}. Since \texttt{validate} is overridden by \texttt{MyAccount}, we are left with the operation \texttt{supMtd}. This analysis is governed by the definition of \texttt{SupReq}. Since \texttt{bind} \((\text{MA, A#update}) = \text{FA}\), application of this rule gives the operation \texttt{verify} \((\text{FA, update, delReq}_{\text{FA}}(\text{AA, A#update}))\). By applying the definition of \texttt{delReq} as found in Appendix A.1, function \texttt{delReq}_{\text{FA}}(\text{AA, A#update}) evaluates to \texttt{R}_{\text{AA}}(\text{AA, A#update}). During the analysis of \texttt{AuthAccount}, the call \texttt{A#update} was bound to \texttt{Account}. However, as \texttt{MyAccount} introduces a diamond in the class hierarchy, the call is bound to \texttt{FeeAccount} for instances of \texttt{MyAccount}. In order to rely on the specification \texttt{S(MA, A, withdraw)} we therefore need to ensure that the requirement \texttt{R(FA, A#update)} holds for \texttt{update} as implemented by \texttt{FeeAccount}. The requirement follows by entailment from specification \texttt{S(MA, FA, update)}, which means that the \texttt{verify} operation succeeds by rule \texttt{ReqDer}. This concludes the verification of \texttt{module}(\texttt{L}), and \( \mathcal{E}_{14} \) is the resulting environment.

The analysis of imposed requirements means that when \texttt{withdraw} is executed on an instance of \texttt{MyAccount}, we may rely on the specifications of this method given by the different superclasses of \texttt{MyAccount}, i.e.

\[ \texttt{S(MA, A, withdraw)} \cup \texttt{S(AA, A, withdraw)} \cup \texttt{S(FA, A, withdraw)}. \]

\textbf{5. External specification by interfaces}

Whereas lazy behavioral subtyping provides a flexible framework for reasoning about open object-oriented programs. However, when reasoning about an external call \texttt{e.m(F)} with \texttt{e : E}, the pre/post assertion pair of the call must follow from the \texttt{R} requirements for \texttt{m} that have been established for class \texttt{E} (rules \texttt{EXTERNAL} and \texttt{extReq} in Fig. 6). As these \texttt{R} requirements are generated by the internal analysis, they may not provide a suitable basis for reasoning about external properties. One solution to this problem is to extend the \texttt{R} requirements of \texttt{E} by analysis of the external call. However, this can trigger new verification tasks, which make the approach less modular; it will be necessary to verify that the new pre/post assertion pair of the external call holds for \texttt{E} and all subclasses of \texttt{E}. Another approach is to let the programmer provide \texttt{R} requirements for methods which suffice for reasoning about external calls. This approach results in a more modular version of lazy behavioral subtyping in the sense that a class need only be analyzed once, but slightly restricts method redefinition.

In this section we use \textit{behavioral interfaces} as a means to specify and reason about external calls. A behavioral interface describes the visible methods of a class and their specifications, and inheritance may be used to form new interfaces from old ones. Behavioral interfaces are used to type object variables (references). Subtyping follows the inheritance hierarchy of interfaces, but need not follow the class hierarchy. A class definition explicitly declares which interface it implements.

This section develops this approach in terms of a language \textit{MI}, which extends \textit{MI}, in which objects and object references are typed by interfaces and each class implements a single interface. We develop the corresponding reasoning framework \texttt{LBSI(PL)} for reasoning about \textit{MI} programs, based on proof environments \texttt{IEEnv} which include interface information and, as before, an underlying sound program logic \texttt{PL}.

\textbf{5.1. A language with behavioral interfaces}

The programming language \textit{MI} extends \textit{MI} with interfaces and has the syntax given in Fig. 12. The syntax for classes is modified such that a class implements a single interface. An interface \textit{I} may extend a list \( I \) of superinterfaces, and
declare a set $\mathcal{E}$ of method signatures, where behavioral constraints are given as (pre, post) specifications. An interface may provide specifications of methods which are not found in its superinterfaces, and it may declare additional specifications for methods found in its superinterfaces. Interfaces form an inheritance hierarchy in which the relationship between interfaces is restricted to a form of behavioral subtyping; if $I'$ extends $I$, then $I'$ is a subtype of $I$ and $I$ is a supertype of $I'$. Thus, an interface may not declare method specifications that are in conflict with specifications declared by its superinterfaces. Let $\geq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces. Thus, $I' \geq I$ if $I'$ extends $I$ or if $I'$ (directly or indirectly) extends $I$. We say that an interface $I$ exports the methods which are declared in $I$ or inherited from the superinterfaces of $I$, with the associated constraints on method use.

A class $C$ implements the interface $I$ given by the implements clause in the class definition. All the methods exported by $I$ must be defined, satisfying the specifications of $I$. The analysis of the class must ensure that this requirement holds. Only the methods exported by $I$ are available for external invocations on references typed by $I$. The class may implement additional auxiliary methods for internal use. An instance of $C$ supports $I$ and all superinterfaces of $I$, ensuring that the object provides the methods exported in $I$ and adheres to the specifications imposed by $I$ on these methods. Objects of different classes may support the same interface, corresponding to different implementations of the interface behavior.

If $C$ implements $I$, some class $D$ may extend $C$ without implementing the behavior specified by $I$: if $D$ implements interface $J$, then $J \preceq I$ need not hold. If $J$ is not a subtype of $I$, the specifications declared by $I$ will not be imposed on $D$, and instances of $D$ will not support $I$. As a consequence, a subclass may reuse and redefine superclass methods within in the framework of lazy behavioral subtyping, since it is free to violate the interface specifications of its superclasses. In this manner, the type and class inheritance hierarchies are separated. Fields in $MIL$ are typed by interfaces; if an object supports $I$ (or a subtype of $I$) then the object may be referenced by a field $v$ typed by $I$, i.e., $v$ may refer to an instance of class $C$. However, if $J$ is not a subtype of $I$, the field may not refer to an instance of class $D$. Static type checking of an assignment $v := e$ must then ensure that the expression $e$ denotes an object supporting the declared interface of $v$. In this setting, the substitution principle for objects can be reformulated as follows:

For an object variable $v$ with declared interface $I$, the object that $v$ refers to at run-time will satisfy the behavioral specification $I$.

Reasoning about an external call $e.m(\mathcal{E})$ can then be based on the declared interface type of the object expression $e$; the interface hides the actual class of the object referred to by $e$. This simplifies external to simply check interface contracts, and require operations are no longer needed in the proof system. Observe that internal method calls $m(\mathcal{E})$ and $m@C(\mathcal{E})$ circumvent the interface mechanism, whereas $x.m(\mathcal{E})$ depends on the declared interface of $x$ (even if $x$ can be reduced to this).

5.2. The proof environment of LBSI(PL)

In LBSI(PL), a class name is bound to a tuple $(\mathcal{D}, \mathcal{I}, \mathcal{F}, \mathcal{M})$ of type IClass, and the interface implemented by a class is accessible by the observer function impl. An interface name is bound to a tuple $(\mathcal{I}, \mathcal{E})$ of type Interface, where the list of superinterfaces $\mathcal{I}$ and the method specifications $\mathcal{E}$ are accessible by the observer functions inh and specs, respectively. The proof environments of LBSI(PL) are defined as follows.

Definition 5 (Proof Environments with Interfaces). A proof environment $\mathcal{E}$ of type IEnv is a tuple $(L_\mathcal{E}, K_\mathcal{E}, S_\mathcal{E}, R_\mathcal{E})$ where $L_\mathcal{E} : \text{Ctd} \rightarrow \text{IClass}, K_\mathcal{E} : \text{lid} \rightarrow \text{Interface}$ are partial mappings and $S_\mathcal{E}, R_\mathcal{E} : \text{Ctd} \times \text{Ctd} \times \text{Mid} \rightarrow \text{Set(APair)}$ are total mappings. Below, we use the notation $S_\mathcal{E}(C, D, m)$ for $S_\mathcal{E}(C, D, m)$ and $R_\mathcal{E}(C, D#m)$ for $R_\mathcal{E}(C, D, m)$.

In LBSI(PL), $I \in \mathcal{E}$ denotes that $K_\mathcal{E}(I)$ is defined. We introduce some auxiliary functions, which are formally defined in Fig. 13. Let the function $\text{public}_\mathcal{E}(I)$ denote the set of method identifiers exported by $I$, so $m \in \text{public}_\mathcal{E}(I)$ if $m$ is declared in $I$ or inherited from a supertype of $I$. If $I' \preceq I$ then $\text{public}_\mathcal{E}(I) \subseteq \text{public}_\mathcal{E}(I')$, since a subtype can only add methods to those
of a supertype. Let the function \( \text{spec}_\xi(I, m) \) return a set of type \( \text{Set}[\text{APair}] \) with the behavioral constraints imposed on \( m \) by \( I \). Note that these constraints may stem from \( I \) or from a supertype of \( I \) and that a subinterface may provide additional specifications of methods inherited from superinterfaces. If \( m \in \text{public}_\xi(I) \) and \( I' \preceq I \), then \( \text{spec}_\xi(I, m) \subseteq \text{spec}_\xi(I', m) \).

The binding of method calls. The internal calls discard the interfaces implemented by the different classes, so the binding of internal late bound and static calls remain as defined in Section 2.4. For the binding of external calls, the definition from Section 2.4 is no longer suitable, since the binding is not restricted by the declared class of the callee in \( M\text{IL} \). However, to obtain a healthy binding strategy as explained in Section 2.4, an external call to \( m \) on an instance of class \( C \) will be bound by \( \text{bind}(C, C\#m) \). For class \( C \), the specifications of this implementation is given by \( S(I, \text{bind}(C, C\#m), m) \). For convenience, we let \( S(I, C, m) \) denote this set.

Sound environments. The definition of sound environments is revised to account for interfaces. In Condition 1 below, the pre/post assertion pair of an external call must now follow from the interface specification of the called object. Consider a pre/post assertion pair \( (r, s) \) stemming from the analysis of an external call \( e.m(D) \) in some proof outline, where \( e : I \). As the interface hides the actual class of the object referenced by \( e \), the call is analyzed based on the interface specification of \( m \).

The assertion pair \( (r, s) \) must thereby follow from the specification of \( m \) given by type \( I \), expressed by \( \text{spec}_\xi(I, m) \rightarrow (r, s) \).

**Condition 2** for sound environments is unchanged from **Definition 4**, but a third condition is introduced, expressing that a class satisfies the specifications of the implemented interface. If \( C \) implements an interface \( I \), the class defines (or inherits) an implementation of each \( m \in \text{public}_\xi(I) \). For each such method, the behavioral specification declared by \( I \) must follow from the specification of the method to which external calls will be bound.

**Definition 6 (Sound Environments).** A proof environment \( \xi \) of type \( I\text{Env} \) is sound if it satisfies the following conditions for each \( C : \text{Cid} \) and \( m : \text{Mid} \).

1. \( \forall(p, q) \in S_\xi(C, B.m) \Rightarrow \exists O \cdot O \vdash_{\text{PL}} \text{body}_\xi(B, m) : (p, q) \land \text{Internals}_\xi(C, B, O) \land \text{Internals}_\xi(O) \land \text{Static}_\xi(C, O) \)
2. \( m \in C\text{mid} \Rightarrow S_\xi(C, C.m) \rightarrow R_\xi(C, m) \)
3. \( \forall m \in \text{public}_\xi(I) : S_\xi(C, m) \rightarrow \text{spec}_\xi(I, m) \), where \( I = C\impl \)

where

\[
\text{Internals}_\xi(C, B, O) \triangleq \forall\langle v \rangle. e.C \cdot S_\xi(D, B#n) \rightarrow (r', s') \]
\[
\text{Internals}_\xi(O) \triangleq \forall\langle v \rangle. e.C \cdot S_\xi(I, n) \rightarrow (r', s') \]
\[
\text{Static}_\xi(C, O) \triangleq \forall\langle v \rangle. e.C \cdot S_\xi(B, B#n) \rightarrow (r', s') \]

and \( r' = (r[Z_0/O, X = s[\text{return}/v][Z_0/O]) \), \( s' = s[\text{return}/v][Z_0/O] \). \( X \) are the formal parameters of \( n \), and \( Z \) are the local variables of the calling method \( m \).

**Lemma 1** is adapted to environments of type \( I\text{Env} \). The proof is given in **Appendix C.1**.

**Lemma 2.** Assume sound environment \( \xi : I\text{Env} \) and a sound program logic \( \text{PL} \). Let \( B, D : \text{Cid} \), \( m : \text{Mid} \), and \( (p, q) : \text{APair} \) such that \( B, D \in \xi \) and \( (p, q) \in S_\xi(D, B.m) \). Then \( \models_D m(\xi) : (p, q)\{\text{body}(B, m)\} \).

For environment updates, we define an operation to update a proof environment with a new interface, and redefine the operation for updating a proof environment with a new class:

- \( \xi \oplus \text{extl}(C, I, \overline{I}, \overline{M}) \triangleq \langle L_\xi[C \mapsto \langle \overline{D}, I, \overline{M} \rangle], K_\xi, S_\xi, R_\xi \rangle \)
- \( \xi \oplus \text{extK}(I, \overline{I}, \overline{M}) \triangleq \langle L_\xi, K_\xi[I \mapsto \langle \overline{I}, \overline{M} \rangle], S_\xi, R_\xi \rangle \)

5.3. The calculus \( \text{LBSI}(\text{PL}) \)

In \( \text{LBSI}(\text{PL}) \), judgments have the form \( \xi \vdash A \), where \( \xi \) is the proof environment and \( A \) is a sequence of interfaces and classes. As before we require that superclasses are analyzed before subclasses, and in addition that superinterfaces are analyzed before subinterfaces. Furthermore, we assume that an interface is analyzed before it is used by a class. Consequently, whenever a class is analyzed, its implemented interface is already part of the environment, and for each external call statement \( v := e.m(D) \) in the class where \( e : I \), the interface \( I \) is in the environment. These assumptions ensure that the analysis of a class will not be blocked due to a missing superclass or interface.

External calls are now verified against the interface specifications of the called methods, so the analysis of a class can be completed without imposing require operations on other classes. Consequently, it suffices to consider individual classes and interfaces as the granularity of program analysis in the revised calculus, and the module layer of \( \text{LBSI}(\text{PL}) \) is omitted. The syntax for analysis operations in \( \text{LBSI}(\text{PL}) \) is given by:

---

2 For external calls where \( r, s \) range over \textbf{this}, we also need to replace \textbf{this} with a fresh name in \( r' \) and \( s' \).
The extended inference system \( \text{LBSI}(PL) \), where \( \mathcal{P} \) is a (possibly empty) sequence of classes and interfaces. Rules \text{NewClass}' and \text{External}' replace \text{NewClass} and \text{External} from \text{LBS}(PL). The other rules of \text{LBS}(PL) are preserved.

\[
\text{LBSI}(PL) \quad \mathcal{A} ::= \mathcal{P} | \langle C : \emptyset \rangle \cdot \mathcal{P} \\
\mathcal{P} ::= K | L | \mathcal{P} \cdot \mathcal{P} \\
\emptyset ::= \epsilon | \text{anReq}(\overline{M}) | \text{anSpec}(\overline{MS}) | \text{anCalls}(C, O) | \text{verify}(C, m, \overline{R}) | \text{supCls}(\overline{C}) | \text{supMtd}(C, \overline{M}) | \text{intSpec}(\overline{M}) | \emptyset \cdot \emptyset
\]

The new operation \( \text{intSpec}(\overline{M}) \) is used to analyze the interface specifications of methods \( \overline{M} \) with regard to implementations found in the considered class.

The calculus \( \text{LBSI}(PL) \) for \( \text{MII} \) consists of a (sound) program logic \( PL \), a proof environment \( \mathcal{E} : \mathcal{I}Env \), the inference rules listed in Fig. 14, and modified versions of the inference rules of \( \text{LBS}(PL) \), except \text{NewClass}, \text{EXTERNAL}, \text{NEWINT}, and \text{ENFMODULE}. Rules \text{NewClass} and \text{EXTERNAL} are replaced by \text{NewClass}' and \text{EXTERNAL}' as shown in Fig. 14. Rule \text{EXTREQ} is superfluous because external calls are analyzed in terms of interface specifications, and rules \text{ENFMODULE} and \text{ENFMODULE} are redundant as \( \text{LBSI}(PL) \) is without modules. The remaining rules of \( \text{LBS}(PL) \) are modified by removing the module operations as illustrated by \text{NewClass}' and \text{EXTERNAL}'. The complete set of rules for \( \text{LBSI}(PL) \) can be found in Appendix B, including structural rules.

The main differences between \( \text{LBS}(PL) \) and \( \text{LBSI}(PL) \) are captured by the rules in Fig. 14. Rule \text{NEWINT} extends the environment with a new interface, but no further analysis of the interface is required. The specifications of the interface will be analyzed with respect to each class that implements the interface. (Recall that interfaces are assumed to appear in the sequence \( \mathcal{P} \) before they are used.) Rule \text{NewClass}' is similar to the rule from \( \text{LBS}(PL) \), but an operation \text{intSpec} is introduced to analyze the specifications of the implemented interface. Rule \text{EXTERNAL}' is used for external calls; here, the call is analyzed based on the interface specification of the callee. For each public method of the class, the rule \text{intSpec} is used to verify the interface specification follows from the specification of the method implementation to which external calls will be bound.

Example 6. Reconsider the classes in Figs. 4 and 11. As a subclass need not satisfy the superclass type, the different classes may implement different interfaces. Assuming a suitably expressive interface specification language, for instance using invariants over communication histories as in [14], one may provide abstract specifications of how class instances interact with their environment. The class \text{Account} may implement an interface \text{I} defining exactly how the balance is calculated by \text{deposit} and \text{withdraw}. One may then specify the result of method \text{getbal} as a function over the history restricted to completed executions of \text{deposit} and \text{withdraw}. The interface \text{I} will be violated by the subclass \text{FeeAccount}, where also executions of method \text{monthly} will change the balance. This means that this class implements a different interface than \text{I}, since the result returned by \text{getbal} will depend on previous executions of \text{monthly} (in addition to \text{deposit} and \text{withdraw}). However, the internal analysis of the classes is performed as before, and the different \( 5 \) specifications are used to establish the interface properties.

Soundness. In order to show the soundness of \( \text{LBSI}(PL) \), Theorem 1 is first modified as follows:

Theorem 3. Let \( PL \) be a sound program logic, \( \mathcal{E} : \mathcal{I}Env \) a sound environment, and \( I \) be an interface or a class definition. If a proof of \( \mathcal{E} \vdash I \) in \( \text{LBSI}(PL) \) has \( \mathcal{E}' \) as its resulting proof environment, then \( \mathcal{E}' \) is also sound.

The proof of this theorem is given in Appendix C.2. We now show soundness for \( \text{LBSI}(PL) \):
Theorem 4 (Soundness). If PL is a sound program logic, then LBSI(PL) constitutes a sound proof system, in the sense that the environment resulting from the analysis of a program is sound.

Proof. In LBSI(PL), a program is analyzed as a sequence of classes and interfaces. It follows directly from Definition 6 that the empty environment is sound. Theorem 3 and Lemma 2 guarantee that the environment remains sound during the analysis of classes and interfaces. □

6. Related work

Multiple inheritance is commonly used in modeling notations such as UML [6], as a concept naturally inherits from several other concepts. However, it has not found its way into prominent programming languages such as Java. This may be due to the complexity of resolving method binding, as discussed in Section 2, which may easily cause ambiguities. However, multiple inheritance is supported in, e.g., C++ [38], CLOS [13], Eiffel [29], Ocaml [26], POOL [2], Self [10], and Creol [22]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Name conflicts also occur in the context of multiple dispatch, or multi-methods [13]. Multi-methods give a powerful binding mechanism, but reasoning about multi-methods and redefinition is difficult. For the prototype-based language Self [39], which supports the even more flexible mechanism of dynamic inheritance, an elegant prioritized binding strategy for multiple inheritance has been proposed in [10]. Each superclass is given a priority. With equal priority, the superclass related to the caller class is preferred. However, explicit class priorities may cause surprises in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller, binding fails.

The approach presented here is directly applicable to Creol since methods are the code structuring mechanism of Creol. The additional features of concurrency control and processor release points can be treated as in [14] (where inheritance is not considered).

There are surprisingly few high-level formal models of multiple inheritance in the literature. Such formalizations have traditionally used the objects-as-records paradigm, especially when dealing with (statically) typed languages. Cardelli [9] gives a denotational semantics of “multiple inheritance,” concentrating on typing aspects; i.e., the proposed notion of inheritance corresponds to subtyping in modern terminology. Rossie et al. [36] formalize multiple inheritance using subobjects, a run-time data structure used for virtual pointer tables [24,38]. This work focuses on compile-time issues and does not clarify multiple inheritance at the abstraction level of the programming language. A natural semantics for late binding in Eiffel models the binding mechanism at the abstraction level of the program [5]. Recently, an operational semantics and a type safety proof inspired by C++ have been formalized in Isabelle/HOL [40]. Due to its relative complexity compared to single inheritance, multiple inheritance has been seen as a mixed blessing: one the one hand desirable, on the other hand ambiguous. Thus, a number of proposals have been put forward to allow more flexible modes of code reuse, without being based on traditional multiple inheritance. These include dynamic inheritance [10], nested inheritance [30], and, perhaps most prominently, mixin- or trait-based inheritance [7]. The latter is used instead of multiple inheritance in, e.g., Scala [31]. The notion of healthiness proposed in this paper also serves to remove ambiguities from multiple inheritance by avoiding accidental overriding, making multiple inheritance easier to use.

Work on behavioral reasoning for object-oriented programs address languages with single inheritance (e.g., [34,35,8]). For late binding, different variations of behavioral subtyping are most common [27,1,25], as discussed above. Pierik and de Boer [34] present a sound and complete reasoning system for late bound calls which does not rely on behavioral subtyping. This work, also for single inheritance, is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental reasoning.

Lately, incremental reasoning, both for single and multiple inheritance, has been considered in the setting of separation logic [28,11,33]. These approaches support a distinction between static specifications, given for each method implementation, and dynamic specifications used to verify late bound calls. The dynamic specifications are related to our R requirements in the sense that both are imposed on subclass overriding. The dynamic specifications are given at the definition site, in contrast to our work where R requirements are generated by call-site analysis.

7. Conclusion and future work

This paper extends the framework of lazy behavioral subtyping from languages with single inheritance to languages with multiple inheritance. Thus, a contribution of the paper is a proof system for an object-oriented kernel language with multiple inheritance. Another contribution is to show that lazy behavioral subtyping can be adapted from single to multiple inheritance in a natural way, making it well-suited for different object-oriented systems. Lazy behavioral subtyping supports incremental reasoning under an open world assumption, where class hierarchies can be gradually extended by inheritance. The approach is more flexible than traditional behavioral subtyping, since user given S specifications are not imposed on overriding methods in subclasses. Only the minimal R requirements resulting from the analysis of internal late bound calls must be preserved by overriding methods. This is demonstrated by the main example. The S specifications of a method definition are used in the verification of static calls to the method and to establish interface properties if the method is public. S specifications may in addition be used in order to establish class invariants.
The paper investigates two versions of the reasoning framework, one for a purely class-based language and another for a language with behavioral interfaces. We have presented our formalisms in terms of a small kernel language which captures the main features of object-oriented programming, excluding language features not central to the discussion. Therefore, the results of the paper are applicable to many languages, assuming a healthy binding strategy. We show soundness of the reasoning framework in both cases. A running example illustrates how the approach can be used for formal reasoning in a setting which is more flexible than traditional behavioral subtyping, and how independent class and interface hierarchies can be used for flexible reuse of code.

It has been argued that multiple inheritance is too complex to be applicable in a safe manner. This is mainly due to horizontal name conflicts and method binding which behaves in unexpected ways, causing ambiguities. This paper proposes healthiness requirements on method binding and the use of static calls to reduce these factors. The paper gives a healthy binding mechanism where renaming is not needed, thereby avoiding theoretical and practical problems related to explicit renaming. In this way, our formalism contributes to making the concept of multiple inheritance more attractive. The motivation behind multiple inheritance is to support flexible code reuse. This coincides with the motivation for lazy behavioral subtyping, which is to reason about flexible code reuse. It is therefore interesting to combine these mechanisms. Combining healthiness and lazy behavioral subtyping, two independent class hierarchies will not accidentally interfere with each other when the hierarchies are merged in a common subclass. This suffices to allow incremental reasoning for external calls as well as internal calls.

Appendix A. Soundness of LBS(PL)

A.1. Auxiliary function definitions

To establish soundness of LBS(PL), we define the auxiliary functions as listed in Fig. 15. They are needed to track the behavioral constraints during the analysis of an inheritance hierarchy.

Function $S\uparrow : Cid \timesomid \to Set[APair]$ (taking the environment as an implicit argument) is defined such that $S\uparrow(D, B\#m)$ returns the union of $S(B, C, B\#m)$ for all $D \leq B$. Function $R\uparrow : Cid \timesomid \to Set[APair]$ is defined such that $R\uparrow(D, m)$ returns the union of all $R(C, B\#m)$ for $D \leq B$, i.e., all requirements towards $m$ that are imposed by classes above $D$. This function is defined in terms of the function $req : Cid \timesomid \to Set[APair]$. Function $commSup : Cid \to Set[Cid]$ returns the set of common superclasses, and is defined in terms of $com : Cid \to Set[Cid]$ and $sup : Cid \to Set[Cid]$. Function $delReq : Cid \timesomid \to Set[APair]$, is used to compute delayed requirements. The set $R(D, B\#m)$ is contained in $delReq(D, B\#m)$ if the following two conditions are met:

- $D \leq B$
- There is no class $G$ where $bind\_d(G, B\#m) = bind\_d(D, B\#m)$ for $D \leq G \leq B$, i.e., for all classes that are strictly above $D$ and below $C$, the call $m\#B$ binds to a different implementation than the one found by $bind\_d(D, B\#m)$.
The function `delReq` is defined in terms of `bel`: List[Cid] × Cid → ... O⟩ ;L] · M (decompSupMtd)
E ⊢ [⟨C: supMtd(D, m1) · supMtd(D, m2) · O⟩ ;L] · M
E ⊢ [⟨C: supMtd(D, m1m2) · O⟩ ;L] · M

A.2. Structural rules of LBS(PL)

This section presents LBS(PL) rules for decomposing list-like structures and handling trivial cases. Here \( \mathcal{M} \) is a (possibly empty) list of analysis operations.

\[
\begin{align*}
\text{(noSpec)} & \quad E \vdash [\langle C : \emptyset \rangle ; L] · M \\
\text{(decompSpec)} & \quad E \vdash [\langle C : \text{anSpec}(\emptyset, \emptyset) \rangle ; L] · M \\
\text{E} & \vdash [\langle C : \text{anSpec}(\emptyset_1, \emptyset_2) \rangle · \text{anSpec}(\emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anSpec}(\emptyset_1, \emptyset_2) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anReq}(\emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anReq}(\emptyset_1, \emptyset_2) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anReq}(\emptyset_1, \emptyset_2) · \text{anReq}(\emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anReq}(\emptyset_1, \emptyset_2) · \text{anReq}(\emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset_1, \emptyset_2) · \text{verify}(D, m, \emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset_1, \emptyset_2) · \text{verify}(D, m, \emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1) · \text{anCalls}(D, t_1) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1) · \text{anCalls}(D, t_1) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset_1, \emptyset_2) ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset_1, \emptyset_2) · \text{verify}(D, m, \emptyset_2, \emptyset) ; L] · M \\
\text{E} & \vdash [\langle C : \text{verify}(D, m, \emptyset_1, \emptyset_2) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1) · \text{anCalls}(D, t_1) · \emptyset ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) ; L] · M \\
\text{E} & \vdash [\langle C : \text{anCalls}(D, t_1 · t_2) · \emptyset ; L] · M
\end{align*}
\]

- Class C is at depth 0 below C.
- Class D is at depth d (where d > 0) below class C if there is a class D' ∈ D.inh such that D' is at depth d − 1 below C, and for all classes D'' ∈ D.inh the depth of D'' below C is less or equal to d − 1 or D'' is not below C.
A.3. Proof of Lemma 1

Lemma 1 shows the important property of sound environment (cf. Definition 4):

Assume a sound environment $\mathcal{E} : Env$ and a sound program logic $PL$. Let $B, D : \text{Cid}, m : \text{Mid}$, and $(p, q) : \text{APair}$ such that $B, D \in \mathcal{E}$ and $(p, q) \in S_2 \uparrow (D, B.m)$. Then $\vdash_D m(\overline{x}) : (p, q)\{body_\mathcal{E}(B, m)\}$.

