Project N°: 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 D4.3
Correctness

Due date of deliverable: (T0+48)
Actual submission date: 1st March 2013

Start date of the project: 1st March 2009          Duration: 48 months
Organisation name of lead contractor for this deliverable: BOL

Final version

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

Correctness

This document summarises Deliverable D4.3 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](http://www.hats-project.eu).

This deliverable reports on analysis of correctness of systems modeled and realized with the HATS methodology; in particular, several lines of work which are aimed at assessing the correctness of ABS specifications taking into accounts a plethora of aspects: program invariants, e.g. JML-like assertions, coping with all basic mechanisms of object-oriented programming, such as dynamic binding and object creation, and the ABS methodology, such as delta modeling expressing variants in software families; exploitation of composition patterns and data-flow policies, with applications to security; deadlock detection coping with both classical resource-based deadlocks and process-based deadlocks; program transformation correctness, determinism and non-interference; and analysis of non-terminating behaviours.

The analysis techniques considered for checking such properties are of several kinds: compositional approaches both at the component-level, where the analysis is based on dynamic logic, and at system-level, where patterns/interaction policies are considered; global analyses, e.g., to analyze deadlocks; analysis based on relational logic to reason about program transformations; and weak bisimilarity with big-step semantics.

Such techniques are often based on abstract behavioral representations with formal methods: they require formal models/languages to be introduced, which are simpler with respect to the ABS language, so to formally reason about system properties. Moreover, almost all the presented techniques have also been implemented, such as the compositional deductive verification system KeYABS for ABS, which is based on a dynamic logic.

List of Authors

Wolfgang Ahrendt (CTH)
Gilles Barthe (IMDEA)
Mario Bravetti (BOL)
Richard Bubel (TUD)
Juan Manuel Crespo (IMDEA)
Elena Giachino (BOL)
Stijn de Gouw (CWI)
Reiner Hähnle (TUD)
César Kunz (IMDEA)
Ilham W. Kurnia (UKL)
Andreas Lundblad (KTH)
Keiko Nakata (IoC)
Tarmo Uustalu (IoC)
## Contents

1 Introduction

1.1 Deviations from the DoW .................................................. 5
1.2 List of Papers Comprising Deliverable D4.3 ............................. 6

2 Component-level Verification

2.1 Functional Verification ..................................................... 10
  2.1.1 Background ................................................................. 10
  2.1.2 The Verification Workflow ............................................ 13
2.2 Extension to Abstract Object Creation .................................. 16
  2.2.1 Core Language: Syntax and Semantics ............................... 17
  2.2.2 Applications of Updates ............................................... 19
  2.2.3 Discussion ................................................................. 19
  2.2.4 Conclusion ................................................................. 21
2.3 Verification in Delta-Oriented Programming ............................ 21
  2.3.1 Introduction ................................................................. 21
  2.3.2 Verification Framework ............................................... 22
  2.3.3 Delta-Oriented Reuse of Programs and Contracts ................. 25
  2.3.4 Abstract Method Calls ................................................. 27
  2.3.5 Application Scenarios .................................................. 29
  2.3.6 Related Work ............................................................. 31
  2.3.7 Conclusion ................................................................. 31

3 System-level Verification

3.1 Verification of Composition Patterns .................................. 32
  3.1.1 Kernel Language and Trace Characterization ...................... 32
  3.1.2 Class and Component Characterization .............................. 35
  3.1.3 Specification .............................................................. 37
  3.1.4 Composition Patterns .................................................. 39
  3.1.5 Related Work ............................................................. 42
  3.1.6 Conclusion ................................................................. 42
3.2 Verification of Data Flow Policies ...................................... 42
  3.2.1 Theoretical Formalization ............................................. 42
  3.2.2 Practical Evaluation and Case Studies .............................. 44

4 Deadlock Analysis

4.1 Petri Net Based Approach ................................................. 50
  4.1.1 Rescheduling .............................................................. 51
  4.1.2 Places and Tokens ...................................................... 52
  4.1.3 Transitions ............................................................... 52
4.2 Contract Based Approach .................................................. 55
  4.2.1 Contracts and the Contract Inference System ...................... 57
Chapter 1

Introduction

To assess correctness of systems modeled and realized with the HATS methodology, encompassing also the features of variability in ABS models, several novel tool supported techniques have been pursued, reflecting the variety of expertise in the consortium: operational techniques, co-inductive techniques, type systems, static analysis, (dynamic) logics, and logical relations. The inherent complexity of systems modeled with ABS and the variety of properties of interest in such systems, encompassing, e.g., also deadlock analysis and security, has led to several lines of work taking into account different aspects: program invariants, e.g., JML-like assertions, coping with all basic mechanisms of object-oriented programming, such as dynamic binding and object creation, and the ABS methodology, such as delta modeling expressing variants in software families; exploitation of composition patterns and data-flow policies, with applications to security; deadlock detection coping with both classical resource-based deadlocks and process-based deadlocks; program transformation correctness, determinism and non-interference; and analysis of non-terminating behaviours.

The techniques adopted are often based on abstract behavioral representations with formal methods: they require formal models/languages to be introduced, which are simpler with respect to the ABS language, so to formally reason about system properties. Moreover, almost all the presented techniques have also been implemented, such as the compositional deductive verification system KeYABS for ABS.

In the following we outline the techniques in the deliverable. We first distinguish between verification techniques based on specification languages describing the behaviour of classes/components, hereafter called component-level techniques, and verification techniques related to the behavior of the entire (parts of the) system, in terms of compositional patterns/policies, hereafter called system-level verification techniques.

In Chapter 2 we present component-level techniques. Section 2.1 describes the work of CTH and TUD in the development of the core machinery. This work is based on a compositional proof system using histories, proof obligations expressed in dynamic logic, JML-like assertions as specification language and the KeYABS tool. The section focuses on the creation of proof-obligations and how they are obtained from attribute grammar based specifications. In Section 2.2 CTH and TUD in collaboration with CWI extend this machinery by encompassing object creation (at the abstraction level of the programming language, i.e., independently from a specific implementation), dynamic binding and arrays. Section 2.3 presents the work of TUD on extending the above proof machinery to the verification of programs in the presence of delta modelling. A new technique called abstract method calls is introduced which, together with first-order reasoning, structured contracts and proof caching, allows us to reuse proofs. This enables the verification of software systems with a high degree of variability and steady adaptation.

In Chapter 3 we present system-level techniques. Section 3.1 describes the work of UKL on a theory for the compositional verification of functional properties of open active object systems. The framework is based on behavioral specifications of interaction of system components. The central goal is the development of proof rules for common composition patterns. Section 3.2 presents the work of KTH on specification and enforcement, at the code level, of data flow policies using tree automata (which has strong security applications) and on an implementation inliner based on the Taintdroid tool for taint propagation in the Dalvik virtual machine. This supplements earlier work in Task 3.4 which addressed word automata for the
same purpose, at both ABS and code levels.

In Chapter 4 we report about two techniques developed by BOL for deadlock detection in ABS programs. We describe a technique based on type systems, for deadlock and livelock analysis. This technique relies on the definition of abstract descriptions of behavior in form of behavioral types called “contracts”, derived by a type inference algorithm. A potential lock is detected when a circular dependency is found in some state of the automata. This work has been continued from Task 2.5. Moreover, we present a more recent technique, which focuses on process-based deadlocks in addition to resource based classical deadlocks, which is based on translation of ABS programs into Petri nets and on reachability analysis of the derived Petri nets.

In Chapter 5 we present work by IMDEA on verification techniques based on relational logic for program properties like program transformation correctness, non-interference and determinism.

Finally, in Chapter 6 we present the collaboration of IoC with CTH extending the analysis methods to properly account for nonterminating behaviours, by the usage of coinductive big-step semantics. This work combines, for the first time, compositional big-step reasoning, reasoning about nonterminating behaviors (including reasoning about liveness/responsiveness properties), and concurrency.

1.1 Deviations from the DoW

The Description of Work of Task 4.3 in Annex I does not include any activity for UKL and FRH, hence the following should be added: “UKL will study compositional techniques to model the functional behavior of systems based on the specifications of their components using characteristics of ABS models.”.

Moreover, the description of UPM (IMDEA) activity in Annex I should be modified by removing the sentence “including termination analysis”. This has instead been studied in Deliverable D4.2 [36].

IoC employed coinduction in the context of big-step operational semantics and Hoare-style program logics to reason about nontermination in combination with interaction and concurrency and found elegant solutions a number of technical problems one has to address in combinations of inherently compositional big-step reasoning with nontermination and communication/concurrency. Notably IoC overcame the technical difficulties that had remained unresolved in the highly acclaimed Compcert effort on a certified compiler, where coinductive big-step semantics was considered for a basic non-interactive sequential setting only. Owing to lessons learned in the course of this progress, instead of developing type systems fragments of coinductive program logics and transformation of proofs in such logics along compilation, IoC concentrated on automatic transformation of proofs between the standard inductive big-step semantics based program logics and the coinductive program logic. Effectively the standard program logics are fragments of the coinductive logic, with a less expressive assertion language. The work of IoC on correctness of compilation was reported in Deliverable 1.4 [39].

1.2 List of Papers Comprising Deliverable D4.3

This section lists all the papers that comprise this deliverable, indicating where they were published, and it explains how each paper is related to the main text of a specific chapter in this deliverable.

The papers are not directly attached to Deliverable D4.3, but they are made available on the HATS website at the following url: [http://www.hats-project.eu/sites/default/files/D4.3/](http://www.hats-project.eu/sites/default/files/D4.3/)

Paper 1: Abstract Object Creation in First-Order Dynamic Logic: State of the Art in KeY

This paper extends the work on abstract object creation by Ahrendt et al. [2] to classes, arrays and failures, and adds various soundness and completeness results. Its content is the main subject of Section 2.2. This paper was written by Stijn de Gouw, Frank S. de Boer, Wolfgang Ahrendt, and Richard Bubel and it is currently under review.

(Download Paper 1)
Paper 2: Verification of Object-Oriented Programs: A Transformational Approach

This paper presents a Hoare logic for object-oriented programs, justified formally by soundness and completeness theorems, by means of a transformation to programs with recursive procedures. Of interest to us is especially the transformation for programs with dynamic binding, as stated in Section 2.2.

This paper was written by Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog and Stijn de Gouw and it was accepted for publication in the Journal of Computer and System Sciences.

(Download Paper 2.)

Paper 3: Weak Arithmetic Completeness of Object-Oriented Inductive Assertion Networks

This paper contains an investigation into the expressive power of assertion languages in the presence of auxiliary array variables to specify properties of the heap. Part of the final completeness result is based on the above work on abstract object creation. The content of the paper is the main subject of Section 2.2.3.

This paper was written by Stijn de Gouw, Frank S. de Boer, Wolfgang Ahrendt, and Richard Bubel and it was accepted for publication in the proceedings of the 39th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2013.

(Download Paper 3.)

Paper 4: Reuse in Software Verification by Abstract Method Calls

This paper introduces abstract method calls, a new verification rule for method calls that can be used in most contract-based verification settings. By combining abstract method calls, structured reuse in specification contracts, and caching of verification conditions, it is possible to detect reusability of contracts automatically via first-order reasoning. The content of the paper is the main subject of Section 2.3.

This paper was written by Reiner Hähnle, Ina Schaefer and Richard Bubel and has been accepted at CADE-24.

(Download Paper 4.)

Paper 5: A Relational Trace Logic for Simple Hierarchical Actor-Based Component Systems

This paper describes a sound trace-based logic for a chain composition pattern in ABS programs without object groups and futures. Together with the Paper 6, this paper is the main subject of Section 3.1.

This paper was written by Ilham W. Kurnia and Arnd Poetzsch-Heffter and is published in the Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE! ’12.

(Download Paper 5.)

Paper 6: Verification of Concurrent Object Component System

This paper extends the previous paper by describing the connection between operational and trace semantics of open concurrent object systems. In particular, this paper discusses how run-time components, an abstraction level between object groups and systems, are formed. Together with Paper 5, this paper is the main subject of Section 3.1.

This paper was written by Ilham W. Kurnia and Arnd Poetzsch-Heffter and is under review.

(Download Paper 6.)
Paper 7: TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies

This paper [27] proposes a novel approach to security policy monitoring that uses tree automata to capture constraints on the way data are processed along an execution. The paper investigates some of the model’s meta-properties, and shows how the ideas can be implemented using labels corresponding to automaton states. Finally an implementation targeting the Dalvik VM is presented and evaluated in five real world case studies. The results and statistics of this evaluation is the main focus of Section 3.2.

This paper was written by Mads Dam, Gurvan Le Guernic and Andreas Lundblad and published in the proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012.

(Paper 7)

Paper 8: A Petri Net Based Analysis of Deadlocks for Active Objects and Futures

This paper [31] describes the technique based on translation into Petri nets for detecting resource and process based deadlocks in ABS programs without object groups and where futures cannot be passed as arguments to methods. This is the main subject of Chapter 4, apart from Section 4.2.

This paper was written by F.S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen and G. Zavattaro, and it was published in the Proceedings of the 9th International Workshop on Formal Aspects of Component Software, FACS 2012.

(Paper 8)

Paper 9: Deadlock Analysis of Concurrent Objects: Theory and Practice

This paper [55] presents a framework for statically detecting deadlocks in ABS, which is made of an inference algorithm to extract abstract descriptions of method’s behaviors (contracts), and an evaluator, which computes a fixpoint semantics, to return finite state models of contracts. A potential deadlock is detected when a circular dependency is found in some state of the model. The prototype implementation of the framework and its validation against an industrial case study are also discussed. This paper is the subject of Section 4.2.

This paper was written by E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt and P. Y. H. Wong, and was submitted for publication in Proceedings of 10th International Conference on Integrated Formal Methods.

(Paper 9)


This paper [58] is an introduction to the framework for the deadlock analysis, defined in [59]. The algorithm for deciding deadlock-freeness is discussed by means of a number of paradigmatic examples. This paper is the subject of Section 4.2.

This paper was written by E. Giachino and C. Laneve, and was published in Proceedings of 7th International Symposium on Trustworthy Global Computing.

(Paper 10)

Paper 11: Relational Verification Using Product Programs

This paper introduces the general notion of product program that supports a direct reduction of relational verification to standard verification; this is the main subject of Section 5.1.

This paper was written by Gilles Barthe, Juan Manuel Crespo, and César Kunz and was published in the proceedings of the 17th International Symposium on Formal Methods.

(Paper 11)
Paper 12: From Relational Verification to SIMD Loop Synthesis

This paper explores a new approach to auto-vectorization by exploiting the advances on the techniques of program product construction; this is them main subject of Section 5.3.

This article is written by Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron and was accepted for publication in the proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

(Download Paper 12)

Paper 13: Coinductive Big-Step Semantics of Concurrency

This paper [95] presents a framework of big-step semantics for share-variable concurrency that also handles non-termination based on coinductive resumptions and coinductive evaluation. The approach is applicable to both preemptive and cooperative scheduling. It is discussed in Chapter 6.

This paper was written by T. Uustalu and is to appear in the proceedings of 6th Workshop on Programming Language Approaches for Concurrency and Communication-Centric Software, PLACES 2013.

(Download Paper 13)

Paper 14: A Hoare Logic for the Coinductive Trace-Based Big-Step Semantics of While

This paper [85] develops a sound and complete Hoare-style logic for the trace-based coinductive big-step semantics for a simple sequential language [82, 84]. The assertion language bears some reminiscence to that of interval temporal logic, but adopted for a setting allowing right-infinite intervals. This logic facilitates compositional reasoning about both safety and liveness-type properties of possibly non-terminating programs. It is demonstrated that it subsumes standard partial-correctness and total-correctness logics, witnessed by automatic transformations of proofs between these logics.

This paper was written by K. Nakata and T. Uustalu is under consideration for publication in Logical Methods in Computer Science, currently under final revision.

(Download Paper 14)
Chapter 2

Component-level Verification

2.1 Functional Verification

For the ABS language, a compositional logic and calculus for state-based properties based on the rely-guarantee principle, has been developed in ABS. The logic and calculus are described in detail in [38, 3, 4]. One of the main characteristics of the calculus is that it does not require to talk explicitly about threads on the syntactic level. Instead it manages to stay in a sequential setting without the need to represent the active threads of different concurrent object groups (COG). Nevertheless, the proven functional properties carry over to the general concurrent and distributed case thanks to the calculus’ compositional design.

In the following sections we explain briefly the main idea behind the generation of proof obligations and how the translation from expressions using a higher-level specification language into dynamic logic is realized in principle.

2.1.1 Background

proof obligations

We introduce here only the necessary parts of ABS DL needed to follow this chapter. For a detailed overview we refer to [1, 3, 38].

ABS Dynamic Logic

ABS DL is a sorted first-order dynamic logic with two modalities $[\cdot]$ (box) and $⟨·⟩$ (diamond). Let $p$ be a sequence of executable ABS statements and $φ$ any ABS DL formula, then:

- $[p]φ$ is a formula in ABS DL and expresses that if $p$ terminates then in its final state the property $φ$ holds.
- $⟨p⟩φ$ is a formula in ABS DL and expresses that $p$ terminates and in the reached final state the property $φ$ holds.

ABS DL has one additional modality called update. Updates originate from JavaCard Dynamic Logic [12, 90]. An update keeps track of state changes and it can intuitively be seen as an explicit substitution. We distinguish elementary updates

$$lhs := rhs$$

from parallel updates

$$lhs_1 := rhs_1 || \ldots || lhs_n := rhs_n$$

where $lhs, lhs_i$ are program variables and $rhs, rhs_i$ are terms representing the values assigned to these variables. Updates $u$ can be applied to terms $\{u\} t$ or to formulas $\{u\} φ$. Parallel updates are applied simultaneously and in case of conflicts ($lhs_i = lhs_j, i \neq j$) a last-win semantics is used as conflict resolution. We explain updates and their meaning by an example:
1. Evaluating the formula \( \{ i := 3 + j \} \) \( i \geq j \) in a state \( s \) (mapping program variables to values) is equivalent to evaluating \( i \geq j \) in a state \( s' \) which differs from \( s \) only in the assignment of \( i \) which is evaluated to \( s'(i) = s(j) + 3 \). Similarly, the evaluation of the above formula in a state \( s \) returns the same result as evaluating the formula \( 3 + j \geq j \) in \( s \). This shows that updates can be also seen as weakest precondition transformers.

2. The evaluation of the formula \( \{ i := j \mid | j := i \} \phi \) evaluates the formula \( \phi \) in a state where the values of \( i \) and \( j \) have been swapped.

We introduce now the modeling of two central concepts, namely, the heap and the event history. In ABS object fields are private and shared memory communication is only possible within COGs.

ABS DL is a compositional logic with the nice property that it is sufficient to consider only single objects in isolation when proving, e.g., that invariants are preserved or a method satisfies its contract. Compositionality allows then that the proven property holds for the overall system.

For instance, when proving that a certain method implementation \( m \) satisfies its contract or preserves its class or interface invariant, we consider only the method invocation of \( m \) on a single object of interest. The ABS DL formulas refer to this object using the distinguished program variable \( \text{this} \).

We model the heap as an instance of the theory of arrays using \select \ and \store \ expressions to access and assign values to objects fields. For each attribute \( \text{attr} \) of class \( C \) a field symbol (constant) \( \text{attr}@C \) of (almost) the same name is introduced. Wellformedness of the heap ensures that all of its objects belong to the same COG. On the syntactic level, the heap belonging to the COG of \( \text{this} \) is accessed using the global program variable \( \text{heap} \).

We can express that executing a method \( m() \{ \text{body} \} \) preserves a class invariant stating that a field \( \text{balance} \) must always be non-negative, as follows

\[
\text{select(\text{heap, this, balance})} \geq 0 \Rightarrow \text{[body]}\text{select(\text{heap, this, balance})} \geq 0
\]

The most important concept for expressing and reasoning about (asynchronous) events is that of a history. In the following we give an excerpt of the theory of histories based on the 4-event history as introduced in [44, 38].

The global system history is a sequence of system events such as asynchronous method invocation, method completion or the creation of a new object and COG. To stay compositional, the logic itself does not refer to the system history, but to the so called \textit{local communication history} which projects (order preserving) the global history to a subhistory containing exactly those events in which the objects known to the \text{this} object participated. The local history is formalized as a sequence of events. An incomplete list of history events is given below:

- **Invocation events:** An invocation event \( \langle o, \to, o', f, m, \bar{e} \rangle \) is appended to the current history at the moment on which the object \( o \) invokes method \( m \) on \( o' \) with parameters \( \bar{e} \), and with future \( f \) holding the result when \( m \) completes.