**Proof.** By definition of $S_1^\prime$, the assumption $(p, q) \in S_2 \uparrow (D, B.m)$ implies that there exists some class $C$ such that $D \leq_\mathcal{E} C \leq_\mathcal{E} B$ and $(p, q) \in S_2 (C, B.m)$. By Definition 4, there must be a proof outline $O$ for the method body such that $O \vdash_{PL} body_\mathcal{E}(B, m) : (p, q)$. The proof of the lemma proceeds by induction over the call structure of $m$.

**Base case:** The execution $[p]body_\mathcal{E}(B, m)[q]$ does not lead to any method calls. Then $\vdash_D m(\overline{x}) : (p, q)\{body_\mathcal{E}(B, m)\}$ follows by the soundness of $PL$.

**Induction step:** Each kind of method call is considered in a separate case. If a call to some method $n$ in $body_\mathcal{E}(B, m)$ is bound to a definition in class $H$ for search class $G$, we take $\vdash_G n(\overline{y}) : \langle t, u \rangle(body_\mathcal{E}(H, n))$ as the induction hypothesis, for each $(t, u) \in S_2 \uparrow (G, H.n)$. For a call to $n$ with precondition $r$, postcondition $s$, and actual parameters $\overline{e}$, we let $r' = (r[\overline{e}/\overline{z}] \land \overline{y} = \overline{\overline{y}/\overline{z}})$ and $s' = s[\text{return}/v][\overline{e}/\overline{z}]$, where the return value of the method is assigned to the variable $v$ by the caller. We consider each kind of method call in $O$ by a separate case.

**Late bound internal calls** $[r]v := n(\overline{e})\{s\}$. By the assumptions of the lemma, internal calls are bound with search class $D$. Thus, in the induction hypothesis we thereby have $G = D$, and $H = \text{bind}_{\mathcal{E}}(D, B\#n)$. Consequently, it suffices to ensure $S_\mathcal{E} \uparrow (D, H.n) \rightarrow (r', s')$, which follows by Definition 4.

**Static calls** $[r]v := n\#A(\overline{e})\{s\}$. For the induction hypothesis, we have $G = D$ and $H = \text{bind}_{\mathcal{E}}(A, A\#n)$. By Definition 4, we then have $S_\mathcal{E} \uparrow (C, H.n) \rightarrow (r', s')$. The desired $S_\mathcal{E} \uparrow (D, H.n) \rightarrow (r', s')$ then follows by the definition of $S_1^\prime$ since $D \leq_\mathcal{E} C$, i.e., $S_\mathcal{E} \uparrow (C, H.n) \subseteq S_\mathcal{E} \uparrow (D, H.n)$.

**External calls** $[r]v := e : E.n(\overline{e})\{s\}$. This call can bound with respect to any class $G$ such that $G \leq_\mathcal{E} E$, and let $H = \text{bind}_{\mathcal{E}}(G, E\#n)$. By the induction hypothesis, it suffices to ensure:

$$S_\mathcal{E} \uparrow (G, H.n) \rightarrow (r', s')$$

(2)

For the external call, Definition 4 gives the following assumption:

$$S_\mathcal{E} \uparrow(E, \text{bind}_{\mathcal{E}}(E, E\#n), n) \rightarrow (r', s') \quad \text{and} \quad R_\mathcal{E} \uparrow(E, n) \rightarrow (r', s')$$

(3)

The relation $S_\mathcal{E} \uparrow (G, H.n) \rightarrow (r', s')$ is proved by induction over the depth $d$ of $G$ below $E$.

**Base case:** $d = 0$, i.e., $G = E$. In this case, we have $H = \text{bind}_{\mathcal{E}}(E, E\#n)$. Relation (2) then follows by assumption (3).

**Induction step:** $d = d' + 1$, i.e., $G \leq_\mathcal{E} E$ at depth $d$. As the induction hypothesis, we may assume that for any class $G'$ at depth $d'$ below $E$, that $S_\mathcal{E} \uparrow (G', \text{bind}_{\mathcal{E}}(G', E\#n), n) \rightarrow (r', s')$. We consider two cases.

Case 1: $n \in G.mtds$. By Definition 4, Condition 2, we then have $S_\mathcal{E} \uparrow (G, n) \rightarrow R_\mathcal{E} \uparrow(G, n)$. By Fig. 15, we have $S_\mathcal{E} \uparrow (G, G.n) = S_\mathcal{E} \uparrow (G, G.n)$. Since $\text{bind}_{\mathcal{E}}(G, E\#n) = G$ and $H$ in Relation (2) therefore equals $G$ in this case, it suffices to ensure $R_\mathcal{E} \uparrow (G, n) \rightarrow (r', s')$. By Fig. 15, we have $R_\mathcal{E} \uparrow(E, n) \subseteq R_\mathcal{E} \uparrow(G, n)$ for $G \leq_\mathcal{E} E$. Relation (2) thereby follows by (3) and transitivity of entailment:

$$R_\mathcal{E} \uparrow(G, n) \rightarrow R_\mathcal{E} \uparrow(E, n) \rightarrow (r', s')$$

Case 2: $n \notin G.mtds$. Since $H = \text{bind}_{\mathcal{E}}(G, E\#n)$ and $n \notin G.mtds$, there must by Definition 1 of function bind exist some $G' \in G.\text{inh}$ such that $\text{bind}_{\mathcal{E}}(G', E\#n) = H$. As the induction hypothesis applies to $G'$, we have $S_\mathcal{E} \uparrow (G', H.n) \rightarrow (r', s')$. The desired relation $S_\mathcal{E} \uparrow(G, H.n) \rightarrow (r', s')$ then follows by the definition of $S_1^\prime$, since $G \leq_\mathcal{E} G'$.

\[ \square \]

A.4. Proof of Theorem 1

The theorem establishes the soundness of the inference system of Section 4.1:

Let $\mathcal{E} : Env$ be a sound environment and $\mathcal{E}$ a set of class definitions. If a proof of $\mathcal{E} \vdash \text{module}(\overline{T})$ in $LBS(PL)$ has $\mathcal{E}'$ as its resulting environment, then $\mathcal{E}'$ is also sound.

**Proof.** Given a sound environment, we prove that the environment extensions preserve soundness. We consider each of the two conditions of Definition 4 in isolation.

**Condition 1 of Definition 4.** The proof proceeds by induction over the inference rules. The only rule that extends $S_\mathcal{E}(C, B.m)$ is $\text{Req\text{-}Nor\text{-}Def}$, and this rule ensures that there is a proof outline for $body_\mathcal{E}(B, m)$. The set $S_\mathcal{E}(C, B.m)$ is only extended during analysis of $C$. Thus, for any $(p, q) \in S_\mathcal{E}(C, B.m)$ we have an $O$ such that $O \vdash_{PL} body_\mathcal{E}(B, m) : (p, q)$ and $C \leq_\mathcal{E} B$.
If the specification \((p, q)\) is included in \(S_\ell(C, B.m)\) by \text{ReqNotDer}, an operation \text{anCalls}(B, O)\) is generated and analyzed in the context of \(C\). Each call statement in the proof outline is analyzed by either \text{Internal}, \text{Static} or \text{External}. We consider each kind of method call in isolation. For a call to \(n\) with precondition \(r\), postcondition \(s\), and actual parameters \(\overline{e}\), we let \(r' = (r[Z_0/Z] \land \overline{y} = \overline{e[Z_0/Z]})\) and \(s' = \{\text{return} / v\}[Z_0/Z]\), where \(\overline{y}\) are the formal parameters and the return value of the method is assigned to the variable \(v\) by the caller.

**Late bound internal calls.** For each \([r \triangleright v := n(\overline{e})\] in \(O\), we have \(n \in \text{called}_d(B)\) and an operation \text{anCalls}(B, [r \triangleright v := n(\overline{e})\] is analyzed. Rule \text{Internal} applies to this operation, ensuring \((r', s') \in R_\ell(C, B\#n)\). As the class hierarchy is extended, we then need to ensure

\[
S_\ell\uparrow(D, \text{bind}_d(D, B\#n), n) \rightarrow (r', s')
\]

for each class \(D\) below \(C\) as defined by Definition 4. This is done by induction over the depth \(d\) of \(D\) below \(C\).

**Base case:** \(d = 0\), i.e., \(D = C\). By (4), we need to ensure \(S_\ell\uparrow(C, H.n) \rightarrow (r', s')\) for \(H = \text{bind}_d(C, B\#n)\). This case is handled by analysis of the \text{anCalls}(B, [r \triangleright v \triangleright := n(\overline{e})\] operation that is generated by the analysis of \(C\). The application of \text{Internal} leads to an operation verify\((H, r', s')\). Since the analysis of this operation succeeds, \text{ReqDER or ReqNotDer} is applied. The relation \(S_\ell\uparrow(C, H.n) \rightarrow (r', s')\) must hold directly if \text{ReqDER} is applied. Otherwise, if \text{ReqNotDer} is applied, the set \(S_\ell(C, H.n)\) is extended with \((r', s')\). The desired relation then holds by transitivity of entailment since \(S_\ell\uparrow(C, H.n) \rightarrow S_\ell(C, H.n) \rightarrow (r', s')\).

**Induction step:** \(d = d' + 1\), i.e., \(D \triangleleft C\) at depth \(d\). As the induction hypothesis, we may assume \(S_\ell\uparrow(D', \text{bind}_d(D', B\#n), n) \rightarrow (r', s')\) for any class \(D'\) such that \(D < D' \leq C\), i.e., \(D'\) is at depth less or equal to \(d'\) below \(C\). We consider the two cases \(n \in D.mtds\) and \(n \notin D.mtds\) separately.

**Case 1:** \(n \in D.mtds\). Method \(n\) is defined in \(D\), which means that \text{bind}_d(D, B\#n) = \text{D}. By (4), the relation \(S_\ell\uparrow(D, D.n) \rightarrow (r', s')\) must then be ensured. By the definition of \(R\), we have \(R_\ell(C, B\#n) \subseteq R_\ell(D, \text{inh}, n)\) since \(D < C\).

Since \(n \in D.mtds\), \text{NewMtd} will be applied to \(n\) by the analysis of \(D\), generating an operation verify\((D, n, (r', s'))\). This operation either succeeds by \text{ReqDER or ReqNotDer}, both ensuring the desired \(S_\ell\uparrow(D, D.n) \rightarrow (r', s')\).

**Case 2:** \(n \notin D.mtds\). For this case, we consider \(B \notin \text{commSup}_d(D)\) and \(B \in \text{commSup}_d(D)\) separately.

**Case 2a:** \(B \notin \text{commSup}_d(D)\). In this case, there is exactly one \(D' \in D\text{.inh}\) such that \(D' \leq B\). Since \(D < C \leq B\), we then have \(D' \leq B\). By the definition of \(R\), we have \(R_\ell(C, B\#n) \subseteq R_\ell(D, \text{inh}, n)\) since \(D < C\).

Since \(n \notin D.mtds\), \text{NewMtd} will be applied to \(n\) by the analysis of \(D\), generating an operation verify\((D, n, (r', s'))\). This operation either succeeds by \text{ReqDER or ReqNotDer}, both ensuring the desired \(S_\ell\uparrow(D, D.n) \rightarrow (r', s')\).

**Case 2b:** \(B \in \text{commSup}_d(D)\). Let \(E = \text{bind}_d(D, B\#n)\). By (4), we need to ensure \(S_\ell\uparrow(D, E.n) \rightarrow (r', s')\) where \((r', s') \in R_\ell(C, B\#n)\).

For all classes \(G\) such that \(D < G \leq E\), the induction hypothesis gives \(S_\ell\uparrow(G, \text{bind}_d(G, B\#n), n) \rightarrow (r', s')\). There are two possibilities:

- **Exist a such that** \(D < a \leq G \leq E\) and \text{bind}_d(G, B\#n) = \text{E}. Then the induction hypothesis, we then have \(S_\ell\uparrow(G, E.n) \rightarrow (r', s').\) Proof obligation \(S_\ell\uparrow(D, E.n) \rightarrow (r', s')\) then follows by the definition of \(S\). Since \(G\) is above \(D\).

- **Exist no class** \(G\) such that \(D < G \leq E\) and \text{bind}_d(G, B\#n) = \text{E}. By the definition of \text{delReq}, the set \(R_\ell(C, B\#n)\) is then included in \text{delReq}_d(D, B\#n).

When class \(D\) is analyzed by \text{NewClass}, a \text{supCls}(\text{commSup}_d(D))\) operation is generated and analyzed in the context of \(D\). Since \(B \in \text{commSup}_d(D)\), decomposition of this operation leads to a \text{supCls}(B)\) operation. Furthermore, since \(n \in \text{called}_d(B) \setminus D.mtds\), the application of rule \text{SupMtd} and further decomposition leads to an operation \text{supMtd}(B, n). By rule \text{SupReq}, we then arrive at an operation verify\((E, n, \text{delReq}_d(D, B\#n))\). By decomposition of the requirement set, we then arrive at an operation verify\((E, n, (r', s'))\). As in the base case, application of either \text{ReqDER or ReqNotDer} then ensures \(S_\ell\uparrow(E, E.n) \rightarrow (r', s')\).

**Static calls.** For each call \([r \triangleright v := n(\overline{A}(\overline{e}))\] in the proof outline \(O\), rule \text{Static} will be applied. The operation verify\((\text{bind}_d(A, A\#n), n, (r', s'))\) will be generated by this rule, and the operation is verified during analysis of \(C\). This operation succeeds by \text{ReqDER or ReqNotDer}, both ensuring the desired relation \(S_\ell\uparrow(C, \text{bind}_d(A, A\#n), n) \rightarrow (r', s')\) of Definition 4.

**External calls.** For each \([r \triangleright v := e : E.n(\overline{e})\] in \(O\), \text{External} is applied. This rule will generate a require\((E, n, (r', s'))\) operation. This operation succeeds by \text{extrReq}, which ensures \(S_\ell\uparrow(E, \text{bind}_d(E, E\#n), n) \rightarrow (r', s')\) and \(R_\ell\uparrow(E, n) \rightarrow (r', s')\) as required by Definition 4.

**Conclusion of Definition 4.** For each method \(m \in C.mtds\), we must establish \(S_\ell(C, C.m) \rightarrow R_\ell(C, C.m)\). When class \(C\) is analyzed by \text{NewClass}, an operation \text{anReq} is generated for each method defined in \(C\). By \text{NewMtd}, possibly followed by decomposition of the requirement set, an operation verify\((C, m, (p, q))\) is generated for each \((p, q) \in R_\ell(C, \text{inh}, m)\). These operations succeed either by \text{ReqDER or ReqNotDer}, ensuring \(S_\ell(C, C.m) \rightarrow R_\ell(C, \text{inh}, m)\).
Since \( R_e \uparrow (C, m) = \text{req}_e(C, C, m) \cup R_e \uparrow (C.\text{inh}, m) \) by the definition of \( R \uparrow \), it then remains to ensure \( S_0(C, C.m) \rightarrow \text{req}_e(C, C, m) \). If \((r, s) \in \text{req}_e(C, C, m)\), there exists by the definition of \( \text{req} \), a class \( B \) such that \( C \leq_B B \) and \((r, s) \in R_e(C, B\#m)\). In order for \((r, s)\) to be included in this set, \( \text{Internal} \) is applied during analysis of \( C \). This rule will generate an operation \( \text{verify}(\text{bind}_e(C, B\#m), m, (r, s)) \) which equals \( \text{verify}(C, m, (r, s)) \) since \( m \) is defined in \( C \). Again, analysis of this operation succeeds by \( \text{ReqDer} \) or \( \text{ReqNotDer} \), which both ensures the desired \( S_0(C, C.m) \rightarrow (r, s). \)

**Appendix B. LBSI(PL) inference rules**

This section shows the complete set of inference rules for LBSI(PL). More specifically, the syntax of the analysis operations used by the rules here are given in Section 5.3, together with an explanation of the characteristic rules for dealing with interfaces.

**B.1. Main rules**

For the rules \( \text{Internal}, \text{Static}, \text{and External} \), we have that \( r' = (r[\mathcal{I}_0/\mathcal{I}] \wedge \mathcal{R} = \mathcal{I}[\mathcal{I}_0/\mathcal{I}]) \), \( s' = s[\text{return}/v][\mathcal{I}_0/\mathcal{I}] \), \( \mathcal{R} \) are the formal parameters of \( n \), and \( \mathcal{I} \) are the local variables of the calling method.

\[
\begin{align*}
&I \notin \xi & I \in \xi & \xi \odot \text{extK}(I, I, \mathcal{I}) \vdash \rho \\
&l \in \xi & C \notin \xi & \mathcal{D} \in \xi & \mathcal{E} = \text{commSup}(C) & \xi \odot \text{extK}(C, D, I, I, \mathcal{I}, \mathcal{M}) \vdash (C : \text{anSpec}(\mathcal{M}) \cdot \text{anReq}(\mathcal{M}) \cdot \text{supCls}(E) \cdot \text{intSpec}(\text{public}_z(I))) \cdot \rho \\
&\xi \vdash (\text{class } E \text{ extends } D \text{ implements } I (\mathcal{M}, \mathcal{K})) \cdot \rho \\
&\xi \vdash (C : e) \cdot \rho & \xi \vdash (C : \text{anSpec}(\text{m@D}(\mathcal{I}) : (p, q))) \cdot \rho \\
&\xi \vdash (C : \text{verify}(C, m, R_e \uparrow (C.\text{inh}, m)) \cdot \theta) \cdot \rho & \xi \vdash (\text{ReqNotDer}) & O \in R_e \odot \text{body}_e(D, m : (p, q)) & \xi \vdash (C : \text{anCalls}(D, O) \cdot \theta) \cdot \rho \\
&\xi \vdash (C : \text{verify}(D, m, (p, q))) \cdot \rho & \xi \vdash (\text{ReqDer}) & S_e \uparrow (C, D, m) \rightarrow (p, q) & \xi \vdash (C : \theta) \cdot \rho \\
&\xi \vdash (C : \text{verify}(D, m, (p, q))) \cdot \rho & \xi \vdash (\text{ INTERNAL }) & E = \text{bind}_e(C, D\#m) & \xi \vdash (C : \text{verify}(E, m, (r', s')) \cdot \theta) \cdot \rho \\
&\xi \vdash (C : \text{anCalls}(D, [r] v := m(\mathcal{I})(s)) \cdot \theta) \cdot \rho & \xi \vdash (\text{ STATIC }) & E \vdash (C : \text{verify}(\text{bind}_e(B, B\#m), m, (r', s')) \cdot \theta) \cdot \rho \\
&\xi \vdash (C : \text{anCalls}(D, [r] v := m@B(\mathcal{I})(s)) \cdot \theta) \cdot \rho & \xi \vdash (\text{ EXTERNAL }) & e : I & I \in \xi & \text{spec}_e(I, m) \rightarrow (r', s') & \xi \vdash (C : \theta) \cdot \rho \\
&\xi \vdash (C : \text{anCalls}(C, [r] v := e.m(\mathcal{I})(s)) \cdot \theta) \cdot \rho & \xi \vdash (\text{ INTSPEC }) & S_e \uparrow (C, m) \rightarrow \text{spec}_e(C.\text{impl}, m) & \xi \vdash (C : \theta) \cdot \rho \\
&\xi \vdash (C : \text{intSpec}(m) \cdot \theta) \cdot \rho & \xi \vdash (\text{ SUPMTD }) & E \vdash (C : \text{supCls}(D) \cdot \theta) \cdot \rho \\
&\xi \vdash (C : \text{supMtd}(D, \text{called}(D) \setminus C.mtDs) \cdot \theta) \cdot \rho & \xi \vdash (\text{ SUPREQ }) & \xi \vdash (C : \text{supMtd}(D, m) \cdot \theta) \cdot \rho \\
&\xi \vdash (C : \text{verify}(E, m, \text{delReq}(C, D\#m)) \cdot \theta) \cdot \rho &
\end{align*}
\]

Please cite this article in press as: J. Dovland et al., Incremental reasoning with lazy behavioral subtyping for multiple inheritance, Science of Computer Programming (2010), doi:10.1016/j.scico.2010.09.006
B.2. Structural rules

\[
\begin{align*}
\text{(noSpec)} & \quad E \vdash (C : \emptyset) \cdot P \\
\text{(decompSpec)} & \quad E \vdash (C : \text{anSpec}(\emptyset) \cdot \emptyset) \cdot P \\
\text{(decompSupMtd)} & \quad E \vdash (C : \text{anSpec}(M_1 \cdot M_2) \cdot \emptyset) \cdot P \\
\text{(noMtds)} & \quad E \vdash (C : \emptyset) \cdot P \\
\text{(decompMtds)} & \quad E \vdash (C : \text{anReq}(\emptyset) \cdot \emptyset) \cdot P \\
\text{(decompSeq)} & \quad E \vdash (C : \text{anCalls}(D, t_1 \cdot t_2) \cdot \emptyset) \cdot P \\
\text{(decompIf)} & \quad E \vdash (C : \text{anCalls}(D, \text{if } b \text{ then } t_1 \text{ else } t_2) \cdot \emptyset) \cdot P \\
\text{(skip)} & \quad E \vdash (C : \emptyset) \cdot P \quad \text{does not contain call statements} \\
\text{(nossupCls)} & \quad E \vdash (C : \emptyset) \cdot P \\
\text{(decompSupCls)} & \quad E \vdash (C : \text{supCls}(\emptyset) \cdot \emptyset) \cdot P \\
\text{(nosupMtd)} & \quad E \vdash (C : \text{supMtd}(D_1 \cdot D_2) \cdot \emptyset) \cdot P \\
\text{(decompSupMtd)} & \quad E \vdash (C : \text{supMtd}(D, M_1) \cdot \emptyset) \cdot P \\
\text{(decompIntSpec)} & \quad E \vdash (C : \text{intSpec}(\emptyset) \cdot \emptyset) \cdot P \\
\text{(decompIntSpec)} & \quad E \vdash (C : \text{intSpec}(M_1) \cdot \emptyset) \cdot P \\
\text{(decompIntSpec)} & \quad E \vdash (C : \text{intSpec}(M_2) \cdot \emptyset) \cdot P \\
\end{align*}
\]

Appendix C. Soundness of LBSI(PL)

As for the basic calculus without interfaces in Appendix A, we show the proofs for two central properties, Lemma 2 (sound environments) and Theorem 3 (soundness of the calculus).
C.1. Proof of Lemma 2

This lemma corresponds to the analogous Lemma 1 (for the setting without interfaces), shown earlier.

Assume a sound environment $\mathcal{E} : IEnv$ and a sound program logic $\mathcal{L}$. Let $B, D : \text{Cid}_m : \text{Mid}$, and $(p, q) : \text{APair}$ such that $B, D \in \mathcal{E}$ and $(p, q) \in S_{\mathcal{E}}(D, B, m)$. Then $\models_D m(\mathcal{X}) : (p, q)(\text{body}_{B}(B, m))$.

Proof. This proof is equal to the proof of Lemma 1, except for the treatment of external calls in the induction step.

By definition of $S_{\mathcal{E}}^\uparrow$, the assumption $(p, q) \in S_{\mathcal{E}}^\uparrow(D, B, m)$ implies that there exists some class $C$ such that $D \subseteq_C C \subseteq_{\mathcal{E}} B$ and $(p, q) \in S_{\mathcal{E}}(C, B, m)$. By Definition 6 there must be a proof outline $O$ for the method body such that $O \vdash_{\mathcal{L}} \text{body}_{C}(B, m) : (p, q)$. The proof of the lemma proceeds by induction over the call structure of $m$.

Base case: The execution $(p) \text{body}_{B}(B, m)[q]$ does not lead to any method calls. Then $\models_D m(\mathcal{X}) : (p, q)(\text{body}_{B}(B, m))$ follows by the soundness of $\mathcal{L}$.

Induction step: Each kind of method call is considered by a separate case. For a call to some method $n$ in body$_B(B, m)$ is bound to a definition in class $H$ for search class $G$, we take $\models_C n(\mathcal{Y}) : (t, u)(\text{body}_H(H, n))$ as the induction hypothesis, for each $(t, u) \in S_{\mathcal{E}}^\uparrow(G, H, n)$. For a call to $n$ with precondition $r$, postcondition $s$, and actual parameters $\mathcal{E}$, we let $r' = (r[\mathcal{E}/Z_I] \land \mathcal{E} = \mathcal{E}[\mathcal{E}/Z_I])$ and $s' = s[r:\text{return}/\mathcal{E}[\mathcal{E}/Z_I]]$, where the return value of the method is assigned to the variable $v$ by the caller. We consider each kind of method call in $O$ by itself.

Late bound internal calls $\{r\} v := n(\mathcal{E}) [s]$. By definition, internal calls are bound with search class $D$. Thus, in the induction hypothesis we have $G = D$, and $H = \text{bind}_C(D, B, #n)$. Consequently, it suffices to ensure $S_{\mathcal{E}}^\uparrow(D, H, n) \rightarrow (r', s')$, which follows by Definition 6.

Static calls $\{r\} v := e : I.n(\mathcal{E}) [s]$. By definition, Condition 1, we have $\text{spec}_C(I, n) \rightarrow (r', s')$. Consider some class $E$, with $\text{impl} = J$. If it is possible for the call to bind in context $E$, then $n \in \text{public}_E(I)$ and $I \subseteq_{\mathcal{E}} E$, which gives $\text{spec}_E(I, n) \rightarrow \text{spec}_E(I, n)$. For the induction hypothesis, we then have $G = E$ and $\text{bind}_C(E, E, #n) = H$, and the induction hypothesis ensures that for all $(t, u) \in S_{\mathcal{E}}^\uparrow(E, H, n)$ we have $\models_E n(\mathcal{Y}) : (t, u)(\text{body}_E(H, n))$. By Definition 6, Condition 3, we have $S_{\mathcal{E}}^\uparrow(E, H, n) \rightarrow \text{spec}_C(E, I, n) \rightarrow \text{spec}_E(I, n) \rightarrow (r', s')$. □

C.2. Proof of Theorem 3

Let $\mathcal{L}$ be a sound program logic, $\mathcal{E} : IEnv$ a sound environment, and $I$ be an interface or a class definition. If a proof of $\mathcal{E} \vdash I$ in $\text{LBSI}([\mathcal{L}])$ has $\mathcal{E}$ as its resulting proof environment, then $\mathcal{E}'$ is also sound.

Proof. Soundness is trivially maintained if $\mathcal{E}$ is extended by a new interface, and interfaces are assumed to be contained in the environment before they are used. For the analysis of some class $C$, we consider each condition of Definition 6 by itself. For Conditions 1 and 2, these proofs are similar to the ones in Appendix A.4. For brevity, we therefore refer to Appendix A.4 whenever some part of the proof is unchanged.

Condition 1 of Definition 6. The proof goes by induction over the inference rules. The only rule that extends $S_{\mathcal{E}}(C, B, m)$ is $\text{ReqNotDef}$, and this rule ensures that there is a proof outline for $\text{body}_{B}(C, B, m)$. The set $S_{\mathcal{E}}(C, B, m)$ is only extended during analysis of $C$. Thus, for any $(p, q) \in S_{\mathcal{E}}(C, B, m)$ we have an $O$ such that $O \vdash_{\mathcal{L}} \text{body}_{B}(C, B, m) : (p, q)$ and $C \subseteq_{\mathcal{E}} B$.

If the specification $(p, q)$ is included in $S_{\mathcal{E}}(C, B, m)$ by $\text{ReqNotDef}$, and operation $anCalls(B, O)$ is generated and analyzed in the context of $C$. Each call statement in the proof outline is analyzed by either $\text{Internal, Static of External}$. We consider each kind of method call in isolation. For a call to $n$ with precondition $r$, postcondition $s$, and actual parameters $\mathcal{E}$, we let $r' = (r[\mathcal{E}/Z_I] \land \mathcal{E} = \mathcal{E}[\mathcal{E}/Z_I])$ and $s' = s[r: \text{return}/\mathcal{E}[\mathcal{E}/Z_I]]$, where $\mathcal{E}$ are the formal parameters and the return value of the method is assigned to the variable $v$ by the caller.

Late bound internal calls. The proof of this case is unchanged from Appendix A.4.

Static calls. The proof of this case is unchanged from Appendix A.4.

External calls. For each $\{r\} v := e : I.n(\mathcal{E}) [s]$ in $O$, $\text{External}$ will be applied. As the rule succeeds, the rule directly ensures $\text{spec}_E(I, n) \rightarrow (r', s')$ as required by Definition 6.

Condition 2 of Definition 6. The proof of this case is unchanged from Appendix A.4.

Condition 3 of Definition 6. For class $C$ and $I = C.\text{impl}$, we must ensure that for each $m \in \text{public}_C(I)$ we have $S_{\mathcal{E}}^\uparrow(C, m) \rightarrow \text{spec}_E(I, m)$, where $S_{\mathcal{E}}^\uparrow(C, m)$ denotes the set $S_{\mathcal{E}}^\uparrow(C, \text{bind}_C(C, C, #m))$.

When class $C$ is selected for analysis by rule $\text{NewClass}$, an operation $\text{intSpec(public}_C(I))$ is generated and analyzed in the context of $C$. For each $m \in \text{public}_C(I)$, decomposition leads to an operation $\text{intSpec}(m)$. The analysis of this operation succeeds by application of $\text{IntSpec}$ (Appendix B), which ensures the required relation $S_{\mathcal{E}}^\uparrow(C, m) \rightarrow \text{spec}_E(I, m)$ □
References


[21] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23(3) (2001) 396–450.


C.2 Compositional Verification of Temporal Safety Properties
CVPP: A Tool Set for Compositional Verification of Control–Flow Safety Properties

Marieke Huisman\textsuperscript{1} and Dilian Gurov\textsuperscript{2,*}

\textsuperscript{1} University of Twente, Netherlands
\textsuperscript{2} Royal Institute of Technology, Stockholm, Sweden

\textbf{Abstract.} This paper describes CVPP, a tool set for compositional verification of control–flow safety properties for programs with procedures. The compositional verification principle that underlies CVPP is based on maximal models constructed from component specifications. Maximal models replace the actual components when verifying the whole program, either for the purposes of modularity of verification or due to unavailability of the component implementations at verification time. A characteristic feature of the principle and the tool set is the distinction between program structure and behaviour. While behavioural properties are more abstract and convenient for specification purposes, structural ones are easier to manipulate, in particular when it comes to verification or the construction of maximal models. Therefore, CVPP also contains the means to characterise a given behavioural formula by a set of structural formulae. The paper presents the underlying framework for compositional verification and the components of the tool set. Several verification scenarios are described, as well as wrapper tools that support the automatic execution of such scenarios, providing appropriate pre– and post–processing to interface smoothly with the user and to encapsulate the inner workings of the tool set.

1 Introduction

To enable verification of realistic software, verification techniques have to be compositional and algorithmically decidable. Compositionality ensures that the verification task can be split up in smaller pieces, while algorithmic decidability ensures that verification can be done automatically, without any user interaction. Moreover, for many application domains, compositionality and algorithmic decidability are essential.

For example, in a dynamically reconfigurable distributed system, components can join and leave the system at run–time dynamically. For such an open system, appropriate verification techniques are necessary to support safe downloading, i.e., to determine without any user interaction whether a newly arriving component will not corrupt the well–functioning of the global system. These techniques require the relativisation of the correctness of the system on the specifications

\textsuperscript{*} Partially funded by the EU FET project FP7–ICT–2009–3 HATS.
and the local correctness of its components. This relativisation can also be used for the purposes of *modularity*. Modular verification is a means of controlling the complexity of verifying large software. It allows an independent local evolution of the implementations of individual modules without affecting the global correctness of the program.

The CVPP tool set is designed to tackle exactly this kind of verification problems by supporting an algorithmic technique for compositional verification. Its focus is on control–flow safety properties of programs with (possibly recursive) procedures. Such properties typically describe sets of allowed sequences of method invocations, and are conveniently expressed in temporal logic. The underlying program model is that of *flow graphs*, abstracting completely from program data to allow efficient algorithmic modular verification. However, the model can be enhanced with exception information or multi-threading. Even though the tool set is developed with compositionality in mind, it can also be used for non-compositional control–flow verification problems of programs with procedures. In particular, it allows to reduce infinite-state verification of behavioural properties to finite-state verification of structural properties.

Abstracting away from all data may seem like a severe restriction, but still many useful properties can be expressed, such as:

- within atomic transactions, there are no calls to non-atomic methods;
- in a voting system, candidate selection has to be finished, before the vote can be confirmed;
- a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible;
- in a door access control system, the password has to be checked before the door is unlocked, and it can only be changed when the door is unlocked.

Extending the technique with data over finite domains will allow for a wider range of properties and possible applications, but needs to be combined with abstraction techniques to control the complexity of verification. Such an extension will be investigated in future work.

The present paper describes CVPP, its underlying compositional verification framework, and its implementation. We describe three important verification scenarios: (i) open system verification, (ii) modular verification, and (iii) non-compositional verification. We also discuss the encapsulation of the inner workings of CVPP by means of wrapper tools that automate the various scenarios.

Previous work by the authors on tool support and case studies has been reported in 2004 [16]. The current version of the tool set, discussed in this paper, includes later extensions: (i) an inliner to abstract private methods [11], (ii) more general program models concerning exceptions, threads and open flow graphs [15,13], and (iii) a property translation from behavioural to structural properties [12,13]. The last extension allows local assumptions to be behavioural, whereas in the previous version they had to be structural. Further, we have unified the inputs and outputs to allow interoperability of the individual tools, and have started work on wrapper tools, automating the verification scenarios.
**Related Work.** Maven is a modular verification tool addressing temporal properties of procedural languages in the context of aspects [9]. A non-compositional verification method based on a program model closely related to ours is presented by Alur and others [3]. It proposes a temporal logic CaRet for nested calls and returns (generalised to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures. Another example of a successful system for the non-compositional verification of temporal safety properties, applied to C programs, is ESP [8]. This system combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user-defined source code patterns.

Most of the existing work on modular verification of safety properties is based on Hoare logic. Müller was the first to propose a sound modular Hoare-style verification technique for object-oriented languages [18]. A typical verification tool within this line of work is Spec# [4].

Recent work by Alur and Chauhuri proposes a unification of Hoare-style and Manna–Pnueli-style temporal reasoning for procedural programs, presenting proof rules for procedure-modular temporal reasoning [2].

**Organisation.** Sections 2 and 3 sketch the tool set’s theoretical background and underlying verification method. Section 4 describes the different tools that make up CVPP, followed by a description of typical verification scenarios in Section 5. Section 6 exemplifies some typical verification tasks when using CVPP. We conclude with possible extensions that would make CVPP applicable to a larger class of problems (without changing the underlying methodology).

## 2 Program Model and Logic

This section summarises the program model and logic that underlies CVPP. For a more detailed account, the reader is referred to [14].

As mentioned earlier, a characteristic feature of CVPP is the distinction between structural and behavioural properties. Usually, we are interested in properties of the behaviour of a program, while its structure is just a means for accomplishing the desired behaviour. In particular, the same behaviour can be produced by several structures. It is thus more natural and more abstract to specify programs with behavioural properties than with structural ones.

However, algorithmic techniques for program analysis and verification are computationally considerably more expensive on the level of program behaviour than on the level of program structure. Program correctness problems are therefore often phrased in terms of the program structure rather than in terms of its behaviour. Furthermore, many behavioural properties have natural structural counterparts, e.g., tail recursion, while other behavioural properties can be characterised through finite sets of structural ones (see Section 3). Therefore, CVPP is set up in such a way that structural properties can be used whenever this is possible and meaningful.
2.1 Model and Logic

Our program model is control–flow based and thus over–approximates actual program behaviour. It defines two different views on programs: a structural and a behavioural one. Both views are instantiations of the general notions of model, defined below. Notice in particular that these instantiations yield a structural and a behavioural version of the logic, and that this enables a uniform treatment of structure and behaviour whenever possible.

**Definition 1. (Model)** A model is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where $S$ is a set of states, $L$ a set of labels, $\rightarrow \subseteq S \times L \times S$ a labelled transition relation, $A$ a set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state $s$ the set of atomic propositions that hold in $s$. An initialised model is a pair $(\mathcal{M}, E)$, with $\mathcal{M}$ a model and $E \subseteq S$ a set of entry states.

As property specification language we use the fragment of the modal $\mu$-calculus [17] with boxes and greatest fixed-points only is used. This temporal logic is capable of characterising simulation (cf. [14]) and is thus suitable for expressing safety properties. Throughout, we fix a set of labels $L$, a set of atomic propositions $A$, and a set of propositional variables $V$.

**Definition 2. (Logic)** The formulae of our logic are inductively defined by:

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a] \phi \mid \nu X.\phi$$

where $p \in A$, $a \in L$ and $X \in V$.

Satisfaction on states $(\mathcal{M}, s) \models \phi$ is defined in the standard fashion [17]. For instance, formula $[a] \phi$ holds of state $s$ in model $\mathcal{M}$ if $\phi$ holds in all states accessible from $s$ via an edge labelled $a$. A model $(\mathcal{M}, E)$ satisfies a formula $\phi$, denoted $(\mathcal{M}, E) \models \phi$, if all its entry states $E$ satisfy $\phi$. The constant formulae $true$ (denoted $tt$) and $false$ (ff) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \lor \phi$. We assume that formulae have pair–wise distinct fixed–point binders, and unless stated otherwise, are closed and guarded (cf. [23]).

2.2 Control–Flow Structure and Behaviour

**Control–Flow Structure.** We abstract away from all data, therefore program structure is defined as a collection of control–flow graphs (or flow graphs), one for each of the program’s methods. Let $\text{Meth}$ be a countably infinite set of method names. A method graph is an instance of the general notion of model.

**Definition 3. (Method graph)** A method graph for $m \in \text{Meth}$ over a finite set $M \subseteq \text{Meth}$ of method names is an initialised model $(\mathcal{M}_m, E_m)$, where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ a non–empty set of entry points of $m$. $V_m$ is the set of control nodes of $m$, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ is defined so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are return points.
class Number {
  public static boolean even(int n) {
    if (n == 0)
      return true;
    else
      return odd(n - 1);
  }
  public static boolean odd(int n) {
    if (n == 0)
      return false;
    else
      return even(n - 1);
  }
}

Fig. 1. A simple Java class and its flow graph

Example 1. Figure 1 shows a simple Java class and the (simplified) flow graph it induces. The flow graph consists of two method graphs - one for method `even` and one for method `odd`. Entry nodes are depicted as edges without source.

Flow graph `interfaces` are defined as pairs $I = (I^+, I^-)$, where $I^+$, $I^- \subseteq \text{Meth}$ are finite sets of names of `provided` and (externally) `required` methods, respectively. A flow graph $G$ with interface $I$ is denoted $G : I$. The flow graph of a program is essentially the (disjoint) union $\sqcup$ of its method graphs. Flow graphs can only be composed if their interfaces match. A flow graph is `closed` if $I^- = \emptyset$, i.e., it does not require any external methods. Satisfaction, instantiated to flow graphs, is called structural satisfaction $\models_s$.

Example 2. Consider the flow graph in Example 1. The property “on every path from a program entry node, the first encountered call edge goes to a return node” is formalised by the structural formula $\nu X. [\text{even}] r \land [\text{odd}] r \land [\varepsilon] X$, in effect specifying that the program is tail-recursive.

Control-Flow Behaviour. Next, we instantiate models on the behavioural level. Transition label $\tau$ designates internal transfer of control, $m_1 \text{ call } m_2$ invocation of method $m_2$ by method $m_1$, and $m_2 \text{ ret } m_1$ the corresponding return.

Definition 4. (Behaviour) Let $G = (M, E) : I$ be a closed flow graph where $M = (V, L, \rightarrow, A, \lambda)$. The behaviour of $G$ is defined as the initialised model $b(G) = (M_0, E_0)$, where $M_0 = (S_0, L_0, \rightarrow_b, A_0, \lambda_0)$, such that $S_0 = V \times V^*$, i.e., states are pairs of control points $v$ and stacks $\sigma$ (also called configurations), $L_0 = \{m_1 \leftrightarrow m_2 \mid k \in \{\text{call, ret}\}, m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_0 = A$, $\lambda_0((v, \sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_0 \times L_0 \times S_0$ is defined by the rules:

\begin{align*}
\text{[transfer]} & \quad (v, \sigma) \xrightarrow{\nu_b} (v', \sigma) \quad \text{if } m \in I^+, v \xrightarrow{m} v', v \models \neg \tau \\
\text{[call]} & \quad (v_1, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v_1', \sigma) \quad \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1, v_1' \models \neg \tau, v_2 \models m_2, v_2 \in E \\
\text{[return]} & \quad (v_2, v_1', \sigma) \xrightarrow{m_2 \text{ ret } m_1} (v_1, \sigma) \quad \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \land \tau, v_1 \models m_1
\end{align*}

1 We only require $I^-$ to contain methods that are not provided by $I^+$. This is different from our earlier work (e.g., [14]), but in line with the tool set implementation.
The set of initial configurations is defined by \( E_b = E \times \{ \epsilon \} \), where \( \epsilon \) denotes the empty sequence over \( V \).

The definition is easily extended to open flow graphs (see [13]). Flow graph behaviour can alternatively be defined via pushdown automata (PDA) [14, Def. 34] and approximated with the related notion of pushdown systems (PDS). We exploit this by using PDS model checking for verification of behavioural properties (see [6]). Currently, our tool set relies on the external tool Moped [20]; however, this requires the properties to be presented in LTL.

Example 3. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite-state behaviour. One example execution of the program is represented by the following path (in the branching structure) from an initial to a final configuration:

\[
(v_0, \epsilon) \xrightarrow{\text{even call odd}} (v_1, \epsilon) \xrightarrow{\text{odd call even}} (v_2, \epsilon) \xrightarrow{\text{even call odd}} (v_3, \epsilon) \xrightarrow{\text{odd call even}} (v_4, \epsilon) \xrightarrow{\text{even call odd}} (v_5, v_3) \xrightarrow{\text{odd call even}} (v_6, v_3) \xrightarrow{\text{even call odd}} (v_7, v_3) \xrightarrow{\text{odd call even}} (v_8, v_3) \xrightarrow{\text{even call odd}} (v_9, v_3) \xrightarrow{\text{odd call even}} (v_3, \epsilon)
\]

Also on the behavioural level, we instantiate the definition of satisfaction: we define \( G|_{b} \phi \) as \( b(G) \models \phi \). The resulting behavioural logic is powerful enough to express the class of security policies defined by finite state security automata [19].

Example 4. For the flow graph from Example 1, the behavioural formula \( \text{even} \Rightarrow \nu X. [\text{even call even}] \text{ff} \land [\tau] X \) expresses the property “in every program execution starting in method even, the first call is not to method even itself”.

Extensions. This section presents the basic program model and logic, considering only normal, sequential control-flow. Extensions with exceptions and with multi-threaded behaviour (with synchronisation on locks) exist [15], and are supported in CVPP. The extension to open flow graphs mentioned above is also supported. In ongoing work we address further extensions to Boolean programs, as well as to richer fragments of the \( \mu \)-calculus; this is not incorporated in CVPP yet.

3 Framework for Compositional Verification

The compositional verification method underlying our tool set is based on the computation of maximal models from component specifications and the instantiation of components with these models when model checking global system properties. For finite-state systems, this approach was introduced by Grumberg and Long [10] and since then it has become a standard technique for reducing the verification of correctness of property decompositions to model checking.

Maximal Models for Compositional Verification. A model is said to be maximal for a given property \( \phi \), if it satisfies \( \phi \) and simulates \( \text{(w.r.t. a suitable property-preserving simulation relation \( \leq \))} \) all models satisfying \( \phi \). For models in the sense
of Definition 1 and formulae in the logic of Definition 2, maximal models exist and are unique up to isomorphism (see [14]). To compute a maximal model for a property $\phi$, we present the formula as a modal equation system (see [5]), which is then transformed into a canonical form, the so-called simulation normal form.

A formula $\phi$ in simulation normal form can be directly mapped into a (finite) model $M$ that simulates all models that satisfy $\phi$; i.e., for every model $M'$, $M' \leq M$ iff $M' \models \phi$. Due to this close connection between simulation and satisfaction, we obtain the following sound and complete verification principle [14]:

**Compositional verification principle for models:**

Compositional verification principle for models: to show $M_1 \uplus M_2 \models \psi$, it suffices to show $M_1 \models \phi$ (i.e., component $M_1$ satisfies a suitably chosen local assumption $\phi$) and $M_\phi \uplus M_2 \models \psi$ (i.e., component $M_2$, when composed with the maximal model $M_\phi$ for $\phi$, satisfies the global guarantee $\psi$).

Completeness of the principle guarantees that no false negatives exist: if $M_\phi \uplus M_2 \models \psi$ fails, then there is a model $M$ such that $M \models \phi$ but $M \uplus M_2 \not\models \psi$.

Adaptation of this principle to flow graphs (as models) and structural and behavioural properties presents us with certain difficulties. Given a structural or behavioural flow graph property $\phi$, there is no guarantee that the maximal model of $\phi$ is a legal flow graph structure or behaviour.

**Maximal Flow Graphs from Structural Specifications.** For structural properties this problem can be solved for a given flow graph interface $I$, because we can characterise precisely the flow graphs having interface $I$ as models through a structural formula $\theta_I$ in our logic. Let $I = \{m_1, m_2\}$ be a closed flow graph interface. A model is a flow graph with this interface exactly when it satisfies the formula $\theta_I = (\nu X. m_1 \land [m_1, m_2, \varepsilon]X) \lor (\nu Y. m_2 \land [m_1, m_2, \varepsilon]Y)$, which essentially expresses that edges in the flow graph do not cross method boundaries. Then, for every structural formula $\phi$, the maximal model of the formula $\phi \land \theta_I$ is a flow graph $G_{\phi, I}$ that simulates structurally all flow graphs with interface $I$ that satisfy $\phi$. We term this flow graph the maximal flow graph for formula $\phi$ and interface $I$. The above compositional verification principle can then be adapted to structural properties of flow graphs, yielding the following sound and complete compositional verification principle, presented as a proof rule (see [14] for technical details):

$$
\frac{
G_1 \models s \phi \quad G_\phi, I_{G_1} \uplus G_2 \models s \psi
}{
G_1 \uplus G_2 \models s \psi
}\quad (\text{struct − comp})
$$

**Maximal Flow Graphs from Behavioural Specifications.** In the case of behavioural flow graph properties, however, there is no such way to characterise in our logic all models that constitute behaviours of flow graphs with a given interface (intuitively, this is because the logic is not capable of expressing context–free properties). Furthermore, these models are infinite–state and cannot be constructed explicitly; what we actually need is a way to construct the maximal flow graph for a given behavioural formula $\phi$ and interface $I$. It turns out, however, that in
general there is no such single flow graph, but rather a set of flow graphs having the
property that every flow graph satisfying $\phi$ is simulated by some flow graph in the
set. To compute such a set, we have developed a translation from behavioural
flow graph properties $\phi$ to equivalent sets of structural properties $\Pi_I$ for a
given interface $I$. The translation is based on a tableau construction that conceptually amounts to symbolic execution of the behavioural formula, collecting structural constraints along the way. By keeping track of the subformulae that have been examined, recursion in the structural constraints is identified and captured by fixed–point formulae (for details see [12]). Combining this translation with maximal flow graph generation for structural properties yields the following sound and complete compositional verification principle for flow graphs and behavioural properties, presented as a proof rule:

\[ (\text{beh} - \text{comp}) \]
\[ G_1 \models_b \phi \quad \{ G_{x, t_G_2} \uplus G_2 \models_b \psi \} \quad x \in \Pi_{I_1, G_2} (\phi) \]
\[ G_1 \uplus G_2 \models_b \psi \quad G_1 : I_{G_1} \]

In addition, we have also developed a “mixed” rule [14], where local structural assumptions are combined with global behavioural guarantees.

The presented proof rules are flexible, allowing reasoning about a combination of concrete components (i.e., given through their implementation) and abstract components (i.e., given though their specification), both at the structural and the behavioural levels. Section 5 shows typical verification scenarios, where these proof rules are applied for open system and modular verification. A possible instantiation of this approach is to choose individual methods as components. The proof rules then give rise to a procedure–modular verification technique for temporal properties (see [21]).

4 Tool Support for Compositional Verification

This section describes the different internal data formats and tools within the CVPP tool set. It also exemplifies the different input formats used. A high–level overview of CVPP’s architecture is shown in Figure 2 (where rounded boxes denote data formats, squared boxes tool components, and dashed lines denote external formats or tools).
As program input format, currently the Java bytecode format is used. Internally, there are three important data formats:

- **Model**: the program model representation, containing nodes, edges, a valuation and a set of entry points.
- **Formula**: the property representation. We support behavioural and structural formulae in our logic, both in recursive and in equation system form.
- **Interface**: the interface representation, containing lists of provided and of externally required methods. Interfaces are used as auxiliary information by almost all tool components, and are therefore not included explicitly in Figure 2.

The components of the tool set are the following:

- **Analyser**: from Java classes to flow graphs. Java bytecode classes are abstracted into flow graphs. The tool is build on top of the Soot framework [22].
- **Graph**: transformations on the program model representations. The main operations supported are flow graph composition, pretty printing in different formats (in particular as CCS process terms and as PDS of the induced behaviour), and inlining of private methods. The use of the latter operation, called Graph Inliner, is briefly explained in Section 5.1 (see also [11]).
- **Formula**: transformations on the property representations. The main operations supported are the simplification of formulae, the conversion from one property format to another (such as the translation of our logic from recursive to equation system form, needed for maximal model construction), pretty printing as a CWB or LTL formula (as input for Moped), as well as the characterisation of behavioural formulae by structural ones. The latter operation is referred to as Beh2Struct. In addition, we allow properties to be expressed using so-called patterns. Patterns provide abbreviations for commonly used specification constructs. They increase readability and make the property more independent of the interface. The Formula component translates patterns into our logic.
- **MaxMod**: the maximal model construction as described in Section 3. This component uses formulae expressed as equation systems.
- **ModCheck**: model checking, using external tools: for structural properties we use CWB, the Edinburgh Concurrency Workbench [7], while for behavioural properties we rely on Moped, a PDS model checker for LTL [20].

To conclude this section, we show how the examples from Section 2 are written in CVPP’s input formats. Consider again the flow graph from Figure 1. The method graph of method even is written as follows:

```
node 0 meth(even) entry
node 1 meth(even) edge 0 1 eps
node 2 meth(even) edge 1 2 eps
node 3 meth(even) ret edge 2 3 odd
node 4 meth(even) ret
```
The interface and structural and behaviour properties are written as follows in CVPP’s input format:

**interface for Number: provided even, odd**

struct. formula Ex. 2: $\nu X.((\text{[even]} r) \land (\text{[odd]} r) \land (\text{[eps]} X))$

beh. formula Ex. 4: $\text{meth(even)} \Rightarrow \nu X.((\text{[even call even]} ff) \land (\text{[tau]} X))$

5 Typical Verification Scenarios

Section 3 presented two compositional verification principles; this section describes in detail some typical scenarios supported by CVPP and these verification principles. In addition, we also describe how CVPP can be used for non-compositional verification. This is in particular interesting for behavioural properties: by means of the translation of behavioural properties into structural ones, CVPP provides an effective way to reduce the verification problem for behavioural properties to the computationally simpler problem for structural ones.

5.1 Open System Verification

The most general application of the proof rules presented in Section 3 is to open system verification, where some components are given by an implementation (referred to here as concrete components), while others are only given by a specification (abstract components). This can typically happen with dynamically reconfigurable or evolving software, where some components are either not known or simply not statically fixed at verification time.

Thus, verification of a global property of an open system has to be relativised on the local specifications of the abstract components. For instance, if all specifications are behavioural, this is achieved by consecutively applying rule (beh − comp) on every abstract component. The implementations of the abstract components, once available, are checked against their local specifications.

An additional complexity stems from the detail of information in the concrete components. Typically, these contain information about private methods. In contrast, the abstract components and global properties are described in terms of the public interface. Therefore, the implementation details in the concrete components are abstracted away, by the Graph Inliner, to the publicly visible behaviour, before composing the components.

The overall verification task thus divides into two independent tasks, supported by our tool set as follows:

1. **Local correctness:** Check whether the implementation, once available, of every abstract component meets its local specification as described below in Section 5.3.

2. **Global correctness:**
   (a) for every concrete component, from its implementation, extract a flow graph using the Analyser, and use the Graph Inliner to construct its publicly visible behaviour;
(b) for every abstract component, if its local specification is behavioural, translate the property to an equivalent set of structural ones using Beh2Struct;

(c) for every structural property, being either a local specification of an abstract component itself or resulting from step 2(b), compute a maximal flow graph using MaxMod;

(d) for all instantiations of abstract components by corresponding constructed maximal flow graphs, and instantiations of concrete components by their extracted flow graphs, compose the graphs using Graph to produce a global flow graph of the system, and model check the latter against the global specification as described below in Section 5.3.

5.2 Modular Verification

In the modular software design paradigm the goal is to verify the modules of a software system locally, i.e., independently of each other, and then to combine the local correctness arguments into a global correctness proof of the whole system. In our verification framework, modular verification is simply an instance of the more general case of open system verification described above, with modules as components and where all components are abstract. This eliminates task 2(a) and simplifies conceptually task 2(d).

One can view the notion of module on different levels of granularity. One (rather extreme) case in procedural programming languages is when every procedure itself is considered a module and is equipped with a specification. In this case we obtain procedure–modular verification, similar to many Hoare logic based verification approaches. We have recently shown on a case study that it is indeed possible and convenient to reason at this level of granularity about control–flow safety properties of an application [21].

5.3 Non–compositional Verification

The open system and modular verification scenarios above give rise to several non-modular verification tasks. In fact, CVPP can also be applied in a non-compositional setting. This is in particular useful when reasoning about behavioural properties. Due to unbounded recursion, verification of behavioural properties for procedural programs is infinite–state, even when all data is abstracted away as in our case. On the other hand, verification of structural properties is finite–state. Thus, by applying our translation from behavioural to sets of structural properties, one can reduce verification of behavioural properties to a finite number of finite–state verification tasks. Given a Java application and a property specification (either behavioural or structural), this is done as follows:

1. extract the flow graph of the application using the Analyser (and if necessary, use the Graph Inliner to abstract away from implementation details);
2. if the property is structural, cast the flow graph as a CCS term using Graph, and model check the term against the property using the CWB;
3. If the property is behavioural, there are two alternatives: either
   (a) cast the flow graph as a pushdown system using Graph, and model check it against the property using Moped; or
   (b) translate the property to an equivalent set of structural ones using Beh2Struct, and perform step 2 for each one of these.

Step 3(b) is particularly meaningful in settings where the behavioural specifications are known in advance (such as the security policies of mobile platforms) and are relatively stable; the property translation can then be applied prior to the verification task itself.

5.4 Wrapper Tools for Standard Verification Scenarios

The different scenarios described above require the use of several of the tools of CVPP in a particular pre–defined order. To make CVPP easier to use, and to hide away the internal formats and translations within the tool set, wrapper tools are being developed that perform the typical verification scenarios automatically. A wrapper implements a pre– and a post–processor that translates input and output of the tool set, and performs the different verification steps automatically. The post–processor appropriately handles feedback from the model checkers: when a structural property is violated, it is indicated where in the program this violation occurs; when a behavioural property is violated the model checking counter example is translated back into a program trace.

The first wrapper tool that we developed is ProMoVer [21]. It automates procedure–modular verification of Java programs annotated with global and method–local specifications. ProMoVer was evaluated on a small but realistic case study: we verified the absence of calls to non–atomic methods within Java Card transactions for a Java Card electronic purse application². In the near future, we plan to develop wrapper tools for the other scenarios.

6 Executing the Verification Scenarios

To illustrate how CVPP is used, this section discusses how parts of the different verification scenarios described in the previous section are applied on concrete examples. For a larger example discussing our experiences with ProMoVer for the verification of the safe use of the Java Card transaction mechanism in an e-commerce application for smart cards, we again refer the reader to [21].

6.1 Generating Maximal Flow Graphs for a Behavioural Property

One important subtask in the compositional verification scenarios discussed in the previous section is the construction of maximal flow graphs from a behavioural specification of a component; see steps 2(b,c) of the open system

² A web–based interface to ProMoVer is available from: http://www.csc.kth.se/˜siavashs/ProMoVer/promover.php
verification scenario. As explained in Section 3, this is achieved by translating the behavioural property into an equivalent set of structural ones, and by constructing a maximal flow graph for each of the latter.