- **Invocation reaction events:** An invocation reaction event \( \langle o, \to, o', f, m, \bar{e} \rangle \) is recorded when the execution of the method \( m \) as response to an invocation event is actually scheduled.

- **Completion events:** A completion event \( \langle o, \leftarrow, o', f, m, \bar{e} \rangle \) corresponds to the termination of the method execution of \( m \).

- **Completion reaction events:** A completion reaction event is recorded after the completion event when the future is successfully resolved.

On the syntactic level there is a global program variable \( \mathcal{H} \) which refers to the local communication history. Further, there are datatype constructors for each event type. For instance, the function \( \text{invocEv}(...) \) corresponds to the above listed method invocation event.
In addition to the history formalization as a sequence of events, there is a number of auxiliary and convenience predicates that allow to express common properties concerning histories. For example, predicates like \( \text{wfHist} \) (History), \( \text{beginsWith} \) (History, Event), \( \text{endsWith} \) (History, Event), \( \text{nriE} \) (History, MethodLabel), etc., are used, respectively, to specify wellformedness of histories, the first or last event contained in a history or the number of method invocations for a given method and history.

We have now reached a stage where we can express system invariants like that the number of push invocations on a stack object must always be greater or equal than the number of pop invocations. We can also relate class invariants to the history. For instance, we may state that the value of a field balance is always equal to the last invocation of method setBalance.

**Sequent Calculus**

Before describing the general verification workflow with KeYABS, we give a short introduction into the Gentzen-style sequent calculus used to reason about ABS programs. A sequent is a data structure of the form:

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

which has the same meaning as the formula

\[
\bigwedge_{i \in \{1\ldots m\}} \phi_i \rightarrow \bigvee_{j \in \{1\ldots n\}} \psi_j
\]

A sequent rule

\[
\begin{array}{c}
\text{name} \\
\text{premise} \\
\begin{array}{c}
\vdots \\
S_1 \\
\ldots \\
S_n
\end{array} \\
\text{conclusion}
\end{array}
\]

\((s, s_i, i \in \{1\ldots n\})\) are sequents) has a name, a premise consisting of a possibly empty sequence of sequents and a conclusion. A sequent rule is called correct if the validity of the premise implies the validity of the rule’s conclusion. An axiom is a sequent rule without premise.

A sequent proof is a tree where each node is labelled with a sequent and there exists a sequent rule for each inner node such that the conclusion of \( r \) matches the node’s sequent and the rule’s premises match the sequents of the node’s children. A branch (of the proof tree) is called closed if the last rule application was an axiom. A proof is called closed if and only if all its branches are closed.

The sequent calculus as realized in ABS DL essentially simulates a symbolic interpreter for ABS. The assignment rule for a local program variable is:

\[
\text{assign} \quad \frac{\Gamma \Rightarrow \{v := e\} \ [\text{rest}] \phi, \Delta}{\Gamma \Rightarrow \{v = e; \text{rest}\} \phi, \Delta}
\]

where \( v \) is a local program variable and \( e \) a pure (side effect free) expression. The rule rewrites the formula by moving the assignment from the program into an update. During the symbolic execution updates are accumulated in front of the modality. Once the program has been completely executed the updates are applied to the formula resulting in a pure first-order logic formula (assuming there are no nested modalities).

An example for a rule that causes the proof tree to split is

\[
\text{ifSplit} \quad \frac{\Gamma, e \doteq \text{True} \Rightarrow [p;\text{rest}] \phi, \Delta \quad \Gamma, e \doteq \text{False} \Rightarrow [q; \text{rest}] \phi, \Delta}{\Gamma \Rightarrow \text{if} \ (e) \ \{ \ p \ } \ \text{else} \ \{ \ q \ } \ \text{rest} \phi, \Delta}
\]

where for each branch of the conditional statement a corresponding proof branch is created. Each of the two branches has to be considered and closed in order to prove that the property \( \phi \) holds when the ABS program terminates.
We conclude this section with the rules for asynchronous method invocation and the await statement:

\[
\begin{align*}
\text{asyncMC} & \\
\Gamma \Rightarrow o \neq \text{null} \land \text{wfHist}(\mathcal{H}), \Delta \\
\Gamma \Rightarrow \{\mathcal{U}\}(\text{futureUnused}(frc, \mathcal{H}) \rightarrow \{fr := frc \mid \mathcal{H} := \text{append}(\mathcal{H}, \text{invocEv}(\text{this, o, frc, m, e})))\}[\text{rest}][\phi] \}
\end{align*}
\]

In case of an asynchronous method invocation the proof splits into two branches: The first branch ensures that the callee is not null and that the history is wellformed. The second branch introduces a new constant \(frc\) which represents the future (placeholder for the method’s return value). The left side of the implication ensures that the future is new and it has not yet been used (\textit{futureUnused}) and updates the history by appending the invocation event for the asynchronous method call. Afterwards the normal program execution is continued with the remaining program (\textit{rest}).

Finally, the sequent rule for the \textit{await} statement is as follows:

\[
\begin{align*}
\text{awaitComp} & \\
\Gamma \Rightarrow \text{ClInv}(C)(\text{heap, this}), \Delta \\
\Gamma \Rightarrow \{\text{heap} := \text{newHeap} \mid \mathcal{H} := \text{append}(\mathcal{H}, \text{append}(\text{newHist}, \text{compREv}(\ldots))))\} \\
(\text{ClInv}(C)(\text{heap, this}) \land \text{wfHist}(\mathcal{H}) \rightarrow [\text{rest}][\phi]), \Delta \\
\end{align*}
\]

The await statement releases control allowing other threads to take over. When the await condition is satisfied (here: the future becomes resolved), the waiting thread can be rescheduled. As control of the COG is released, we must ensure that a state has been reached in which the class invariant is satisfied as the continuing thread will rely on it. The fulfillment of the class invariant is checked by the first branch.

The second branch assumes that the await condition is satisfied and continues the execution in a state where the completion reaction event has been appended to the extended history. Extended means that the history \(\mathcal{H}\) as before execution of the await statement has been prolonged by an arbitrarily long sequence of events (using the fresh Skolem constant \textit{newHist}) representing those events that occurred between control release and resume. In our rely guarantee based setting, we can safely assume that upon resume of control, the class invariant has been established by the previous thread and the class invariant is valid again. But the heap might have been changed and all previous accumulated knowledge must be removed. This is achieved by assigning to the heap an unknown value (represented by the freshly introduced Skolem constant \textit{newHeap}).

### 2.1.2 The Verification Workflow

Figure 2.1 exemplifies the verification workflow using KeYABS. The ABS model and its specification is loaded and handed over to the proof-obligation generator. The proof-obligation generator constructs from a given ABS model and ABS specification, formulas that express that a method \(m\) preserves the class invariant or the interface invariants or that the method fulfills its contract. These formulas are called \textit{proof obligations}.

#### Generation of Proof Obligations

In [44] a number of proof obligations have been proposed to ensure correctness of the overall system. Among them are schemas for formulas in dynamic logic that allow verifying that

1. after object construction and initialization the class invariant of the object is established,
2. a method preserves the interface/class invariants,
3. a method satisfies its contract, i.e., if upon invocation time its precondition is satisfied then at completion time (when executing the return statement) its postcondition is satisfied.

For ABS DL we could reuse their approach with only minor modifications concerning technical details.
From Specification Language to Dynamic Logic

Expressing proof obligations and method specifications directly in dynamic logic is cumbersome. More convenient is to specify the properties in a higher level specification language like one of those described in [35].

In this section we sketch how to translate a specification expression written in the attribute grammar based specification language presented in [35].

A central concept of that specification language are the so called communication views (or short views). A communication view allows to specify the history events that are of interest for the current specification. Communication views can be specified in the language defined by the following grammar which is taken from [35]:

\[
V ::= \text{view } v \{ P \} \quad \text{communication view}
E ::= \text{call | resolve} \quad \text{asynchronous calls and future resolutions}
S ::= \text{I\_in.m | I\_out.m} \quad \text{method names in interfaces}
M ::= E \_ S \mid \text{new } C \quad \text{messages: calls/resolutions and creations}
P ::= (M \_ t)^* \quad \text{(message M terminal t) pairs}
\]

On the logic level a communication view \( v \) defines a function of the same name \( v \) which is interpreted as an order-preserving projection on the system history \( \mathcal{H} \). I.e., \( v(\mathcal{H}) \) is evaluated to an order-preserving subsequence of \( \mathcal{H} \) which contains exactly those event types from \( \mathcal{H} \) that are contained in the view specification. For each communication view function its definition is added in terms of calculus rules to the verification system. In addition, lemmas that express order preservance and similar properties of these projection functions are generated and added as well.

The communication view allows us to separate different specification concerns. To actually express properties on allowed call sequences and other functional properties, attribute grammars are used: Restrictions on the call sequences of a view (or the history) are expressed by the words belonging to the language generated by the grammar, while the grammar attributes are used to express data related properties.

We explain this along an example and give here the attribute grammar used to specify valid call sequences and contracts of the methods \texttt{push} and \texttt{pop} as presented in [35]:

![Figure 2.1: Verification workflow with ABS DL](image-url)
The context-free grammar restricts the valid call sequences to those sequences in which the number of push invocation reaction events is equal to or exceeds the number of pop method invocation reaction events.

In our example below, the view `stackhist` projects the local communication history onto an order preserving subsequence consisting only of push method invocation events and pop method completion events. The view `stackhist` is specified as

```plaintext
view stackhist { call_Stack.push push, resolve_Stack.pop pop }
```

With the help of such a communication view, the methods declared by the Stack interface can be specified as follows (z is a free variable used only for specification purposes):

```plaintext
interface Stack {
  @ requires z == stackhist.stack()
  @ ensures equals(stackhist.stack(), append(z, item))
  Unit push(Int item);

  @ requires z == stackhist.stack() and not(isEmpty(z))
  @ ensures equals(stackhist.stack(), tail(z)) and result == head(z)
  Int pop()
}
```

The specification of push captures in the precondition the view on the history at invocation reaction time, i.e., at the time the method is actually being scheduled and executed. Its postcondition ensures that at method completion time the updated view on the history is equal to the view captured in the precondition with the pushed element appended. The contract of pop is similar and it specifies that if executed on a non-empty stack then, after method execution, the top element has been removed from the stack and returned.

Handing over the above ABS model and specification of pop to the proof-obligation generator results in the creation of an ABS DL formula similar to:

```plaintext
wfHist(\mathcal{H}, heap) \land wfHeap(heap) \land this \neq null \land \forall z.(z \equiv stackhist(\mathcal{H}, push, pop) \land \neg isEmpty(z) \rightarrow [result=this.pop()]stackhist(\mathcal{H}, push, pop) \equiv tail(z) \land result \equiv head(z))
```

Two remarks on the generated formula:

1. The prestate value of the stack history view is “stored” in the logical variable z using universal quantification. Logical variables are rigid, i.e., their value cannot be changed by a program. This ensures that when using the variable behind the box modality, it still evaluates to the same value as before.

2. The function `stackhist` realizes a projection function, which maps the local communication history to an order preserving subsequence. These projection functions are generated from the views defined by the attribute grammars as part of the proof-obligation generation process.

The formula needs then to be proven valid using the described sequent calculus.
2.2 Extension to Abstract Object Creation

In object-oriented programming languages like ABS, objects as first-class citizens in the domain of values, introduce a general mechanism of indirection. This high-level mechanism of indirection 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 user-defined operations (i.e., methods) which can be performed on objects. The only other operations that can be applied on objects are built-in (such as new, equality and conditional expressions), provided by the ABS. Moreover, these operations can only be performed on the created objects. The objects not (yet) created do not exist and therefore can not be referred to by any programming construct. This ensures memory safety, relieves the programmer of (error-prone) manual memory management, helps portability and allows compiler optimizations to freely move objects in memory. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language, following Wittgenstein:

*Whereof one cannot speak, thereof one must be silent.*

In [30] a Hoare logic is presented to verify 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. This abstraction requires new techniques for computing the weakest precondition of object creation statements because in the state prior to the creation of the object this new object does not exist so that we cannot refer to it. Hence, note that we cannot simply obtain the weakest precondition as usual, by substitution in the post-condition of the variable to which the new object is assigned, because there does not exist a term for it in the state prior to the creation of the object! Moreover, because of the abstraction level in the assertion language, quantification over objects only involves created objects. Consequently, the scope of the quantifiers is also affected by object creation which has therefore to be taken into account by the weakest precondition calculus.

The main contribution of this work is twofold. First, we provide a new formalization of a weakest precondition calculus for abstract object creation in dynamic logic which extends the initial work in [2] to classes, arrays and dynamic binding. Dynamic binding is supported using a transformational approach, see [8] for details. The logic allows the specification and verification of object-oriented programs at the abstraction level of the ABS. That is, besides the above operations on objects, the logic only supports quantification over created objects, including array objects. Consequently, objects not (yet) created can not be referred to by any construct in the logic. We show that the standard first-order sequent calculus of this logic which forms the basis of the KeY theorem prover [12] adequately captures the abstract data type of objects. All new proof rules have been fully implemented in a variant of KeY. As the second contribution, we extend the dynamic logic with auxiliary variables and we investigate the expressiveness of the resulting assertion language. Since arrays in ABS are also objects, this indirection allows for the first-order specification of properties of object structures which cannot be expressed directly in first-order logic, like reachability.

Related work

Specification languages like the Java Modeling Language (JML) [74] and the Object Constraint Language (OCL) [86] also abstract from the underlying representation of objects. To the best of our knowledge, all known tools for the (deductive) verification of object-oriented programs are based on some explicit representation of objects, e.g., objects are represented by natural numbers and counters are used to model object creation. Such an explicit representation can be useful in the context of programming languages which support for example pointer arithmetic, but it does not comply with the abstract data type of objects as provided by object-oriented languages like ABS. Further, we show in this section that also the verification engine itself does not require such an explicit (internal) representation.

Pure first-order logic without auxiliary variables is not expressive enough to assert, for example, reachability properties. For example, in [6] a formalization of abstract object creation is given using inductively
defined predicates (so-called pure methods). Such predicates are typically used in specifying the footprint of a program, which is crucial in tools based on separation logic (VeriFAST [70], jStar [47], Slayer [18] and Smallfoot [17]) to facilitate local reasoning. Another approach is taken by Why3 [54], PVS [96] and Isabelle [71], which generate verification conditions in standard higher-order logic (without introducing additional logical connectives such as the separating conjunction and points-to operator of separation logic). However in general higher-order logic seriously complicates proof theory (in contrast to first-order logic, the validities are not recursively enumerable) and thus automation.

We show that KeY extended with an abstract data type of objects and auxiliary variables allows both the first-order specification and verification of programs at the abstraction level of the programming language. For the verification, dynamic logic is used as a systematic formalization of abstract object creation, as a generator of first-order verification conditions. KeY itself is one of the state-of-the-art verification tools. To the best of our knowledge our extension of KeY is the first tool which supports abstract object creation as described above.

2.2.1 Core Language: Syntax and Semantics

We focus only on the core features of ABS which are relevant for our extension of object creation to arrays, classes and dynamic binding. Further we briefly present a corresponding assertion language based on first-order dynamic logic (DL). A full account of KeY style DL is found in [14].

The language is strongly typed and contains the primitive types Nat and Boolean. Additionally there are user-defined classes C, and predefined classes T[ ] of unbounded arrays in which the elements are of type T. Arrays are dynamically allocated and are indexed by natural numbers. Formally arrays are not included directly in the ABS, however they can easily be defined in ABS as classes containing methods a.get(i) and a.set(i,v) for respectively getting and setting the element at index i of array a. Thus adding arrays does not increase the expressiveness of ABS, they are merely added here for convenience. Clearly it would be desirable to treat all implementations of array classes in a uniform (i.e. implementation independent) manner. We do so here by adding the operations of getting and setting to the formal syntax of our language, with the more usual notation a[e] instead of a.get(i) (and similarly for a.set(i,v)). Multi-dimensional arrays are modeled (as in Java) as arrays in which the elements themselves are arrays. For instance, Nat[ ] is the type of one dimensional arrays of natural numbers and Nat[[ ]] is the type of two-dimensional arrays of natural numbers. We will not be concerned with type checking, but only note that it can be done statically, and it is orthogonal to the semantics of the language.

Expressions of our language are side-effect free. The following grammar generates the language of expressions, respectively statements:

\[
e ::= u \mid e.x \mid \text{null} \mid e_1 = e_2 \mid \text{if } b \text{ then } e \text{ fi} \\
   \text{if } b \text{ then } e_1 \text{ else } e_2 \text{ fi} \mid e_1[e_2] \mid f(e_1, ..., e_n) \mid C(e)
\]

\[
s ::= \text{skip} \mid s_1; s_2 \mid \text{if } b \text{ then } s_2 \text{ else } s_3 \text{ fi} \mid \text{while } e \text{ do } s \text{ od} \mid \text{abort} \\
   m(e_1, ..., e_n) \mid u ::= \text{new} \mid u := e \mid e_1[e_2] := e \mid e_1.x := e
\]

The function \( f(e_1, ..., e_n) \) denotes an arithmetic or Boolean operation of arity \( n \). For class types C the Boolean expression \( C(e) \) is true if and only if the (dynamic) type of \( e \) is exactly C. Formally dynamic logic formulas are generated by the following grammar:

\[
\phi ::= b \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \phi \rightarrow \phi_2 \mid \exists l : \phi \mid \forall l : \phi \mid \{U\} \phi \mid [s] \phi \mid \langle s \rangle \phi
\]

In this grammar, \( b \) is a Boolean expression, \( s \) is a statement, \( U \) is an update and \( l \) is a logical variable of any type of our core language. A formula \( \langle p \rangle \phi \) is true in a model \( M \) if the execution of \( p \) terminates when started in \( M \) and results in a model where \( \phi \) is true. A formula \( [p] \phi \) is true in a model \( M \) if the execution of \( p \), when started in \( M \), either does not terminate or results in a model in which \( \phi \) is true. In other words, the difference between the operators is the one between total and partial correctness.
**Semantics.** The basic notion underlying the semantics of both the programming language and the assertion language is that of a many-sorted model with disjoint domains \( \text{dom}(T) \) for each type \( T \) and an interpretation \( I \). The interpretation assigns a function (a meaning) to each function symbol, and a relation to each relation symbol. The domain for \( \text{Nat} \) is interpreted in the standard way, i.e., denotes the set of natural numbers. We assume that the arithmetical operations defined on this domain includes at least addition. Moreover we assume a binary function symbol \([\ ]\), which when interpreted in a model \( M \), given a number \( n \) and an array \( o \) yields the value of the array element \( o[n] \) in \( M \). We sometimes use the notation \( M([\ ]_T)(n)(o) \) for this value.

The meaning of an expression \( e \) of type \( T \) is a (total) function \([e]\) that maps a model \( M \) to an individual of \( M(T) \). This function is defined in the standard way. Statements in our language are deterministic and can fail (\text{abort}) or diverge. Their meanings are defined operationally, and use the (quite common) notation
\[
\langle s, M \rangle \rightarrow M'
\]
to express that executing \( s \) in the model \( M \), terminates in the model \( M' \).

The meaning of normal assignments, conditional statements and while loops is defined in the standard way. We focus on the semantics of an array creation:
\[
\langle u := \text{new}, M \rangle \rightarrow M'
\]
where \( u \) has type \( T[\ ] \) and \( M' \) satisfies:

1. \( o = \nu(M,T[\ ]) \).
2. \( M'(T[\ ]) = M(T[\ ]) \cup \{o\} \).
3. \( M'([\ ]_T)(n)(o) = \text{init}_T \) for all \( n \in M(\text{Nat}) \).
4. \( M'(u) = o \).
5. The domains other than \( \text{C} \) and the interpretations of the other non-logical symbols in \( M' \) are the same as in \( M \).

For the selection of a new object of class \( \text{C} \) we use a choice function \( \nu \) on a model \( M \) and class \( \text{C} \) to get a fresh object \( \nu(M, \text{C}) \) of class \( \text{C} \) which satisfies \( \nu(M, \text{C}) \notin M(T) \) for any class \( T \) (in particular, \( \nu(M, \text{C}) \notin M(\text{C}) \)). The third clause states that all elements in the array are initialized to their default value.

We write \( M \models \phi \) if the formula \( \phi \) is true in the model \( M \) (with the standard Tarski definition of truth). A formula \( \phi \) is valid if \( M \models \phi \) holds for every model \( M \). Interestingly, even though we allow quantification over arrays, all assertions are \textit{first-order} \( \nu \) dynamic logic formulas. This is because of a subtle difference in meaning between modeling arrays as sequences (not first-order), or as pointers to sequences (first-order).

In the first case \( \exists s : s[0] = 0 \) expresses that there exists an array \( s \) of natural numbers, of which the first is 0. The array itself is not an element of the domain of a model for our many-sorted dynamic logic language, but rather a sequence of elements of the domain \( \text{Nat} \). In this interpretation the above formula is valid.

The second option of modeling arrays as pointers is taken in this work. For a logical variable \( l \) of type \( T \) we have the following semantics of existential quantification:
\[
M \models \exists l. \phi \text{ iff for some } o \in M(T) : M' \models \phi.
\]
where \( M' \) differs from \( M \) only in \( M'(l) = o \). The semantics of universal quantification can be derived using the equivalence \( \forall l : \phi \leftrightarrow (\exists ! l : \neg \phi) \). In this interpretation for quantification, if \( a \) is a logical variable of type \( \text{Nat}[\ ] \) then \( \exists a : a[1] = 0 \) asserts the existence of an array object (an individual of the domain for \( \text{Nat}[\ ] \)) in which currently the first element is 0. This formula is not valid. It is false in all models in which no such array object exists. Note also that the \textit{extensionality} axiom \( \forall a, b, n : a[n] = b[n] \to a = b \) for arrays is also not valid.
2.2.2 Applications of Updates

In this section, we define a rewrite relation on dynamic logic formulas with updates, to standard first-order logic formulas without updates. This relation is necessary to reason about formulas containing updates. Updates are essentially delayed substitutions. They are resolved by application to the succeeding formula, e.g., \( \{u := e\}(u > 0) \) leads to \( e > 0 \). Update application is only allowed on formulas not starting with either a diamond, box or update modality.

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: \( \{U\} e \leadsto e' \).

For the definition of this rewrite relation for standard cases, see [90, 12]. Thus we focus on object creations (possibly of an array type). To define update application on \( \{u := \text{new}\} e \), simply substituting \( u \) by \( \text{new} \) is not possible, since \( \text{new} \) is not an expression. In fact, prior to the creation of the new object, there is no term which denotes the new object. Hence a simple substitution of \( u \) by some term is not sufficient. However, object expressions can only be compared for equality, dereferenced, accessed as an array if the object is of an array type, or appear as arguments of a class predicate or conditional expression. Since object expressions do not appear as arguments of any other function, we define update application by a contextual analysis of the occurrences of \( u \) in \( e \). The rules are summarized in the Table 2.2 and illustrated on the left side in Figure 2.3.

2.2.3 Discussion

Object Creation vs. Object Activation

Proof systems for object-oriented languages [1] usually achieve the uniqueness of objects via an injective mapping, here called \( \text{obj} \), 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 model, the successor of this maximum index is stored in a ghost variable, here called \( \text{next} \). In case of Java, \( \text{next} \) would be a static field, for each class. Object creation increases the value of \( \text{next} \), which conceptually is more an activation than a creation. Quantifiers cover the entire co-domain of \( \text{obj} \), including “not yet created” objects.

In order to restrict a certain property \( \phi \) to the “created” objects, the following pattern is used: \( \forall l. (\psi \rightarrow \phi) \), where \( \psi \) restricts to the created objects. Formulas of the form \( \exists n. (n < \text{next} \land \text{obj}(n) = l) \) are the approach taken in ODL [15]. To avoid the extra quantifier, a ghost instance variable of Boolean type, here called \( \text{created} \), can be used to indicate for each object whether or not it has already been “created” [14]. In this case we set the \( \text{created} \) status of the “new” object (identified by \( \text{next} \)) and increase \( \text{next} \). The assertion \( \forall n. (\text{obj}(n).\text{created} \Leftrightarrow n < \text{next}) \) retains the relation between the \( \text{created} \) status and the object counter \( \text{next} \) on the level of the proofs. In both cases, we need further assertions to state that fields of created objects always refer to created objects.

To state in this setting that a new object indeed is new, we need to add conditions to restrict the scope of quantification to created objects: \( \forall l. (l.\text{created} \rightarrow (u := \text{new}) \neg (u = l)) \). An object activation style proof of this is given in Figure 2.3 on the right (abbreviating \( \text{created} \) by \( \text{cr} \)). Many steps in this proof are caused by the particular details of the explicit representation of objects and the simulation of object creation by object activation.

Expressiveness

Standard first-order logic cannot express reachability properties. We have proposed first-order logic together with auxiliary variables to specify properties of the heap. The question arises how expressive our approach is in general. In the presence of general abstract data types, Tucker and Zucker [94] observe that for expressing, for example, strongest post-conditions, standard arithmetic coding techniques do not apply. Therefore Tucker and Zucker prove expressibility of strongest post-conditions in a weak second-order language which contains quantification over finite sequences. We show in [52] that the strongest post-condition of a formula in the
\( \{U\} u' \leadsto u' \)

where \( u, u' \) are different variables

\( \{U\} e.x \leadsto e' \)

\( \{U\}(e.x) \leadsto e' \)

where \( e \) is neither \( u \)

nor a conditional expression

\( \{U\}(e[e_1]) \leadsto e' \)

\( \{U\}(e[e_1]) \leadsto e' \)

where \( e \) is neither \( u \)

nor a conditional expression

\( \{U\}(e = e') \leadsto \text{true} \)

where \( e \) and \( e' \) are \( u \)

\( \{U\}(C(e)) \leadsto \text{true} \)

where \( e \) is \( u \) and \( u \) is of type \( C \)

\( C(\{U\}e) \leadsto e' \)

\( \{U\}(C(e)) \leadsto e' \)

where \( e \) is neither \( u \)

nor the conditional expression

\( \forall l. (\{U\} \phi) \leadsto \psi \)

\( \{U\} \forall l. \phi \leadsto \psi \)

where \( l \) is a logical variable

of the same type as \( u \)

\( (\{U\} \phi[u/l]) \land \forall l. (\{U\} \phi) \leadsto \psi \)

\( \{U\} \forall l. \phi \leadsto \psi \)

where \( l \) is a logical variable

of a different type as \( u \)

Figure 2.2: Simplification of object creation with \( U \equiv u := \text{new} \)
language of Presburger arithmetic, and a program instrumented with auxiliary variables in a suitable way is definable in Presburger arithmetic itself. This is surprising, since the standard approach to show that the strongest post-condition is definable is based on the usual Gödel encoding of partial recursive functions, which relies on the presence of multiplication in the assertion language, but multiplication is not available in Presburger arithmetic. The basic idea is that one can instrument any program to store the computation in auxiliary array variables. The computation can then be recovered in an assertion by accessing these auxiliary variables.

2.2.4 Conclusion

We showed how the assertion language used in KeY can be used conveniently together with auxiliary variables to provide a powerful way to express properties of the heap of ABS programs. Moreover the assertion language supports abstract object creation (including dynamically created arrays), abstracting from irrelevant implementation details of object creation, which in general complicate proofs. First-order dynamic logic was used as a systematic way to formalize the inductively defined rewrite relation needed to reason about abstract object creation, and to generate verification conditions. Tool support is provided by a special version of KeY available on [http://keyaoc.hats-project.eu/](http://keyaoc.hats-project.eu/). A larger case study which illustrates the proof rules and rewrite relation can be found in paper 1 from the list of papers.

Future Work. A main line of future research concerns the integration of different techniques to further support modularity, i.e., local reasoning as supported by the separating conjunction of separation logic and dynamic frames.

2.3 Verification in Delta-Oriented Programming

2.3.1 Introduction

In this section we focus on the variability and evolvability aspect when verifying ABS programs. Current state-of-the-art approaches to formal specification and verification do not support fast-paced changes: specification and verification effort is largely wasted when changes occur, systematic reuse is not possible (see Sect. 2.3.6 for a discussion).

To improve the situation, two things are required: (i) formal specifications and verification proofs must be equipped with a systematic reuse principle; (ii) the reuse principle employed in the targeted code must
match the one in specifications and proofs, so it is possible to reflect code changes when reasoning formally about them.

The standard composition principle of modern programming languages are procedures or method calls. To render formal verification scalable with the size of target programs, most approaches use a formalization of Meyer’s design-by-contract principle [78], where the behavior of a method is captured in the form of a contract between caller and callee. Contract-based specification has been realized for industrial target languages such as Java by specification languages such as JML [75] or in specific program logics, e.g., [12, Ch. 3]. The central idea of contract-based formal verification is to substitute a method call in the target code with a declarative specification of the effect of the call, obtained from the obligation that the callee ensures in its contract. For this to work, two things are necessary: first, the called method must have been successfully verified against its contract; second, the application requirements of the contract must be fulfilled in the call context. The problem of keeping up with target code changes during verification can be formulated in this framework as follows: Assume we have successfully verified a given piece of code \( p \). Now, one of the methods \( m \) called in \( p \) is changed, i.e., \( m \)'s contract in general is no longer valid. Therefore, this contract cannot be used in our proof of \( p \) which is accordingly broken and must be redone with the new contract of \( m \). If \( p \) contains loops, then new invariants must be found, which is time consuming and expensive.

A rather restrictive approach to the change problem is Liskov’s principle [77]: here, the new contracts must be substitutable for the old ones or, equivalently, only code changes that respect the existing contracts are permitted. But, even with optimizations [48, 49], this is too restrictive in practice, because already very simple code modifications tend to break existing contracts. A more fundamental solution is called for, and this is the contribution of this section.

Our approach consists of two elements: the first element is an extension of the delta-oriented programming approach followed in ABS. We extend deltas to be used as a structured reuse principle for both code and contracts. Deltas in contracts foster reuse of specifications and make reused parts syntactically explicit. The details are in Sect. 2.3.3. But the problem remains that modified contracts are in general no longer applicable in a proof. It is impossible to figure out at method-call time whether a modified contract (or parts of it) might still be applicable or useful for the proof at hand. Therefore, the second ingredient of our approach is to disentangle in proofs the analysis of program code and the application of method contracts. This is achieved with abstract contracts in Sect. 2.3.4. In Sect. 2.3.5 we show that by combining abstract contracts, structured reuse of contract-based specification, and caching of first-order goals, it is possible to establish reusability of contracts by first-order reasoning. The result is a verification framework for programs under change, where reusable verification tasks are detected automatically.

### 2.3.2 Verification Framework

As mentioned above, we work in a contract-based [78] verification setting using ABS as target language and deltas to express variability. In our terminology we follow closely that of KeY and JML [12, 75]. We use an obvious notation to access classes \( C \) and methods \( m \) within a program \( P \): \( P.C, P.C.m \), etc.

**Definition 2.3.1.** A program location is an expression referring to an updatable heap location (variable, formal parameter, field access, array access). A contract for a method \( m \) consists of:

1. a first-order formula \( r \) called precondition or requires clause;
2. a first-order formula \( e \) called postcondition or ensures clause;
3. a set of program locations \( a \) (called assignable clause) that occur in the body of \( m \) and whose value can potentially be changed during execution.

We extend our notation for accessing class members to cover the constituents of contracts: \( C.m.r \) is the requires clause of method \( m \) in class \( C \), etc.

**Definition 2.3.2.** Let \( m(p) \) be a call of method \( m \) with parameters \( p \). A total correctness expression has the form \( ⟨m(p)⟩Φ \) and means that whenever \( m \) is called then it terminates and in the final state \( Φ \) holds where \( Φ \)
is either again a correctness expression or it is a first-order formula. (Partial correctness adds nothing to our discussion: we omit it for brevity.)

In first-order dynamic logic [12] correctness expressions are just formulas with modalities. One may also encode correctness expressions as weakest precondition predicates and use first-order logic as a meta language, as typically done in verification condition generators (VCGs). Either way, we assume that we can build first-order formulas over correctness expressions, so we can state the intended semantics of contracts: Validity of the formula \( r \rightarrow \langle m(p) \rangle e \) expresses the correctness of \( m \) with respect to the pre- and postcondition of its contract. In addition we must state correctness of \( m \) with respect to its assignable clause: one can assume [52] there is a formula \( A(a, m) \) whose validity implies that \( m \) can change at most the value of program locations in \( a \). To summarize:

**Definition 2.3.3.** A method \( m \) of class \( C \) satisfies its contract if the following holds:

\[
\models C.m.r \rightarrow \langle m(p) \rangle C.m.e \land A(C.m.a, C.m) \tag{2.1}
\]

The presence of contracts makes formal verification of complex programs possible, because each method can be verified separately against its contract and called methods can be approximated by their contracts (see method contract rule below). The assignable clause of a method limits the program locations a method call can have side effects on.

**Example 1.** Fig. 2.4 shows a simple bank account interface and its implementation. Contract elements appear before the method they refer to and start with a \( @ \). The method deposit(x) is specified with a contract whose precondition in the @requires clause says that the balance should be positive. The postcondition in the @ensures clause expresses that the balance after the method call is equal to the balance before the method call.

We are aware that this basic technique is insufficient to achieve modular verification. Advanced techniques for modular verification, e.g. [11], would obfuscate the fundamental questions considered in this section and can be superimposed.
plus the value of parameter \(x\). For simplicity, we use (similar to JML) the keyword \(\text{old}\) to access prestate values, but saving old values in renamed locations is equally possible. The obvious dual method \(\text{withdraw}(x)\) is not shown. Method \(\text{move}(\text{amount},a,b)\) moves an amount between accounts. This should not increase the overall balance as stated in its \@ensures\ clause.

In a calculus for verification of a method like \(\text{move}(\text{amount},a,b)\) against its contract, methods calls (here, \(\text{deposit}(x)\) and \(\text{withdraw}(x)\)) must be replaced by their contracts using the method contract rule to achieve scalability:

\[
\text{methodContract} \quad \frac{\Gamma \Rightarrow m.r \quad \Gamma \Rightarrow U_{m,a}(m.e \rightarrow (\langle \omega \rangle \Phi))}{\Gamma \Rightarrow (m(p)) ; \Phi} \tag{2.2}
\]

The rule is applied to the conclusion below the line: in a proof context \(\Gamma\) (a set of formulas) we need to establish correctness of a program starting with a method call \(m(p)\) with respect to a postcondition \(\Phi\) (typically, an ensures clause). The rule uses the contract of \(m\) and reduces the problem to two subgoals. The first premise establishes that the requires clause is fulfilled, i.e., the contract is honoured by the callee. That is exploited in the second premise, where the ensures clause can now be used to prove that the remaining program \(\omega\) is correct. But one must be careful with the possible side effects the call might have had on the values of locations listed in the assignable clause of \(m\)'s contract. As we cannot know these, we use a substitution \(U_{m,a}\) to set all locations occurring in \(m.a\) to fresh Skolem symbols not occurring elsewhere. It is intentional that we do not commit to a specific calculus or program logic. Soundness of the method contract rules is formally stated here:

**Theorem 2.3.4.** If the implementation of \(m\) satisfies its contract, then rule (2.2) is sound.

**Proof.** The method contract rule is fairly standard except for the use of the substitution \(U_{m,a}\) which encodes the assignable clause of the contract. In [16] a theorem is shown from which the correctness of (2.2) follows as a special case.

In general, a contract may involve several specification cases, connected by the keyword \text{also}. It is possible to reduce this by propositional reasoning to proof obligations of the form that occur in the premises of rule (2.2). So we assume wlog to deal with a single specification case at verification time. Similarly, each specification case may have multiple requires and ensures clauses, implicitly connected by conjunction. For simplicity, we assume wlog the presence of at most one requires/ensures clause per specification case.

The question we focus on in the following is: What happens with a correctness proof when the implementation of a called method changes? Liskov’s well-known substitutability principle [77], rephrased in terms of contracts, gives one answer.

**Definition 2.3.5 (Substitutable Contract).** For two methods \(m, m'\), with contracts \(m.r, m'.r', m.e, m'.e', m.a, m'.a',\) the second method’s contract is substitutable for the first if the following holds:

\[
(m.r \rightarrow m'.r') \land (m'.e' \rightarrow m.e) \land (m'.a' \subseteq m.a) \tag{2.3}
\]

The next lemma is immediate by the definition of contract satisfaction (Def. 2.3.3), propositional reasoning over (2.3), and monotonicity of postconditions in total correctness formulas.

**Lemma 2.3.6.** If a method \(m'\) satisfies its contract, then it satisfies as well any contract substitutable for it.

This justifies Liskov’s principle and guarantees that a proof stays valid, whenever we replace a method by one whose contract is substitutable for it. As we shall soon see, substitutability is much too restrictive to be practically useful.
2.3.3 Delta-Oriented Reuse of Programs and Contracts

If a program evolves due to bug fixes, newly added features or other modifications, the program code itself and also its specification in form of method contracts changes. To enable systematic reuse for verification, we need to represent changes explicitly. In HATS we use delta-oriented programming [91, 21, 35] (DOP) as code reuse technique. To enable verification reuse we extend deltas to represent anticipated and unanticipated changes of programs and specifications in a structured manner.

Program Deltas

Deltas as described in [35] may add, remove, and modify elements of the target program. At the class level, we use the keywords modifies, adds, and removes preceding the changed class declaration. With adds, new classes can be created and with removes a whole class can be removed. A modified class contains further directives that may change its method and field declarations, for which the same keywords are used (for fields, only adds and removes are permitted). In general, the modifies directive is a mere convenience, as it can be replaced by a suitable combination of removes and adds.

A modified method declaration can be completely replaced or it can be a wrapper using the original call. The keyword original stands for a call to the most recent version of the currently modified method and it must match its type signature. The original construct makes code reuse possible and is reminiscent of super calls in standard OO languages. A main difference between original and super is that the former, as all other changes contained in a delta, are resolved at compile time, when applying a delta to an existing program. The same method can be modified and wrapped in several subsequent delta applications capturing individual changes.

Example 2. We want to extend the account class of Ex. 1 with a feature to charge a fee for each deposit. This is realized in the delta in Fig. 2.5. Delta declarations begin with the keyword delta, followed by a name and optional parameters, here, the amount of the transaction fee. The delta modifies class Account and its method deposit(x) by wrapping and reusing the previous version with an original call. The conditional around the call to original ensures that the deposited amount is not lower than the fee to avoid counter-intuitive results.

Obviously, the application of a delta to a given program may fail. The modifies and removes directives implicitly assume the existence of program elements that may be missing, the type signature of an original call may not match, etc. Therefore, compilation in DOP is a two-stage process: in a first step, deltas are applied in a user-specified sequence, where the well-definedness of each delta application is checked, and the result is a flattened, delta-free program to be processed further with a standard compiler. To specify products resulting from delta applications, we use the keyword product, followed by a name and a sequence of the deltas that are to be applied.

Example 3. Fig. 2.6 shows the declaration of a bank account product with deposit fee derived from Ex. 2 (for which the name Base is used by convention) and subsequent application of the DFee delta of Ex. 3, where the parameter is instantiated with 2 units. The declaration of class Account is generated automatically by the delta compiler.
product AccountWithFee(Base, DFee(2));
// results in:
interface IAccount { Unit deposit(Int x); }
class Account implements IAccount {
    Int balance = 0;
    Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); }
}

Figure 2.6: Result of applying the DFee delta to the bank account

delta DFee(Int fee);

modifies class Account {
    // modify the only existing specification case
    modifies @requires \original ∧ x >= fee;
    modifies @ensures balance == \old(balance) + x - fee;
    // add a new specification case
    adds also @case TrivialAmount
        @requires \original ∧ x < fee;
        @assignable \nothing;
    modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); }
    modifies @requires \original ∧ amount>=fee
    of Unit move(Int amount, IAccount a, IAccount b) }

Figure 2.7: Delta changing the specification of the deposit method

Contract Deltas
Changing program code typically requires to change method contracts since a change might intentionally cause a different functionality that has to be reflected in the contract. Hence, it is natural to extend the concept of deltas to contracts. Following \[19\,63\], we permit the keywords \textit{adds}, \textit{modifies}, and \textit{removes} in front of specification cases and requires/ensures/assignable clauses. If there is more than one specification case we use names to distinguish them. A name clause is of the form ”@case <name>;” and this name can be used to qualify a \textit{modifies} or \textit{removes} directive.

Going beyond \[19\], and in analogy to program deltas, we allow reuse of specifications in modified and added contract clauses using the keyword \textit{original}. As with program deltas, this means that the most recent version of the contract clause is replaced for the \textit{original} keyword when the delta is applied. If there is more than one specification case, \textit{original} can be qualified with a name.