For example, consider a component specified by an interface where methods even and odd are provided and no external methods are required, and by the behavioural property “in every program execution starting in method even, the first call is not to method even itself” formalised in Example 4. Providing this interface and formula to Beh2Struct, and optimising the result with the simplification facility of Formula, we obtain one structural formula: even ⇒ νX. [even] ff ∧ [ε] X. To compute a maximal flow graph, we first apply the conversion facilities of Formula to transform the formula into a modal equation system, which is then passed on, together with the original interface, to MaxMod. The resulting maximal flow graph is shown in Figure 3. Notice that the method graphs for even and odd are isomorphic, but the graph of method even has two entry nodes while the graph of method odd has four; as a result, the former restricts the behaviour in that, once called, method even can only call method odd as a first method call, while the latter makes no restrictions on the behaviour whatsoever. This maximal flow graph can now be substituted for the given component when model checking global system properties.

6.2 Closed System Model Checking of a Behavioural Property

Consider again the component of the previous subsection, described by the interface where methods even and odd are provided and no external methods are required, and by the behavioural property in Example 4. We want to show that the class Number defined in Example 1 is an appropriate implementation of this component. This is an instance of the non-compositional verification scenario in Section 5.3. Thus, using the Analyser, we first extract the flow graph, resulting in the flow graph as in Figure 1. For this application, there is no difference between public and private interface, thus there is no need to use the Graph Inliner.

The property is behavioural, thus we have a choice (cf. step 3, Section 5.3). (a) We can model check the behavioural property directly. We use Graph to produce the PDS from the flow graph, and Formula to transform the property to an LTL formula. Then Moped is used to verify that class Number indeed respects this property. (b) As in the previous subsection, we can compute the structural
formula that characterises the behavioural formula by using Beh2Struct. We use Graph to pretty print the flow graph as CCS term and Formula to pretty print the formula in CWB’s input format. Then CWB is used to verify that class Number indeed respects this structural property.

7 Conclusion

CVPP is a tool set for compositional verification of control–flow safety properties of procedural programs. It supports a completely automatic verification method based on maximal models. The underlying general compositional verification principle instantiates to two important verification scenarios, namely open system verification and modular verification. By means of an algorithmic translation of behavioural into structural properties, the tool is also applicable to non–compositional verification, allowing infinite–state PDA model checking to be reduced to standard finite–state model checking. The various scenarios can be supported by wrapper tools, such as ProMoVer, that encapsulate the inner workings of the tool set and provide a smooth interface to the user.

The largest CVPP case study so far is the verification of absence of illicit applet interactions in a smart card application [14,6]. This has been redone with the later extensions of the tool set. It is future work to develop more case studies, similar in size and complexity, but taking advantage of the different wrapper tools. For all three verification scenarios appropriate wrappers will be developed. Further, we will provide support for other property specification formalisms, in particular security automata. Support for flow graph extraction from source code will be improved, developing a modular and extensible tool. Other extensions concern the program model, where we plan to add data to flow graphs to represent Boolean programs faithfully, and to develop a solution for multi–threaded programs. Finally, we plan to extend the logic to include liveness properties; these become meaningful when the flow graphs model program behaviour faithfully, or at least provide under–approximations of the guaranteed behaviour.

Acknowledgements. We thank everybody who contributed to CVPP: Irem Aktug (Analysers), Christoph Sprenger (MaxMod), Siavash Soleimanifard (ProMoVer), and Afshin Amighi (property simplification). We are also indebted to Stefan Schwoon, who extended the input language of Moped to serve our needs.

References

ProMoVer: Modular Verification of Temporal Safety Properties

Siavash Soleimanifard¹, Dilian Gurov¹, and Marieke Huisman²

¹ Royal Institute of Technology, Stockholm, Sweden
² University of Twente, Enschede, Netherlands

Abstract. This paper describes ProMoVer, a tool for fully automated procedure–modular verification of Java programs equipped with method–local and global assertions that specify safety properties of sequences of method invocations. Modularity at the procedure–level is a natural instantiation of the modular verification paradigm, where correctness of global properties is relativized on the local properties of the methods rather than on their implementations, and is based here on the construction of maximal models for a program model that abstracts away from program data. This approach allows global properties to be verified in the presence of code evolution, multiple method implementations (as arising from software product lines), or even unknown method implementations (as in mobile code for open platforms). ProMoVer automates a typical verification scenario for a previously developed tool set for compositional verification of control flow safety properties, and provides appropriate pre– and post–processing. Modularity is exploited by a mechanism for proof reuse that detects and minimizes the verification tasks resulting from changes in the code and the specifications. The verification task is relatively light–weight due to support for abstraction from private methods and automatic extraction of candidate specifications from method implementations. We evaluate the tool on a number of applications from the smart card domain.

1 Introduction

In modern computing systems, code changes frequently. Modules (or components) evolve rapidly or exist in multiple versions customized for various users, and in mobile contexts, a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready–made off–the–shelf components, and each component may be dynamically replaced by a new one that provides improved or additional functionality. The static and dynamic variability makes it more important to provide formal correctness guarantees for the behaviour of such systems, but at the same time also more difficult. Modularity of verification is a key to providing such guarantees in the presence of variability.

* Soleimanifard’s work is funded by the ContraST project of the Swedish Research Council VR, and Gurov’s work by the EU FET project FP7–ICT–2009–3 HATS.
In modular verification, correctness of the software components is specified and verified independently (locally) for each module, while correctness of the whole system is specified through a global property, the correctness of which is verified relative to the local specifications rather than relative to the actual implementations of the modules. It is this relativization that enables verification of global properties in the presence of static and dynamic variability. In particular, it allows an independent evolution of the implementations of individual modules, only requiring the re-establishment of their local correctness.

Hoare logic provides a popular framework for modular specification and verification of software, where it is natural to take the individual procedures as modules, in order to achieve scalability, see e.g., [16]. While Hoare logic allows the local effect of invoking a given procedure to be specified, temporal logic is better suited for capturing its interaction with the environment, such as the allowed sequences of procedure invocations. This paper shows that procedure–modular verification is also appropriate for safety temporal logic: for each procedure the local property specifies its legal call sequences, while the system’s global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

To support our approach, we have developed a fully automated verification tool, PROMoVeR. It takes as input a Java program annotated with global and method–local correctness assertions written in temporal logic and it automatically invokes a number of tools from cvpp, a previously developed tool set for compositional verification [12], to perform the individual local and global correctness checks. Essentially, PROMoVeR is a wrapper that performs a standard verification scenario in the general tool set, which demonstrates that procedure–modular verification of temporal safety properties can be applied automatically. PROMoVeR is available via a web–based interface [18]. To make the annotation procedure comparatively lightweight, together with PROMoVeR we also provide a facility to extract a candidate local property specifying the legal call sequences by means of static analysis, given a concrete procedure implementation. A user then only has to inspect the extracted specifications and potentially remove superfluous constraints in order to accommodate possible evolution of the code. A further reduction of effort is achieved by focusing on the public procedures only, the private ones being considered merely as an implementation means. Finally, PROMoVeR also provides proof storage and reuse: only the properties that are affected by a change (either in implementation or in specification) are reverified, all other results are reused.

Validity of the approach is shown on some typical Java Card e-commerce applications. Such security–relevant applications are an important target for formal verification techniques. For these applications we verify the absence of calls to non–atomic methods within transactions. Such properties, specifying legal sequences of calls to security–related methods, are an important class of platform–specific security properties. The PROMoVeR web interface allows the user to verify similar platform–specific security properties, for which a ready–made formalization is provided.
To allow efficient algorithmic modular verification, the tool set currently abstracts away from all data, thus considering safety properties of the control flow; in particular, method calls in Java programs are over-approximated by non-deterministic choice on possible method implementations that the virtual call resolution might resolve to. This rather severe restriction on the program model is imposed by the maximal model construction that is the core of our modular verification technique (see [9] for a proof of soundness and completeness for this program model). Still, many useful properties can be expressed at this level of abstraction. These include the platform-specific security properties discussed above, and application-specific properties such as: (i) a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible; or (ii) in a voting system, candidate selection has to be finished, before the vote can be confirmed. Extending the technique with data, either over finite domains or over pointer structures, will allow for a wider range of properties and possible applications, but requires a non-trivial generalisation of the maximal model construction, and needs to be combined with abstraction techniques to control the complexity of verification and of model extraction from a program. We are currently investigating this.

Control flow safety properties can be expressed in various formalisms, e.g., automata-based or process-algebraic notations, as well as in temporal logics such as LTL [20] and the safety fragment of the modal μ-calculus [14]. Internally, cvpp uses the latter, but ProMoVer allows the user to write the specifications in LTL, which is usually considered more intuitive. It is future work to extend ProMoVer also with other notations, in particular graphical ones.

The work in this paper is closely related with the development of cvpp, a tool set for compositional verification of control flow safety properties [12]. As already pointed out, ProMoVer is essentially a wrapper that automates a typical verification scenario for cvpp, where modularity is applied at the procedure-level. In addition, ProMoVer provides support for proof reuse, and specification extraction, a collection of ready-formalised properties, and translates between the different intermediate formats and formalisms.

A non-compositional verification method based on a program model closely related to ours is presented by Alur et al. [3]. It proposes a temporal logic CARET for nested calls and returns (generalized to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures. ESP is another example of a successful system for non-compositional verification of temporal safety properties, applied to C programs [5]. It combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user-defined source code patterns. MAVEN is a modular verification tool addressing temporal properties of procedural languages, but in the context of aspects [7]. Recent work by Alur and Chauhuri proposes a unification of Hoare-style and Manna-Pnueli-style temporal reasoning for procedural programs, presenting proof rules for procedure-modular temporal reasoning [2].

3
The rest of this paper is organized as follows. Section 2 presents the use of ProMoVer from a user’s point–of–view. Section 3 recapitulates the verification framework, describing the underlying program model and logic, and the compositional verification method based on constructing maximal models. Then, Section 4 describes the ProMoVer tool, while Section 5 describes three small but realistic case studies using the tool. Finally, the last section draws conclusions and suggests directions for future research.

Preliminary results on an earlier version of ProMoVer were reported in a workshop paper [19]. There, we did not address certain issues such as abstraction from private methods, automatic specification extraction, and proof reuse, and provided less experimental results.

2 ProMoVer: A User’s View

We start by illustrating how ProMoVer is used on a small example. Both local and global properties are provided as assertions in the form of program annotations: programs are annotated with global program properties and methods are annotated with local method properties. We use a JML–like syntax for annotations (cf. [15]). ProMoVer is procedure–modular in the sense that correctness of the global program property is relativized on the local properties of the individual methods. Thus, the overall verification task naturally divides into two independent subtasks:

(i) a check that each method implementation satisfies its local property, and
(ii) a check that the composition of local properties entails the global property.

Notice that the second subtask only relies on the local properties and does not require the implementations of the individual methods. Thus, changing a method implementation does not require the global property to be reverified, only the local property. If the second subtask fails, ProMoVer provides a counter example in the form of a program behavior that violates the respective property.

In addition to the properties, the technique also requires global and local interfaces to be specified. A global interface consists of a list of the methods provided (i.e., implemented) and required (i.e., used) by the program. The local interface of method \( m \) contains a list of the methods required by the method (as the provided method is obvious). The user only has to specify the local required methods, the global interface can be deduced from the local method information.

Example 1. Consider the annotated Java program in Figure 1. It consists of two methods, \( \text{even} \) and \( \text{odd} \). The program is annotated with a global control flow safety property, and every method is annotated with a local property and an interface specifying the required methods. As mentioned above, the global interface is extracted from the local interfaces.

This section only gives an intuitive description of the properties specified in the example; a formal definition of the temporal logic LTL is given below in Definition 4. The global property expresses that “in every program execution
Fig. 1: A simple annotated Java program

starting in method `even`, the first call is not to method `even` itself”. The local property of method `even` expresses that “method `even` can only call method `odd`, and after returning from the call, no other method can be called”. The local property of method `odd` is symmetric.

As explained above, the annotated program is correct if (i) methods `even` and `odd` meet their respective local properties, and (ii) the composition of local properties entails the global one. In fact, the annotated program is correct and our tool therefore returns an affirmative result.

Example 2. If we now change in the previous example the global property to “in every program execution starting in method `even`, no call to method `odd` is made”, the tool detects this change and only rechecks the global property for the already computed composition of local properties. This check fails, since the annotated program is now incorrect. To demonstrate this ProMoVer returns the following program execution that is allowed by the local properties, but violates the global one, as a counter example (adapted for user understandability by replacing program points with the names of the methods they belong to, cf. Definition 3):

\[
\text{(even, c)} \xrightarrow{\text{even call odd}} \text{(odd, even)} \xrightarrow{\text{odd ret even}} \text{(even, c)}
\]

3 Framework for Modular Specification and Verification

Next, we briefly present the formal framework underlying the ProMoVer tool that supports this style of procedure-modular verification. It is heavily based on our earlier work on compositional verification [9, 8].

3.1 Program Model and Logic

First, we formally define the program model and property specification logic.
Definition 1 (Model). A model is a (Kripke) structure $M = (S, L, \rightarrow, A, \lambda)$ where $S$ is a set of states, $L$ a set of labels, $\rightarrow \subseteq S \times L \times S$ a labeled transition relation, $A$ a set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state $s$ the set of atomic propositions that hold in $s$. An initialized model is a pair $(M, E)$ with $A$ a model and $E \subseteq S$ a set of initial states.

Our program model is based on the notion of flow graph, abstracting away from all data in the original program. It is essentially a collection of method graphs, one for each method of the program. Let $\text{Meth}$ be a countably infinite set of methods names. A method graph is an instance of the general notion of initialized model.

Definition 2 (Method graph). A method graph for method $m \in \text{Meth}$ over a set $M \subseteq \text{Meth}$ of method names is an initialized model $(M_m, E_m)$ where $M_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ is a non-empty set of entry nodes of $m$. $V_m$ is the set of control nodes of $m$, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are return points.

Notice that methods can have multiple entry points. Flow graphs that are extracted from program source have single entry points, but the maximal models that we generate for compositional verification can have multiple entry points.

Every flow graph $G$ is equipped with an interface $I = (I^+, I^-)$, denoted $G : I$, where $I^+, I^- \subseteq \text{Meth}$ are the provided and externally required methods, respectively. These are needed to construct maximal flow graphs (see Section 3.2).

A flow graph is closed if its interface does not require any methods, and it is open otherwise. Flow graph composition is defined as the disjoint union $\uplus$ of their method graphs.

Example 3. Figure 2 shows the flow graph of the program from Figure 1. Its interface is $\{(\text{even, odd})\}, \emptyset$, thus the flow graph is closed. It consists of two method graphs, for method even and method odd, respectively. Entry nodes are depicted as usual by incoming edges without source.

Flow graph behavior is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label $\tau$ for internal transfer of control, $m_1$ call $m_2$ for the invocation of method $m_2$ by method $m_1$ when method $m_2$ is provided by the program and $m_1$ call! $m_2$ when method $m_2$ is external, and $m_2$ ret $m_1$ respectively $m_2$ ret? $m_1$ for the corresponding return from the call.

Definition 3 (Behavior). Let $G = (M, E) : (I^+, I^-)$ be a flow graph such that $M = (V, L, \rightarrow, A, \lambda)$. The behavior of $G$ is defined as initialized model $b(G) =$ Fig. 2: Flow graph of EvenOdd
$(\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = (V \cup I^-) \times V^*$, i.e., states are pairs of control points $v$ or required method names $m$, and stacks $\sigma$, $L_b = \{ m_1 k m_2 | k \in \{ \text{call, ret} \}, m_1, m_2 \in I^+ \} \cup \{ m_1 \text{ call} \ m_2 | m_1 \in I^+, m_2 \in I^- \} \cup \{ m_2 \text{ ret} | m_1 | m_1 \in I^+, m_2 \in I^- \} \cup \{ \tau \}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:

\[
\begin{align*}
\text{[transfer]} \quad (v, \sigma) &\xrightarrow{\tau} (v', \sigma) \quad \text{if } m \in I^+, v \xrightarrow{\tau} m, v', v \models -r \\
\text{[call]} \quad (v_1, \sigma) &\xrightarrow{\text{m1\ call\ m2}} (v_2, v_1') \quad \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{\text{m2}} m_1, v_1', v_1 \models -r, v_2 \models m_2, v_2 \in E \\
\text{[ret]} \quad (v_2, v_1 \cdot \sigma) &\xrightarrow{\text{m2\ ret\ m1}} (v_1, \sigma) \quad \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \land r, v_1 \models m_1 \\
\text{[call]} \quad (v_1, \sigma) &\xrightarrow{\text{m1\ call\ m2}} (m_2, v_1') \quad \text{if } m_1 \in I^+, m_2 \in I^- \not\models m_2, v_1 \xrightarrow{\text{m2}} m_1, v_1', v_1 \models -r \\
\text{[ret?]}} \quad (m_2, v_1 \cdot \sigma) &\xrightarrow{\text{m2\ ret\ m1}} (v_1, \sigma) \quad \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \models m_1 \\

\text{The set of initial states is defined by } E_b = E \times \{ \varepsilon \}, \text{ where } \varepsilon \text{ denotes the empty sequence over } V \cup I^-.
\]

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behaviour. This simplification is justified, since we abstract away from data in the model and the behaviour is thus context-free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

**Example 4.** Consider the flow graph from Example 3. An example run through its (branching, infinite–state) behavior, from an initial to a final state, is:

\[
(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even\ call\ odd}} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau} (v_8, v_3) \xrightarrow{\text{odd\ ret\ even}} (v_3, \varepsilon)
\]

Now, consider just the method graph of method even as an open flow graph, having interface $\{ \text{(even}, \{ \text{odd}) \}$. The local contribution of method even to the above global behavior is the following run:

\[
(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even\ call\ odd}} (\text{odd}, v_3) \xrightarrow{\text{odd\ ret\ even}} (v_3, \varepsilon)
\]

**Pushdown systems (PDS)** are an alternative way to express flow graph behavior. We exploit this by using PDS model checking, concretely the tool MOPED [13], for verifying program behavior against temporal formulas.

As mentioned above, safety properties can be expressed in many different formalisms. In this paper, we use safety LTL which consists of the safety–fragment of Linear Temporal Logic (LTL), using the weak until–operator. Internally, however, the whole machinery is based on the safety fragment of the modal $\mu$–calculus. Safety LTL is somewhat less expressive than the latter fragment and can be uniformly encoded in it. This translation is implemented as part of PtoMoVeR. In our LTL formulas, we use an additional atomic proposition entry that holds for entry nodes. It is removed by the translation into the modal $\mu$–calculus.
Definition 4 (Safety LTL). Let \( p \in A_b \cup \{\text{entry}\} \) and \( m \in M \). The formulae of Safety LTL are inductively defined by:

\[
\phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 \mathcal{W} \phi_2
\]

Satisfaction on states \((\mathcal{M}_b, s)\) |= \( \phi \) for LTL is defined in the standard fashion [20]:

- formula \( X \phi \) holds of state \( s \) in model \( \mathcal{M}_b \) if \( \phi \) holds in the next state of every run starting in \( s \);
- \( G \phi \) holds if for every run starting in \( s \), \( \phi \) holds in all states of the run; and
- \( \phi \mathcal{W} \psi \) holds in \( s \) if for every run starting in \( s \), either \( \phi \) holds in all states of the run, or \( \psi \) holds in some state and \( \phi \) holds in all previous states.

Example 5. Consider the global property of class \texttt{EvenOdd} in Figure 1 (where \texttt{&&} is ASCII notation for \( \land \)) and its intuitive meaning given in Example 1. By the way flow graphs are extracted and constructed, entry nodes are only accessible via calls; hence, if control starts and remains in method \texttt{even}, execution can be at an entry node only as the result of a self-call. The formula thus expresses precisely that “if program execution starts in method \texttt{even}, method \texttt{even} is not called until method \texttt{odd} is reached”, which coincides with the interpretation given in Example 1.

3.2 Compositional Verification

Our method for \textit{algorithmic compositional verification} is based on the construction of maximal flow graphs from component properties. For a given property \( \psi \) and interface \( I \), consider the set of all flow graphs with interface \( I \) satisfying \( \psi \).

A \textit{maximal flow graph} for \( \psi \) and \( I \), denoted \( \text{Max}(\psi, I) \), satisfies exactly those properties that hold for all members of the set. Thus, the maximal flow graph can be used as a representative of the set for the purpose of property verification. For details the reader is referred to [9].

For a system with \( k \) components, our principle of compositional verification based on maximal flow graphs can be presented as a proof rule with \( k + 1 \) premises, that states that the composition of components \( G_1 : I_1, ..., G_k : I_k \) satisfies a global property \( \phi \) if there are local properties \( \psi_i \) such that (i) each component \( G_i \) satisfies its local property \( \psi_i \), and (ii) the composition of the \( k \) maximal flow graphs \( \text{Max}(\psi_i, I_i) \) satisfies \( \phi \).

\[
G_1 \models \psi_1 \cdot \cdot \cdot G_k \models \psi_k \quad \bigcup_{i=1,...,k} \text{Max}(\psi_i, I_i) \models \phi \\
\bigcup_{i=1,...,k} G_i \models \phi
\]

As mentioned above, in the context of \texttt{ProMoVer}, we consider individual program methods as components. If we instantiate the above compositional verification principle to procedure-modular verification, we obtain the verification tasks stated informally in Section 2 (where \( M \) is the set of program methods, with \( k = |M| \), and \( \psi_i \) and \( C_i \) are the specification and the implementation of method \( m_i \), respectively):
Annotated Java Program
ProMoVer
YES/NO+ YES/NO+
Prop.
Pre−Processor
Graph Tool
Graph Tool
CWB
Max. Model (i) (ii)
Post−Processor
YES/NO+ Counter ex. or
Method name
Counter ex.
Method name
Local Global
Prop.

Fig. 3: Overview of ProMoVer and its underlying tool set

(i) Checking $C_i \models \psi_i$ for $i = 1, \ldots, k$: For each method $m_i \in M$, (a) extract the method flow graph $G_i$ from $C_i$, and (b) model check $G_i$ against $\psi_i$. For the latter, we exploit the fact that flow graphs are Kripke structures, and apply standard finite–state model checking.

(ii) Checking $\bigcup_{i=1}^{k} \max(\psi_i, I_i) \models \phi$: (a) Construct maximal flow graphs $\max(\psi_i, I_i)$ for all method specifications $\psi_i$ and interfaces $I_i$, then (b) compose the graphs, resulting in flow graph $G_{\max}$, and finally (c) model check $G_{\max}$ against global property $\phi$. For the latter, represent the behavior of $G_{\max}$ as a PDS and use a standard PDS model checker.

Example 6. Consider again the annotated Java program from Example 1. ProMoVer first extracts the method flow graphs of methods even and odd, denoted $G_{\text{even}}$ and $G_{\text{odd}}$, respectively. Next, ProMoVer checks $G_{\text{even}} \models \psi_{\text{even}}$ and $G_{\text{odd}} \models \psi_{\text{odd}}$ by standard finite state model checking. Independently, it constructs the maximal flow graphs of methods even and odd, denoted $\max(\psi_{\text{even}}, I_{\text{even}})$ and $\max(\psi_{\text{odd}}, I_{\text{odd}})$, respectively, and composes the graphs to obtain $G_{\max} = \max(\psi_{\text{even}}, I_{\text{even}}) \oplus \max(\psi_{\text{odd}}, I_{\text{odd}})$. Finally, ProMoVer translates $G_{\max}$ to a PDS and model checks the latter against the global property.

Our framework provides an inlining based technique for abstracting away private methods, regarding these merely as a means for implementing the public ones (see [9] for details). ProMoVer implements this, requiring only the public methods to be specified with local properties.

4 The ProMoVer Tool

Next we describe how ProMoVer works internally. As mentioned above, it is build as a wrapper on top of cvpp [12]. ProMoVer is implemented in Python and can be tested online via a web interface [18].

Figure 3 gives a schematic overview of how ProMoVer wraps up the individual cvpp tools. As input, ProMoVer accepts annotated Java programs
as exemplified in Section 2. The pre-processer uses the Java Doclet API [6] to parse the annotations and passes the properties and interfaces to the other tools.

Task (i) starts by invoking the Analyzer tool to extract the method graphs of the program. This builds on Sawja [11] to extract flow graphs from Java bytecode. Then, the Graph Tool is used, which implements a collection of algorithms on flow graphs, including flow graph composition ⊎ and translations of flow graphs into different formats. Here it is used to abstract away from private methods: all calls to private methods are inlined into the code of the public methods, thus producing method flow graphs that only describe the public behavior [9]. Next, the Graph Tool translates the public flow graph of each method into a CCS model. These are then model checked against the respective local method specifications using the Concurrency Workbench (cwb) [4].

Task (ii) starts by constructing a maximal flow graph for every method using the Maximal Model Tool. Then the Graph Tool composes the generated flow graphs and converts the result into a PDS. Finally Moped [13] is used to model check the PDS against the global property.

The post-processer collects all model checking results and converts these into a user-understandable format. It returns a positive result if all collected model checking tasks succeed, and a negative one otherwise. If one of the local model checking tasks fails, ProMoVer returns the name of the method that violates its specification. If the global model checking task fails, ProMoVer returns the counter example provided by Moped, transformed into a program execution.

Procedure-modular verification splits the overall verification problem into a number of smaller verification tasks. To exploit this in the context of program evolution, we provide a mechanism for proof storage and reuse. The first time that a program is verified by ProMoVer, all extracted method flow graphs and constructed maximal flow graphs are stored. If later the implementation of a method \( m \) is changed, the new method flow graph is extracted and checked against \( m \)'s local property. If the local specification \( \phi_m \) of method \( m \) is changed, the existing flow graph of method \( m \) is model checked against \( \phi_m \), and the maximal flow graph of \( m \) is constructed from \( \phi_m \) and then composed with the other maximal flow graphs (recovered from storage). Finally, the result of the composition is model checked against the global property.

To reduce the effort to write specifications, we provide an automatic specification extractor. This extracts the (over-approximated) order of method invocations in a method’s implementation. The extracted specifications can be inspected by the user to identify and remove superfluous dependencies. ProMoVer extracts the specifications in the form of modal equation systems, which serve as input for the construction of maximal flow graphs. In the future, we plan to also extract to other specification languages such as LTL.

5 Experimental results with ProMoVer

We used ProMoVer to verify a standard control flow safety property on a number of Java Card applications. Java Card technology provides a secure environ-
ment to support applications on smart cards, developed by Sun Microsystems. It is one of the leading interoperable platforms for smart cards. Many smart card applications are security–critical, implementing for example e-commerce applications.

As mentioned above, for platforms such as Java Card, collections of control flow safety properties exist that programs should adhere to in order to provide minimal security requirements. We focus on such a property of the Java Card transaction mechanism. This mechanism ensures that data remains consistent upon power loss. Safe use of it demands that certain methods are not called within a transaction. We show how this global safety property can be expressed in our setting, and be verified with PROMoVer for several applications, where we apply specification extraction to annotate the public methods of the applications.

The Java Card Transaction Mechanism. Smart cards have two types of writable memory, persistent memory (EEPROM or Flash) and transient memory (RAM). The Java Card memory model adheres to this. Transient memory needs constant power supply to store information, while persistent memory can store data without power. Smart cards do not have their own power supply; they depend on the external source that comes from the card reader device. Therefore, a problem known as card tear may occur: a power loss when the card is suddenly disconnected from the card reader. If a card tear occurs in the middle of updating data from transient to persistent memory, the data stored in transient memory is lost and may cause the smart card to be in an inconsistent state.

The transaction mechanism is provided to prevent this. It can be used to ensure that several updates are executed as a single atomic operation, i.e., either all updates are performed or none. The mechanism is provided through methods `beginTransaction` for beginning a transaction, `commitTransaction` for ending a transaction with performed updates, and `abortTransaction` for ending a transaction with discarded updates [10] – all declared in class `JCSystem` of the Java Card API.

In addition, the Java Card API contains some non–atomic methods that cannot be used when a transaction is in progress. Notably, the class `javacard.framework.Util`, that provides functionality to store and update byte arrays, contains methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` that may not be used within a transaction (for safe array updating within a transaction, the class provides the atomic method `arrayCopy`). We use PROMoVer to verify in a procedure–modular way that applications comply with this Safe Transaction Policy.

The Applications. For this experiment we use several public examples of Java Card applications. All are realistic e-commerce applications developed by Sun Microsystems to demonstrate the use of the Java Card environment for developing e-commerce applications. AccountAccessor is an application to keep track of account information. It is to be used by a wireless device connected via a network service. It contains methods to look up and to modify the account balance. TransitApplet implements the on-card part of a system that connects to
Table 1: Applications details

<table>
<thead>
<tr>
<th>Application</th>
<th>#LoC</th>
<th>#Methods (Public)</th>
<th>#Calls (Relevant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountAccessor</td>
<td>190</td>
<td>9 (7)</td>
<td>38 (4)</td>
</tr>
<tr>
<td>TransitApplet</td>
<td>918</td>
<td>18 (5)</td>
<td>106 (5)</td>
</tr>
<tr>
<td>JavaPurse</td>
<td>884</td>
<td>19 (9)</td>
<td>190 (25)</td>
</tr>
</tbody>
</table>

an authenticated terminal and provides account information and operations to modify the account balance. **JavaPurse** is a smart card electronic purse application providing secure money transfers. It contains a balance record denoting the user’s current and maximum credits, and methods to initialize, perform and complete a secure transaction. Further, it also contains methods to update information related to a loyalty program, and to validate and update the values of transactions, balance and PIN code.