Example 4. For the contract of the modified \textit{deposit(x)} method in DFee (Fig. 2.5), we want to reuse the contract of the \textit{original} method shown in Fig. 2.4. This can look as in Fig. 2.7. Note that we need two specification cases: one when the fee does not exceed the deposited amount and one when it does. The first is obtained as a modification of the existing contract: in the requires clause, a suitable precondition is added to the \textit{original} requires clause. The previous version of the ensures clause is replaced by a new version which takes the deduction of the fee into account. The assignable clause is untouched. The second case is obtained by \textit{adds}. Again, the \textit{original} precondition is reused. In this case, the balance of the account remains unchanged,
@requires x > 0 ∧ x >= 2;
@ensures balance == \old(balance) + x - 2;
@assignable balance;
also
@case TrivialAmount
@requires x > 0 ∧ x < 2;
@assignable \nothing;
Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); }

Figure 2.8: Result of applying the DFee program and contract delta

which is implied by the new assignable clause. While this is sufficient, it is hard to detect and, therefore, to exploit.

Example 5. If we compute the product shown in Fig. 2.6 by delta application including the base contract and the contract of the delta we obtain the contract of the modified deposit(x) method shown in Fig. 2.8.

Verification of Deltas

The main question in a formal verification context is: how to prove that a program delta satisfies its contract delta? In general, this is not possible for a delta in isolation, for two reasons: the first is, before actually applying a delta, its code and its contract are partially unknown. In [63] we addressed this issue by imposing a number of constraints: occurrences of original in requires clauses must be of the form original ∨ r', in ensures clauses of the form original ∧ e', and in assignable clauses of the form original\remove{a}. This ensures substitutability of reused contracts and makes Lemma 2.3.6 applicable. The second problem is that a method satisfying its contract might cease to do so after modification of either code or contract. In [63] we imposed two conditions relative to a given partial order ≺ of delta applications: (i) modified contracts in ≺-larger deltas must be substitutable, and (ii) calls to methods that might have been modified are replaced with the ≺-minimal contract of that method. Under these conditions, satisfaction of each contract in the base and in all deltas implies that all contracts in any ≺-compatible product are satisfied [63, Thm. 1]. Unfortunately, these restrictions are too severe in practice:

Example 6. Consider the contracts in Figs. 2.4, 2.7. The ensures clause of the modified contract introduces the parameter fee and bears no logical relation to the clause it replaces. In the added specification case, the implicitly given default ensures clause “@ensures true;” is weaker, not stronger as required.

2.3.4 Abstract Method Calls

Ex. 6 shows that already minor changes to programs violate Liskov’s principle. This makes reuse of verification effort problematic in general:

Example 7. Consider method move() from Fig. 2.4 which transfers money between two accounts. Its contract states that money might be lost, because of fees or similar, but it strictly excludes generation of money. Its implementation calls deposit() to credit the receiving account. To prove that move() satisfies its contract requires to apply the method contract rule (2.2) to deposit(). Changing the contract of deposit() to the version in Fig. 2.8 entails that neither of the two specification cases satisfies Liskov’s principle (see Ex. 6). Consequently, in the verification of move() no reuse is possible.

The previous example highlights an unfortunate property of the usual verification setup: assuming we use a complete verification method, intuitively, the logical information in the proof with the call to the

2Relatively complete, with respect to Peano arithmetic, of course.
Figure 2.9: Shape of an abstract method contract

Figure 2.10: Abstract method specification for deposit without fee

original deposit() should be sufficient to justify the contract of move() even for the version with a fee. But
this cannot be detected easily, because the proof of move()’s contract uses the ensures clause of deposit().
Now, when deposit()’s contract is changed, it is impossible to disentangle the information from deposit()’s
contract and the steps used to prove move(). To achieve such a separation we need a technique that splits
method invocation from the actual contract application. This is the central contribution of our work and
explained now.

The main technical idea is to introduce a level of indirection into a method contract that allows to delay
the substitution of its concrete requires and ensures clause. We call this an abstract method contract. It has
the shape shown in Fig. 2.9. Its abstract section consists of the standard requires, ensures, and assignable
clauses. As before, the assignable clause specifies all locations that might be changed by the specified method.
The requires and ensures clauses, however, now merely contain placeholders \textit{R}, \textit{E}, and \textit{\textbackslash def}(\textit{l}_i), which are
defined by concrete formulas and terms in the definitions section, which must contain a definition for each
placeholer in the abstract section. The ensures clause is a conjunction of equations specifying the post
value of each possibly changed location in the assignable clause and additional properties \textit{E} on the post
state. Please note that Fig. 2.9 is merely a convenient notation. The formal definition of an abstract method
contract is given in Def. 2.3.7 below.

The main restriction inherent to abstract method contracts is that the assignable clause explicitly lists
updatable locations (i.e., it is not abstract). Nevertheless, it is part of the abstract section, so that it is
shared by all clients of the contract. This is necessary to ensure that applying any abstract contract for
a method has the same result before definitions are unfolded. Another minor restriction is the equational
form, which enforces that the post value for any assignable location is well-defined after contract application.
Field accesses occurring in definitions are expressed using getter methods. This ensures that their correct
value is used when definitions are unfolded.

\textbf{Example 8.} Fig. 2.10 reformulates the contract of method \texttt{deposit()} in terms of an abstract method contract.

Abstract method contracts are fully compatible with contract deltas, with the restriction that assignable
clauses may not be changed. The only difference is that all changes specified in a delta are acted upon in
the definitions section of an abstract contract—the abstract section remains completely unchanged. In our
example, the application of the delta in Fig. 2.7 results in an abstract contract with two specification cases,
Figure 2.11: Abstract method specification case for a successful deposit() with fee

one of which is shown in Fig. 2.11.

It is perhaps surprising that original still occurs after delta application. The explanation is that the abstract shape of contracts does not force us anymore to unfold original references immediately. As we shall see in Sect. 2.3.5, it can have advantages not to do so. To indicate that the original has been in fact resolved, we add a reference (here: Base) to its container. Now we can define abstract method contracts formally:

Definition 2.3.7 (Abstract method contract). An abstract method contract $C_m$ for a method $m$ is a quadruple $(r, e, U, \text{defs})$ where

- $r, e$ are logic formulas representing the contract’s pre- and postcondition,
- $U$ is an explicit substitution representing the assignable clause, and
- $\text{defs}$ is a list of pairs $(\text{defSym}, \xi_{\text{defSym}})$ where defSym are non-rigid (i.e., state dependent) function or predicate symbols used as placeholders in $r$, $e$, and $\xi$ their defining term or formula. For each $\text{def}(l_i)$ there is a unique function symbol in defs. For simplicity, we refer to both with $\text{def}(l_i)$, as long as no ambiguity arises.

Placeholders must be non-rigid to prevent the program logic calculus to perform simplifications over them that are invalid in some program states. To ensure soundness of the abstract setup we add the definitions of the placeholders (i.e., the contents of the definition section of each abstract contract) as a theory to the logic, just like other theories, such as arithmetic, etc. This means that the notion of contract satisfaction (Def. 2.3.3) is now able to consider defined symbols in abstract contracts. Additionally the rule expandDef on the right that substitutes placeholders by their definitions is now obviously sound. The advantage of this setup is that we can still use the old method contract rule (2.2), which simply ignores the definition section. As we changed neither the satisfaction of contracts nor the method contract rule, Thm. 2.3.4 still holds.

2.3.5 Application Scenarios

Abstract Verification

Rule (2.2) now uses only the abstract section of an abstract method contract. Hence, its application yields the same result for all method contracts that share the same abstract section. This allows to define the following general verification process: Assume we want to establish that a method $m$ satisfies a contract $C$. In the first phase the rules of the underlying calculus are applied until only first-order proof goals are remaining. This can be done, for example, with symbolic execution or with a verification condition generator (VCG) and typically involves manual annotation of the target program with suitable loop invariants. During this phase all calculus rules, including SMT (satisfiable modulo theory) and first-order solvers, except for expandDef may be used to close subgoals. If the implementation of $m$ contains a call to another method $n$
we use an abstract contract \( C_n \) for the latter. Because of this, some first-order subgoals will usually remain open. Let us call this partial proof \( p \).

There are two things one can do at this stage: first, we can use the \texttt{expandDef} rules of the definition section of \( C_n \) and first-order reasoning on the open subgoals of \( p \). If \( m \) satisfies \( C \) and suitable invariants were chosen in \( p \), this complete the proof by first-order reasoning. Second, assume now we made changes to \( n \) and modified the definition section of its contract, let’s call it \( C'_n \). As long as \( C_n \) and \( C'_n \) have the same abstract section, we can reuse proof \( p \) completely. To test whether \( p \) still satisfies \( C \) after the change, it is sufficient to use the \texttt{expandDef} rules of the definition section of \( C'_n \) on \( p \). Again, this is a first-order problem. This is significant, because coming up with the right invariants is usually much more expensive than first-order reasoning.

\textbf{Example 9.} Applying the verification rules of KeY \cite{key} to show that \texttt{move()} satisfies its contract (Fig. 2.4), while using abstract contracts of \texttt{deposit()} and \texttt{withdraw()}, results in a partial proof \( p \) with the open first-order goal

\[
\text{\texttt{def}(a.balance)} + \text{\texttt{def}(b.balance)} \leq \text{\texttt{old}(a.balance)} + \text{\texttt{old}(b.balance)}
\]

If we use the \texttt{expandDef} rules gained from the definition sections (cf. Fig. 2.10) we obtain the goal

\[
\text{\texttt{old}(a.getBalance())} - x + \text{\texttt{old}(b.getBalance())} + x \leq \text{\texttt{old}(a.balance)} + \text{\texttt{old}(b.balance)}
\]

which is trivial for an SMT solver. In addition, we can reuse \( p \) after applying the \texttt{DFee} delta to \texttt{deposit()}, because the abstract contracts in Figs. 2.10, 2.11 have identical abstract sections. With the rules gained from the definition section of Fig. 2.11 the resulting subgoal is still first-order provable.

\textbf{Liskov for Free}

A nice feature of our approach is that preservation of changed contracts as justified by Liskov’s substitutability principle (Sect. 2.3.3) is detected automatically. Let \( n' \) be a method whose contract \( C'_{n} \) is substitutable (2.3) for \( n \)’s contract \( C_n \) and assume \( m \) invokes \( n \). Abstract verification first constructs a partial proof \( p_m \) for \( m \) and its contract that has open first-order verification conditions \( V_m \). These contain placeholders from \( C_n \). Assume we can finish \( p_m \) by expanding their definitions plus first-order reasoning.

To verify that \( m \) still satisfies its contract when \( n \) is replaced with \( n' \), we proceed as follows: in \( V_m \) substitute each placeholder from \( C_n \) with its corresponding placeholder from \( C'_{n} \). This is possible, because both contracts have identical abstract sections. After expanding the definitions, by substitutability (2.3), one first-order subsumption step is enough to obtain the definitions from \( C_n \), which have been proven already. Therefore, (complete) first-order reasoning will automatically detect such a situation.

\textbf{Experiments}

We performed preliminary experiments using the KeY verification system. The necessary abstract method contracts and definition expansion rules have been provided manually. Manual steps for saving and loading of the partial proof were also necessary, but they will be fully automatic once proper support for abstract method calls is implemented.

The example used as running example throughout this section showed only modest gains by abstract method calls, which is not surprising considering the low complexity of the involved methods. Verifying the more complex method \texttt{requestTransaction()} which calls \texttt{deposit()} and contains a loop, we achieved savings of 90\%. The following table summarizes our results (proof complexity in number of nodes (branches)):

<table>
<thead>
<tr>
<th>Example</th>
<th>Cached Proof</th>
<th>Proof (Base)</th>
<th>Savings</th>
<th>Proof (DFee)</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>100 (6)</td>
<td>477 (10)</td>
<td>21%</td>
<td>517 (10)</td>
<td>19%</td>
</tr>
<tr>
<td>reqTrans</td>
<td>887 (20)</td>
<td>976 (23)</td>
<td>91%</td>
<td>979 (23)</td>
<td>91%</td>
</tr>
</tbody>
</table>
Program Evolution

For verifying a program that evolves by changing methods via delta operations, we can proceed as follows: For each contract $C_m$ of each method $m$ contained in the initial program, we construct a proof (e.g., by VCG or symbolic execution) using the abstract method contracts. We store the proof and also the open subgoals in a cache for future reference. Then, we unfold the definitions in the abstract method contract $C_m$ and use, e.g., an SMT solver to close the open subgoals to verify the method.

If the program evolves by delta application, we consider several cases: If the implementation of a method $m$ and its contract $C_m$ as well as all abstract sections of contracts $C_n$ for methods $n$ called by $m$ remain unchanged, we can completely reuse the stored partial proof for $C_m$ as in Sect. 2.3.5. For contracts of called methods $n$ that are unchanged, we can reuse previous proof goals stored in the cache. If the contracts of the called methods $n$ are changed, but substitutable for $C_n$, we obtain the proof as described in Sect. 2.3.5. If the contracts of called methods are changed in other ways, we unfold their definitions and obtain the proof by first-order reasoning and store the new proof goals in the cache. If the implementation of method $m$ or its contract changed, we need to construct a new proof for $C'_m$ as for the initial program. Here, we do not unfold the definitions of original when the partial proof is stored in the cache in order to be able to reuse partial proofs also for different instantiations of original. If, in the newly constructed proof, contracts for methods $n$ called by $m$ did not change or their contracts are substitutable wrt. previous contracts, we can reuse proof goals stored in the cache or apply the principle of Sect. 2.3.5. If a method is newly added, it has to be verified from scratch and the proof is stored in the cache.

2.3.6 Related Work

Previous approaches to deductive verification of evolving programs [13, 89] propose proof replay techniques to ameliorate verification effort. The old proof is replayed as long as possible, and at a mismatch a new proof rule is guessed. The paper [13] uses a similarity function to determine which parts of a previous proof can be reused, while [89] uses differencing operations. Unlike our work, proof replay is unrelated to the program or specification reuse principle.

In [49], a set of allowed changes to evolve an OO program is introduced which is similar to delta operations. For verified method contracts, a proof context is constructed which keeps track of the shown proof obligations. Program changes cause the proof context to be adapted so that the proof obligations that are still valid are preserved and new proof goals are created. The proof context is similar to the proof cache proposed in Sect. 2.3.5 but reuse only happens at the level of contracts, not on the level of (partial) proofs as in our work.

Several approaches have been proposed recently that target efficient verification of delta-oriented programs. These works are set in the context of software product line engineering where static program variability is considered in contrast to program evolution which is considered in this work. In [19], it is assumed that one program variant has been fully verified. From the structure of a delta to generate another program variant it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. The main result of [19], and its restrictions, are discussed in Sect. 2.3.3. In [28], methods in a delta are verified based on a contract which makes assumptions on the contracts of the called methods explicit. The main difference to our presented work is that reuse in the above approaches only happens at the level of the proof obligations limiting their reuse potential.

2.3.7 Conclusion

We presented a framework for systematic reuse of verification effort for programs and specification under change. Its distinctive feature is that reuse takes place at the level of code, specification, and proofs with a matching reuse principle. Our work highlights the importance of automated first-order reasoning in verification of programs that undergo frequent changes. Detaching the usage from the validation of contracts turns the test for reusability of previously cached results from a specific verification problem into a general
first-order problem. A logical next step is to investigate the nature and the relative difficulty of the resulting first-order problems.
Chapter 3

System-level Verification

3.1 Verification of Composition Patterns

In this section, we look into how to verify functional behavior of systems by modularly using the verified specifications of their components. In the ABS setting, the verification process can be done in two levels as illustrated by Fig. 3.1. On the first level, the ABS classes are verified against their specifications. The verification of component and system specifications is carried out in the second level. The class specifications are used as the units or primitives for component or system verification. Smaller component specifications that have been verified can also be used to verify larger components. The main benefit of this approach is that the verification of the system does not have to be done on the object level. Stated differently, in verification techniques that go down to object level for verifying the behavior of components (e.g., [4, 44]), all objects that are created during the execution of a component must be considered explicitly in the verification. The two-tier approach requires only the specification of the subcomponents to verify the behavior of components.

Applying this approach to verify the behavior of components and systems brings up challenges such as coming up with an appropriate notion of components and its semantics (in particular with respect to the unknown environment the components are placed in), identifying composition patterns, and building logics that correspond to the patterns. Through an analysis of a kernel language of ABS that focuses on the functional aspects of its behavior at the actor level (where a COG in ABS is taken as an actor), one way to construct a component is by keeping track of the actor creation tree [73]. Following the actor creation tree allows the use of class names to refer to the component instances for specifying their behavior. Traces of observable events are used as the semantics and specification basis because they capture fully abstract observable behavior of actors without needing state information [68, 24]. By observing how actors in components are created and interact with each other, several composition patterns are identified. It is not part of the goal to obtain completeness, however there should be sound logics that handle common patterns as this helps reduce the verification effort. For several patterns, we have developed a sound trace-based logic that achieves the goal by allowing internal interactions of actors of the components to be hidden [72].

Section Structure. The section starts with a short description of the kernel language derived from ABS and describes the changes to the operational semantics to include observable trace information, followed by the definition of closed and open systems or components and their respective trace semantics. Section 3.1.3 illustrates the Hoare-style specification technique. Section 3.1.4 presents the patterns and provides a sound trace-based logic that handles some of the patterns. In Section 3.1.5 we shortly review the related work.

3.1.1 Kernel Language and Trace Characterization

The work presented in this section is based on a kernel language derived from ABS which focuses on the higher concurrency level of actors. The kernel language features asynchronous method calls without futures, as integrating futures into the framework is still work in progress. The kernel language ignores ABS features that are orthogonal to the verification framework, including interface encapsulation, object groups, local
synchronous calls and product lines. With respect to object groups, from ABS the actor names can be seen as the group identities while an actor class is the set of classes that the owner of an object group (i.e., the initial object of the object group) uses to create its local objects. To simplify the semantics, both actor creations and asynchronous method calls are seen as statements, instead of expressions with side effects as in Deliverable 1.1a [34]. More precisely, actor creation is performed as an assignment to a variable \((v = \text{new } C(\tau))\) where \(C\) stands for the class name and \(\tau\) represents pure expressions that are passed on to the constructor of \(C\), and asynchronous method calls are of the form \(e!mtd(\tau)\) stating an asynchronous call of method \(mtd\) with parameters \(\tau\) to the actor referenced by \(e\).

As an example, Fig. 3.2 contains a variant of an industrial distributed database system investigated by Arts and Dam [9]. The server receives requests from the client, where each of these requests contain a query. The goal of the server system is to respond to the requests with the appropriate query computation results. To serve each request, the server creates a worker and pass on the query to be computed. The query may be divided into multiple subqueries or handled as a whole using some model function \(\text{compute}\). If the query can be divided into multiple subqueries (detected by using some model function \(\text{querySize}\)), more concurrency can be introduced in the following way. Before each worker processes the first subquery (extracted by \(\text{firstQuery}\)), it creates another worker to which the rest of the query is passed on. When the computation of the first subquery is finished, the worker merges (using function \(\text{merge}\)) the previous result with this computation result and propagates the merged result to the next worker. Eventually all chunks of the task are processed, and the last worker sends back the final result to the client.

```java
class Server() {
    Unit serve(Client c, Query q) {
        // querySize(q) ≥ 1
        Worker w = \text{new } Worker();
        w!do(q);
        w!propagateResult(null, c);
    }
}
class Worker() {
    Value myResult = null;
    Worker nextWorker = null;
    Unit do(Query q) {
        if (querySize(q) > 1) {
            nextWorker = \text{new } Worker();
            nextWorker!do(restQuery(q));
        } else {
            nextWorker = null;
        }
        myResult = compute(firstQuery(q));
    }
    Unit propagateResult(Value v, Client c) {
        await nextWorker != null;
        if (nextWorker == null) {
            c!response(merge(myResult, v));
        } else {
            nextWorker!propagateResult(merge(myResult, v), c);
        }
    }
}
```

Figure 3.2: Client-Server example
Operational Semantics

The operational semantics is defined by reduction rules on configurations. The configurations contain the code being executed, the heap with the instantiated actors and the trace of observable communication between actors in the system. The execution of an actor can be described by different communication events which represent the interaction between the actor and its environment. The traces over observable events then describe the observable behavior of a system [24, 68], see also Sect. 2.1.

In this section, we consider events where the sending and receiving of a message are considered as a single event. An adaptation to 4-event semantics [35, 15, 60] is relatively straightforward, but for the system-level verification, it is more important to focus on hiding actors that are not of interest.

Traces are represented using the sequence data structure \( \text{Seq}(T) \), with \( T \) denoting the type of the sequence elements. An empty sequence is denoted by \([\ ]\) and \( \cdot \) represents sequence concatenation. The function \( \text{Pref}(s) \) produces the set of all prefixes of a sequence \( s \). Assume the set of actor names \( a, b \in A \) and the set of messages \( m \in M \) that can be communicated between actors. The function \( \text{class}(a) \) returns the class of an actor \( a \). A message \( m \) can either be an actor creation \( \text{new } C(\overline{p}) \) or a method call \( \text{mtd}(\overline{p}) \). A parameter is a data value \( d \in D \) or an actor name.

The set of events \( E \) is built on the messages. An event \( e \in E \) represents the occurrence of a message \( m = \text{msg}(e) \) being sent by the caller actor \( a = \text{caller}(e) \) to the callee actor \( b = \text{callee}(e) \). If \( m \) is a creation message, \( b \) is the name of the created actor while \( a \) is its creator. An event \( e \) is represented by \( a \rightarrow b = \text{new } C(\overline{p}) \) or \( a \rightarrow b!\text{mtd}(\overline{p}) \) when the message is an actor creation or a method call, respectively. The inclusion of the caller information allows distinction between input and output events with respect to a single actor or a group of actors. The function \( \text{acq}(e) \), short for \( \text{acquaintance} \), collects the actor names occurring in the parameter list of a method call event. The caller name is transparent to the callee, so it is not part of the acquaintance. The function \( cr(e) \), short for \( \text{created} \), returns \( b \) when \( e \) is a creation event.

We take traces \( t \in \text{Seq}(E \cup \{\sqrt{}\}) \) to be a sequence of events. The \( \sqrt{\} \) symbol present at the end of \( t \) indicates that an execution has finished (i.e., no more operational rules can be applied). In other words, the execution is maximal (cf. [10, p. 96]). \( \sqrt{\} \) can only appear at most once in any trace as the last element of the sequence. Both \( acq \) and \( cr \) are lifted to the traces. To observe the local behavior of an actor we use the projection operator, denoted \( \downarrow e \), stating the projection of \( t \) to \( a \) where all events whose neither caller nor callee is \( a \) are removed from \( t \). When necessary, the actor parameter can be enriched with \( \text{callee} \) or \( \text{caller} \) to denote that we are focusing on the events where \( a \) is the callee or the caller, respectively. This operator is lifted to a set of traces \( T \) and a set of actors \( A \subseteq A \) in a natural way.

Following the approach in Deliverable 1.1a [34], a configuration is a multiset of actors, tasks and a representation of the trace. The binary operator \( \parallel \) denotes the multiset constructor that is \textit{associative} and \textit{commutative}. The reduction rules are interpreted modulo associative and commutative equations. An actor \( a \) is represented in the configuration with the form \( a[C,\sigma,l] \), denoting its class \( C \), the class parameter values \( \sigma \) and its lock \( l \). A lock \( l \) can be on, \( \top \), or off, \( \bot \). A task \( n \) has the form \( n(a,\sigma,s) \) where \( a \) is the actor to which \( n \) belongs, \( \sigma \) contains the local variables, and \( s \) is the statement that still needs to be executed. The trace \( t \) is represented by \( \text{tr}(t) \).

To describe the operational semantics, the syntax is enriched with two pieces of auxiliary syntax that appear only in the reduction rules:

\[
 s ::= \ldots | \text{grab}(a) | \text{release}(a) .
\]

As in ABS, these pieces allow scheduling control of tasks of an actor. The \( \text{grab}(a) \) statement turns a lock of an actor \( a \) on when it is unlocked, and \( \text{release}(a) \) does the opposite. In addition, we use the following helper functions. \( \text{eval}(e,\sigma,\sigma') \) takes an expression \( e \), the actor state \( \sigma \) and the local variables of a task \( \sigma' \) and computes the resulting value. This function is lifted to take multiple expressions. As a generic substitution mechanism, \( p[\overline{x}/\overline{r}] \) denotes the substitution of all (free) occurrences of variables \( \overline{x} \) by \( \overline{r} \). \( \text{body}(C,m) \) returns the statement body \( s(\overline{x}) \) of the method \( m \) in class \( C \), where \( \overline{x} \) are the local variables.

Figure 3.3 gives the reduction rules that involve the trace unit of the configuration. Rules \text{R-ACALL} and \text{R-NEW} allow the corresponding asynchronous calls and new actor creation to be appended to the
trace $t$, respectively. Changes to the local stores of both actors and tasks are denoted by $\sigma[x \mapsto \text{val}].$

Other reduction rules handling other aspects of the kernel language remain close to the corresponding rules defined in Deliverable 1.1a \[33\]. By using induction on the reduction rules, one can obtain the following well-formedness property of the generated traces.

**Definition 3.1.1** (Well-formed traces). Let $t$ be a trace and $e$ a place holder for a method call event $a \rightarrow b.m(\overline{p})$. A trace $t$ is well-formed if

\[
\forall a \in A, t' \cdot e \in \text{Pref}(t) \bullet \{b\} \cup \text{acq}(e) \subseteq \text{acq}(t_{vA,\text{caller}}) \cup \text{cr}(t_{vA,\text{caller}}).
\]

The definition above states that for every method call an actor makes, it must know the name of the actor it is calling and also the names of each actor present as parameters of the method call. Assuming there is no self-call, the well-formedness property implies that a trace $t$ begins with a creation event. From here on, we assume only well-formed traces.

**Example 10.** Let $c, s, w$ be Client, Server and Worker actors, respectively and $q$ be a query. Assuming $m$ is some Main actor that creates $c$ and $s$, the following (not yet finished) well-formed trace may occur from the client-server model in Fig. 3.2

\[
m \rightarrow s = \text{new Server}() \cdot m \rightarrow c = \text{new Client}(s) \cdot c \rightarrow s!\text{serve}(c, q) \cdot s \rightarrow w = \text{new Worker}() \cdot s \rightarrow w!\text{do}(q).
\]

### 3.1.2 Class and Component Characterization

Having an operational semantics for a group of actors is the basis to understand how a system behaves. One natural way to structure the grouping is to use classes to form components. Using the component notion, the system’s behavior can be understood by composing the behavior of the components. As the main interest in verifying the behavior of a system, the component notion used here refers to the run-time instances of the groups of actors. The classes are used as a convenient way to identify the behavior of the particular (groups of) actor instances and hide the actual actors that are instantiated. The following approach is derived from the characteristics of well-known component models, such as OSGi \[88\], CCM \[62\] and COM \[79\].

#### Closed System

To model a system in $\text{ABS}$, one needs to define a set of classes $C$. When $C$ contains all classes of each actor instance that appears during the execution of the system including the initial actor (the actor that represents the initial main block), this class set is called definition-complete. For example, the set of $\{\text{Client, Server, Worker, Main}\}$ is definition-complete, when in the definition of both Client and Main no other classes apart from Server are required. This notion represents the full knowledge of both the environment and the system itself. Thus, this kind of system is named closed system.

**Definition 3.1.2** (Closed system). A closed system $CS = (C, C_0)$ is a definition-complete set of classes $C$ with a distinguished activator class $C_0 \in C$. 


A closed system is described by having all class definitions needed to know precisely how each actor within the system behaves. An activator class is the class of the initial actor of the system. This initial actor should then create other actors necessary for the system to run.

**Definition 3.1.3** (Closed system trace semantics). The trace semantics of a closed system \( CS = (C, C_0) \) is the prefix-closed trace set \( Tr(CS) \) where for each \( t \in Tr(CS) \), there is a sequence of reduction rule applications starting from the initial configuration \( a(C_0, \sigma_{init}, T) \parallel n(a, \sigma_{main}, s_{main}) \parallel tr([]) \) that leads to a configuration containing \( tr(t) \).

The trace semantics of a closed system is a trace set which collects all traces that can be made from an initial configuration whose member is the initial configuration of the actor of activator class. Using the closed system definition, we can define the trace semantics of a class.

**Definition 3.1.4** (Class trace semantics). The trace semantics of a class \( C \) is a set of traces \( Tr(C) \) where for each trace \( t \in Tr(C) \), there is a closed system \( CS = (C, C_0) \) such that \( C \in C \) and there is a trace \( t' \in Tr(CS) \) such that \( t = t' \downarrow a \) for each actor \( a \) where \( class(a) = C \).

The trace semantics of a class \( C \) is obtained by taking all traces of all closed systems that contain \( C \) and then projecting all those traces down to actors of class \( C \). We denote this trace semantics as \([C]\).

**Example 11.** Taking the trace from example [10] as the base for projection to the Server actor \( s \), a trace of the Server class is \( m \rightarrow s = \text{new Server}() \cdot c \rightarrow s!\text{serve}(c, q) \cdot s \rightarrow w = \text{new Worker}() \cdot s \rightarrow w!\text{do}(q) \).

**Component**

A closed system deals with a group of actors that are completely executable without any influence from outside, while single classes only deal with the behavior of each of those actors. This shows a gap between single classes and closed systems because of the lack of possibility to know the behavior only for a subgroup of actors of that closed system. This gap is filled with an abstraction in terms of components. Components should share the characteristics of closed systems and single classes. Furthermore, they should also allow hiding behavior within the components.

The component abstraction chosen here is based on the actor creation tree. Translating this requirement in terms of sets of classes is done through the notion of creation-completeness. A set of classes \( C \) is creation-complete if for each actor creation \( \text{new } D \) appearing in the body of the method definition of any class \( C \in C \), \( D \) is also in \( C \). This notion allows the behavior of some actors whose references are passed on within the interaction to be unknown and thus fits with the goal of filling the gap between classes and closed systems.

**Definition 3.1.5** (Component). A component \( C = (C, C_0) \) is a creation-complete set of classes \( C \subseteq CL \) with some activator class \( C_0 \in C \).

A component is essentially a system that starts with an actor of some activator class. This notion of activator class comes from OSGi [58], where the component is instantiated through BundleActivator. The model where the component is instantiated by creating a single object of the activator class also coincides, for example, with the object adaptor of CORBA Component Model [62] and the class factory of COM [79]. Using the definition above, we can identify \((\{\text{Worker}\},\text{Worker})\) and \((\{\text{Server},\text{Worker}\},\text{Server})\) as components, where as \((\{\text{Server}\},\text{Server})\) is not.

To obtain the trace semantics of a component, the context in which the component is used has to be present. With this context, there is complete information on the interaction between all actors within the component (and also the context). Similar to a component, a context is also a set of classes with some activator class. However, this set of classes must be such that the class definition of the component is completed. In other words, the context provides the means to transform the component into a closed system. As a closed system, the trace semantics is clear. The main issue is to exclude parts of the traces where only actors of the environment are interacting. The function \( extAct(t, C) \) handles this issue by keeping track of actors created by the initial actor of the component of class \( C \) with respect to a trace \( t \). The trace
semantics of a component can be produced by projecting the traces of the closed system to the groups of actors produced by $extAct$.

**Definition 3.1.6** (Component plain trace semantics). Let $C = (C, C_0)$ be a component. The plain trace semantics of $C$ is a set of traces $Tr(C)$ where for each trace $t \in Tr(C)$, there is a context $X = (C^x, C_0^x)$ of $C$ such that in the resulting closed system $CS = (C^x \cup C, C_0^x)$, there is a trace in $t' \in Tr(CS)$ and $t = t', t_{\downarrow extAct}(t, C_0)$.

The term “plain” refers to the lack of hiding performed on the internal interaction between actors within the component. Implicit within the definition is the usage of actor creation tree to define the instances of components. The closed system used to obtain the traces is just one way to compose the component with the context. The activator class of the resulting closed system is taken from the context because it is the context which decides when the component is instantiated.

To finalize the abstraction between classes and closed systems, the internal behavior that happens between actors within the component should be hidden. Such a view allows bottom-up reuse of the components, for example, when the component is a library or a framework. Moreover, the view is suitable to deal with open systems, in particular systems with a non-software environment (e.g., GUI interaction with a human user). By using $extAct$, hiding is straightforward to define. Now we define the component trace semantics that hides the internal interaction between actors of the component.

**Definition 3.1.7** (Component trace semantics). Let $C = (C, C_0)$ be a component and $Tr(C)$ its plain trace semantics. The component trace semantics is the trace set $Tr([C_0]) = Tr(C)\downarrow_{A-\text{extAct}}(Tr(C), C_0)$.

To distinguish the component plain trace semantics from the one where hiding is performed, we use the notation $[C_0]$. This notation, read as boxed $C_0$, comes from the way we structure our component instances, where the component instance can be thought as a box of actors starting from the initial instance of the activator class $C_0$. By using the projection to the set of actors of the environment, only the observable behavior which captures the communication with the context is reflected. The trace semantics of $C$ is denoted by $[[C_0]]$.

**Example 12.** Taking the context where only a single Client actor is instantiated and it sends a single request to the server, $[[\text{Server}]]$ contains the following trace:

$$m \rightarrow s = \text{new Server}() \cdot c \rightarrow s!\text{serve}(c, q) \cdot w \rightarrow c!\text{response}(\text{compute}(q)) \cdot \sqrt{\cdot}.$$

### 3.1.3 Specification

Using the trace semantics of actor classes and component characterization based on the actor creation tree, their functional behavior can be specified in a trace-based manner. We use a specification format similar to the Hoare triple [67], where we relate input traces to output traces. That is, the specifications are of the form $\{p\} D \{q\}$, where $p$ and $q$ are trace assertions and $D$ is either a class or a component. Informally, this triple means that if an input trace of $D$ satisfies $p$, the output trace will satisfy $q$. Specifications that are given in terms of class invariants (see, e.g., the work of CTH [3] and UIO [21, 10]) can be transformed into such triples using counting and splitting of input and output events, provided the input and output events are not mutually dependent.

A trace assertion is a first-order logic formula in which the special trace constant $\$ can be used. In the input (output) condition, $\$ represents the caller suppressed input (output) trace. This suppression reflects the lack of knowledge on the receiver side (i.e., the entity being specified) of which actor the caller of a message is.

**Definition 3.1.8** (Trace assertions). Let $\$ be a trace constant representing a trace. Trace assertions $p, q$ are defined inductively by the following first-order logic clauses:

- Boolean expressions are assertions ($\$ may be present).
• If $p, q$ are assertions and $x$ is a variable, then $\neg p \land q$, $\exists x : p$ are also assertions.

The other logical operators, e.g., $\lor$, $\implies$ and $\forall$, are derived in the usual way. Logical variables will be represented by underlined serif characters. To define the semantics of a trace assertion, we substitute all occurrences of the trace constant with the actual trace. We assume that all variables and all substitutions are correctly typed. Using first-order logic, we then map the assertion to boolean values $\{\text{true}, \text{false}\}$. Given a trace $t$, we write $\models_{\text{FOL}} p[t/\$]$ if it is mapped to $\text{true}$, and $\not\models_{\text{FOL}} p$ if for any trace $t$, $\not\models_{\text{FOL}} p[t/\$]$. The FOL index indicates that the variable assignment is done using the first-order logic semantics.

**Example 13.** $\$ = $\langle$this$ = \text{new Worker()} \cdot \text{this!do(q) \cdot this!propagate(v,c)} \rangle$ is a trace assertion stating that the trace starts with a creation of a Worker actor, stored into the free variable this. Then, the worker is sent a task computation request followed by a result propagation. Only this sequence appears in the trace.

To define the triple semantics, a trace $t$ needs to be split into input and output traces, $ti$ and $to$ respectively, here represented by $\text{split}(t, L) = (ti, to)$. The function needs the set of actors $L$ that represents the entity $D$ at run-time.

In the case where the entity represented by $D$ is a class $C$, $L$ is taken to be a singleton actor $a$ whose class is $C$. We call $\{p\} C \{q\}$ a class triple. The split function ensures that only the interaction done by $a$ appears in the input and output traces that are being considered in the semantics of the specification triple.

**Definition 3.1.9** (Class triple semantics). Let $C$ be a class, $\llbracket C \rrbracket$ its semantics, and $a$ an actor such that $\text{class}(a) = C$. $\llbracket C \rrbracket$ satisfies $\{p\} C \{q\}$, written $\models_{\text{FOL}} \{p\} C \{q\}$, if for all maximal traces $t \in \llbracket C \rrbracket$ with $\text{split}(t, \{a\}) = (ti, to)$ the following holds:

$$\models_{\text{FOL}} p[ti/\$] \implies q[to/\$].$$

**Example 14.** The following specification of the server class states that when a server is created and a request comes, the server creates a new worker, gives the worker the query and tells it to start propagating the result.

\[
\{ S = \langle \text{this} = \text{new Worker()} \cdot \text{this!do(q) \cdot this!propagate(v,c)} \rangle \} \text{ Server}
\]

When $D$ is a component represented by its activator class $\llbracket C \rrbracket$, the set of actors $L$ of the run-time component needs to be extracted from the semantics. Because only one creation event of the initial actor is present in any trace of $\llbracket C \rrbracket$, the set of actors of the component that lies on the boundary of that run-time component can be extracted. This set is represented using the function $\text{boundary}(T)$.

**Definition 3.1.10** (Component triple semantics). Let $C$ represent a component, $\llbracket C \rrbracket$ its semantics and $B = \text{boundary}(\llbracket C \rrbracket)$ the set of actors on the boundary of the component. $\llbracket C \rrbracket$ satisfies $\{p\} C \{q\}$, written $\models_{\text{FOL}} \{p\} C \{q\}$, if for all maximal traces $t \in \llbracket C \rrbracket$ with $\text{split}(t, B) = (ti, to)$ the following holds:

$$\models_{\text{FOL}} p[ti/\$] \implies q[to/\$].$$

**Example 15.** As a component, the worker replies to the client by merging the partial result passed on to the component with the computation of the remaining query as a whole. This property can be specified as follows.

\[
\{ S = \langle \text{this} = \text{new Worker()} \cdot \text{this!do(q) \cdot this!propagate(v,c)} \rangle \} \text{ Worker}
\]

The input part states that a new Worker actor is created, then a query comes followed by a request to propagate the computation result. When the input part is satisfied, the worker component produces a response back to the client by computing the whole remaining task and merging it with the given partial result.

These triples have a partial semantics. To obtain a more precise description of the behavior of an actor class or a component, multiple triples can be used. A trace set representing the actual behavior of the class or the component must satisfy every triple.

**Definition 3.1.11** (Specifications). Let $D$ be a class or a component. A specification for $D$ is a set of specification triples $S = \{\{p_1\} D \{q_1\}, \ldots, \{p_n\} D \{q_n\}\}$. $\llbracket D \rrbracket$ satisfies $S$, written $\models S$, if

$$\forall \{p_i\} D \{q_i\} \in S \models \{p_i\} D \{q_i\}.$$
3.1.4 Composition Patterns

How actors are composed depends mainly on their reference exposure. The exposure can occur in two ways: by creating another actor so that the creator knows the reference of the newly created actor; and through parameters of method calls. That is, they are exactly the two types of events as described above. The composition also depends on how the communication proceeds, i.e., how the actors interact with each other.

In Fig. 3.4 we identify several composition patterns by using these two criteria. The figure uses the boxed capital characters A, B, C to denote an actor of class A, B or C and a dashed box to denote the boundary of the composed component. For simplicity we assume an actor of class A to be the initial actor of the component. In the logic, B can actually also be a component. The solid arrows indicate method calls, whereas the dashed arrows show which actors are exposed outside of the boundary. The exposure allows the context of the component to call the exposed actors. We do not show the exposure of actors within the boundary. All other actors within the boundary are created transitively by the initial actor. In the figure, the actors are created in the direction of the method call (except for 3.4(e) where B is created by A).

There are six patterns that have been identified and not in any way are these patterns representing an exhaustive list of all possible patterns. The simplest pattern is the sink pattern (a) which represents a passive composition, such as logging, where there is no feedback given to the context. Pattern (b) is a fixed pipelining composition, similar to the way how processors process the machine instructions. Pattern (c) extends pattern (b) by allowing a number of actors to be created in the same way, depending on the parameter given into the component. A typical example of this pattern is a generator, such as a prime number generator that applies Sieve of Erastothenes. The Worker component also falls into this pattern. The next pattern shows a typical case of a basic client-server setting, where each request coming to the server is delegated to newly created workers. Pattern (e), similar to publisher-subscriber pattern, allows a more complicated client-server setting where the worker may communicate back to the server for more information. The last pattern in the figure allows input to come not only from the initial actor of the component, but also from its subcomponent actors. That is, other actors of the component are exposed to the context. In all these patterns, the context has to expose some references of their own to obtain feedback from the component.