Table 1 shows information about the size, number of methods (total and public), and number of method invocations (total and relevant for the global property) of these applications.

**Specification of Safe Transaction Policy.** As discussed above, we want to ensure formally that the non-atomic methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not invoked within a transaction. Hence, applications have to adhere to the following global control flow safety property:

In every program execution, after a transaction begins, methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not called until the transaction ends.

This safety property can be expressed formally with the following LTL formula:

\[
G (\text{beginTransaction} \rightarrow ((\neg \text{arrayCopyNonAtomic} \land \neg \text{arrayFillNonAtomic}) W \text{commitTransaction}))
\]

**Extracting Local Method Specifications.** The specification extractor is used to obtain local specifications for every public method. Basically, these describe the order of method invocations. We inspected those for immaterial orderings, and translated the adjusted representations into safety LTL. The intention is that local method specifications capture the allowed sequences of method calls made from within the specified method, but in an abstract way, allowing for possible evolution of the method implementations.

**Verification Results.** After annotating the applications, they are passed to ProMoVer. The tool extracts the flow graph of the applications, and partitions them into the individual method graphs to verify adherence to the local properties. Further, for each local property a maximal flow graph is constructed, and their composition is verified w.r.t. the global property above. The statistics for these verifications are given summarized in Table 2. The table shows: the time
Table 2: Verification Results

<table>
<thead>
<tr>
<th>Application</th>
<th>PPT</th>
<th>GE</th>
<th>#NEF</th>
<th>LMC</th>
<th>MFC</th>
<th>#NMF</th>
<th>GMC</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountAccessor</td>
<td>1.4</td>
<td>3.8</td>
<td>435</td>
<td>0.5</td>
<td>0.7</td>
<td></td>
<td>20</td>
<td>0.9</td>
</tr>
<tr>
<td>TransitApplet</td>
<td>1.4</td>
<td>4.7</td>
<td>897</td>
<td>0.3</td>
<td>0.9</td>
<td>30</td>
<td>0.9</td>
<td>13.2</td>
</tr>
<tr>
<td>JavaPurse</td>
<td>1.5</td>
<td>6.5</td>
<td>1543</td>
<td>0.5</td>
<td>13.0</td>
<td>48</td>
<td>1.1</td>
<td>22.5</td>
</tr>
</tbody>
</table>

Table 3: Proof Reuse Results

<table>
<thead>
<tr>
<th>Application</th>
<th>New TT</th>
<th>% TT</th>
<th>MFC</th>
<th>New TT</th>
<th>% TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountAccessor</td>
<td>6.0</td>
<td>68</td>
<td>0.1</td>
<td>4.6</td>
<td>52</td>
</tr>
<tr>
<td>TransitApplet</td>
<td>7.2</td>
<td>54</td>
<td>0.1</td>
<td>5.0</td>
<td>37</td>
</tr>
<tr>
<td>JavaPurse</td>
<td>9.0</td>
<td>40</td>
<td>0.1</td>
<td>5.4</td>
<td>24</td>
</tr>
</tbody>
</table>

spent by the pre–processor (PPT) and the graph extractor (GE), the number of nodes in the extracted flow graphs (#NEF), the time spent for local model checking (LMC) and for constructing maximal flow graphs (MFC), the number of nodes in the maximal flow graph composition (#NMF), the time spent for global model checking (GMC), and the total time spent for the whole verification task including conversions between formats and post–processing (TT). All times are in seconds, and were obtained on a SUN SPARC machine.

We also experimentally evaluated the advantages of proof storage and reuse mechanism. After the first verification, when method and maximal flow graphs are stored, for each application, we once changed the source code and once the local specification of a public method, and used ProMoVer to reverify the application. The result of proof reuse are shown in Table 3. The numbers show that proof reuse can reduce significantly the verification time for larger applications.

6 Conclusion

This paper describes ProMoVer, a tool that supports automatic procedure–modular verification of control flow safety properties of sequences of method invocations. ProMoVer takes as input a Java program annotated with temporal correctness assertions. It essentially implements a particular verification scenario for the cvpp tool set that supports compositional verification of programs with procedures [9].

Modularity is understood here as the relativization of global program correctness properties on the correctness of its components, and is seen as the key to program verification in the presence of static and or dynamic variability due to code evolution, code customization for many users, or as yet unknown or unavailable code such as mobile code. We illustrate two important points: (i) temporal safety properties provide a meaningful abstraction for individual methods:
and (ii) procedure–modular verification of temporal safety properties can be performed automatically. Moreover, ProMoVer implements a mechanism for proof storage and reuse, so that only relevant parts have to be reverified after a system change. This makes the verification method advocated by ProMoVer suitable to be used in a context where systems evolve frequently, as is the case e.g., for software product lines or mobile code. The modularity of the verification allows an independent evolution of the implementations of the individual methods, only requiring the re-establishment of their local correctness.

We believe that writing properties at the procedure–level is intuitive for a programmer. Still, to decrease the effort of annotating programs, we provide support for specification extraction in the case of post–hoc specification of already implemented methods, an inlining–based private method abstraction that requires only public methods to be specified, and a library of standard global safety properties.

Experiments with realistic Java Card applications show that useful safety properties of such programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer.

Still, some issues remain to be resolved in order to increase the utility of ProMoVer. Both for pre– and post–hoc method specification, notations based on automata or process algebra may prove more convenient than LTL, and may also allow more efficient maximal flow graph construction. Ultimately, our goal is that all specifications (local and global) can be written in various temporal logics and notations, or to use patterns to abbreviate common specification idioms. The tool set will provide translations into the underlying uniform logic, which is currently the safety fragment of the modal μ-calculus. However, because of limitations on the currently available PDS model checkers, global properties have at present to be written in LTL.

Many important safety properties require program data to be taken into account. As a first step towards handling data, work has begun on extending our verification framework and tool set to Boolean programs. We are also currently investigating how to generalize our method for the program model of Rot et al. that models object references in the presence of unbounded object creation [17].

Finally, to investigate the scalability of the approach, we plan to perform a significantly larger case study.

Acknowledgment We are indebted to Wojciech Mostowski and Erik Poll for their help in finding a suitable case study, to Afshin Amighi and Pedro de Carvalho Gomes for helping with the implementation of cvpp and ProMoVer, and to Stefan Schwoon for adapting the input language of Moped to our needs.

References

17. J. Rot, F. de Boer, and M. Bonsangue. A pushdown system representation for unbounded object creation. In Informal pre-proceedings of Formal Verification of Object-Oriented Software (FoVeOOS ’10), 2010.
C.3 Compositional Verification of Software Product Lines
Compositional Algorithmic Verification of Software Product Lines*

Ina Schaefer¹, Dilian Gurov², and Siavash Soleimanifard²

¹ Chalmers University of Technology, Gothenburg, Sweden
² Royal Institute of Technology, Stockholm, Sweden

Abstract. Software product line engineering allows large software systems to be developed and adapted for varying customer needs. The products of a software product line can be described by a hierarchical variability model specifying the commonalities and variabilities between the individual products. The number of products generated by such a hierarchical model is exponential in its size, which poses a serious challenge to software product line analysis and verification. For an analysis technique to scale, the effort has to be linear in the size of the model rather than in the number of products it generates. This is only possible if compositional verification techniques are applied that allow the analysis of products to be relativized on the properties of their variation points. In this paper, we propose simple hierarchical variability models (SHVM) with explicit variation points as a novel way to describe a set of products consisting of sets of methods. We generalize a previously developed compositional technique and tool set for the automatic verification of control-flow based temporal safety properties to product lines defined by SHVMs. The desired scalability is achieved by variation point specifications that relativize the global product property towards the associated variants. We evaluate the proposed technique on a number of test cases.

1 Introduction

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to adapt to their application context. Software product line engineering [19] aims at developing a set of systems variants with well-defined commonalities and variabilities by managed reuse in order to decrease time to market and to improve quality. During family engineering reusable core assets are developed, that are used to realize the actual products during application engineering.

A software product line can be described by a hierarchical variability model. In this model, on each level of hierarchy the commonalities of the products are specified in a common core, while the variabilities are represented by explicit

* This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) and the EU project FP7-231620 HATS.
variation points. Each variation point is associated with a set of variants that represent choices for realizing the variation points in different products. A variant can itself contain commonalities defined in a common core and variabilities specified by variation points introducing a new level of hierarchy. The number of products defined by a hierarchical variability model is exponential in the size of the model. This exponential explosion of the number of products poses serious problems to ensure critical product requirements by static analysis or other formal verification techniques. In general, it is infeasible to verify all products individually. Formal verification techniques only scale if their complexity is linear in the size of the hierarchical variability model and not in the number of products, which is exponential in the size of the variability model. In order to achieve this scalability, the verification techniques have to be compositional allowing to relativize the product properties towards properties of variation points.

In this paper, we generalize a previously developed compositional verification technique (and the corresponding tool set) for the automatic verification of control-flow based temporal safety properties [12, 14] to the compositional verification of hierarchically defined product lines. We propose simple hierarchical variability models (SHVM) as a novel way to specify products consisting of sets of public and private methods by defining common core methods and explicit variation points on different hierarchical levels. The properties that can be handled fully automatically specify illegal sequences of method invocations, such as improper usage of API methods, in terms of temporal logic formulas, abstracting from the computed data. Compositionality, and in particular the ability to relativize global SHVM properties on local assumptions for the core methods and the variation points, is achieved by means of maximal flow graphs that are derived algorithmically from the local assumptions. The flow graphs replace the assumptions when verifying global properties. The local specifications of core methods are verified by extracting flow graphs from the method implementations and model checking the induced behaviours against their specification.

The paper is organized as follows. In Section 2, we present SHVMs to hierarchically represent product lines. In Section 3, we describe the foundations of our compositional verification technique. In Section 4, we explain the compositional verification procedure for product lines. In Section 5, we present tool support and an evaluation of the compositional verification technique. In Section 6, we review related work and conclude the paper in Section 7.

2 Hierarchical Variability Modelling

A product in the context of this work is defined by a set of methods. Products are not necessarily closed, i.e., they may still require additional methods such as API methods. A method \( m \) from a set of methods \( \text{Meth} \) is understood as a method definition, consisting of a method name, the types of the return value and the parameters, and its implementation (method body). The methods of a product are partitioned into public and private methods. Public methods are visible to the outside of the product, while private methods are only visible within
products and can be viewed as a means of implementing the public methods. For a product, the methods defined in the product are called provided, while the called methods that are not provided themselves are referred to as required.

A product line $PF$ is defined as a set of method sets $PF \subseteq 2^{Meth}$ and can be represented by a hierarchical variability model, with the common methods of all products captured by a core set of methods separated into public and private methods. The differences between the products are represented by variation points. To each variation point, a set of variants is attached. The variants represent different possibilities to realize the variability described by this variation point. A variant can either comprise a set of core methods or be a hierarchical variability model itself, i.e., consisting of core set of methods and a set of variation points. A product is derived by resolving the variabilities, i.e., by selecting variants at the variation points on all levels of hierarchy.

In this paper, we introduce a variant of the hierarchical variability modelling approach called simple hierarchical variability model (SHVM). An SHVM is a hierarchical variability model that requires exactly one variant to be selected at each variation point to obtain a product. In an SHVM, there is no means for defining constraints between different variants and variation points to represent that the selection of a variant at one variation point requires a specific variant at another variation point to be selected, thus restricting the number of derivable products.

**Definition 1 (Simple Hierarchical Variability Model).** A simple hierarchical variability model (SHVM) $S$ is inductively defined as:

(i) a ground model consisting of a core set of methods $M_C = (M_{pub}, M_{priv})$, partitioned into public and private methods $M_{pub}, M_{priv} \subseteq Meth$, or

(ii) a pair $(M_C, \{VP_1, \ldots, VP_N\})$ consisting of a core set of methods $M_C$ defined as above, and a non-empty set of variation points. Every variation point $VP_i$ is a set of at least 2 SHVMs called variants.

The variant interface of a pair $(M_C, \{VP_1, \ldots, VP_N\})$ is defined as the union of all sets of provided and required methods in the core and the variation points. We assume the following two well-formedness constraints on SHVMs. First, all variants attached to a variation point provide and require the same set of public methods. This guarantees that the variation point provides the same set of methods independently of the selected variant and ensures that in any product the methods required by a variant (that should not be externally provided, like API calls) are indeed provided by some other variant. This interface is called the variation point interface. Second, in order to enforce that a derivable product does not contain several methods with the same name, it is required that the provided methods in each variation point interface are disjoint from each other and the core method set.

**Example 1.** As running example throughout this paper, we consider a product line of cash desks that is a simplified version of the trading system product line case study proposed in [20]. The cash desks process purchases by retrieving the
Fig. 1: The Cashdesk SHVM

The common purchase process of all cash desks is modelled by the public core method \texttt{sale}. The private methods \texttt{updateStock} and \texttt{writeReceipt} represent internal details of the sale process. The two variation points \texttt{@EnterProducts} and \texttt{@Payment} represent the variabilities of the cash desks. The variation point \texttt{@EnterProducts} has the associated variants \texttt{Keyboard} and \texttt{Scanner} for entering product by keyboard or by scanner. Both provide the public method \texttt{enterProd} that is internally realized by the different private methods \texttt{useKeyboard} and \texttt{useScanner}, respectively. Similarly, the variation point \texttt{@EnterProducts} has the two associated variants \texttt{Cash} and \texttt{Card} that both provide the public method \texttt{payment} which is internally realized by different private methods in the two variants.

\begin{verbatim}
CashDesk = (((sale),{updateStock,writeReceipt}),
\{@EnterProducts,@Payment\})
where @EnterProducts = \{Keyboard,Scanner\}
    @Payment = \{Cash,Card\}
Keyboard = ((\{enterProd\},\{useKeyboard\}))
Scanner = ((\{enterProd\},\{useScanner\}))
Cash = ((\{payment\},\{cashPay\}))
Card = ((\{payment\},\{enterCard,cardPay\}))
\end{verbatim}

prices for all items to be purchased and calculating the total price. After the customer has paid, a receipt is printed and the stock is updated accordingly. The commonality of all cash desks is that every purchase is processed following the same process. However, the cash desks differ in the way how the items are entered, either using a keyboard or using a scanner, and how the payment is received, either by cash or by credit card. This set of cash desks is defined by an SHVM as follows:
An SVHM can be seen as a tri-partite directed graph having an SHVM-node as root, where SHVM-nodes have one core methods leaf child (split in public and private methods) and optional VP-node children that have two or more SHVM-node children. For the cashdesk example, a graphical presentation is shown in Figure 1. In the figure, SHVM-nodes are depicted by rounded boxes, core methods nodes by ovals, and VP-nodes by diamonds. The dotted rounded boxes depict what we call modules of the SHVM, defining the boundaries between SHVMs at different levels of hierarchy. The size of an SHVM is defined as the number of modules in its graph.

An SHVM induces a set of products \(PF\) through all possible ways of resolving the variabilities of the SHVM. Variability resolution means to recursively select exactly one variant for each variation point. The set of products induced by a ground model containing only core methods is the singleton set comprising the set of core methods (and, thus, representing one product). The set of products induced by a variation point is the union of the product sets induced by its variants. Finally, the set of products induced by an SHVM with a non-empty set of variation points is the set of all products consisting of the core methods and of exactly one product from the set induced by each variation point.

**Definition 2 (Variability Resolution).** Let \(S\) be an SHVM as defined above. The set \(\text{products}(S) \subseteq 2^{\text{Meth}}\) induced by \(S\) is inductively defined as follows:

\[
\begin{align*}
\text{products}(\text{MC}) &= \{ \text{MC} \} \\
\text{products}(\text{VP}) &= \bigcup_{S \in \text{VP}} \text{products}(S) \\
\text{products}(\text{MC}, \{ \text{VP}_1, \ldots, \text{VP}_N \}) &= \{ \text{MC} \cup \bigcup_{1 \leq i \leq N} M_i \mid M_i \in \text{products}(\text{VP}_i) \}
\end{align*}
\]

For a given SHVM, let \(\text{AND}\) and \(\text{OR}\) denote the maximal branching factors at SHVM and variation points nodes, respectively, and let \(\text{ND}\) be its nesting depth. The number of products induced by the SHVM is bound by \(\text{OR} \cdot (\text{AND}^{\text{ND}} - 1)\) and is thus exponential in the size of the SHVM, which is bound by \(\text{OR} \cdot (\text{AND}^{\text{ND}+1} - 1)\).

**Example 2.** The SHVM defined in Example 1 induces the products:

\[
\text{products}(\text{CashDesk}) = \{ P_1, P_2, P_3, P_4 \}
\]

where:

\[
\begin{align*}
P_1 &= \{ \text{sale, updateStock, writeReceipt, enterProdKeyboard, } \\
& \quad \text{useKeyboard, paymentCash, cashPay} \} \\
P_2 &= \{ \text{sale, updateStock, writeReceipt, enterProdScanner, } \\
& \quad \text{useScanner, paymentCash, cashPay} \} \\
P_3 &= \{ \text{sale, updateStock, writeReceipt, enterProdKeyboard, } \\
& \quad \text{useKeyboard, paymentCard, enterCard, cardPay} \} \\
P_4 &= \{ \text{sale, updateStock, writeReceipt, enterProdScanner, } \\
& \quad \text{useScanner, paymentCard, enterCard, cardPay} \}
\end{align*}
\]
To disambiguate methods with the same name, but coming from different variants, we add as subscript the name of the parent SHVM–node, for instance, \texttt{enterProd}\textsubscript{Keyboard} refers to the method \texttt{enterProd} of the variant \texttt{Keyboard}.

Notice that in SHVMs with a small nesting depth as in the example above, the exponential blow–up in the number of products is not observed: With branching factors of 2 and a nesting depth of 1, we have at most 4 products, but 5 modules. However, adding just another level of hierarchy, e.g., variability in the accepted type of cards, immediately results in an explosion (cf. Section 5).

3 Framework for Compositional Verification

This section outlines the theoretical framework for verification of temporal safety properties upon which our compositional verification technique for product lines (described in the next section) is based. It relies on our earlier work on compositional verification (see e.g. [12, 11]).

\textit{Program Model} In order to reason algorithmically about sequences of method invocations, we abstract the set of methods defining our program by ignoring all data. An initialized model serves as an abstract representation of a program’s structure and behaviour.

\textbf{Definition 3 (Model).} A model is a (Kripke) structure \( M = (S,L,\rightarrow,A,\lambda) \) where \( S \) is a set of states, \( L \) a set of labels, \( \rightarrow \subseteq S \times L \times S \) a labeled transition relation, \( A \) a set of atomic propositions, and \( \lambda : S \rightarrow \mathcal{P}(A) \) a valuation, assigning to each state \( s \) the set of atomic propositions that hold in \( s \). An initialized model is a pair \((M,E)\) with \( M \) a model and \( E \subseteq S \) a set of initial states.

A method graph is an instance of an initialized model which is obtained by ignoring all data from a method implementation. A flow graph is a collection of method graphs, one for each method of the program. A flow graph is the abstraction of a set of methods that is analyzed with our technique.

\textbf{Definition 4 (Method graph).} Let Meth be a countably infinite set of methods names. A method graph for method \( m \in \text{Meth} \) over a set of method names \( M \subseteq \text{Meth} \) is an initialized model \((M_m,E_m)\) where \( M_m = (V_m,L_m,\rightarrow_m,A_m,\lambda_m) \) is a finite model and \( E_m \subseteq V_m \) is a non-empty set of entry points of \( m \). \( V_m \) is the set of control nodes of \( m \), \( L_m = M \cup \{\varepsilon\} \), \( A_m = \{m,r\} \), and \( \lambda_m : V_m \rightarrow \mathcal{P}(A_m) \) so that \( m \in \lambda_m(v) \) for all \( v \in V_m \) (i.e., each node is tagged with its method name). The nodes \( v \in V_m \) with \( r \in \lambda_m(v) \) are return points.

Note that methods according to the above definition can have multiple entry points. Flow graphs that are extracted from a program source have single entry points, but the maximal models that we generate for compositional verification can have multiple entry points.

Every flow graph \( \mathcal{G} \) is equipped with an interface \( I = (I^+,I^-) \), denoted \( \mathcal{G} : I \), where \( I^+,I^- \subseteq \text{Meth} \) are the provided and externally required methods,
respectively. Interfaces are needed when constructing maximal flow graphs (see Section 3). A flow graph is closed if its interface does not require any methods, and it is open otherwise. Flow graph composition is defined as the disjoint union \( \uplus \) of their method graphs.

Example 3. Figure 2 shows a simple Java class and the (simplified) flow graph it induces. It consists of two method graphs, for method even and method odd, respectively. Entry nodes are depicted as usual by incoming edges without source. Its interface is \((\{\text{even}, \text{odd}\}, \emptyset)\), thus the flow graph is closed.

The behavior of a flow graph is also defined as an initialized model. We use transition label \( \tau \) for internal transfer of control, \( m_1 \text{ call } m_2 \) for the invocation of method \( m_2 \) by method \( m_1 \) when method \( m_2 \) is provided by the program, \( m_2 \text{ ret } m_1 \) for the corresponding return from the call, and label \( m_1 \text{ caret } m_2 \) for the (atomic) invocation of and return from an external method \( m_2 \) by method \( m_1 \).

Definition 5 (Behavior). Let \( G = (M, E) : (I^+, I^-) \) be a flow graph such that \( M = (V, L, \rightarrow, A, \lambda) \). The behavior of \( G \) is defined as initialized model \( b(G) = (M_b, E_b) \), where \( M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b) \), such that \( S_b = V \times V^* \), i.e., states (or configurations) are pairs of control points \( v \) and stacks \( \sigma \), \( L_b = \{m_1 \mid k \in \text{call, ret}, m_1, m_2 \in I^+ \} \cup \{m_1 \text{ caret } m_2 \mid m_1 \in I^+ \land m_2 \in I^- \} \cup \{\tau\} \), \( A_b = A \), \( \lambda_b((v, \sigma)) = \lambda(v) \), and \( \rightarrow_b \subseteq S_b \times L_b \times S_b \) is defined by the rules:

- \([\text{transfer}]\) \((v, \sigma) \stackrel{\tau}{\longrightarrow} (v', \sigma)\) if \( m \in I^+ \), \( v \stackrel{m}{\rightarrow}_m v' \), \( v \models \neg r \)
- \([\text{call}]\) \((v_1, \sigma) \stackrel{m_1 \text{ call } m_2}{\longrightarrow} (v_2, v'_1, \sigma)\) if \( m_1, m_2 \in I^+ \), \( v_1 \stackrel{m_2}{\rightarrow}_{m_2} v'_1 \), \( v_1 \models \neg r \), \( v_2 \models m_2 \), \( v_2 \in E \)
- \([\text{ret}]\) \((v_2, v_1, \sigma) \stackrel{m_2 \text{ ret } m_1}{\longrightarrow} (v_1, \sigma)\) if \( m_1, m_2 \in I^+ \), \( v_2 \models m_2 \land r \), \( v_1 \models m_1 \)
- \([\text{caret}]\) \((v_1, \sigma) \stackrel{m_1 \text{ caret } m_2}{\longrightarrow} (v'_1, \sigma)\) if \( m_1 \in I^+ \), \( m_2 \in I^- \), \( v_1 \stackrel{m_2}{\rightarrow}_{m_2} v'_1 \), \( v_1 \models \neg r \)
The set of initial configurations is defined by $E_b = E \times \{\varepsilon\}$, where $\varepsilon$ denotes the empty sequence.

Return transitions always hand back control to the caller of the method. Calls to external methods are modeled with caret transitions that jump immediately from the external method invocation to the corresponding return, without considering the intermediate behavior. This treatment of method calls is inspired by the temporal logic $\text{CARet}$ [1]. An alternative way to express flow graph behavior is to use pushdown systems (PDS). This can be exploited by using pushdown system model checking for verifying behavioral properties.

Example 4. Consider the flow graph from Example 3. One example run through its (branching, infinite–state) behavior, from an initial to a final configuration, is:

$$(v_0, \varepsilon) \xrightarrow{\text{even call odd}} (v_1, \varepsilon) \xrightarrow{\text{odd ret even}} (v_2, \varepsilon) \xrightarrow{\text{even caret odd}} (v_3, \varepsilon)$$

Now, consider just the method graph of method $\text{even}$ as an open flow graph, having interface $\{\text{even}\}, \{\text{odd}\}$. The local contribution of method $\text{even}$ to the above global behavior is the following run:

$$(v_0, \varepsilon) \xrightarrow{\text{even caret odd}} (v_1, \varepsilon) \xrightarrow{\text{odd ret even}} (v_2, \varepsilon) \xrightarrow{\text{even caret odd}} (v_3, \varepsilon)$$

We refine this program model to allow an explicit partitioning of method names into public and private ones, and introduce the notions of public interface and public behaviour in order to abstract away from private methods which are used as a means of implementing the desired public behavior. On the flow graph level, such an abstraction is accomplished through inlining of private methods. For details the reader is referred to [12].

Specification The specification language for behavioral properties, around which our compositional verification method is centered, is simulation logic, the fragment of the modal $\mu$–calculus [15] with boxes and greatest fixed–points only. This temporal logic is adequate for expressing safety properties.

Definition 6 (Simulation Logic). The formulae of simulation logic are inductively defined by:

$$\phi ::= p | \neg p | X | \phi_1 \land \phi_2 | \phi_1 \lor \phi_2 | [a]\phi | \nu X. \phi$$

where $p \in A$, $a \in L$ and $X$ ranges over propositional variables.

Satisfaction on states $(M_b, s) \models \phi$ is defined in the standard fashion [15]. For instance, formula $[a]\phi$ holds of state $s$ in model $M_b$ if $\phi$ holds in all states accessible from $s$ via an edge labeled $a$. An initialized model $(M_b, E_b)$ satisfies a formula $\phi$, denoted $(M_b, E_b) \models \phi$, if all its initial configurations $E_b$ satisfy $\phi$.

Safety properties can also be expressed in other formalisms and logics. In this paper, we use the safety–fragment of Linear Temporal Logic (LTL) that uses the weak version of until as an alternative to simulation logic since it is generally easier to understand. The formulae of $s\text{LTL}$ are inductively defined by:

$$\phi ::= \cdots$$
\( \phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 W \phi_2 \)

where \( p \in A_b \). Satisfaction on states \((M_b, s) \models \phi\) is again defined in the standard fashion (see e.g. [23]).

**Compositional Verification** Our method for compositional verification is based on the construction of maximal flow graphs for properties of sets of methods. For a given property \( \psi \) and interface \( I \) consisting of provided and required methods, consider the class of all flow graphs with interface \( I \) satisfying \( \psi \). A maximal flow graph for \( \psi \) and \( I \) is a flow graph \( \text{Max}(\psi, I) \) that satisfies exactly those properties that hold for all members of the class. Thus, the maximal flow graph can be used as a representative of the class for the purpose of checking properties. Using maximal models for compositional verification was first proposed in [10] for finite-state systems, and was generalized for flow graphs in [12, 11].

The main principle of compositional verification based on maximal flow graphs can be presented, for a system that is partitioned into \( k \) sets of methods, as a proof rule with \( k + 1 \) premises:

\[
\begin{align*}
G_1 \models \psi_1 & \quad \cdots \quad G_k \models \psi_k & \quad \bigcup_{i=1,\ldots,k} \text{Max}(\psi_i, I_i) \models \phi \\
\end{align*}
\]

The principle states that the composition of the sets of methods with the respective interfaces \( G_1 : I_1, \ldots, G_k : I_k \) satisfies a global property \( \phi \) if for some local properties \( \psi_i \) satisfied by the corresponding sets of methods \( G_i \), the composition of the maximal flow graphs for \( \psi_i \) and \( I_i \) satisfies property \( \phi \). For details the reader is again referred to [12].

### 4 Compositional Verification of SHVMs

In order to allow scalable verification of a temporal safety property of all products in a product line, we propose a compositional reasoning approach that is linear in the number of modules in the SHVM definition of the product line, rather than linear in the number of products which would be exponential in the number of modules. This approach is an instantiation of the compositional verification principle presented above to SHVMs.