Oftentimes systems are composed in such a way that these patterns are combined. For example, the main component of an industrial distributed database system [9] combines patterns (c), (d) and (e). We have handled a variation of this database system that involves pattern (c) by developing a logic that also serves as an example of the verification framework [72].

Parameterized Pipelining

The logic presented in Fig. 3.5 consists of a few inference rules. Each premise can be a trace assertion, which is applied to any trace using first-order logic semantics, or a specification triple. A proof is a tree of inference rule applications. Each node is a (possibly empty) premise and each edge is a rule application. The root of
the proof is the main goal: a specification triple. The leaves are either valid trace assertions or class triples

to be verified against the implementation.

**Boxing** transforms a class triple into a component triple, when the output trace assertion states that

no actor is created (represented by the predicate *nonCr*) by the instance of that class. This coincides with

the component trace semantics definition, because already the class traces that satisfy this specification

guarantee that no actor creation is observed.

**BoxedComposition** composes a class triple with a subcomponent triple. Suppose the triple of class *C*

is restricted enough to guarantee that its instance only creates a subcomponent, and the output of the instance

as a whole becomes the input of the instance of the subcomponent. Then, we can infer the component triple

of *C* by taking the input trace assertion of the original class triple and the output trace assertion of the

subcomponent triple. In other words, the instance of *C* encapsulates the subcomponent.

More precisely, first, the class triple *C* must guarantee that the actor’s name will not be exposed. The

predicate *noSelfExp*, short for *no self exposure*, takes variable *j* representing the input trace. It guarantees a

one way interaction because the current actor is not exposed by checking that the acquaintance of the output

trace does not contain that actor. Second, the triple of *D* must ensure that no foreign actor is created by

the instance of *D* as represented by the predicate *nonCr*. Third, the output produced by the actor of class *C*

must match the input of the instance of *D*. In other words, the actor of class *C* exclusively feeds the

instance of *D* in this particular case. This matching is handled by predicate **match**.

\[
\text{match}(q, q', D) \equiv q \implies \exists a \in A \cdot \text{firstCreated}(a) \land \text{classOf}(a, D) \land q'
\]

The predicate **firstCreated** checks if the first event is an actor creation and *a* represents the created actor.
The predicate **classOf** checks if the created actor is of class *D*. The **match** predicate relies on the fact that the

trace set begins with an actor creation. For **match** to hold, the free variables of *q* and *q’* should coincide.

The **Induction** rule deals with the case where multiple actors of the same class are created to handle the

same input, with some parameter reducing to some base case, as indicated by the measure variable *m*. In a way, this is similar to making a recursive call inside an actor. The main difference is that due to the

concurrent nature of the actors, each input parameter may be processed independently by each created actor, possibly optimizing the computation.

In addition to the inference rules above, the logic also includes Hoare logic rules such as **Consequence**, **Invariance** and **Substitution**. The inference rule **Consequence** allows the input trace assertion to be weakened and the output trace assertion to be strengthened. **Invariance** allows a predicate containing no trace constant to strengthen both input and output trace assertions of a triple. **Substitution** allows a free variable to be substituted by some predicate or expression not containing the trace constant. We can show that the logic above is sound as the proposition below states.

**Proposition 3.1.12** ([72]). The logic in Fig. 8.3 is sound with respect to the specification semantics.

To show how the logic is applied, we verify the following simple specification of the server component

stating: A request from the client is replied by a response to the client with the computed result.

\[
\{ \$ = \langle \text{this} = \text{new Server()} \cdot \text{this\_serve}(c, q) \rangle \} \text{Server} \{ \$ = \langle \text{response}(\text{compute}(q)) \rangle \}
\]
are substituted by

\[ \text{INV} \]
\[
\{ \text{ISrv} \} \hspace{1em} \text{Server} \setminus \{3w \cdot \text{OSrv} \}
\]
\[
\{ \text{ISrv} \land w = \text{cse}(\text{ISrv}) \} \hspace{1em} \text{Server} \setminus \{3w \cdot \text{OSrv} \land w = \text{cse}(\text{ISrv}) \}
\]
\[
\exists w \cdot \text{OSrv} \land w = \text{cse}(\text{ISrv}) \implies \exists w \cdot \text{OSrv} \land \text{noSelExp}(j)
\]
\[
\{ \text{ISrv} \land w = \$ \} \hspace{1em} \text{Server} \setminus \{3w \cdot \text{OSrv} \land \text{noSelExp}(j) \}
\]

\[ \text{Cons} \]
\[
\exists w \cdot \text{OSrv} \land w = \text{cse}(\text{ISrv}) \implies \exists w \cdot \text{OSrv} \land \text{noSelExp}(j)
\]
\[
\{ \text{ISrv} \land w = \$ \} \hspace{1em} \text{Server} \setminus \{3w \cdot \text{OSrv} \land \text{noSelExp}(j) \}
\]

\[ \text{Comp} \]
\[
\text{match} \{3w \cdot \text{OSrv} \land \text{OWrkC}\} \hspace{1em} \text{Server} \setminus \{3w \cdot \text{OSrv} \land \text{OWrkC} \land \text{null} \}
\]
\[
\{ \text{ISrv} \} \land \text{OSrv} \hspace{1em} \text{OWrkC}\}
\]

Figure 3.6: Server component proof tree

Figure 3.6 provides a proof tree how the server component triple above can be verified using the logic. In the proof tree, the following abbreviations represent the trace assertions of the specification triples.

- \( \text{ISrv} \equiv \text{ISrvC} \equiv \$ = (\text{this} = \text{new Server}() \cdot \text{this!serve}(c, q)) \)
- \( \text{OSrv} \equiv \$ = (w = \text{new Worker}() \cdot w!\text{do}(q) \cdot w!\text{propagate}(\text{null}, c)) \)
- \( \text{OWrkC} \equiv \$ = (\text{this} = \text{new Worker}() \cdot \text{this!do}(q) \cdot \text{this!propagate}(v, c)) \)
- \( \text{OWrkC} \equiv \$ = (c!\text{response}(\text{merge}(v, \text{compute}(q)))) \)
- \( \text{OSrvC} \equiv \$ = (c!\text{response}(\text{compute}(q))) \)

The abbreviations are chosen such that \( \text{ISrv} \), for example, represents the input event content equality of the server class triple, whereas \( \text{OWrkC} \) represents the output event content equality of the worker component triple. The \( C \) suffix indicates the assertion is used in a component triple. We also introduce the function \( \text{cse} \), short for content sequence extractor, to extract the event content sequences from these abbreviations.

To achieve the inference of the server component specification, we work backwards until the server class and worker component specifications are obtained. The suitable rule for this inference is [boxedcomposition]BoxedComposition|The input to the server component is handled fully by the server actor, while the output of the server actor is captured completely by the worker component instance. In order to match the output trace assertion of the server class triple, the partial result variable in the input trace assertion of the worker component instance has to be initialized to \( \text{null} \).

As both triples left as proof obligation in the proof tree above are not of the triples assumed to be valid, they must be transformed. The left subproof tree shows how the server class triple can be obtained. We need to store the input trace and transfer it to the output trace assertion of the triple, in order to determine whether a reference to the created server actor is not exposed. In this example, the content sequence extractor function is used to get the suppressed input trace that we need. Note that the input trace assertions of the server class and server component triples are the same.

The transformation for the worker component triple is straightforward to obtain. All occurrences of variable \( v \) are substituted by \( \text{null} \) and the \( \text{nonCr} \) predicate is trivially implied by the component definition. Thus, we have completed the inference proof of the server component triple.

The worker component triple can be verified using the [induction]Induction rule. A complete account how the worker component is verified from the worker class triple can be found in [72].

Other Patterns

The work on a logic that treats other patterns is ongoing. To handle multiple requests (and also patterns where the initial actor creates multiple subcomponents of different initial classes), [boxedcomposition]BoxedComposition|can be extended to include more triples of the subcomponents. The difficulty lies in combining the output of these subcomponents and the initial actor to imply the output of the component.

These patterns are for communication without futures. More patterns emerge when futures as a first-class entity, as featured in ABS, are present, because futures that can be passed around may expose references at a later time than when the future is actually resolved. Nevertheless, the patterns presented above are still applicable given some adaptation to the lessened exposure of the context to the component.
3.1.5 Related Work

Misra and Chandy [80], Soundarajan [92] and Widom et al. [97] proposed proof methods handling network of concurrent processes using traces. Soundarajan related invariants on process histories to the axiomatic semantics of a parallel programming language. Widom et al. discussed the necessity of having prefix-closed trace semantics and partial ordering between messages of different channels to reach a complete proof system. They deal only with closed systems of fixed finite processes (and channels) and, because of their generality, make no use of the guarantees and restrictions of the actor model.

Ahrendt and Dylla [4] and Din et al. [44, 46] extended Soundarajan’s work to deal with actor systems. They consider only finite prefix-closed traces, justifying it by having only a finite number of actors to consider in the verification process. Din et al. particularly verified whether an implementation of a class satisfies its triples by transforming the implementation in a simpler sequential language, applying the transformational method proposed by Olderog and Apt [87]. The main difference to the approach described in this section is on the notion of component that hides a group of actors into a single entity. It avoids starting from the class specifications of each actor belonging to a component when verifying a property of the component.

3.1.6 Conclusion

In this section, we have constructed the framework of two-tier verification of actor-based (or COG-based) component systems. Through analysis of the way new actors are created, we are able to obtain an abstraction level of run-time components. Combined with the trace-based semantics extracted from the programming language, a specification technique that refers only to the class names is developed. This leads to the construction of a sound logic that handles specific patterns of interactions between the run-time components that are based on the class specifications. A logical next step is to investigate more patterns and introduce futures into the trace-based semantics in a similar way as done in [44, 38].

3.2 Verification of Data Flow Policies

This section presents a framework for monitoring and enforcing policies concerning direct data flows. In relation to our previous work, [25, 26] and similar work by others, [65, 66, 42], our technique can be positioned somewhere between control flow monitoring and information flow monitoring. It is similar to control flow monitoring in the sense that the framework is capable of monitoring the order of observable actions, but rather than enforcing temporal policies such as “f may be evaluated after g but not vice versa”, the framework handles policies such as “f may be applied to the result of g but not vice versa”. It is also similar to information flow monitoring in the sense that the flow of data plays a central role. It should however be made clear that the focus is not on arbitrary information flow but rather on direct and explicit flows.

The work presented in this section is a continuation of the work presented in D4.1, [37]. Before presenting the prototype implementation and the case studies (which have not been presented in earlier deliverables) we provide a brief background of the theory.

3.2.1 Theoretical Formalization

We start by introducing the theoretical background required to understand how the prototype works and how the policies in the case studies are expressed.

Observable Actions

The observable actions, i.e. the actions performed by the target program which are observed (and possibly rejected) by the execution monitor, are in this framework defined to be the external function calls. This is a common choice in runtime monitoring, c.f. [66, 42]. What is novel in our case is that the framework not only takes the function identifier and arguments into account, but also the full history of function calls (referred to as the computational history) that were performed to compute the arguments. This means that the
// Each time a socket is connected, it is given a taint unique for its host.
Socket.connect(String host) = fresh(host)

// The taint is copied to its input/output stream.
Socket.getInputStream(Socket this: A) = A
Socket.getOutputStream(Socket this: A) = A

// The taint is copied to any data read from such input/output stream.
InputStream.read(InputStream this: A) = A

// Any data written to a (tainted) output stream must either be untainted or
// have the same taint as the output stream. (The result is always untainted.)
OutputStream.write(OutputStream this: A, int x : B) = (  
    A = 0 -> 0  
    B = 0 -> 0  
    A = B -> 0  
  )

Figure 3.7: Policy stating that data from one host may not be sent to another host. (Presented in the syntax accepted by the prototype implementation.)

framework is able to enforce a policy stating for instance that \( f \) may be applied to \( g(x) \) but not to \( h(x) \) even though \( g \) and \( h \) are extensionally equal. A more realistic example would be a policy stating that \( \text{sanitize} \) accepts any string as argument, while the function \( \text{runQuery} \) only accepts strings returned by \( \text{sanitize} \).

Efficient Implementation

Each observable action corresponds to a tree of function applications. These trees fully capture the computational history of the values observed in the execution. From a theoretical point of view, these trees are convenient, but in practice they are intractable to maintain during runtime. Instead of the full tree of function applications, we use abstractions referred to as labels. Each label represents a (possibly infinite) set of concrete function application trees which all share some security relevant property.

Policies and Enforcement

A policy is formalized as a predicate over the set of sequences of observable actions. In other words, a policy either accepts or rejects an execution based on the observable actions. In the research presented here we focus on local and subtree closed policies. A local policy decides if a given sequence of observable actions is accepted or not by inspecting each action in isolation. For a (local) subtree closed policy, the set of observable actions is also subtree closed, which means that a policy can not for instance accept \( f(g()) \) but reject \( g() \). The first property relieves the framework from dealing with global execution monitor state. A solution, which could easily be adapted to this framework, for dealing with this type of monitor state has been presented in our earlier work, [25, 26]. The second property allows us to provide results regarding completeness.

Furthermore we present the syntax and semantics of a language for describing policies. An example policy, which makes sure that the data received from one host are not allowed to be sent to another host is presented in Figure 3.7. The semantics is conveniently defined in terms of tree automata over function application trees (i.e., trees constructed from function \( F \) nodes and constant \( C \) leaves ). The states \( Q \) of the automaton correspond to the set of labels and each clause in the policy is translated into a set of automata moves \( \Delta \). The automaton corresponding to the policy in Figure 3.7 is shown in Figure 3.8. As can be seen, it propagates the label of a socket to its input and output streams (\( \Delta_1 \) and \( \Delta_2 \)) and ensures that the label of the result of a read operation complies with the label of the input stream (\( \Delta_3 \)), and that the label of an
\[ Q = \mathbb{N} \cup \{ \text{unit} \} \]
\[ F = \mathbb{C} \cup \mathbb{F} \]
\[ Q_f = Q \]
\[ \Delta = \{ c \rightarrow \text{unit}(c) \mid c \in \mathbb{C} \} \]
\[ \cup \{ \text{Socket}.\text{connect}(q(x)) \rightarrow \text{fresh}(x)(\text{Socket}.\text{connect}(x)) \mid q \in Q \} \]
\[ \cup \{ \text{Socket}.\text{getInputStream}(q(x)) \rightarrow q(\text{Socket}.\text{getInputStream}(x)) \mid q \in Q \} \quad (\Delta_1) \]
\[ \cup \{ \text{Socket}.\text{getOutputStream}(q(x)) \rightarrow q(\text{Socket}.\text{getOutputStream}(x)) \mid q \in Q \} \quad (\Delta_2) \]
\[ \cup \{ \text{InputStream}.\text{read}(q(x)) \rightarrow q(\text{InputStream}.\text{read}(x)) \mid q \in Q \} \quad (\Delta_3) \]
\[ \cup \{ \text{OutputStream}.\text{write}(q(x),q'(y)) \rightarrow \text{unit}(\text{OutputStream}.\text{write}(x,y)) \mid q, q' \in Q \land (q = \text{unit} \lor q' = \text{unit} \lor q = q') \} \quad (\Delta_4) \]

Figure 3.8: Tree automaton corresponding to the syntax in Figure 3.7.

A policy defined in this language is by construction both local and subtree closed.

3.2.2 Practical Evaluation and Case Studies

We have developed a prototype implementation of the framework targeting Java bytecode (thus the tool applies to compiled ABS programs as well) which is based on taint analysis. Taint analysis (also known as taint tracking) is a common technique for tracking data flows in runtime. The technique relies on (1) having points at which data is originally tainted (taint sources), (2) making sure that taints propagate along with every data flow (taint propagation) and (3) having points at which taints of output data is intercepted (taint sinks). In our work we rely on taint propagation for tracking data flows by letting taints represent labels. The notion of taint sources and sinks however are factored out and handled by program rewriting. To make sure that the return values are tainted according to the policy, our program rewriter adds code after each method invocation to explicitly set the taint according to the policy.

In our implementation we have chosen to use a framework called TaintDroid by William Enck et al. which targets the Android Platform. The TaintDroid framework is based on a modified version of the Dalvik VM which taints data coming from various privacy related sources such as the GPS, camera, microphone etc. and monitors the taints of data being sent on the network. As mentioned above we will utilize the taint propagation mechanism for tracking data flows but ignore the built-in taint sources and sinks. The program rewriter is a modified version of our ConSpec monitor inliner used in our previous work.

The remainder of this section describes the case study applications and policies and concludes by discussing the results of the evaluation.

Case Study 1: DroidLocator

Just as the popular application Find My Phone for iPhone, the DroidLocator application allows the user to locate a lost or stolen Android device through a web service. As opposed to Find My Phone and other similar services however, DroidLocator prevents server administrators and third parties from using the location data maliciously. It does so by encrypting the location data, based on a user-provided key, before uploading it to the server. When the user later retrieves the encrypted location data, he or she can decrypt it without revealing the location to anyone else.
Application  DroidLocator is a small application written for the purpose of this case study. It retrieves the location data from the GPS hardware, uses the `javax.crypto` package to encrypt it with a key retrieved through `EditText.getText`, and submits it to the server using the standard socket API.

Policy  The desired policy states that (A) the location may not be sent over the network unless it is encrypted and that (B) the key used in the encryption must be provided by the user (retrieved through `EditText.getText` on an object with no prior invocations of `EditText.setText`). Figure 3.9 shows the policy expressed in mathematical notation: Each clause consists of a method (and formal parameters for the labels of the arguments) and a guarded command which determines the label of the result. Examples of function application trees accepted and rejected by this policy are found in Figure 3.10.

The set of labels corresponding to this policy is: \{0^{16}1^{16} (L), 10^{15}1^{16} (sock), 010^{14}1^{16} (conf), 0010^{13}1^{16} \}

Figure 3.10: Accepted (top) and rejected (bottom) FATs for the DroidLocator policy.
(nonuser), $0^{17}1^{15}$ (userenc), $0^{16}1^{14}$ (userinp)), and the bitwise-or operator is used for $\oplus$.

To ensure security, the label specifying that data contain location information is encoded using an \textit{or}-flag (a bit-flag propagated through an \textit{or}-operator) and the label specifying that data are encrypted is encoded using an \textit{and}-flag (a bit-flag propagated through an \textit{and}-operator).

**Results**  The behavior was unaffected by monitor inlining since the original application adhered to the policy. When the code was changed to use as encryption key a predefined string literal (such as, in Figure 3.10, the empty string), the execution was terminated before the location was uploaded.

**Case Study 2: Sms2Group**

An application, \textit{Sms2Group}, requiring the SEND_SMS permission, allowing users to send SMS-messages to groups of contacts, is studied. The policy in this study restricts which numbers messages may be sent to.

**Application**  \textit{Sms2Group} has been developed for the purpose of this study. The application allows the user to automate the task of sending text messages to a group of contacts. It relies on the group attribute in the contact book, fetched using the content provider API and uses the ordinary \textit{SmsManager.sendTextMessage} method to send SMSes.

**Policy**  Messages are prevented from being sent to arbitrary numbers by ensuring that destination numbers (first argument of \textit{sendTextMessage}) originate from the local address book. This general policy naturally separates legitimate executions from malicious ones. Using traditional inlining techniques this type of policy would be expressed using a guard that scans the address book and checks that the destination number is present. There are two conceptual differences between these approaches. As opposed to a policy that relies on scanning the address book, our policy expresses that an SMS may not be sent to numbers with arbitrary origin \textit{even} if the number is present in the address book. In this sense our policy is stricter. Another difference is that a scan of the address book is typically a linear operation, whereas checking the taint of a value is a constant time operation.

**Results**  The inlining did not affect the functionality of the original program as it adheres to the policy. When changing the code so that the program attempts to send an SMS to a hard-coded number or a number from the address book concatenated with an arbitrary string, the policy is violated and the program is terminated as expected.

**Case Study 3: Bankdroid**

This case study examines an internet banking application, \textit{Bankdroid}, which allows users to review account information from several different banks. The application has many security concerns as the information it handles (balances, recent transactions, etc.) is usually considered confidential. The main objective of the case study is to demonstrate how standard security policies can be applied \textit{transparently} on real world honest applications, while still blocking dishonest variants of the same applications.

**Application**  \textit{Bankdroid} (40k lines of code) is distributed through Google Play and currently installed on 100,000+ devices \[61\]. It uses the Apache HttpClient library to communicate with the banks. To allow the policy to be expressed at the level of sockets (instead of at the level of the Apache HttpClient API), the library has been included in the client code base which adds another 60k lines of code.

**Policy**  The policy is a Chinese-Wall like policy which states that data received from host $A$ may be sent back to host $A$ but not to some other host $B$. As mentioned in the above paragraph, the policy is expressed at the level of sockets which makes it general and applicable to many other applications requiring Internet access. Examples of accepted and rejected function application trees are found in Figure 3.11.
Results The application was modified to leak the current balance of each bank account to a host controlled by a potential attacker. The policy was then inlined in the modified application. When the leak was about to take place, the inlined code successfully terminated the execution.

Case Study 4: Auto Birthday SMS

*Auto Birthday SMS* is a popular application distributed on Google Play which allows the user to automatically send SMS-messages to friends on their birthdays. It is free of charge and displays ads which are retrieved over the network. It requires the *internet* and *send_sms* permissions. Applications requiring this combination of permissions are interesting to study since trojans sending premium-rate SMS messages are relatively common [53] and can potentially transform the phone into an SMS spamming bot. As demonstrated in this case study, TreeDroid is useful even for honest coders in order to harden their applications by inlining generic security policies.

Application Application data, including numbers to send messages to, are stored in a SQLite database. The code turns out to be vulnerable to SQL-injection attacks which can be exploited by any application with permission to modify the address book data. The code calls *SQLiteDatabase.execSQL*, which updates the database, with an unsanitized query containing the name of a contact. The contact name should be sanitized by *DatabaseUtil.sqlEscapeString* before running the query.

Policy The policy applied is a general sanitize-before-query policy stating that a query passed to *execSQL* must be a string literal, a result of *sqlEscapeString* or a concatenation of such strings. The label used for sanitized values is encoded using an *and*-flag to ensure that the concatenation of sanitized and unsanitized strings are considered sanitized. An example of an accepted function application tree is found in Figure 3.12. Omitting the call to *sqlEscapeString* would result in a tree which would be rejected by the policy.

Results The inlined code prevents the application from performing queries containing unsanitized arguments, such as raw contact names, in the SQL statements. The original application violates the policy upon certain user actions, in such cases execution is successfully terminated by the monitor.
Figure 3.12: Accepted FAT for the SQL policy.

Case study: Lovetrap

Lovetrap is a real world SMS-trojan detected by Symantec in July 2011 [93]. Among other bad behaviors, it sends premium rate SMS messages (which is the focus of this study). This case study demonstrates the efficiency of TreeDroid on real world attacks.

Application  Lovetrap, which looks like a regular game, starts a service which downloads a list of numbers and messages which it repeatedly tries to send by SMS.

Policy  The policy from case study 1 is reused without modification, which is an indication of the policy’s genericity.

Results  By locally redirecting requests going to the host of the attacker to our own server, we managed to supervise the actions of the trojan. The monitor inlining at the bytecode level proceeds as expected without special tweaking. After inlining, the trojan’s service is terminated immediately and therefore no longer able to send SMS messages as intended.

Statistics

Case studies statistics have been collected in Table 3.13. For applications where we have access to the source code, business logic execution time has been measured. In Bankdroid we measured the time it takes to update
the accounts, for DroidLocator we measured the time it takes to encrypt and upload the location, and for Sms2Group we measured the time it takes to collect the group information and send the SMS. Taint tracking runtime overhead has been estimated by TaintDroids authors to about 14% on a Google Nexus One [51]. Our measurements (significantly higher, as expected since they are performed using Dalvik in debug mode on an Android emulator) are included for comparison with the runtime overhead due to the inlined code. The bytecode size overhead in the Auto Birthday SMS study is due to the fact that the relatively common operation of concatenating strings is considered policy relevant.
Chapter 4

Deadlock Analysis

The ABS methodology enables the design of systems that support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In this scenario, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. In other words, the correct termination of each of the two process activities depends on the termination of the other. Since there is a circular dependency, termination is not possible.

In ABS so-called futures are used to manage return values from asynchronous calls. Futures can be accessed by means of either a get or an await primitive: the first one blocks the object until the return value is available, while the second one is not blocking as the control is released. The combination of blocking and non-blocking mechanisms to access futures may give rise to complex deadlock situations which require a rigorous formal analysis. In our work we give two different notions of deadlock for systems based on active objects and futures. One is based on blocked objects and conforms with the classical definition of deadlock by Coffman Jr. et al. [50]. The other one is an extended notion of deadlock based on blocked processes which is more general than the classical one.

We start by considering the most popular definition of deadlock that goes back to an example titled deadly embrace given by Dijkstra [43] and the formalization and generalization of this example given by Coffman Jr. et al. [50]. Their characterization describes a deadlock as a situation in a program execution where different processes block each other by denial of resources while at the same time requesting resources. Such a deadlock can not be resolved by the program itself and keeps the involved processes from making any progress.

A more general characterization by Holt [69] focuses on the processes and not on the resources. According to Holt a process is deadlocked if it is blocked forever. This characterization subsumes Coffman Jr.’s definition. A process waiting for a resource held by another process in the circle will be blocked forever. In addition to these deadlocks Holt’s definition also covers deadlocks due to infinite waiting for messages that do not arrive or conditions, e.g., on the state of an object, that are never fulfilled.

We now explain our notions of deadlock by means of an example. Consider two objects $o1$ and $o2$ belonging to classes $C1$ and $C2$, respectively, with $C1$ defining methods $m1$ and $m3$ and $C2$ defining method $m2$. Moreover suppose method $m1$ of object $o1$ to be the method executed when the program starts. Such methods are defined as follows (for simplicity type declarations are omitted):

```java
m1() {x1 = o2!m2(); x1.get; return;}
m2() {x2 = o1!m3(); x2.get; return;}
m3() {return;}
```

The variables $x1$ and $x2$ are futures, accessed (in this case) with the blocking get statement. This program clearly creates a deadlock because the execution of $m1$ blocks the object $o1$ and the execution of $m2$ blocks
the object o2. In particular, the call to m3 cannot proceed because the object o1 is being blocked by m1 waiting on its get. We call classical deadlocks these cases in which there are groups of objects such that each object in the group is blocked by a get on a future related to a call to another object in the group.

Consider now the case in which the method m2 is defined as follows:

\[
\text{m2() \{x2 = o1!m3(); \texttt{await x2?; x2.get; return;}\}}
\]

In this case, object o2 is not blocked because m2 releases the control by performing an await instead of (just) a get. Nevertheless, the process executing m2 will remain blocked forever. We call extended deadlock this case of deadlock at the level of processes.

In [31] we have formalized the notions of classical and extended deadlock, and we have proven that the latter includes the former. Moreover, as the main technical contribution, we have shown a way for proving extended deadlock freedom. The idea is to consider an abstract semantics of programs expressed in terms of Petri nets which are analyzed for reachability of a distinct marking representing deadlocks. This will be the content of the following Section 4.1.

We also considered another approach for detecting extended deadlock freedom, which while being less precise (in that, as we will discuss, the analysis is not really carried out at the process level, but at the object level instead, thus producing false positives), it is based on a more efficient static analysis technique, that we also implemented as part of the ABS toolchain: abstract descriptions of method behaviors are extracted in the form of behavioral types and circular dependency among objects are then searched in the states of the obtained transition system. This will be the content of the following Section 4.2.

4.1 Petri Net Based Approach

The idea is to consider an abstract semantics of programs expressed in terms of Petri nets. In order to reduce to Petri nets, we abstract away several details, in particular, we represent futures as quadruples composed of the invoking object, the invoking method, the invoked object, and the invoked method. For instance, the future x1 in the example presented in the previous section is abstractly represented by o1.m1@o2.m2.

Notice that a plain usage of this abstraction (without any special treatment) is not satisfactory: in the abstract semantics a process could access a wrong future simply because it has the same abstract name. Consider, for instance, the following example (where we make the same assumptions as those in the example presented previously):

\[
\text{m1() \{x1 = o2!m2(1); x2 = o2!m2(2); x2.get; await x1?; x1.get; return;\}}
\]

\[
\text{m2(x1) \{ if \{ x1 == 1 \} then return; else \{ x2 = o1!m3(); await x2?; x2.get; return;\}} \}\}
\]

\[
\text{m3() \{return;\}}
\]

Both the futures x1 and x2 of method m1 will be represented by the same abstract name o1.m1@o2.m2. For this reason, even if this program originates a deadlock when get is performed on x2 by m1, according to the abstract semantics the system could not deadlock. In fact, the return value of the first call could unblock the get as the two futures have the same name in the abstract semantics. To overcome this limitation, we add in the abstract semantics marked versions of the methods: when a method m is invoked, the abstract semantics nondeterministically selects either the standard version of m or its marked version denoted with m?. Both method versions have the same behavior, but the return value will be stored in two futures with two distinct abstract names. For instance, in the example above, if we consider that the first call to m2 actually activates the standard version m2 while the second one activates the marked version m2?, there will be no confusion between the two futures as their abstract names will be o1.m1@o2.m2 and o1.m1@o2.m2?, respectively. In this case, the system will deadlock also under the abstract semantics.

In general, in the abstract semantics, we can non-deterministically choose to use a marked future: this models the fact that the call is assumed to be part of a deadlock. Technically, to apply this idea, for every method code, we move all internal choices up front and for each choice we make different assumptions (all
possible ones) on which future (at most one) is to be marked. More precisely, we perform a transformation of the initial program, which, for every method body, yields a set of abstract trace executions. During such a transformation we also make a data abstraction, remove superfluous internal steps, such as multiple get and await primitives on the same futures, and duplicated choices from the program, to reduce the size of the Petri net. For the technical details of the transformation we refer the reader to [31]. This transformation can add spurious deadlocks but it cannot remove them because the new abstract model is an overapproximation of the original system. We also assume methods of objects belonging to the same component object group to be all methods of a common single object, so that locks are associated simply to such objects and not to groups.

The Petri net based abstract semantics allows us to obtain a decidable way for proving extended deadlock freedom. In fact, reachability problems are decidable in Petri nets, and we are able to reduce extended deadlock detection to a reachability problem in the abstract Petri net semantics: reachability of particular Petri net configurations corresponding to deadlock scenarios.

We now describe the basic ideas of the Petri net construction.

### 4.1.1 Rescheduling

The translation of conditional rescheduling, originating from the await statement, deviates from the ABS behaviour. In particular, differently from ABS, the object lock is always released upon reaching the await statement. The statement await x? is translated to the sequence release(o); get@o’.m*; grab(o), where release(o) and grab(o) represent acquire and release of a lock on the object o executing the await statement and get@o’.m*, where o’.m* is either o’.m or o’.m?, represents a get on the future x (i.e., x.get) identified either by o.l@o’.m or o.l@o’.m? respectively, where o and l are the object and the method executing the await statement. In the ABS case, instead, the lock is only released if the result that the process is waiting for is not available. In case the result is available the process continues its execution without rescheduling.

This deviation is motivated by the preliminary transformation that we make on the program and the asynchronous nature of the communication. In the case of ABS semantics the rules have to cover both the first await/get of a result and the subsequent await/get. In case of a subsequent result the await statement has to be executed without rescheduling since the existence of the result has been proven by the previous await. In the abstract semantics, consecutive claims have been removed, i.e., each await in the abstract case is the first await of the result. Moreover, due to the asynchronous nature of the communication, we the delivery of results can be delayed.

### 4.1.2 Places and Tokens

The Petri net obtained from a program contains two kinds of places (see Figure [4.1]):

**Locks.** Places identifying the locks of objects. Each object has its designated lock place labeled by the unique name g of the object. A token in such a place represents the lock of the corresponding object being available. There is at most one token in such a place.

**Process.** Places identifying a particular process in execution or the future as a result of the execution of a process. These places are labeled with l(t) where l is an abstract label identifying the future related to the call that originated the process execution and t is the abstract method code (trace) still to be executed. A token in this place represents one instance of such a process in execution or a future (if t is empty). In case of a future, the token is consumed if the future is consumed via await/get.

### 4.1.3 Transitions

The transitions of the Petri net are determined by the translation of the semantic steps of the operational semantics of a program, which are applied to obtain the transitions for the places labeled with l(t) representing processes in execution. Formally, we give the translation for the individual execution steps according to the operational semantics of the program (see [31]).
Initial Transitions. A program execution starts from an initial configuration composed of a set of classes, a set of objects and an initial process. This is the main method in the program, which is not called by any other process. The selection of the abstract trace of the main method to be executed (among those obtained during the main method transformation) is done by the initial transition depicted in Figure 4.2. To do this we have included an auxiliary place \textit{start}. This place will be the initial place of the Petri net.

Method Calls. We present the Petri net transitions for a method call in Figure 4.3. As we already explained, a process place in the Petri net is labeled with a tuple \(o1.l@o2.l2\). We abbreviate parts of the label by \(c@o.l\) resp., \(o.l@c\) or the whole label by \(l\) if details are not needed. In Figure 4.3, we represent method calls for both marked and unmarked futures. The method body of the called process \(t'\) is in both cases of the form \(\text{grab}(o); t_{o,l}; \text{release}(o)\), where \(t_{o,l}\) is an abstract trace execution of the method \(l\) of the object \(o\). This means that, in general, several (alternative) transitions are generated for a method call: a transition for each trace \(t_{o,l}\).
Lock Handling. To execute the \texttt{grab}(o) statement, the object lock of object \(o\) must be available. When releasing the lock of an object \(o\) by \texttt{release}(o), a token is added to the place representing the object lock (Figure 4.4).

Waiting for Results. We present the Petri net transitions for waiting for the result of a method call in Figure 4.5. The notations "\(o.l^+\)" or "\(o.l^*\)" denote that \(o.l\) can be marked or not: formally, + and * are metavariables that can be either the empty string or ?. As it was explained before, to avoid the token confusion of sequential calls, the tokens are consumed. Notice that removing the result is not problematic with respect to multiple claims of a value because subsequent claims are removed in the preliminary transformation.

Example. In Figure 4.6 we show a fragment of the Petri net obtained for the extended deadlock example considered at the end of the introductory text of the current chapter about deadlock analysis. Concerning labels for places, only significant ones are shown, and, we have used the following abbreviations. Concerning the caller part of labels we have that: \(c^{main}\) is \(o^{main}.main\), i.e. the initial \textit{main} method executed by an initial fictitious \(o^{main}\) object, \(c^1\) is \(o1.m1?\) and \(c^2\) is \(o2.m2?\). Concerning the code part of labels we have that:

- \(t1\) is \texttt{grab(o1);} \texttt{o2!m2?;} \texttt{get@o2.m2?;} \texttt{release(o1)},
- \(t2\) is \texttt{grab(o2);} \texttt{o1!m3?;} \texttt{release(o2);} \texttt{get@o1.m3?;} \texttt{grab(o2);} \texttt{release(o2)} and
- \(t3\) is \texttt{grab(o1);} \texttt{release(o1)}.

Notice that the Petri net in Figure 4.6 is a fragment for the following reasons: (i) The call by the main method to the place labeled by \(c^{main}@o1.m1?(t1)\), which here is shown as already having received a token, is not represented (and also the execution of the main method itself from the initial place “start” of the Petri...
Figure 4.6: Translation of the example on extended deadlock.

net, giving rise to two abstract trace executions, one calling m1 without the mark o and one calling m1 with the mark o, the latter giving rise to the subnet represented in Figure 4.6. (ii) Such a call and the call originating the transition leading to the place labeled by $c^1[@o.m2?]@t_2$ also give rise to other transitions leading to places labeled by $c^mmain@o.m1?@t_1'$ and $c^1[@o.m2?]@t_2'$, respectively, where $t_1'$ is like $t_1$ apart from $m2$ not being marked with $o$ and, similarly, $t_2'$ is like $t_2$ apart from $m3$ not being marked with $o$. From such places replicas of the corresponding subnets are then produced with a different combination for the marks.

A deadlock is detected for this net in that the following marking is reachable (see the definition of deadlocked markings in [31]). Tokens are only in the following places: the place labeled by $cmain@o1.m1?$ and one calling $c1@o2.m3?$ (i.e. $m1$ is blocked on a get for the future identified by $o2.m2?$ and $m2$ is blocked on a get for the future identified by $o1.m3?$) and, finally, the place labeled by $c^2@o1.m3?@t_3$ (i.e. $m3$ is blocked on the grab for the lock on object $o1$ which is possessed by $m1$).

4.2 Contract Based Approach

In this section we describe the theoretical framework for deadlock analysis for ABS presented in [56, 57, 58, 59, 54].

Deadlocks may be particularly insidious to detect in systems where the basic communication operation is asynchronous (e.g. ABS asynchronous method invocation) and the synchronization explicitly occurs when the value is strictly needed (e.g. ABS get operation). In this context, when a thread running within the object group $x$ performs a get operation on a thread within the object group $y$, then it blocks every other thread that is competing for the lock on $x$. This blocking situation corresponds to a dependency pair $(x, y)$, meaning that the progress on $x$ is possible provided the progress of threads on $y$. A deadlock then corresponds to a circular dependency in some configuration, such as a collection of pairs of the form $(x, x_1), (x_1, x_2), \ldots, (x_n, x)$.

Similarly, being the await operation a synchronization as well, it introduces a dependency between the current group $x$ and the awaited group $y$ of a slightly different kind, noted $(x, y)^w$. In this case, in fact, the current object while waiting it releases the lock instead of keeping it. The semantics of await requires the task to compete again for the lock with other tasks in the same object, and then try again for the result. If it is available the computation proceeds, otherwise the lock is released again and so on. In case the converse dependency $(y, x)$ holds at the same time, then $y$ results to be blocked waiting for $x$. Releasing the lock on $x$ does not change the fact that tasks in $y$ are blocked. The circular dependency still holds, however the system is not completely blocked since there is a task caught in an infinite loop of getting and releasing the lock (a situation that we call livelock).

To illustrate these concurrency features of ABS, we discuss three different implementations of the factorial
function in an hypothetical class `Math`, declared in Figure 4.7, which we are going to use as running example throughout the chapter. The function `factG` is the standard definition of factorial: the recursive invocation `this!factG(n-1)` is post-fixed by a `get` operation that retrieves the value returned by the invocation. Yet, `get` does not release the lock of the caller object; therefore the task evaluating `this!factG(n-1)` is fated to be delayed forever because its object is the same as the one of the caller. The function `factAG` solves this problem by permitting the caller to release the lock with an explicit `await` operation, before getting the actual value with `x.get`. An alternative solution is displayed by the function `factNC`, whose code is very similar to that of `factG`. Yet, in case of `factNC`, the recursive invocation is `z!factNC(n-1)`, where `z` is an object in a new group. This means that the task of `z!factNC(n-1)` may start without waiting for the release of any lock by the caller.

Further difficulties arise in the presence of infinite (mutual) recursion: consider, for instance, systems that create an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and really hard to predict. In addition, deadlocks may not be detected during testing, and, even if they are, it can be difficult to reproduce them and find their causes.

Our deadlock detection framework consists of an inference algorithm that extracts abstract behavioral descriptions out of the concrete program. These abstract descriptions, called `contracts`, retain necessary information for the deadlock analysis (typically all the synchronization information’s are extracted, while data values are ignored).

Contracts represent the input for the analysis algorithm that produces a finite state model, called `deadLock Analysis Model` (or `lam`), whose states contain dependency pairs. If a circular dependency is found in one of these states, then the original program may contain a deadlock.

To overview our analyzer, we observe that, in presence of recursion in the code, the evaluation of the abstract description may end up in an infinite sequence of states, without giving back any answer. As we hinted before, the main challenge is due to the generation of an unbounded number of new names, which must be dealt with statically in a finite state model, so that our analysis always terminates.

This issue was first addressed adopting a saturation technique [57]: we fix a finite number of new names that can be created while evaluating the contract, in order to force the termination by guaranteeing always
a fix-point. The saturation is encountered when all the available new names have been assigned, and, from that moment on, every creation will be assigned the same name. This gives a sound but imprecise solution: if the model is deadlock-free then the ABS program is deadlock-free, but the analysis may find false positives. This approach is also reported in the Deliverable D2.7 [10].

We have then developed another technique that allow us to determine when to stop the evaluation. Informally, when the abstract program is linear – it has (mutual) recursions of the kind of the factorial function – then states reached after some point are going to be equivalent to past states. That point, called saturated state, may be determined in a similar way as the orbit of a permutation [22] (actually, our theory builds on a generalization of the theory of permutations [59]). A saturated state represents the end of a pattern that is going to repeat indefinitely. Therefore it is useless to analyze it again and, if a deadlock has not been encountered up to that point, then it cannot be produced afterwards. Analogously, if a deadlock has been encountered, then a similar deadlock must be present each time the same pattern recurs. When the abstract program is nonlinear – it has (mutual) recursions of the kind of the Fibonacci function – our technique is not precise because it introduces fake dependency pairs. However, it is sound.