For every module \( \langle M_C, \{ VP_1, \ldots, VP_N \} \rangle \) in the SHVM, a specification has to be provided in order to allow for compositional verification. This comprises a specification for every public method \( m \in M_{pub} \) by a public behavioural property \( \psi_m \) and a public interface \( I_m = (I^+_{m}, I^-_{m}) \) declaring the names of the publicly provided and required methods, and a specification for every variation point \( VP \) by a behavioral property \( \psi_{VP} \) and a public interface \( I_{VP} \), and a specification of the SHVM node itself by a behavioral property \( \phi \) and a public interface \( I \). The SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification. The lop-level SHVM is specified by the global product property that should be verified.
Compositional verification of the SHVM proceeds as follows. For every module \((M_C, \{VP_1, \ldots, VP_N\})\) of the SHVM, perform the following two tasks:

(i) for every public method \(m \in M_{pub}\), extract the method graph \(G_m\) from the implementation of \(m\), then inline the already extracted graphs of the private methods, and finally model check the resulting method graph against the specification \(\psi_m\) of \(m\) to establish \(G_m \models \psi_m\). For the latter, we apply standard finite–state model checking.

(ii) for all public methods \(m \in M_{pub}\) with specification \((I_m, \psi_m)\), construct the maximal method graphs \(\text{Max}(\psi_m, I_m)\), and for all variation points \(VP_i\) with specification \((I_{VP_i}, \psi_{VP_i})\), construct the maximal flow graphs \(\text{Max}(\psi_{VP_i}, I_{VP_i})\). Then, compose the graphs, resulting in flow graph \(G_{\text{Max}}\), and model check the latter against the SHVM property \(\phi\), i.e.,

\[
\bigg( \biguplus_{m \in M_{pub}} \text{Max}(\psi_m, I_m) \uplus \bigg) \bigg( \biguplus_{VP_i \in \{VP_1, \ldots, VP_N\}} \text{Max}(\psi_{VP_i}, I_{VP_i}) \bigg) \models \phi
\]

For properties given in sLTL, we represent the behavior of \(G_{\text{Max}}\) as a PDS and use the LTL-model checker Moped [21].

The total number of verification tasks needed to establish the global product line property is, thus, equal to the number of modules, since we have to complete one verification task per module, while the number of products is exponential in the number of modules.

Example 5. To illustrate our compositional verification approach, we use the cashdesk product line described in Example 1. The global property we want to verify is, that “the entering of products is finished, before the payment process is started”. One can formalize this property in temporal logic as follows:

\[
\varphi_{CD} = \text{sale} \rightarrow (\neg\text{payment} \mathcal{U} (r \land \text{enterProd} \land X \text{sale}))
\]

This formula expresses that if control starts in method \text{sale}, method \text{payment} is never called prior to a return from \text{enterProd} to \text{sale}.

First, we have to specify all public core methods and variation points of the cashdesk SHVM. The specification of the \text{sale} method and the \@EnterProd and \@Payment variation points are as follows:

- The interface of method \text{sale} is \(I_{\text{sale}} = (\{\text{sale}\}, \{\text{enterProd}, \text{payment}\})\). The behavioral property that method \text{sale} has to satisfy is that it invokes method \text{enterProd} and, only after returning, invokes method \text{payment}. Formally, this can be expressed by the sLTL formula:

\[
\varphi_{\text{sale}} = \text{sale} \mathcal{U} \text{enterProd} \mathcal{W} \text{sale} \mathcal{W} \text{payment} \mathcal{W} (G \text{sale})
\]

- The interface of variation point \@EnterProducts is \(I_{\text{EP}} = (\{\text{enterProd}\}, \{\text{payment}\})\). The property required for the variation point is that the
enterProd method never calls the payment method, neither directly nor via a call to one of its non-public methods. Formally, this property can be expressed by the formula:

$$\varphi_{EP} = G \neg payment$$

- The interface of variation point @Payment is $$I_P = \{\{\text{payment}\}\}, \{\text{enterProd}\}$$. Similarly to the variation point above, the property required for this variation point is that the payment method never calls the enterProd method:

$$\varphi_P = G \neg enterProd$$

The variants Keyboard, Scanner, Cash and Card inherit the specification of their SHVM node from the respective variation point specification. The specification of public methods enterProd and payment is similar to the specification of the @EnterProd and @Payment variation points.

First, we have to verify that all public methods satisfy their behavioral property. For the sale method, we have to inline the private methods writeReceipt and updateStock to obtain the method graph of the sale method. Then we check that the method graph satisfies the property $$\varphi_{sale}$$ by finite-state model checking. Similarly, we verify the enterProd and payment methods defined in the variants Keyboard, Scanner, Cash and Card.

Second, we have to establish that all SHVMs satisfy their SHVM specification. For the top-level SHVM, we construct the maximal models for the specifications of the variation points @EnterProducts and @Payment and for the public method $$\varphi_{sale}$$, and model check $$\varphi_{CD}$$ against the composition of these maximal models. The properties of the variants Keyboard, Scanner, Cash and Card are easy to verify because each of them contains only one public method. A maximal model for the specification of this public method is constructed and checked against the inherited variation point property.

5 Tool Support and Evaluation

ProMoVer [22] is a fully automated tool for the procedure–modular verification of control flow temporal safety properties of Java programs. It supports compositional verification by relativizing the correctness of a global program property on properties of individual methods and their public interfaces. All interfaces, local and global properties are provided to the tool as assertions in the form of program annotations. ProMoVer accepts a JML–like syntax for annotations (cf. [17]) as special comments called pragmas.

We have adapted ProMoVer for verifying properties of SHVMs according to the compositionality principle described in Section 4. For this adaptation, we have extended the annotation language to support the definition of core methods, variants and variation points and the associated specifications by designated pragmas. The tool takes as input a source code file in which the SHVM to be analysed is represented by annotations. The product property, the variation point properties and the specifications of the public core methods are also
provided by annotations. Figure 3 shows in the left column the annotation for the @EnterProd variation point, while the annotations for its Keyboard variant with core method enterProd are shown in the right column. ProMoVER fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

```java
/**
 * @variation_point: EnterProd
 * @variation_point_interface: provided enterProd()
 * @variation_point_ltl_prop: G ! payment
 * @variants: Keyboard, Scanner
 */

/** @variant: Keyboard
 * @variant_interface: provided enterProd()
 * @local_interface: required
 * @local_ltl_prop: G ! payment
 */
public int enterProd()
{
...
```

Fig. 3: Annotations for variation point @EnterProd and its variant Keyboard

For evaluating our compositional verification approach, we considered the verification of the safety property explained in Example 5 for different versions of the trading system product line [20]. The product lines of cash desks were described as SHVMs with different hierarchical depths and different total numbers of modules. As a basis, we used the product line described in Example 1 and extended it by an optional coupon handling functionality within the sale method, and a variation point for accepting different card types as a hierarchical refinement of variant Card. For each product line, we compared the time required to verify all induced products individually with the time for compositional verification. The experiments were performed on a SUN SPARC machine.

The results are summarized in Table 1 where CD denotes the product line of Example 1, CD/CH the version with coupon handling, CD/CT the version with different card types and CD/CH/CT the version with coupon handling and different card types. As it can be observed from the table, the processing time $t_{ind}$ for verifying every product individually grows dramatically when new modules and levels of hierarchy are added to the SHVM. This is easily explained by the ana-

<table>
<thead>
<tr>
<th>Product Line</th>
<th>Depth</th>
<th># Modules</th>
<th># Products</th>
<th>$t_{ind}$[s]</th>
<th>$t_{comp}$[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>101</td>
<td>26</td>
</tr>
<tr>
<td>CD/CH</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>206</td>
<td>28</td>
</tr>
<tr>
<td>CD/CT</td>
<td>2</td>
<td>9</td>
<td>11</td>
<td>281</td>
<td>29</td>
</tr>
<tr>
<td>CD/CH/CT</td>
<td>2</td>
<td>11</td>
<td>20</td>
<td>518</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 1: Evaluation Results
lytical bounds presented in Section 2. In contrast, the growth of the processing
time $t_{\text{comp}}$ for compositional SHVM verification is insignificant, since the pre-
processing and flow graph extraction is only performed once by ProMoVer for
the complete SHVM. The experiment suggests that for large software products
comprising many products, the compositional verification technique based on
the SHVM representation of the product line increases efficiency of verification
dramatically.

6 Related Work

The existing approaches to represent product line variability on the artefact level
can be classified into three main directions [25]. First, annotative approaches consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [26, 8], presence conditions [6], or separate variability representations, such as orthogonal variability models [19], define which parts of the model have to be removed to derive the model of a concrete product. Second, compositional approaches [3, 25, 18, 2] associate product fragments with product features which are composed for particular feature configurations. Third, transformational approaches, such as [13], represent variability by rules determining how modelling elements of a base model have to be replaced for a particular product model. In this paper, we pursue an alternative approach to model the variability of a software product line by hierarchical variability modelling in SHVMs. Apart from the Koala component model [24] that defines variabilities of components in a fixed component architecture, there is no previous approach to represent the variability of product lines hierarchically.

Most approaches for algorithmic verification of behavioral requirements of
software product lines rely on an annotative model of the product line comprising
all possible product variants in one model. Existing model checking techniques are adapted to deal with optional behavior defined by variant annotations. For instance, in [7], modal transition systems are extended by variability operators from deontic logic. In [9], the process calculus CCS is extended with a variant operator to represent a family of processes. In [16], transitions of I/O-automata are related to variants. In [5], product families are modeled by transition systems where transitions are labelled with features such that state reachability modulo a set of features can be computed.

These approaches do not scale for large product lines since the used annota-
tive product line models get easily very large. To counter this, Blundell et al. [4]
propose a technique for compositional verification of product features. The behavior of a feature is represented by a state machine to which other features may attach in interface states. For a temporal property of a feature, constraints for the interface states are generated that have to be satisfied by composed features. However, the notion of feature composition the compositionality results are based on is fairly restrictive. In contrast, SHVMs allow the flexible and expressive representation of product variability together with a scalable compositional verification technique.
7 Conclusion

We present a novel hierarchical variability model for software product lines, in which products are specified as sets of public and private methods by defining common core methods and variation points at different hierarchical levels. The model allows to adapt a previously developed method and tool set for compositional verification of procedural programs such that the exponential blow-up required for verifying all products individually is avoided: The number of verification tasks resulting from our method is linear in the size of the variability model rather than in the number of products. This is achieved by the introduction of variation point specifications on which product properties are relativized, and the construction of maximal flow graphs that replace the specifications when model checking specifications on the next higher level of hierarchy. The class of properties that can be handled fully automatically is the class of control flow-based temporal safety properties, specifying illegal sequences of method calls. The input to our verification tool is the description of a product line in form of an annotated Java program defining the variability model and providing the necessary specifications. Our first experiments with the tool show a dramatic gain in performance even for models with a low hierarchical depth. In future work, we are planning to extend our variability model with optional variants and constraints between variants.

Acknowledgement We thank Afshin Amighi for his help with flow graph extraction, and Björn Terelius for his help with obtaining the analytical bounds.

References

C.4 Verification of Software Product Lines with Delta-oriented Slicing
Verification of Software Product Lines with Delta-Oriented Slicing

Daniel Bruns¹, Vladimir Klebanov¹, and Ina Schaefer²,*

¹ Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{bruns,klebanov}@kit.edu
² Chalmers University of Technology, 421 96 Gothenburg, Sweden
schaefer@chalmers.se

Abstract. Software product line (SPL) engineering is a well-known approach to develop industry-size adaptable software systems. SPL are often used in domains where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness. In this paper, we present delta-oriented slicing, an approach to reduce the deductive verification effort across an SPL where individual products are Java programs and their relations are described by deltas. On the specification side, we extend the delta language to deal with formal specifications. On the verification side, we combine proof slicing and similarity-guided proof reuse to ease the verification process.

1 Introduction

A software product line (SPL) [18] is a set of software systems (called products) with well-defined commonalities and variabilities. SPL are often used in domains (e.g., communications, medical, transportation) where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness by any method.

Even without formal verification, the dimensions and complexity of product lines make it essential to model the relationships between products explicitly. One of the authors has been working on the software engineering aspects of SPL [22,23,21]. This has resulted in a modeling approach called delta-oriented programming (Sect. 2). Our current effort aims to exploit the structural information available in an SPL model to reuse verification results obtained from verifying one product when considering another product. Where necessary, we enrich the model with semantical information (such as formal specifications, Sect. 3). Considering other possibilities to verify SPL that are more meta-level (like generic or partial proofs) and require more semantical information, we decided to go on with a more light-weight approach first.

* This author has been supported by Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.
The technology that we are using to illustrate our approach is Java for programming single products, JML [13] for formal specifications and the KeY system [4] for deductive verification. However, we only make the following assumptions about the verification system:

– We concentrate on systems that manipulate an explicit proof object in the proof assistant style, but do discuss systems operating in the verifying compiler style (a verification condition-generating tool chain with an SMT solver at its end).
– We support both ways in which verification systems can treat method calls: using the method contract or inlining the implementation. Using the contract is inherently modular while inlining is not, but it still has its advantages. It is simple, does not force the developer to write “trivial” contracts for helper methods, and reduces the number of commitments that need to be updated as the code evolves.
– Our method is also parametric on how a verification system treats invariants. In the worst case, all methods in the program have to be verified to preserve every invariant, as the invariant vocabulary is (in general) unrestricted. In practice, verification systems use criteria such as visibility, syntax and typing, assignable clauses or ownership to reduce the workload. We simply limit ourselves to requiring that all relevant invariants must be checked.

In our approach we analyze the SPL model to determine which parts of the original product are unchanged in the new product and also do not have to be verified again. This analysis constitutes proof slicing (Sect. 4).

For the modified or otherwise affected product parts, we apply a previously-developed proof reuse technique based on the assumption of similarity between the two implementation variants. (Sect. 5).

We present related work in Sect. 6 and draw conclusions in Sect. 7.

2 Delta-Oriented Programming of Software Product Lines

*Delta-oriented programming* (DOP) [22,23,21] is a novel approach for implementing software product lines. Delta-oriented programming offers an expressive and flexible “programming meta-language” for specifying a set of products. Its aim is to relax the restrictions of currently established SPL description formalisms such as feature-oriented programming (FOP) [3] by adding the explicit possibility to remove parts of a program. For a more detailed comparison between delta-oriented and feature-oriented programming, the reader is referred to [22].

In delta-oriented programming, an SPL is implemented as a *core module* together with a set of *delta modules*. The core module contains a complete product implementation for some valid feature configuration, which can be developed by conventional single-application engineering techniques. Delta modules specify changes to be applied to the core module in order to implement other products.
The notation we use for Java programs constituting individual products is the following:

**Definition 1.** A program is a set of class declarations (further called classes) and a binary inheritance relation on this set. We are primarily interested in the transitive closure of this relation \(\sqsubseteq\) and the transitive reflexive closure \(\sqsubseteq^\ast\). \(A \sqsubseteq B\) means that the class \(A\) is below class \(B\) in the inheritance hierarchy. Abstract classes and interfaces are omitted in this paper for brevity.

A class is a set of field and method declarations (which are built up of names, types, parameters, bodies, etc., as appropriate in Java). If \(C\) is a class declaring a method with signature \(m\), then we will refer to this particular implementation as \(C::m\).\(^1\) Vice versa, we identify the method signature \(m\) with a set of classes in a product that declare a method with that signature: \(C \in m\) if \(C::m \in C\).

Modification operations used in delta modules that we consider in this paper are the following:

– adding/removing a class declaration \(C\): \(\text{adds}(C)\), \(\text{removes}(C)\)
– modifying class \(C\) by

\(^1\) For simplicity, we assume the absence of method overloading. In Java, a class may contain several method implementations with the same identifier and compatible parameter types. This renders the lookup procedure far more complicated; c.f. [8, Sect. 15.12.2].
• adding/removing a field $f$: $\text{adds}(C::f), \text{removes}(C::f)$
• adding/removing a method declaration $m$: $\text{adds}(C::m), \text{removes}(C::m)$
• changing the direct superclass of $C$ to $C': \text{reparents}(C,C')$

On an abstract level, the variability of an SPL is defined by the feature set $F$. Valid member products of an SPL are given by the feature model $F \subseteq 2^F$. Each product uniquely corresponds to a combination of features, also called feature configuration. In the following, we identify products and feature configurations in $F$. Each delta module $d$ contains an application condition $\varphi_d$ (the when clause in concrete syntax), which is a propositional formula over the feature set $F$. The application conditions specify which delta modules are necessary for which features. For every pair of valid products $P_1, P_2 \in F$, $\Delta(P_1, P_2)$ is the set of delta modules that have to be applied to the product $P_1$ in order to obtain a product $P_2$ with a different feature configuration.\(^2\) The original delta language proposal \cite{Baier1999} demands a partial order on deltas to guarantee that the result of applying $\Delta(P_1, P_2)$ is unique, as well as certain other syntactical well-formedness conditions, which we are not concerned with in this paper.

**Example 1.** Our running example in this paper is a delta-oriented product line of bank accounts inspired by \cite{Jung2004}. Figure 1a shows the core module of this SPL with the basic Account class. Figure 1b shows the delta module $D\text{Investment}$ for activating the Investment feature, which accumulates a bonus for each deposit made. Figure 1c contains the result of applying the delta module to the core, which is, again, a conventional Java class. Later on, in Example 3, we will also see the Paycheck feature adding the class Employer as a client of Account.

\section{Delta-Oriented Formal Specification of Software Product Lines}

We use the Java Modeling Language (JML) \cite{Dwyer2001} for the formal specification of product properties. In this work, we concentrate on class invariants and method contracts with pre- and post-conditions. As JML specifications are written directly into Java source files as comments, it is possible to include them in the delta language introduced in Sect. 2. A core module is specified just as a conventional program. An example of a core module with JML specifications can be seen in the first listing of Example 3.

For delta modules, we extend the delta language with the following operations to manipulate specifications:

- adding an invariant to a class: $\text{adds}(C,I)$
- removing an invariant from a class: $\text{removes}(C,I)$

\(^2\) This is a slight generalization of the original delta approach, where deltas could only be applied to the core product.
- adding a contract (pre-/post-condition pair) to a method: \( \text{adds}(C::m, ct) \)
- removing a contract from a method: \( \text{removes}(C::m, ct) \)

Note that we only consider pairs of exactly one pre- and post-condition to be added or removed together. In case one of them is trivial (i.e., true), it is omitted.

**Example 2.** Figure 2 shows the delta module \( \text{DInvestmentSpec} \) changing the specifications in class \( \text{Account} \). It is applied for the same configurations as the code delta \( \text{DInvestment} \), since it has the same application condition.\(^3\)

In general, there is no concordance between code deltas and specification deltas for one product. It is perfectly conceivable to change the code without changing the specification or the other way round. However, there are (at least) the following exceptions where code changes influence the specification:

- Removing a class or a method induces the removal of attached specifications.
- JML enforces behavioral subtyping, i.e., subclasses inherit the specifications of the superclass. Changing the inheritance hierarchy, thus, also changes the specification.
- JML by default enforces non-nullness of fields, variables, etc. Adding a field of reference type to a class automatically creates an implicit invariant about this field.
- Changing a (pure) method changes the semantics of specifications using this method.

![delta DInvestmentSpec when Investment { 
modifies class Account {
  removes //@ ensures bonus == \old(bonus);
  from void addBonus(int x);
  adds //@ requires x >= 0;
    //@ ensures bonus == \old(bonus) + x;
  to void addBonus(int x);
}]

**Fig. 2.** A specification delta adds and removes pre- and post-conditions from a method

### 4 Delta-Oriented Slicing

When a new product is derived by delta application, in general, both the implementation as well as the specification change. However, from the structural information available in the used delta modules, we are able to conservatively infer which specifications of the new product remain valid (i.e., the proofs done for the old product are not affected by the change) and which parts have to be (re-)proven in order to establish the specified properties. We call the latter *delta-oriented slice.*

\(^3\) It is possible to specify code and specification changes in the same delta module. The separation at this point is for presentation reasons.
Slicing originated as a program analysis technique answering the question of which program statements influence the value of a given variable. Our algorithm answers the question of which proofs are influenced by a delta module.

Of course, the simplest and safest way to achieve assurance for a changed product is to redo all proofs. However, at the current state of hardware and deduction technology, this approach is too slow for any product of non-trivial size. Our approach is much less computationally expensive as it only involves a deterministic static analysis of different artifacts. This way, proof slicing can quickly provide feedback to the engineer on what impact a certain change to the product will have.

Proof Modularity
The key to obtaining a sound slicing algorithm is identifying non-modular proof steps. The issue of proof non-modularity arises if the validity of certain proof steps in a verification proof is lost when the program that is to be verified is changed or extended.

The change may be explicit, i.e., concerning the source code of the very method being verified, or implicit, i.e., concerning program entities that are only referenced (e.g., other methods called). Explicit changes are easy to detect, and if they are benign, they can be treated by proof reuse (Sect. 5). Implicit changes are more involved, and their impact depends both on the semantics of the programming language, as well as on the particular verification calculus. Implicit change is the case that we concentrate on in the following.

Proof modularity has been recognized as an issue for quite some time, focusing, naturally, on adding/removing classes and overriding methods. Particularly relevant to our effort are a previous account for the KeY system [20, Sect. 6.2] as well as a comprehensive survey for the KIV system [24, Chap. 6]. As our change vocabulary is larger, we have to address this issue anew. In the KeY system, identifying rules resulting in non-modular proof steps is made easier by the fact that the class declarations and the class hierarchy are not part of the original proof obligation. This information is available in the background (i.e., in the prover implementation) and can be introduced into a logical sequent by rules containing metaconstructs (functions that are not logically specified, but programmed in the prover). These functions make non-modular rules easily identifiable syntactically, which we have done for the KeY rule base. In the KeY calculus, we discern rules giving rise to proof steps whose validity is:

(A) not affected by implicit program changes (rewriting, propositional, and the like, but also many symbolic execution rules, e.g., for conditionals, loops, etc.);

(B) affected by presence or absence of classes regardless of their content;\(^4\)

\(^4\) The rules of this type are rare and the KeY system has only two of them: \texttt{TYPEABSTRACT} and \texttt{ARRAYSTORESTATICANALYSE}. The former allows deducing the dynamic type of an object pointed to by an expression with an abstract static type (this rule produces a disjunction over all subclasses). The latter uses a simple static analysis to check whether an array assignment can throw an \texttt{ArrayStoreException}.
(C) affected by methods declared in classes; these rules are the non-modular method invocation rules inlining the method implementations and simulating dynamic binding;

(D) affected by fields declared in classes regardless whether these fields are used in the program; these are the instance creation rules assigning default values to fields;

(E) affected by inheritance relationship between classes; these are the rules for tackling the inheritance predicate $\sqsubseteq$.

The slicing algorithm is based on these findings.

Other systems encode the class hierarchy as axioms that are part of the proof obligation from the start. Here, it is necessary to analyze the proofs constructed by the prover for occurrence of particular axioms. This may be difficult if there is no explicit proof object, but, for instance, the popular SMT prover Z3 often used in verifying compilers provides this information.

**The Algorithm**

In the following, we present the delta-oriented slicing algorithm. As the first step of the algorithm, we copy all finished proofs from product $P_1$ into product $P_2$ regardless of their validity for $P_2$. In the resulting set of proofs for the new product, our algorithm identifies the proofs that do not hold in the new context and marks them as invalid. These proofs have to be redone. The algorithm also identifies new proof obligations that have to be discharged in order to obtain a full set of proofs for the specifications of $P_2$.

**Input:** A set of proofs for a product $P_1$, and the delta $\Delta(P_1,P_2)$

**Output:** A set of valid proofs for the product $P_2 = P_1 + \Delta(P_1,P_2)$

1. Copy all proofs from $P_1$ to $P_2$ (regardless of validity). Weed out all proofs where the vocabulary involved (code or specification) is no longer present.

The following steps refer to the content of the delta module $\Delta(P_1,P_2)$. The algorithm currently considers only the structural change information available in the delta and does not take the content of the modified methods or specifications into account.

2. For each $\textit{adds}(C)$:
   (a) do $\textit{adds}(C::f)$ for each $f \in C$
   (b) do $\textit{adds}(C::m)$ for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B) where $C$ or any of its superclasses appear in the rule conclusion

---

For the sake of the algorithm, we assume that $\Delta(P_1,P_2)$ contains exactly one delta module (i.e., we assume delta module composition).
3. For each removes($C$):
   (a) do removes($C::f$) for each $f \in C$
   (b) do removes($C::m$) for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B)
   where $C$ or any of its superclasses appear in the rule conclusion

Adding and removing methods. When adding methods, we have to distinguish if their invocation is treated by inlining and contract application. If an altered implementation is inlined, the proof, of course, will be invalidated. For a contract, this is different since the altered implementation is expected to fulfill the old contract. Contracts are also not affected by method removal. Even though an implementation has been removed, the contract still applies to some overriding implementation in a subclass.

4. For each adds($C::m$):
   (a) invalidate all pre-existing proofs where $m$ was inlined and $C::m$ would have been among potentially referenced implementations (see Fig. 3)
   (b) proofs using the contracts for $m$ remain valid
   (c) prove that $C::m$ satisfies all specifications of $C$ (either stated directly or inherited), as well as all other invariants

5. For each removes($C::m$):
   (a) invalidate all pre-existing proofs where $m$ was inlined and $C::m$ would have been among potentially referenced implementations (Fig. 3)
   (b) proofs using the contracts for $m$ remain valid

Adding and removing fields. In steps 6–7, it might not be immediately clear why adding or removing a field can invalidate a proof. Consider the following code snippet:

```java
class A { Object f; }
class B extends A { /*@ invariant f == ((A)this).f; @*/ }
```

The invariant in class B holds if and only if no field f is added to class B. Otherwise, the left occurrence of f would refer to B::f, while the right one would continue referring to A::f as fields are bound statically in Java.

Adding or removing fields also invalidates proofs containing instance creation, as this process must assign all fields a default value, resulting in varying intermediate states.

6. For each adds($C::f$):
   (a) find the set of method implementations $M$ referring to $C::f$ in $P_2$
   (b) invalidate all pre-existing proofs about any $C':m \in M$
   (c) invalidate all pre-existing proofs inlining any $C':m \in M$
   (d) invalidate all pre-existing proofs of specifications referring to $C::f$ in $P_2$
   (e) invalidate all pre-existing proofs with proof steps assigning default values (during instance creation) to fields of an object with type $A \sqsubseteq C$
7. For each removes($C::f$): same as step 6, but look for $C::f$ in $P_1$

Class reparenting. Reparenting is an invasive operation, which is illustrated in Fig. 4. reparents($C, C'$) moves $C$ from under its old direct supertype $\tilde{C}$ and beneath $C'$, and with it movedPart = \{ $K$ | $K \sqsubseteq C$ \}. As $\tilde{C}$ we then denote the least common supertype of $\tilde{C}$ and $C'$.

Reparenting class $C$ makes $C$ and its subclasses lose features (implementations and specifications) inherited from oldBranch = \{ $K$ | $\tilde{C} \sqsubseteq K \sqsubseteq \tilde{C}$ \} and inherit new features from newBranch = \{ $K$ | $C' \sqsubseteq K \sqsubseteq \tilde{C}$ \}.

8. For each reparents($C, C'$):
   (a) invalidate all pre-existing proofs inlining method bodies for any virtual method call $e.m()$ with $S$ as the static type of $e$ and
      i. $S \in$ newBranch
      ii. $\tilde{C} \sqsubseteq S$
   or, if at least one method body $K::m$ was inlined such that
   iii. $S \in$ movedPart and $K \in$ oldBranch
   iv. $S \in$ oldBranch and $K \in$ movedPart
   This step reacts to changes in the big case distinction simulating dynamic binding.
   (b) invalidate all pre-existing proofs about/inlining any method implementation $C::m$ containing a method call of the form super.$m'()$ (as the superclass will change)
   (c) invalidate all pre-existing proofs about/inlining any method implementation $K::m$, $K \in$ movedPart that references a field $K'::f$ declared in oldPart (as this reference would change its meaning after the move)
   (d) contracts for methods in reparented classes remain valid unless the contract no longer exists (i.e., it was inherited from oldBranch)
   (e) invalidate proofs for specifications inherited from any class in oldBranch
   (f) prove that all classes $K \in$ movedPart satisfy the specifications inherited from new superclasses in newBranch
   (g) invalidate all proofs containing a proof step deciding the predicate $A \sqsubseteq B$ if $A \sqsubseteq C$ and $B \in$ oldBranch

Adding and removing specifications.

9. For each adds($C::m, ct$)
   (a) prove that the contract $ct$ is fulfilled by all $C'::m$ with $C' \sqsubseteq C$

10. For each removes($C::m, ct$)
    (a) invalidate all pre-existing proofs that use the contract $ct$

11. For each adds($C, I$)
    (a) prove that the invariant $I$ is fulfilled by all relevant implementations

12. For each removes($C, I$)
    (a) invalidate all pre-existing proofs that assume the invariant $I$
For some of the algorithm steps, we need to determine whether an implementation \( \text{C}::m \) is potentially referenced by the method invocation expression \( \text{e.m()} \).

We consider the three different method invocation modes available in Java, defining for each mode a starting point class \( \text{S} \) of method lookup. The relation of \( \text{S} \) and \( \text{C} \) determines the answer:

**Instance or “virtual” mode.** This is the most common mode. The target expression \( \text{e} \) (of type \( \text{S} \)) references an object (it may be an implicit \textbf{this} reference), and the method is not declared static or private. This invocation mode requires dynamic binding.

- The implementation is in \( \text{S} \) or one of its subclasses: If \( \text{C} \subseteq \text{S} \), then “yes”
- The implementation is in a superclass of \( \text{S} \), but it is inherited by \( \text{S} \) or one of its subclasses (i.e., it is not overridden between \( \text{C} \) and \( \text{S} \)): If \( \text{S} \subseteq \text{C} \) such that for all \( \text{K} \) with \( \text{S} \subseteq \text{K} \subseteq \text{C} \) holds \( \text{K} \notin \text{m} \), then “yes” (cf. Fig. 5).
- Otherwise, “no”.

**Static mode (\( \text{m} \) is declared static or private).** In this case, no dynamic binding is performed. The implementation to invoke is determined in accordance with the declared static type \( \text{S} \) of \( \text{e} \). If \( \text{C} = \text{S} \) then “yes”, otherwise “no”.

**Super mode (\( \text{e} \) is the keyword \textbf{super}).** This mode is used to access the methods of the immediate superclass \( \text{S} \) (of the class containing the invocation expression \textbf{super}..\( \text{m}() \)).

- If \( \text{S} \subseteq \text{C} \) and for all \( \text{K} \) with \( \text{S} \subseteq \text{K} \subseteq \text{C} \) holds \( \text{K} \notin \text{m} \), then “yes”.
- Otherwise, “no”.

---

**Fig. 3.** Subroutine: When is a method implementation potentially referenced?

---

**Fig. 4.** Illustration of \textit{reparents}(\( \text{C}, \text{C}' \)). Solid lines represent the direct subtype relation, dotted lines its transitive closure, and dashed lines show relations of the previous product.

**Fig. 5.** Virtual method invocation mode and method overriding.
An Example

Example 3. (i) We return to the bank account example introduced in Sect. 2. The core product with the basic Account class now contains specifications (see below). It can easily be proven that both methods satisfy their contracts and the class invariant.

```java
core Base {  
    class Account extends Object {  
        // @ invariant bonus >= 0;  
        int balance;  
        int bonus;  

        // @ ensures bonus == old(bonus);  
        void addBonus(int x){}  

        /* @ ensures balance == old(balance) + x;  
           @ && bonus >= old(bonus); @*/  
        void update(int x) {  
            balance += x;  
        }  
    }  
}
```

(ii) Next, we apply the delta module shown below in order to generate a new product with the additional feature Paycheck. This module adds an Employer class with a reference to the account and a payday() method with a corresponding specification. In order to determine which proofs for the basic bank account are still valid, we use the delta-oriented slicing algorithm. We perform step 2 for the added class, leading to step 4 for the added method, step 6 for the added field and step 9 for the added contract. Only step 4c is non-trivial, since the method payday() did not exist before. The method can be verified easily – either by inlining the implementation of addBonus() and update() or by applying their contracts. There is no existing proof to reuse. Step 6 is trivial (the set $M$ is

```java
delta DPaycheck when Paycheck {  
    adds class Employer extends Object {  
        Account a;  

        /* @ requires x >= 0 && bonus >= 0;  
        @ ensures a.balance == old(a.balance) + x  
        @ && a.bonus >= old(a.bonus); @*/  
        void payday(int x, int bonus) {  
            a.addBonus(bonus);  
            a.update(x);  
        }  
    }  
}
```
empty) as the field \(a\) did not exist previously. Step 9 is subsumed by step 4 as \textbf{Employer} has no subclasses. No proofs are invalidated.

(iii) If we now want to incorporate the \textbf{Investment} feature as well, we apply the deltas \texttt{DInvestment} (Fig. 1b) and \texttt{DInvestmentSpec} (Fig. 2) to the latest product. These two deltas modify the implementation and specification of the method \texttt{addBonus()} and the implementation of the method \texttt{update()} in the class \texttt{Account}. The slicing steps to take to determine which proofs from the previous product are still valid are: step 4 for the added methods, step 5 for the removed methods, step 9 for the added contract and step 10 for the removed contract.

Steps 4c and 9 dictate that both \texttt{update()} and \texttt{addBonus()} have to be re-proven for conformance with the class invariant and their respective (modified) contracts. Proof reuse is feasible here (see Sect. 5). In contrast, \texttt{payday()} has not changed (neither code nor specification), but the proof that it satisfies its contract is now invalid. The proof has been invalidated by step 4a or 10, since it (the proof) depends on either the implementation or the contract of \texttt{addBonus()}. The proof reuse mechanism may be applied here to find a new proof efficiently. The contract of \texttt{update()} has not changed, and all proofs using it remain valid (step 4b).

\section{5 Proof Reuse for Changed Methods}

In this section, we point to the existing technique of proof reuse \cite{11} as a natural complement to delta-oriented proof slicing. This part of our approach is tailored to interactive verification systems like KeY, where the user provides hints to the prover by manipulating an explicit proof object. In practice (although not in our illustrating example), proofs contain proof steps which cannot be (efficiently) found automatically. Users have to instantiate quantifiers, provide lemmas, loop invariants, and guide proof search in other ways. These efforts can be recycled through proof reuse.

The proof reuse technique has been originally developed for KeY by one of the authors to save verification effort during incremental development (i.e., after fixing a bug). Since then, the method has been applied to a number of different change management scenarios. It uses a similarity measure that determines which proof steps from proofs for the original product can be used to establish the proof obligations for the new product. It is a light-weight technique based on proof replay rather than on proof generation. For a full account of proof reuse in KeY we refer the reader to \cite{11}.

In the delta-oriented slicing step, we have identified which proofs have to be redone for the newly generated product. However, some of the changed method bodies may still have considerable similarities to the ones in the already verified product. The correctness proofs of such modified methods are likely to resemble the old proofs. Here proof reuse can help. Reuse can also be used in case of changed specifications but much less effectively. Specifications are less structured than programs, and proof shapes adhere to implementations rather than specifications, which makes finding reusable subproofs much harder.
6 Related Work

Formal methods are used in the context of software product lines for a variety of applications. A large body of work is concerned with the formal analysis of feature models [1] or product models [14]. Further approaches (e.g., [6]) verify that the variability specified by a feature model is correctly implemented in code. Efficient verification of product behavior, however, is not well established. In testing [15,17] or model checking [12,5] there is work to make validation of product lines more efficient, though.

In [2], a case study for the product line development of a compiler is considered. The compiler is developed by stepwise refinement or extension of the compiler functionality. The correctness proof of the compiler is extended and refined in line with the functional extensions by introduction or adaptation of invariants and the addition of case distinctions. This approach relies on a fixed structure of the induction proof for compiler correctness that allows determining in advance which modifications of the proof are required by functional changes.

Reuse of verification artifacts is also related to a whole plethora of work which is impossible to survey here, such as slicing for debugging [25,27] or model checking [9], reuse of refined specifications [26], change management in theory development [16,10], incremental compilation, refactoring, and software change impact analysis.

An interesting and closely related result from change impact analysis is the tool Chianti [19], which determines whether the results of a test are affected by changes to the source code. Changes to the program are decomposed into “atomic operations”, which are similar to our delta operations. These are then analyzed for their impact on the program’s call graph.

Of course, deriving a new product in a product line is also closely related to evolving a single product. Most verification systems implement some kind of proof management for this case. Alas, system developers apparently—and unjustifiably, we think—tend to consider this important component an implementation detail, as published accounts on this subject are rare.

7 Conclusions

Working on verification of SPL, we have identified several interesting lines of future research. Most of them regard the transition from a syntactic modeling of SPL as in the current delta-oriented programming approach [22] to a more semantic-based modeling of SPL.

In order to define delta operations on specifications in a meaningful way, it is necessary to uniquely identify class invariants and method contracts (e.g., for removal or modification). This could be handled by introducing labels (as most tools probably already do internally).

So far the operations we have defined for specification deltas are rather basic. One reason for this is simplicity. Another reason is that at least with the current calculi, the shape of a proof follows rather closely the shape of the program, but it is much less related to the shape of a specification. It remains to be seen
whether adding more fine-grained change information in the specification deltas helps obtaining new proofs more efficiently. Additional operators that appear promising to us are case distinctions and redundant specifications (lemmas).

Until now, the delta module operations (for code) and their applicability conditions are mostly syntactical. Greater power and precision can be achieved by adding more semantical information. For instance, such a description might dictate that a certain feature is only compatible with another if the base product preserves certain data invariants. New tools could be devised to assist in deriving consistent products with desired behavior based on semantical information.

Finally, getting the formal specification of a product right is difficult, but deriving a correct product from another also has its pitfalls. Even if two products $P_1$ and $P_2$ fulfill the specification $I$ (as ensured by our approach), it is still only syntactically the same specification $I$. The product derivation process may seduce one to believe that $I$ is still an adequate specification for the new product, which might not be the case. In the simplest case $I$ might contain pure methods, which have changed between products. The issue is aggravated by the complicated and sometimes unclear semantics of modern specification languages and requires further investigation.

References

C.5 Interleaving Symbolic Execution and Partial Evaluation
Interleaving Symbolic Execution and Partial Evaluation

Richard Bubel, Reiner Hähnle, and Ran Ji
Department of Computer Science and Engineering
Chalmers University, 41296 Gothenburg, Sweden
bubel|reiner|ran.ji@chalmers.se

Abstract. Partial evaluation is a program specialization technique that allows to optimize programs for which partial input is known. We show that partial evaluation can be used with advantage to speed up as well symbolic execution of programs. Interestingly, the input required for partial evaluation comes from symbolic execution itself which makes it natural to interleave partial evaluation and symbolic execution steps in a software verification setup.

1 Introduction

Symbolic execution [1] and partial evaluation [2] both are generalizations of standard interpretation of programs, however, they generalize in different ways: while symbolic execution permits interpretation of a program with symbolic (i.e., unspecified) initial values, the aim of partial evaluation is to transform a program with partially specified input values into a (hopefully, more efficient) program that has only the unspecified arguments as input. For fully specified input arguments the result of both mechanisms is standard program interpretation.

In this paper we show that both technologies not only are compatible with each other, but that there is considerable potential for synergies. Specifically, we integrate a simple partial evaluator for a Java-like language into the logic-based symbolic execution engine of the software verification tool KeY [3]. This allows to interleave symbolic execution and partial evaluation steps within a uniform (logic-based) framework in a sound way. Intermittent partial evaluation during symbolic execution has the effect that the remaining program that is yet to be executed is continuously simplified relative to the current path conditions and the current symbolic state in each symbolic execution trace.

This paper is organized as follows: in the next section we introduce a small object-oriented programming language which is used for the formal definitions (the actual system is implemented for nearly full-fledged sequential JAVA); we also provide background on symbolic execution and partial evaluation. Sect. 3 defines the program logic and deduction system that we use as a framework for

* This work has been partially supported by the EU project FP7-ICT-2007-3 HATS Highly Adaptable and Trustworthy Software using Formal Methods and the EU COST Action IC0701 Formal Verification of Object-Oriented Software.
the integration. In Sect. 4 we introduce a version of a program specialization
operator that is suitable for logic-based verification and we extend the symbolic
execution calculus with sound rules that permit intermittent partial evaluation.
In Sect. 5 we show the context in which the resulting calculus is applied, and
in Sect. 6 we evaluate the integrated system using formal verification tasks for
a number of JAVA programs. This is followed by a discussion of related work
(Sect. 7). We stress that the particular combination of symbolic execution and
partial evaluation explored in the present paper is by far not the only possible
one. We sketch further possibilities in the final section on future work.

2 Background

2.1 A Simple Programming Language

The object-oriented programming language PL described in this section is basi-
cally a simplified JAVA variant and closely related to the language defined in [4].
We briefly sketch the differences to JAVA:

Unsupported Features. Multi-threading, graphics, dynamic class loading, generic
types or floating point datatypes are not supported by PL nor by the actual
implementation in the KeY tool. Formal specification and verification of these
features is a topic of ongoing research, therefore, left out completely.

Restricted Features. For ease of presentation PL imposes some additional re-
strictions compared to JAVA. The KeY tool and the prototype implementation
of our ideas evaluated in Sect. 6 do not impose these restrictions, but model and
respect the JAVA semantics faithfully. The following restrictions apply to PL:

Inheritance and Polymorphism. For the sake of a simple semantics for dynamic
dispatch of method invocations PL abstains from JAVA-like interfaces and method
overloading. Likewise, with exception of the Null type, the type hierarchy induced
by user-defined class types has a tree structure with class Object as root.
Prohibiting method overloading allows to identify a method within a class unam-
biguously by its name and number of parameters. We allow polymorphism (i.e.
methods can be overwritten in subclasses) but require that their signature must
be exactly the same, otherwise it is a compile-time error.

Visibility. All classes, methods and fields are publicly visible. This restriction con-
tributes also to a simpler dynamic dispatch semantics.

No Exceptions. PL has no support for exceptions. Instead of runtime exceptions like
NullPointerException the program will simply not terminate in these cases.

No class/object Initialization. In JAVA the first active usage of a type or creation
of a new instance triggers complex initialization. PL supports only instance cre-
ation, but does not initialize fields upon creation. In particular, PL does not sup-
port static or instance initializers. Constructors are also missing in PL, a new
instance is simply created by the expression new T().

Primitive Types. Only boolean and int are available. To keep the semantics of
standard arithmetic operators simple, int is an unlimited datatype representing
the whole numbers \( \mathbb{Z} \) rather than a finite datatype with overflow.
A PL program \( p \) is a non-empty set of class declarations with at least one class of name \( \text{Object} \). The class hierarchy is a tree with class \( \text{Object} \) as root. A class \( C_l := (\text{cname}, \text{scname}_{opt}, \text{fld}, \text{mtd}) \) consists of (i) a classname \( \text{cname} \) unique in \( p \), (ii) the name of its superclass \( \text{scname} \) (only omitted for \( \text{cname} = \text{Object} \)), and (iii) a list of field \( \text{fld} \) and method \( \text{mtd} \) declarations.

The syntax for class declaration is the same as in Java. The only lacking features are constructors and static/instance initialization blocks. PL knows also the special reference type \( \text{Null} \) which is a singleton with \( \text{null} \) as the only element. It may be used in place of any reference type and is the only type that is a subtype of all class types.

To keep examples short we agree on the following convention: if not explicitly stated otherwise, any given sequence of statements is seen as if it would be the body of a static, void method declared in a class \( \text{Default} \) with no fields declared.

The syntax of the executable fragment needed for the purpose of this paper as follows:

**Statements**

\[
\text{stmnt ::= stmnt \ stmnt} \mid \text{lvarDecl} \mid \text{locExp}'='\text{exp'}';' \mid \text{cond} \mid \text{loop}
\]

\[
\text{loop ::= \ while ('exp') \ '{' \ stmnt'}\}
\]

\[
\text{lvarDecl ::= \ Type IDENT ( '=' \ exp \ opt )';'}
\]

\[
\text{cond ::= \ if ('exp') \ '{' \ stmnt'}\} \ \text{else} \ '{' \ stmnt'}\}
\]

**Expressions**

\[
\text{exp ::= (exp).optmthdCall} \mid \text{opExp} \mid \text{locExp}
\]

\[
\text{mthdCall ::= mthdName('exp\opt(', 'exp')\opt')}
\]

\[
\text{opExp ::= f(exp\opt, exp\opt)} \mid \text{Z} \mid \text{TRUE} \mid \text{FALSE} \mid \text{null}
\]

\[
\text{f ::= ! | - | < | <= | >= | > | == | & | | | * | / | \% | + | -}
\]

**Locations**

\[
\text{locExp ::= IDENT} \mid \text{exp.IDENT}
\]

Dynamic dispatch works in PL as follows: we need to determine the implementation of a method on encountering a method invocation such as \( \text{o.m(a)} \). To do so, first look up the dynamic type \( T \) of the object referenced by \( \text{o} \). Then scan all classes between \( T \) and the static type of \( \text{o} \) for an implementation of a method named \( \text{m} \) and the correct number of parameters. The first match is taken.

### 2.2 Symbolic Execution

Symbolic execution is an idea from the 1960s [1], but it has only recently been realized efficiently for industrially relevant programming languages. Symbolic execution is a central, very versatile program analysis technique that is used for formal program verification [3, 5, 6], extended static checking and verification [7], debugging [8], and automatic test case generation [9, 10].

In the last decade a number of efficient symbolic execution engines for real heap-based programming and intermediate languages were created including KeY (for JAVA, C, Creol, see [3]), KIV (for JAVA, see [11]), Bogor/Kiasan (for BIR, see [12]), Pex (for MSIL, see [9]), and VeriFast (for C, JAVA, see [13]).
In symbolic execution one permits either uninitialized program locations or,
more generally, program locations that are initialized with symbolic expressions.
The following PL program orders the values of \(x\) and \(y\): after its execution \(x\)
contains the maximum of \(x_0\), \(y_0\) and \(y\) their minimum.

```plaintext
int x = x_0; int y = y_0; int z = max(x,y);
if (x < z) {y = x; x = z;}
```

We use location-value pairs to represent states in symbolic execution. The
expression \(\{l_1 := t_1 \mid \cdots \mid l_n := t_n\}\) denotes a symbolic state in which each
program location of the form \(l_i\) has the expression \(t_i\) as its symbolic value.

After symbolic execution of the first three statements of the program above
we obtain the symbolic state \(U = \{x := x_0 \mid y := y_0 \mid z := max(x_0,y_0)\}\).
Symbolic execution of the conditional splits the execution into two branches,
because the value \(x_0 < max(x_0, y_0)\) of the guard expression is symbolic and
cannot be reduced. The (negated) value of the guard becomes a path condi-
tion relative to which symbolic execution continues. Under the path condition
\(P_1 \equiv x_0 < max(x_0, y_0)\) the body of the conditional is executed which results in
the final symbolic state \(U' = \{x := max(x_0, y_0) \mid y := x_0 \mid z := max(x_0,y_0)\}\).
From \(P_1\) and properties of max one can infer \(max(x_0, y_0) = y_0\) which simplifies
\(U'\) to \(\{x := y_0 \mid y := x_0 \mid z := y_0\}\). The other branch terminates immediately in
state \(U\) under path condition \(P_2 \equiv x_0 \geq max(x_0, y_0)\) \((\equiv x_0 = max(x_0,y_0))\).

It is obvious already from this small example that simplification of inter-
mediate states wrt first-order theories is essential for efficiency and to obtain
intuitive results. Modern symbolic execution engines use SMT solvers [9, 13] and
also powerful built-in theorem provers [3, 11] for this purpose.

The example suggests that a single state during symbolic execution of a
program \(p\) consists of the following three components:

1. A program pointer to the next executable statement of the remaining state-
ments in \(p\) that have to be executed.
2. A path condition \(P\) relative to which the remaining statements are executed.
3. A symbolic state \(U\) relative to which the remaining statements are executed.

Symbolic execution of a program is then arranged as a symbolic execution
tree whose nodes are triples consisting of program pointer, path condition, and
symbolic state.

In general it is not possible to symbolically execute a program fully, because
unbounded loops give rise to infinitely many branches with differing symbolic
path conditions. Loop invariants or induction are required to turn symbolic exec-
ution into a complete method for computing strongest post-states of programs.

### 2.3 Partial Evaluation

The ideas behind partial evaluation go back in time even further than those behind
symbolic execution: Kleene’s well-known \(s_{mn}\) theorem from 1943 states that
for each computable function \(f(x, y)\) where \(x = x_1, \ldots, x_m, y = y_1, \ldots, y_n\) there
is an \( m+1 \)-ary primitive recursive function \( s^m_n \) such that \( \phi_{s^m_n}(f,x) = \lambda y.f(x,y) \).

Partial evaluation can be characterized as the research programme to prove Kleene’s theorem under the following conditions:

1. \( \phi_{s^m_n}(f,x) \) is supposed to run more efficiently than \( f \).
2. \( f \) is a program from a non-trivial programming language, not merely a recursive function.
3. The construction of \( \phi_{s^m_n}(f,x) \) is efficient, i.e., its runtime should be comparable to compilation of \( f \)-programs.

In contrast to symbolic execution the result of a partial evaluator is not the value of output variables, but another program. The known input (named \( x \) above) is also called static input while the general part \( y \) is called dynamic input. The partial evaluator or program specializer is often named \texttt{mix}. Fig. 1 gives a schematic overview of partial evaluation.

![Fig. 1. Partial evaluation schema [2].](image)

The first efforts in partial evaluation date from the mid 1960s and were targeted towards Lisp. Due to the rise in popularity of functional and logic programming languages the 1980s saw a large amount of research in partial evaluation of such languages. A seminal text on partial evaluation is the book by Jones et al. [2].

There has been relatively little research on partial evaluation of JAVA. The paper [14] summarizes the state-of-art until 2002 and discusses the JAVA specializer JSPEC which worked by cross-translation to C as an intermediate language. JSPEC seems to be no longer maintained. We found only one other (commercial) JAVA partial evaluator called JPE\(^1\), but its capabilities and underlying theory is not documented.

The application context of partial evaluation is rather different from that of symbolic execution: in practice, partial evaluation is not only employed to boost the efficiency of individual programs, but often used in meta-applications such as parser/compiler generation.

We illustrate the main principles of partial evaluation by a small control circuit PL program depicted in Fig. 2 on the left. The program approximates

\(^1\) \url{http://www.gradsoft.ua/products/jpe_eng.html}
the value of variable \( y \) to a given \( \text{threshold} \) with accuracy \( \text{eps} \) by repeatedly increasing or decreasing it as appropriate.

\[
y = 80; 
\text{threshold} = 100; 
\]

\[
\text{if} \ (y > \text{threshold}) \ {\{ 
\text{decrease} = \text{true}; 
\} \ \text{else} \ {\{ 
\text{decrease} = \text{false}; 
\}} 
\}
\]

\[
\text{while} \ (|y-\text{threshold}| > \text{eps}) \ {\{ 
\text{if} \ (\text{decrease}) \ {\{ 
\ y-1; 
\} \ \text{else} \ {\{ 
\ y+1; 
\}} 
\}} 
\}
\]

\[\text{Fig. 2. A simple control circuit PL program and its control flow graph.}\]

We can imagine to walk a partial evaluator through the control flow graph (for the example on the right of Fig. 2) while maintaining a table of concrete (i.e., constant) values for the program locations. In the example, that table is empty at first. After processing the two initial assignments it contains \( \mathcal{U} = \{y := 80 || \text{threshold} := 100\} \) (using the update notation of Section 2.2).

Whenever a new constant value becomes known, the partial evaluator attempts to propagate it throughout the current control flow graph (CFG). For the example, this \textit{constant propagation} results in the CFG depicted in Fig. 3 on the left. Note that the occurrences of \( y \) that are part of the loop have \textit{not} been replaced. The reason is that \( y \) might be updated in the loop so that these latter occurrences of \( y \) cannot be considered to be static. Likewise, the value of \( \text{decrease} \) after the first conditional is not static either. The check whether the value of a given program location can be considered to be static with respect to a given node in the CFG is called \textit{binding time analysis} (BTA) in partial evaluation.

Partial evaluation of our example proceeds now until the guard of the first conditional. This guard became a \textit{constant expression} which can be evaluated to \textit{false}. As a consequence, one can perform \textit{dead code elimination} on the left branch of the conditional. The result is depicted in Fig. 3 in the middle. Now the value of \( \text{decrease} \) is static and can be propagated into the loop (note that \( \text{decrease} \) is not changed inside the loop). After further dead code elimination, the final result of partial evaluation is the CFG on the right of Fig. 3.
Partial evaluators necessarily approximate the target programming language semantics, because they are supposed to run fast and automatic. In the presence of such programming language features as exceptions, inheritance with complex localization rules (as in Java), and aliasing (e.g., references, array entries) BTA becomes very complex [14].

3 Dynamic Logic with Updates

3.1 Program Logic

As program logic for PL we use a sorted first-order dynamic logic instantiated by a given PL program $p$. We define formally the family of first-order dynamic logics $\mathcal{DPL}$ used to reason about PL programs. Each concrete instance of this family is associated to exactly one PL program which is then referred to as the context program or sometimes the program context of that logic.

**Definition 1 (Signature).** For any PL program $p$ a $\mathcal{DPL}$ signature $\Sigma_p$ is defined as a tuple $(\text{Types}, \text{FSym}, \text{PSym}, \text{VSym})$, where types is a set of sort names that contains at least $\{\top, \text{boolean}, \text{int}, \text{Object}, \text{Null}\} \cup \text{classes}(p)$. Further, $\text{FSym}$ is a set of function symbols, $\text{PSym}$ a set of predicate symbols, and $\text{VSym}$ a set of logic variable symbols (we omit the subscript $p$ in $\Sigma_p$ whenever it can be unambiguously derived from the context). Function, predicate, and logic variable symbols have a fixed sorted signature. Sorts are ordered wrt a sort hierarchy $\preceq$. The order $\preceq$ models $p$’s type hierarchy with maximum element $\top$.

We distinguish between rigid and non-rigid function and predicate symbols. Intuitively, the semantics of rigid symbols does not depend on the current state of
program execution while non-rigid symbols are state-dependent. (Local) program
variables, arrays, static, and instance fields are modeled as non-rigid function
symbols and together form a separate class of non-rigid symbols called location
symbols. Specifically, local program variables and static fields are modeled as
non-rigid constants, instance fields as unary non-rigid functions, and array access
as a binary non-rigid function. For example, an instance field size of type int
declared in a class List is modeled as a unary non-rigid function size@List : List → int. For terms representing field accesses, such as size@List(head),
we use the more readable short form head.size, if no ambiguities arise (and
similar for array accesses). IΣ denotes the set of all executable PL programs
(i.e., sequence of statements) with locations over signature Σ.

The inductive definition of terms and formulas is standard, but we introduce
a new syntactic category called update to represent state updates with symbolic
expressions. An elementary update has the general shape l := t with terms l, t
and l being a location term (i.e., a program variable, field or array access). It
has the same semantics as an assignment. Updates can be composed into parallel
updates l1 := t1 || l2 := t2 or quantified updates for T x; ϕ; l(x) := t(x).

Definition 2 (Terms, Updates and Formulas). Terms t, updates u and
formulas ϕ are well-sorted first-order expressions of the following kind:

\[
t := x \mid f(t_1, \ldots, t_n) \mid \text{if } (\phi) \text{ then } (t) \text{ else } (t) \mid \{u\}t
\]

\[
u := l := t \mid u || u \mid \text{for } T x; \phi; u
\]

\[
ϕ := q(t_1, \ldots, t_n) \mid ¬ϕ \mid ϕ \circ ϕ (\circ ∈ \{∧, ∨, →, ↔\}) \mid
\{u\}t \mid Qx; ϕ (Q ∈ \{∃, ∀\}) \mid \text{if } (ϕ) \text{ then } (ϕ) \text{ else } (ϕ)
\]

\[
[s]ϕ \mid ⟨s⟩ϕ
\]

s := any element of IΣ

The formula [p]ϕ has the intuitive meaning that if the program p terminates
then in its final state the formula ϕ must hold (partial correctness). The formula
⟨p⟩ϕ means that p terminates and in its final state ϕ holds (total correctness).

All formulas, terms and updates are evaluated with respect to a DPL-Kripke
structure whose states correspond to program states.