A prototype of our framework (following the fix-point approach) has been implemented and tested on large commercial codes, such as the FAS module of the Fredhopper Case Study [33]. For a description of the tool we refer to the Deliverable D5.4 [11] and to [59]. A prototype implementing the approach based on mutations is under development.

In Section 4.2.1 we present the contract inference system. In Section 4.2.2 we describe the fix-point-based approach we adopted for the deadlock analysis. In Section 4.2.3 we describe the decision algorithm produced by the second approach we adopted for the deadlock analysis.

### 4.2.1 Contracts and the Contract Inference System

In order to analyze ABS models, we use abstract descriptions called contracts. The syntax of these descriptions uses variables X, Y, Z, · · ·, and groups a, b, · · ·, and it is defined as follows:

\[
\begin{align*}
\text{c} &::= \text{r} \mid \text{a} \mid \text{a} \rightarrow \text{r} \\
\text{r} &::= \_ \mid X \mid a[f : r] \mid a \rightarrow \text{r}
\end{align*}
\]

#### Future records

r may be a record name X, which represents a variable that may be possibly instantiated by substitutions. The future record a[f : r] defines the group name of the object and the future records of values stored in the fields. The future record a → r specifies that, in order to access to r, one has to acquire the control of the group a (and to release this control once the method has been evaluated). Future records as a associated to method invocations: the object of group a is the receiver object of the invoked method. Alternatively, a contract may also indicate the (type of the) method invocation. The void future record _ is associated to primitive values, that is to values that are not objects.

Contracts c collect the method invocations inside expressions and the group name dependencies. A contract may be empty, noted 0, which expresses that the method behavior is irrelevant for our analysis; or just \((a, a')\) (resp. \((a, a')^\wedge\)) when the dependency is due to a get (resp. an await) operation on a field or an argument of the method. Alternatively, a contract may also indicate the (type of the) method invocation. There are several possibilities:

- \(\text{C.m}(\text{r}) \rightarrow \text{r'}\) (resp. \(\text{C.m}(\text{r}) \rightarrow^s \text{r'}\)) specifies that the method m of class C is going to be invoked asynchronously (resp. synchronously) on an object r, with arguments \(\bar{r}\), and an object r' will be returned;

- \(\text{C.m}(\text{r}) \rightarrow \text{r'}(a, a')\) indicates that the current method execution requires the termination of method C.m running on an object of group a' to release the object with group a. This is the contract of an asynchronous method invocation followed by a get operation on the same future reference.

- \(\text{C.m}(\text{r}) \rightarrow \text{r'}(a, a')^\wedge\), indicating that the current method execution requires the termination of method C.m running on an object of group a' in order to progress. This is the contract of an asynchronous method invocation followed by an await operation, and, possibly but not necessarily, by a get operation. In
The contract $c_1 + c_2$ defines the abstract behavior of (expressions with side-effects in) sequences. The contract $c + c'$ defines the abstract behavior of branching and it is associated to conditionals. For example:

- the term `Math.factG a[\_](\_)\{Maths.factG a[\_](\_) \rightarrow \_.(a,a)\}` \_ is the method contract of method `factG` of class `Maths`. The name `a` in the header refers to the object group name associated to this in the code, and binds the occurrences of `a` in the contract. The contract body shows a recursive invocation to `factG` performed on an object in the same group `a` and followed by a `get` operation, which introduces a dependency pair `(a,a)`. The record \_ corresponds to the parameter and return value, both of type `Int`.

- the term `Math.factAG a[\_](\_)\{Maths.factAG a[\_](\_) \rightarrow \_.(a,a)w\}` \_ is the method contract of `factAG` of class `Maths`. It is similar to the one above, but the presence of an `await` statement in the method body produces a dependency pair `(a,a)w`. The subsequent `get` operation does not introduce any dependency pair, due to the fact that its success is guaranteed, provided the success of the `await`.

- the term `Math.factNC a[\_](\_)\{Maths.factNC b[\_](\_) \rightarrow \_.(a,b)\}` \_ is the method contract of `factNC` of class `Maths`. This method contract differs from the previous ones in that the receiver of the recursive invocation is a free name (i.e., it is not bound by the name `a` in the header). This reflects the fact that the recursive invocation is performed on an object belonging to a newly created group (i.e., different from `a`). The dependency pair due to the `get` operation relates the current group `a` with the new group `b`.

For the inference rules and technical details on the inference system, see [57, 55].

4.2.2 The Analysis

Contracts are inputs to our deadlock analysis technique. The technique returns finite state models, called `lam` (an acronym for `deadLock Analysis Models` [57, 55]), where states are relations on group names.

Lams are illustrated as ovals, representing states; every oval contains pairs of group names. Figure 4.8(i) displays a two-states lam where the object pairs in the states are respectively `{(a,b)}` and `{(a,b), (b,c)}` (the brackets `{...}` are always omitted in the figures). Figure 4.8(ii) displays a one state lam where the relation is `{(a,b)w}`.

The algorithm computing lams takes as input a contract class-table and a main contract to evaluate. These (abstract) descriptions are the output of our inference algorithm. Then, the standard Knaster-Tarski technique is run on the model. The critical issue of this technique is that it may create pairs on fresh names at each step, technically, at every approximant, because of free names in method contracts that correspond to new cogs. As a consequence, the lam model is not a complete partial order (the ascending chains of lams may have infinite length and no upper bound) A paradigmatic example is the model of the recursive method contract of `factNC`. In order to circumvent this issue and to get a decision about deadlock-freedom in a finite number of steps, we use another usual method: running the Knaster-Tarski technique.
up-to a fixed approximant, let us say $n$, and then using a saturation argument. If the $n$-th approximant is not a fix-point, then the $(n+1)$-st approximant is computed by reusing the same object names used by the $n$-th approximant (no additional object name is created anymore). Similarly for the $(n+2)$-nd approximant till a fix-point is reached (by straightforward cardinality arguments, the fix-point does exist, in this case). For example, in the case of the above contract, the $n$-th approximant returns a single state lam containing the relation $\{(a_1, a_2), \ldots, (a_{n-1}, a_n)\}$ whilst the $n+1$-st approximant returns the relation $\{(a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a_n)\}$. This last relation contains a circular dependency – the pair $(a_n, a_n)$ –, which means that the corresponding program may display a deadlock. In this case, this circularity is a false positive that is introduced by the (over)approximation: the original code never manifests a deadlock. A more detailed account of the algorithm may be found in [57].

4.2.3 An Alternative Approach

In [59], we have developed a theoretical framework for defining relations on names (every pair of a relation has meant that the dependency pair is generated by an unrelated function; the original code never manifests a deadlock. A more detailed account of the algorithm may be found in [59].

Lams are defined by terms that use a set of names, ranged over by $x, y, z, \ldots$, and a disjoint set of function names, ranged over $m, m', n, n', \ldots$. A lam program is a tuple $(m_1(\bar{x}_1) = L_1, \ldots, m_\ell(\bar{x}_\ell) = L_\ell, L)$ where $m_i(\bar{x}_i) = L_i$ are function definitions and $L$ is the main lam. The syntax of $L_i$ and $L$ is

$$L ::= 0 \mid (x, y) \mid m(\bar{x}) \mid L | L$$

such that (i) all function names occurring in $L_i$ and $L$ are defined, and (ii) the arity of function invocations matches that of the corresponding function definition.

It is possible to associate a lam function to each method of an ABS program. The purpose of the association is to abstract the object group dependencies that a method will generate out of its definition. For instance, the $\text{factNC}$ method declaration has the associated function declaration in a lam program

$$\text{factNC}(x) = (x, y)\|\text{factNC}(y)$$

where the (first) argument is the name of the object group of $\textit{this}$, which is $x$. In general, in a function $m(x, y, z)$, $x$ refers to the receiver in the original program and $y$ and $z$ refer to the two parameters. The $\text{get}$ operation on a method invocation is recorded in this setting by the fact that the dependency pair and the abstract method invocation are in parallel in the same state (where $\|$ is interpreted as the set union of $(x, y)$ and the dependencies produced by invoking $\text{factNC}(y)$). For instance, if we had $(x, y)\|\text{factNC}(y)$, this would have meant that the dependency pair is generated by an unrelated $\text{get}$ operation.

For example, in the program $(m(x, y, z) = (x, y)|m(y, z, x), m(u, v, w))$, the function $m$ produces a dependency $(x, y)$ and a recursive invocation of $m$; the term $m(u, v, w)$ is the main term. The unfolding of the invocations of $m$ gives the sequence of states

$$m(u, v, w) \rightarrow (u, v)|m(v, w, u) \rightarrow (u, v)|(v, w)|m(w, u, v) \rightarrow (u, v)|(v, w)|(w, u)|m(u, v, w)$$

corresponding to the relations $0$ (initial state), $(u, v)$ (second state), $(u, v)|(v, w)$ (third state), and $(u, v)|(v, w)$ (fourth state).

Because lams feature dynamic name creation – e.g. the function $\text{factNC}(x)$ above, since $y$ is free, creates a new name at every unfolding –, the underlying model is not finite state. Our main contribution is the answer to the

**Problem:** Is it decidable whether the computations of a lam program will ever produce a circularity?

The answer is immediate when the functions are not recursive: it is sufficient to unfold the invocations in the term to evaluate. The answer is also not tough when
(i) functions are linear, that is (mutual) recursions are of the kind of the factorial function, such as the above function \( m \);

(ii) function invocations do not have duplicate arguments and function definitions do not create new names.

When (i) and (ii) hold, recursive functions may be considered as permutations of names – technically we define a notion of associated (per)mutation – and the corresponding theory [22] guarantees that, by repeatedly applying a same permutation to a tuple of names, at some point, one obtains the initial tuple. This point, which is known as the orbit of the permutation, allows one to define the following algorithm for the problem: compute the orbit of the permutation associated to the function in the lam and correspondingly unfold the term to evaluate. For example, the permutation of the above function \( m \) has orbit 3. Therefore, it is possible to stop the evaluation of \( m \) after the third unfolding (at the state \( (u, v)\|((v, w)\|((w, u)\|m(u, v, w)) \)) because every dependency pair produced afterwards will belong to the relation \( (u, v)\|((v, w)\|((w, u)) \).

When the constraint (ii) is dropped, the answer to the problem is not simple anymore. However, the above analogy with permutations has been a source of inspiration for us. Consider the function \( h(x, y, z) = (y, z)\|h(x, x, w) \) and evaluate \( h(u, v, w) \). We obtain

\[
\begin{align*}
\h(u, v, w) & \rightarrow (v, w)\|\h(u, u, w') \rightarrow (v, w)\|((u, w')\|h(u, u, w'') \rightarrow (v, w)\|((u, w')\|((u, w'')\|h(u, u, w''')).
\end{align*}
\]

That is, the evaluation will never return \( h(u, v, w) \), as well as any other invocation in the states, because the definition of \( h \) has a recursive invocation where the first argument is duplicated, the second and third arguments are erased, and a new name is created. Nevertheless, we notice that, from the third state onwards, the invocations of \( h \) are not identical, but may be identified by a map that associates names created in the last step to past names, and it is the identity on other names. The definition of this map, called flashback, requires that the transformation associated to a lam function, called mutation, also records the name creation. For instance, the theory of mutations allow us to map \( h(u, u, w') \) back to \( h(u, u, w') \) by recording that \( w'' \) has been created after \( w' \), e.g. \( w' < w'' \). Analogously \( w'' < w''' \). We generalize the result about permutation orbits:

By repeatedly applying the same mutation to a tuple of names, at some point we obtain a tuple that is identical, up-to a flashback, to a tuple in the past.

As for permutations, this tuple is the orbit of the mutation, which (we prove) it is possible to compute in similar ways.

Linearity of lam programs is a basic concern in our algorithm for determining a saturated state. When lam programs are nonlinear, that is we are also dropping the above item (i), functions have several associated mutations. A paradigmatic example of nonlinear function is when recursions are of the kind of Fibonacci:

\[
(fib(x) = (x, y)\|(x, z)\|fib(y)\|fib(z), fib(u)).
\]

When programs are nonlinear, the generalization of our technique, if any, should require to take into account nondeterminism – a function transforms a tuple into a set of tuples –, which is out of the scope of this contribution. Alternatively, we propose a simpler, but imprecise, technique. Namely, transforming nonlinear lam programs into linear ones by introducing fake dependencies (i.e., false positives in terms of circularities). For example, the \( fib \) lam program is transformed into a linear one (this is a simplified version):

\[
\text{( \quad fib}_{a}^\text{aux}(x, x') = (x, y)\|(x, z)\|(x', y)\|(x', z)\|fib_{a}^\text{aux}(y, z), \quad fib_{a}^\text{aux}(u, u) \quad ) .
\]

To highlight the fake dependencies added by \( fib_{a}^\text{aux} \), we notice that, after two unfoldings, \( fib_{a}^\text{aux}(u, u) \) gives

\[
(u, v)\|((u, w)\|((v, v')\|((w, v')\|((w, v'')\|((w, w'')\|fib(v')\|fib(v'')\|fib(w'')\|fib(w'') \).
\]

On the contrary \( fib \) has a corresponding state (obtained after four steps)

\[
(u, v)\|((u, w)\|((v, v')\|((v, v'')\|((w, w'')\|fib(v')\|fib(v'')\|fib(w'')\|fib(w'') \),
which has no dependency between names created by different invocations. It is worth to remark that these additional dependencies cannot be completely eliminated because of a cardinality argument. The evaluation of a function invocation \texttt{m(\texttt{u})} in a linear program may produce at most one invocation of \texttt{m}, while an invocation of \texttt{m(\texttt{u})} in a nonlinear program may produce two or more. In turn, these invocations of \texttt{m} may create names. When this happens, the creations of different invocations must be \textit{contracted} to names created by one invocation and explicit dependencies must be \textit{added} to account for dependencies of each invocation.

Nevertheless, we prove the soundness of our technique: if the transformed linear program is circularity-free then the original nonlinear one is also circularity-free. In particular, since our analysis lets us determine that the saturated state of \texttt{fibaux} is circularity-free, then we are able to state the same property for \texttt{fib}.

It is possible to apply our technique to verify the deadlock-freeness of programs written in object-oriented languages. In the survey [58], we discuss a number of Java(-like) programs that (may) manifest deadlocks. We informally define an association programs/lams, and derive deadlock-freeness properties of the programs from the circularity-freeness of the associated lams. The technical details, properties and proofs are available in [59].

4.3 Conclusion

In this chapter we presented two techniques based on behavioural types and Petri net translation to detect deadlock in systems made of asynchronously communicating active objects where futures are used to handle return values which can be retrieved via a lock detaining get primitive or a lock releasing await primitive (followed by a get). For both methods our analysis is sound with respect to extended deadlocks (which encompasses also blocked processes in addition to blocked objects considered in the classical notion of deadlock), i.e., if the analysis does not detect any deadlock then we are guaranteed that the original system is deadlock free.

Concerning the other direction, we claim our Petri net based technique to be complete apart from false positives due to the abstraction from data values, i.e., the transformation of “if” primitives into non-deterministic choices (which obviously leads to new behavioral possibilities, hence deadlocks, with respect to the original system).

We now remark on the comparison between the contract based and the Petri net based analysis and future work.

The contract based analysis in Section 4.2 yields transition systems whose states are labeled with sets of dependencies: pairs of objects representing an invocation from an object to another one. The system is deadlock free if no circular dependency is found. The Petri net analysis in Section 4.1 is more precise in that it is process based and not just object based. An example of a false positive detected by the contract based approach follows.

Consider the program consisting of two objects \texttt{o1} and \texttt{o2} belonging to classes \texttt{C1} and \texttt{C2}, respectively, with \texttt{C1} defining methods \texttt{m1} and \texttt{m3} and \texttt{C2} defining method \texttt{m2}. Moreover suppose method \texttt{m1} of object \texttt{o1} to be the method executed when the program starts. Such methods are defined as follows (for simplicity type declarations are omitted):

\begin{verbatim}
m1() {x1 = o2!m2(); await x1?; x1.get; return;}
m2() {x2 = o1!m3(); x2.get; return;}
m3() {return;}
\end{verbatim}

This program would cause a deadlock if we had (just) a get instead of an await in method \texttt{m1}. This is because method \texttt{m1} would call method \texttt{m2}, which in turn would call \texttt{m3} which would not be able to proceed because the lock on object \texttt{o1} would be kept by \texttt{m1} waiting on the get. Differently from the contract based analysis, the Petri net based analysis correctly detects that the system is deadlock free in that method \texttt{m1} is waiting on an await instead of a get.
Concerning language expressivity, contract based analysis additionally considers, with respect to Petri net based analysis, a “new” primitive for object creation and the capability of accounting for (a finite set of) objects used as values (e.g., passed as parameters or stored in fields) in the analysis. Concerning the former, if the theoretical extension described in Section 4.2.3 is not considered, only objects within a finite set of object names can be created (if invocations to the “new” primitive exceed the amount of available object names, as in the case of recursive object creation, old objects are returned), thus such a primitive can be easily encoded in the Petri net based approach by considering all the objects in the set of object names to be present from the beginning (and then “activated”). Concerning the latter, the language abstraction considered in the Petri net based analysis can be quite easily extended by considering objects, out of a finite set, passed to methods (by considering object names as part of the method name). Dealing with objects stored in fields would however require an extension of the encoding into the Petri net, where a different place is considered for each possible object to be stored. We plan to do such extensions and to prove our claim about completeness as a future work.
Relational reasoning provides an effective mean to understand program behavior: in particular, it allows to establish that the same program behaves similarly on two different runs, or that two programs execute in a related fashion. Prime examples of relational properties include notions of simulation and observational equivalence, and 2-properties, such as non-interference and continuity. In the former, the property considers two programs, possibly written in different languages and having different notions of states, and establishes a relationship between their execution traces, whereas in the latter only one program is considered, and the relationship considers two executions of that program.

In spite of its important role, and of the wide range of properties it covers, there is a lack of applicable program logics and tools for relational reasoning. Indeed, existing logics are confined to reasoning about 2-properties of structurally equal programs, and are not implemented. This is in sharp contrast with the more traditional program logics for which robust tool support is available. Thus, one natural approach to bring relational verification to a status similar to standard verification is to devise methods that soundly transform relational verification tasks into standard ones. More specifically for specifications expressed using pre and post-conditions, one would aim at developing methods to transform relational judgments of the form \( \{ \varphi \} c_1 \sim c_2 \{ \psi \} \), where \( \varphi \) and \( \psi \) are relations on the states of the command \( c_1 \) and the states of the command \( c_2 \) (the semantics of relational judgments is discussed below), into Hoare triples of the form \( \{ \tilde{\varphi} \} c \{ \tilde{\psi} \} \), where \( \tilde{\varphi} \) and \( \tilde{\psi} \) are predicates on the states of the command \( c \), and such that the validity of the Hoare triple entails the validity of the original Hoare quadruple; using \( \models \) to denote validity, the goal is to find \( c, \tilde{\varphi} \) and \( \tilde{\psi} \) such that

\[
\models \{ \tilde{\varphi} \} c \{ \tilde{\psi} \} \quad \Rightarrow \quad \models \{ \varphi \} c_1 \sim c_2 \{ \psi \}.
\]

This chapter explores automated constructions for transforming relational verification tasks into standard verification tasks, and their applications. We consider two scenarios, closely related to the notions of evolvability and adaptability that are central to the HATS methodology. In the first scenario, a reference implementation is customized to take maximum advantage of a target execution platform; in this scenario, the goal is to prove semantic equivalence between the optimized implementation and the reference implementation; this property is formalized as a 2-property: for all input memories that are related by the pre-condition, the output memories are related by the post-condition. In the second scenario, we focus on refinement properties, and consider properties of the form: for all initial and final memories of the first program, there exists an initial and final memory of the second program, such that the initial memories are related by the precondition and the final memories are related by the postcondition.

We provide automated constructions that address the needs of both scenarios, and illustrate the applications of our method to several examples from the literature and one specific scenario of great practical interest: auto-vectorization of standard libraries.
\[
\begin{align*}
\vdash \{\phi\} \text{skip} \{\phi\} & \quad \text{(Skip)} \\
\vdash \{\phi[e/x]\} x := e \{\phi\} & \quad \text{(ASSIGN)} \\
\vdash \{\varphi\} \ c \{\phi\} & \quad \vdash \{\phi\} \ d \{\psi\} & \quad \text{(SEQ)} \\
\vdash \{\varphi \land b\} \