Definition 3 (DPL-Kripke structure). A DPL-Kripke structure is a tuple
K = (D, I, S, ρ) where:

- D is a non-empty domain together with a domain function δ : D → Types
  mapping each domain element to its (run-time) type. 
  \[ D_T = \{d ∈ D \mid δ(d) ≤ T\} \]
- I is an interpretation mapping each rigid function symbol f : T_1 × ... × T_n →
  S to a total function I(f) : D_{T_1} × ... × D_{T_n} → D_S and each rigid predicate
  symbol p : T_1 × ... × T_n to a relation I(p) ⊆ D_{T_1} × ... × D_{T_n}.
- S is a set of states. Each state s ∈ S is an interpretation of the non-rigid
  function and predicate symbols.
\( \rho : \Pi \times S \times S \) is a state transition relation relating two states \( s, t \) by a program \( p \) iff \( p \) started in state \( s \) terminates in the final state \( t \). Any set of final states \( \rho(p)(s) \) is either a singleton set or empty as PL is deterministic.

As usual in first-order logic, to define evaluation of terms and formulas in addition to a structure we need the notion of a variable assignment. This is a function \( \beta : \text{Vsymb} \to D \) assigning to logical variables a value in \( D \). The evaluation function \( \text{val}_{K,s,\beta} \) is then defined as usual and summarized in Fig. 4. Due to space reasons we do not give a formal semantics of updates and refer to [3] for details on updates. Instead we explain the meaning intuitively along some examples:

- Elementary updates \( i := j \) have exactly the same meaning as assignments: in a DPL-Kripke structure \( K \) and state \( s \), an update application \( \{ i := j \} \xi \) on a term/formula \( \xi \) yields the same value as if evaluating \( \xi \) in \( K,s' \) where \( s' \) is identical to \( s \) except at \( i \) which is evaluated to \( \text{val}_{K,s,\beta}(j) \) in \( s' \).
- Parallel updates \( u_1 || u_2 \) are evaluated simultaneously and do not interfere with each other. Content swapping of two program variables can thus be expressed by \( i := j || j := i \).
- Quantified updates for \( T v; \phi; u \) allow to update arbitrarily many locations simultaneously. The update “for int \( i \); \( i \geq 0 \land i < a.\text{length}; a[i] := 0 \)”, for example, assigns all array components the value 0.
- In case of parallel and quantified updates conflicts may arise when the same location is assigned different values as in \( i := 0 || i := 1 \). Conflict resolution for parallel updates utilizes a last-wins semantics where the previous update is equivalent to \( i := 1 \). Conflict resolution for quantified updates requires a well-founded order on \( T \) and the update with the smallest value for the quantified variable wins [3].

To summarize, updates are similar to explicit substitutions and allow to express state changes concisely at the syntactic level.

**Definition 4 (Satisfiability and Validity).** A DPL-formula \( \phi \) is

- satisfiable iff there exists a DPL-Kripke structure \( K = (D, I, S, \rho) \), a state \( s \in S \) and a variable assignment \( \beta \) such that \( \text{val}_{K,s,\beta}(\phi) = \text{tt} \) (or in short: \( K,s,\beta \models \phi \));
valid in a DPL-Kripke structure $K$ (we also say that $K$ is a model for $\phi$ and write $K \models \phi$) iff for all states $s \in S$ and variable assignments $\beta$ we have $K, s, \beta \models \phi$;

logically valid iff all DPL-Kripke structures $K$ are models for $\phi$.

We introduce two notions which we will need later on. For technical reasons we must have the possibility to extend a logic’s signature.

**Definition 5 (Signature Extension).** Let $\Sigma, \Sigma'$ denote two signatures. $\Sigma'$ is called a signature extension of $\Sigma$ if there is an embedding $\sigma : \Sigma \subseteq \Sigma'$ that is unique up to isomorphism and enjoys the following properties:

- $\sigma(\text{Types}_\Sigma) = \text{Types}_{\Sigma'}$,
- $\sigma(\text{FSym}_\Sigma) \subseteq \text{FSym}_{\Sigma'}$, where for any arity countably infinite additional function symbols exist (analogously for predicates and logic variables)
- $\sigma(\Pi_\Sigma) \subseteq \Pi_{\Sigma'}$

An important property of signature extensions is the following:

**Lemma 1.** Let $\Sigma' \supseteq \Sigma$ denote a signature extension in the sense of Def. 5. If a DPL-formula $\phi$ over $\Sigma$ has a counter example, i.e., a DPL-Kripke structure $K, s \in S_\Sigma$ with $K, s \not\models \phi$ then $\sigma(K, s) \not\models \phi$. In words, signature extensions are counter example preserving.

Finally, we define the notion of an anonymizing update. The motivation behind anonymizing updates is to erase knowledge about the values of the fields included in the set $\text{mod}$ of locations that can be modified by a program. This is achieved by assigning fresh constant or function symbols to those locations. For example, the anonymizing update for the modifier set $\text{mod}_\Sigma = \{i, j\}$ is $i := c_i || j := c_j$ where $c_i, c_j$ are constants freshly introduced in the extended signature $\Sigma'$.

**Definition 6 (Anonymizing Update).** Let $\text{mod}$ denote a set of terms built from location symbols in $\Sigma$. An anonymizing update for $\text{mod}$ is an update $V_{\text{mod}}$ over an extended signature $\Sigma'$ assigning each location $l(t_1, \ldots, t_n) \in \text{mod}$ a term $f'_l(t_1, \ldots, t_n)$ where $f'_l \in \Sigma' \setminus \Sigma$.

### 3.2 Sequent Calculus

The calculus for reasoning in DPL is a sequent calculus. A sequent is an expression of the form $\Gamma \Rightarrow \Delta$ with $\Gamma, \Delta$ being sets of DPL-formulas. We call $\Gamma$ the antecedent and $\Delta$ the succedent of the sequent. A sequent has the same meaning as the formula

$$\bigwedge_{\phi \in \Gamma} \phi \Rightarrow \bigvee_{\psi \in \Delta} \psi .$$

Sequent rules have the general form

$$\begin{array}{c}
\text{name} \\
\begin{array}{c}
\begin{array}{ccc}
s_1 & \cdots & s_n \\
\hline
s
\end{array}
\end{array}
\end{array}$$
where $s, s_1, \ldots, s_n$ are sequents. The sequents above the line are the rule’s premises while sequent $s$ is called the rule’s conclusion. A sequent without any premises is an axiom.

A sequent proof is a tree whose nodes are labelled with sequents and with a sequent whose validity is to be proven at its root. This proof tree is constructed by applying sequent rules $r$ to leaf nodes $n$ whose sequent matches the conclusion $r$. The premises of $r$ are then added as children of $n$. A branch of a proof tree is closed iff it contains an application of an axiom. A proof tree is closed iff all its branches are closed.

As usual, sequent rules are written in schematic form using schema variables (pattern variables with matching restrictions):

\[
\text{andLeft} \quad \frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \land \psi \Rightarrow \Delta}
\]

\[
\text{close} \quad * \quad \frac{\Gamma, \phi \Rightarrow \phi, \Delta}{\Gamma, \phi, \psi \Rightarrow \Delta}
\]

Here, $\phi$, $\psi$ ($\Gamma$, $\Delta$) are schema variables that can be instantiated with any formula (set of formulas). The sequent rule andLeft is applicable at any leaf sequent that contains a disjunctively connected formula in its antecedent.

To handle formulas containing programs within our sequent calculus we aim to model symbolic execution (see Sect. 2.2). Recall that a node in a symbolic execution tree contains a program pointer to the next active statement, path condition, and a symbolic state relative to which symbolic execution is executed. Accordingly, nearly all sequent rules for programs work on a first active statement $s$ and a current update $U$ in the following general form of a conclusion:

\[
\Gamma \Rightarrow \{U\} | \pi s; \omega | \phi, \Delta
\]

In addition, $\pi$ stands for an inactive prefix containing labels, opening braces or method-frames (see below) and $\omega$ for the remaining program. Path conditions are represented by suitable formulas and accumulate in the antecedent $\Gamma$.

Symbolic execution in our DPL-calculus can be roughly organized into two phases. The first is the rewriting phase where the first active statement is replaced with an equivalent series of simpler statements. A typical rule is

\[
\text{evalIfGuard} \quad \frac{\Gamma \Rightarrow \{U\} | \pi \text{ boolean } b = \text{nse}; \text{ if } (b) \{s1\} \text{ else } \{s2\} \omega | \phi, \Delta}{\Gamma \Rightarrow \{U\} | \pi \text{ if } (\text{nse}) \{s1\} \text{ else } \{s2\} \omega | \phi, \Delta}
\]

where $\text{nse}$ is a schema variable matching any non-simple PL-expression (basically, an expression that is neither a literal nor a program variable). As these kind of rules are pure rewrite rules that can be applied in any possible syntactic context (antecedent, succedent, box, diamond) we use the short form $\xi \leadsto \xi'$ to express that a term/program $\xi$ is replaced with an equivalent term/program $\xi'$:

\[
\text{if } (\text{nse}) \{s1\} \text{ else } \{s2\} \leadsto \text{ boolean } b = \text{nse}; \text{ if } (b) \{s1\} \text{ else } \{s2\}
\]

After the first active statement has been reduced to an elementary statement it is translated into a first-order representation of its semantics with the help of
rules belonging to the second phase. For instance, if the first active statement is a conditional whose guard is a simple expression (a program variable or a boolean literal) then the rule

\[
\Gamma, \{U\} \vdash (b \doteq \text{TRUE}) \Rightarrow \{U\}[\pi \{s1\} \omega] \phi, \Delta
\]

\[
\Gamma, \{U\} \vdash (b \doteq \text{FALSE}) \Rightarrow \{U\}[\pi \{s2\} \omega] \phi, \Delta
\]

splits the current proof branch into two branches, one for the case when the guard evaluates to true, and the other covering the else case. Further important representatives of the rules in this phase are assignment rules like

\[
\text{writeAttribute} \quad \Gamma, \{U\} \neg(o \doteq \text{null}) \Rightarrow \{U\}[o.a := se] \pi \omega, \phi, \Delta
\]

where \(o\) is a schema variable matching program variables, \(a\) matches fields and \(se\) matches simple expressions without side-effects that can be directly translated into a logic term. Fig. 5 shows a small excerpt of a sequent proof illustrating symbolic execution. Finally, we discuss how dynamic dispatch of a method is realized in the calculus. The rule for method invocation translates a dynamic dispatch into a cascade of concrete method calls:

\[
\text{methodInvocation} \quad \Gamma, \{U\} \neg(o \doteq \text{null}) \Rightarrow \{U\}[\pi
\]

\[
\begin{align*}
&\text{if (o instanceof } T_n) \text{ res= } o.m(se)@T_n; \\
&\text{else if (o instanceof } T_{n-1}) \text{ res= } o.m(se)@T_{n-1}; \\
&\text{...} \\
&\text{else res= } o.m(se)@T_1; \\
\end{align*}
\]

\[
\omega] \phi, \Delta
\]

\[
\Gamma \Rightarrow \{U\}[\pi \text{ res = } o.m(se); \omega] \phi, \Delta
\]

- \(o\), \(res\) are schema variables for program variables.
- \(res= o.m(se)@T\) are so called method-body statements. A method-body statement is a place holder for an actual method body namely exactly the method body of method \(m\) with the specified number of parameters as implemented in class \(T\).
– \( T_1, \ldots, T_n \) are all the subtypes of the static type of the program variable against which \( o \) is matched and that contain an actual implementation of the method \( m \). As the most specific implementation has to be taken, the list \( T_1, \ldots, T_n \) fulfills the condition that for all \( 0 < i < j \leq n : T_i \not\preceq T_j \).

4 Interleaving Symbolic Execution and Partial Evaluation

4.1 General Idea

Recall from Section 2.2 that a symbolic execution tree unwinds a program’s control flow graph (CFG). As a consequence, identical code is (symbolically) executed in many branches, however, under differing path conditions and symbolic states. Merging back different nodes is usually not possible without approximation or abstraction [15, 16].

![Symbolic execution tree of the control circuit program.](image)

The hope with employing partial evaluation is that it is possible to factor out common parts of computations in different branches by evaluating them partially before symbolic execution takes place. The naïve approach, however, to first evaluate partially and then perform symbolic execution fails miserably. The reason is that for partial evaluation to work well the input space dimension of a program must be significantly reducible by identifying certain input variables to have static values.

Typical usage scenarios for symbolic execution like program verification are not of this kind. For example, in the program of Fig. 2 in Sect. 2.3 it is unrealistic to classify the value of \( y \) as static. If we redo the example without the initial
assignment $y=80$ then partial evaluation can only perform one trivial constant propagation. The fact that input values for variables are not required to be static can even be considered to be one of the main advantages of symbolic execution and is the source of its generality: it is possible to cover all finite execution paths simultaneously and one can start execution at any given source code position without the need for initialization code.

The central observation that makes partial evaluation work in this context is that during symbolic execution static values are accumulated continuously as path conditions added to the current symbolic execution path. This suggests to perform partial evaluation *interleaved* with symbolic execution.

To be specific, we reconsider the example shown in Fig. 2, but we remove the first statement assigning the static value 80 to $y$. As observed above, no noteworthy simplification of the program’s CFG can be achieved by partial evaluation any longer. The structure of the CFG after partial evaluation remains exactly the same and only the occurrences of variable `threshold` are replaced by the constant value 100. If we perform symbolic execution on this program, then the resulting execution tree spanned by two executions of the loop is shown in Fig. 6. The first conditional divides the execution tree in two subtrees. The left subtree deals with the case that the value of $y$ is too high and needs to be decreased. The right subtree with the complementary case.

All subsequent branches result from either the loop condition (omitted in Fig. 6) or the conditional expression inside the loop body testing the value of `decrease`. As `decrease` is not modified within the loop, some of these branches are infeasible. For example the branch below the boxed occurrence of $y=y+1$ (filled in red) is infeasible, because the value of `decrease` is true in that branch. Symbolic execution will not continue on these branches (at least for simple cases like that), but abandon them as infeasible by proving that the path condition is contradictory. Since the value of `decrease` is only tested *inside* the loop, however, the loop must still be first unwound and the proof that the current path condition is contradictory must be repeated. Partial evaluation can replace this potentially expensive proof search by *computation* which is drastically cheaper.

In the example, specializing the remaining program in each of the two subtrees after the first assignment to `decrease` eliminates the inner-loop conditional, see Fig. 7 (the partial evaluation steps are labelled with `mix`). Hence, interleaving symbolic execution and partial evaluation promises to achieve a significant speed-up by removing redundancy from subsequent symbolic execution.
4.2 The Program Specialization Operator

We define a program specialization operator suitable for interleaving with symbolic execution in DPL. A soundness condition ensures that the operator can be safely integrated into the sequent calculus. This approach avoids to formalize the partial evaluator in DPL which would be tedious and inefficient.

**Definition 7 (Program Specialization Operator).** Let $\Sigma$ be a signature and $\Sigma'$ an extension of $\Sigma$ as in Def. 5 containing countably infinite additional program variables and function symbols for any type and arity. Let $\sigma$ be the embedding of $\Sigma$ in $\Sigma'$ ($\sigma(\Sigma) \subseteq \Sigma'$). The program specialization operator

$$\downarrow_{\Sigma'}: \text{ProgramElement} \times \text{Updates}_{\Sigma'} \times \text{For}_{\Sigma'} \rightarrow \text{ProgramElement}$$

takes as arguments a PL-statement (-expression), an update and a DPL-formula and maps these to a PL-statement (-expression), where all arguments and the result are over $\Sigma'$.

The intention behind the above definition is that $p \downarrow_{\Sigma'} (U, \varphi)$ denotes a "simpler" but semantically equivalent version of $p$ under the assumption that both are executed in a state coinciding with $U$ and satisfying $\varphi$. The signature extension allows the specialization operator to introduce new temporary variables or function symbols.

A program specialization operator is sound iff for all DPL-formulas $\psi \in \text{For}_{\Sigma}$, DPL-Kripke structures $K_{\Sigma'}$, and states $s \in S_{\Sigma'}$

$$K_{\Sigma'}, s \models (p) \downarrow_{\Sigma'} (U, \varphi) \psi \Rightarrow K_{\Sigma'}, s \models U(\varphi \rightarrow (p) \psi).$$

In words, the specialized program $p \downarrow_{\Sigma'} (U, \varphi)$ must be able to reach at least the same post-states as the original program $p$ when started in a state coinciding with $U$ in which (path condition) $\varphi$ holds.

Interleaving partial evaluation and symbolic execution is achieved by introduction rules for the specialization operator. The simplest possibility is:

$$\begin{align*}
\text{introPE} & \quad \Gamma \Rightarrow \{U\} (p) \downarrow (U, \text{true}) \psi, \Delta \\
& \quad \Gamma \Rightarrow \{U\} (p) \psi, \Delta
\end{align*}$$

4.3 Specific Specialization Actions

We instantiate the generic program specialization operator of Def. 7 with some possible actions. In each case we derive soundness conditions.

**Specialization Operator Propagation.** The specialization operator needs to be propagated along the program as most of the different specialization operations work locally on single statements or expressions. During propagation of the operator, its knowledge base, the pair $(U, \phi)$, needs to be updated by additional knowledge learned from executed statements or by erasing invalid knowledge
about variables altered by the previous statement. Propagation of the specializa-
tion operator as well as updating the knowledge base is realized by the following
rewrite rule
\[(p, q) \downarrow \langle U, \phi \rangle \leadsto p \downarrow \langle U, \phi \rangle; q \downarrow \langle U', \phi' \rangle\]

This rule is unsound for arbitrarily chosen \(U', \phi'\). Soundness is ensured under a
number of restrictions:

1. Let \(mod\) denote the set of all program locations possibly changed by \(p\). Then
we require that the DPL-formula \(\{U\} \text{respectStrongModifies}(p, mod)\) is
valid where the predicate \text{respectStrongModifies} abbreviates a formula that
is valid if \(p\) changes at most locations included in \(mod\). “Strong” means that
\(mod\) must contain even locations whose values are only changed temporarily.
Such a formula is expressible in DPL, see [17] for details.

2. Let \(\mathcal{V}_{\text{mod}}\) be the anonymizing update for \(mod\) (Def. 6). By fixing \(U' := \mathcal{U}_{\mathcal{V}_{\text{mod}}}\)
we ensure that the program state reached by executing \(p\) is covered by at least
one interpretation and variable assignment over the extended signature.\(^2\)

3. \(\phi'\) must be chosen in such a way that if \(\mathcal{K}_{\Sigma} \models \{U\}p\phi\) then there exists also
an extended DPL-Kripke structure \(\mathcal{K}_{\Sigma'}\) over an extended signature \(\Sigma'\) such
that \(\mathcal{K}_{\Sigma'} \models \{U'\}\phi'\). This ensures that the post condition of \(p\) is correctly
represented by \(\phi'\). One possible heuristic to obtain \(\phi'\) consists of symbolic
execution of \(p\) and applying the resulting update to \(\phi\). This yields a formula
\(\phi''\) from which we obtain a candidate for \(\phi'\) by “anonymizing” all occurrences
of locations in \(p\) that occur in \(mod\).

The first two soundness conditions can be expressed in DPL, the third one
only in absence of quantified updates. In the latter case, the necessary proofs
could be added as additional nodes that spawn side proofs. A more efficient (and
generally necessary) approach is to show once and for all that the oracle used to
determine \(mod\) and \(\phi'\) is correct wrt the conditions.

\textit{Constant propagation and constant expression evaluation.} Constant propagation
is one of the most basic operations in partial evaluation and often a prerequisite
for more complex rewrite operations. Constant propagation entails that if the
value of a variable \(v\) is known to have a constant value \(c\) within a certain program
region (typically, until the variable is potentially reassigned) then usages of \(v\) can
be replaced by \(c\). The rewrite rule
\[(v) \downarrow \langle U, \varphi \rangle \leadsto c\]
models the replacement operation. To ensure soundness the rather obvious condition
\(U(\varphi \rightarrow v = c)\) has to be proved where \(c\) is a rigid constant. The above
rule can be easily modified to include constant expression evaluation.

\(^2\) It is sufficient to let \(U'\) be any update more general than \(\mathcal{U}_{\mathcal{V}_{\text{mod}}}\).
Dead-Code Elimination. Constant propagation and constant expression evaluation result often in specializations where the guard of a conditional (or loop) becomes constant. In this case, unreachable code in the current state and path condition can be easily located and pruned. A typical example for a specialization operation eliminating an infeasible symbolic execution branch is the rule

\[
\begin{align*}
\text{if} \ (b) \ \{p\} \ \text{else} \ \{q\} & \downarrow (U, \phi) \quad \leadsto \quad p \downarrow (U, \phi)
\end{align*}
\]

which eliminates the else branch of a conditional if the guard can be proved true. The soundness condition of the rule is straightforward and self-explaining:

\[
U(\phi \rightarrow b = \text{TRUE}).
\]

Safe Field Access. Partial evaluation can be used to mark expressions as safe that contain field accesses or casts that may otherwise cause non-termination. We use the notation \(\@ (e)\) to mark an expression \(e\) as safe, for example, if we can ensure that \(o \neq \text{null}\), then we can derive the annotation \(\@ (o.a)\) for any field \(a\) in the type of \(o\). The advantage of safe annotations is that symbolic execution can assume that safe expressions terminate normally and needs not to spawn side proofs that ensure it. The rewrite rule for safe field accesses is

\[
\begin{align*}
o.a & \downarrow (U, \phi) \quad \leadsto \quad \@ (o.a) \downarrow (U, \phi).
\end{align*}
\]

Its soundness condition is \(U(\phi \rightarrow \exists C x ; (o = x))\).

Type Inference. For deep type hierarchies dynamic dispatch of method invocations may cause serious performance issues in symbolic execution, because a long cascade of method calls is created by the method invocation rule (Sect. 3.2, p. 12). To reduce the number of implementation candidates we use information from preceding symbolic execution to narrow the static type of the callee as far as possible and to (safely) cast the reference to that type. The method invocation rule can then determine the implementation candidates more precisely:

\[
\begin{align*}
res = o.m(a_1, \ldots, a_n); \downarrow (U, \phi) & \leadsto res = \@ ((C)_a \downarrow (U, \phi)).m(a_1 \downarrow (U, \phi), \ldots, a_n \downarrow (U, \phi));
\end{align*}
\]

The accompanying soundness condition \(U(\phi \rightarrow \exists C x ; (o = x))\) ensures that the type of \(o\) is compatible with \(C\) in any state specified by \(U, \phi\).

5 Application

As an application of interleaving symbolic execution and partial evaluation, consider the verification of a GUI library. It includes standard visual elements such as Window, Icon, Menu and Pointer. An element has different implementations for different platforms or operating systems. Consider the following program snippet involving dynamic method dispatch:

```java
framework.ui.Button button = radiobuttonX11;
button.paint();
```
The element Button is implemented in one way for Max OS X, while it is implemented in a different way for the X Window System. The method paint() is defined in Button which is extended by CheckBox, Component, and Dialog. Altogether, paint() is implemented in 16 different classes including ButtonX11, ButtonMPC, RadioButtonX11, MenuItemX11, etc. The complete type hierarchy is shown in Fig. 8. In the code above button is assigned an object with type RadioButtonX11 which implements paint(). As a consequence, it should always terminate and the DPL-formula \(\langle\text{gui}\rangle\text{true}\) should be provable where gui abbreviates the code above.

![Fig. 8. Type hierarchy for the GUI example.](image)

First, we employ symbolic execution alone to do the proof. During this process, button.paint() is unfolded into 16 different cases by the method invocation rule (Sect. 3.2, p. 12), each corresponding to a possible implementation of button in one of the subclasses of Button. The proof is constructed automatically in KeY with 161 nodes and 10 branches in the proof tree.

In a second experiment, we interleave symbolic execution and partial evaluation to prove the same claim. The partial evaluator propagates with the help of the Type Inference rule in the previous section the information that the runtime type of button is RadioButtonX11 and the only possible implementation of button.paint() is RadioButtonX11.paint(). All other possible implementations are pruned. Only 24 nodes and 2 branches occur in the proof tree when running KeY integrated with a partial evaluator.

### 6 Evaluation

We implemented a simple partial evaluator for Java and interleaved it with symbolic execution in the KeY system as described above. We formally verified a number of Java programs with KeY with and without partial evaluation.

Table 1 shows the experimental results for a number of small Java programs which can be found in the KeY distribution. The column “Program” shows the name of the program we prove, the column “Strategy” shows the strategy we choose to perform the proof where “SE” means symbolic execution and “SE+PE” means interleaving symbolic execution and partial evaluation; the column “#Nodes” shows the total number of nodes in the proof; the column “#Branches” shows the total number of branches in the proof. The results show
that interleaving symbolic execution with partial evaluation significantly speeds up the proof for `complexEval`, `constantPropagation`, `dynamicDispatch`, `safeAccess`, and `safeTypeCast` which can all be considered to be amenable for partial evaluation. Table 2 shows the experimental results of verifying a larger and more realistic Java e-banking application used in [3, Ch. 10]. The column “Proof Obligation” shows which property we prove; the remaining columns are as in Table 1. The results show that symbolic execution interleaved with partial evaluation can speed up verification proofs even for larger applications. As is to be expected, depending on the structure of the program the benefit varies. It is noteworthy that none of the programs and proof obligations used in the present section have been changed in order to make them more amenable to partial evaluation. In no case we have to pay a significant performance penalty which seems to indicate that partial evaluation is a generally useful technology for symbolic execution and should generally be applied.

The case study in Sect. 5 suggests that it could pay off to take partial evaluation into account when designing programs, specifications, and proof obligations.

### 7 Related Work

Partial evaluation as a technique has been applied in a variety of areas including program optimization, compiler generation and meta-compilation. Partial evaluation has been applied successfully in logic programming [18] as well as for imperative and object-oriented languages like C [19] and Java [14]. A good overview including many references is given in [2]. As far as we know, the present paper is the first application of partial evaluation in formal verification.

Our approach is also related to supercompilation [20]. Supercompilation goes beyond partial evaluation by being able not only to specialize but also to gen-

<table>
<thead>
<tr>
<th>Program</th>
<th>Strategy</th>
<th>#Nodes</th>
<th>#Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>complexEval</td>
<td>SE</td>
<td>261</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>158</td>
<td>3</td>
</tr>
<tr>
<td>constantPropagation</td>
<td>SE</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>56</td>
<td>1</td>
</tr>
<tr>
<td>dynamicDispatch</td>
<td>SE</td>
<td>161</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>methodCall</td>
<td>SE</td>
<td>113</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>108</td>
<td>3</td>
</tr>
<tr>
<td>safeAccess</td>
<td>SE</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>safeTypeCast</td>
<td>SE</td>
<td>73</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SE+PE</td>
<td>45</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1. Symbolic execution and partial evaluation for small Java programs.
Table 2. Symbolic execution and partial evaluation for an e-banking application.

There is a close relationship between the rule for specialization operator propagation (SOP) in Sect. 4.3 and what is known as binding time analysis (BTA) in partial evaluation. Partial evaluation techniques roughly categorize program variables into those which are known to have a constant value independent from any input and those whose value may vary. BTA in partial evaluation determines to which of these categories a variable belongs to. The precision of the analysis has a significant impact on the power of partial evaluation as too early binding prevents certain optimizations. The modifier set mod in the SOP rule influences directly the precision of the BTA performed by our specialization operator. If the oracle determining mod is too conservative (imprecise) too much knowledge of the current state $U$ will be lost and cannot be utilized in later specializations.
8 Conclusions and Future Work

In this paper we concentrated on deductive program verification as the main application scenario, however, as pointed out in Sect. 2.2, symbolic execution has other important usages, such as automatic test case generation [10, 9]. It would be interesting to investigate whether partial evaluation can lead to a reduction of redundant test cases.

We showed that a fairly naive partial evaluator can be used to boost performance of a symbolic execution engine. In Sect. 7 we pointed out that symbolic execution in connection with assignable-clauses amounts to a relatively precise binding time analysis (BTA). As BTA becomes rather tricky for complex languages such as JAVA, it would be interesting to use symbolic execution and our simple partial evaluator to bootstrap a sophisticated partial evaluator for JAVA. It could also be interesting to use symbolic execution in addition to partial evaluation to improve precision, for example, in the test case generation approach of [21] discussed in the previous section.

The example in Sect. 5 shows that interleaving partial evaluation and symbolic execution has potential for speed-up especially for programs that are written generically. This is the case for two software development paradigms that gained much popularity in recent times: model-driven development (MDD) and software product line (SWPL) engineering. In both cases, development takes place as much as possible on a generic level: in MDD programs are modelled in abstract notations (the Platform Independent Model) and code generation is used to derive Platform-Specific Models and actual code; in SWPL one separates Domain Engineering which includes feature modeling and library development from Application Engineering where code is derived via instantiation and composition. In either case the executable code has been derived from generic artefacts and, therefore, verification is likely to benefit from the ability to partially evaluate specific information. We are currently experimenting with an SWPL scenario where we plan to use interleaved partial evaluation and symbolic execution.

References


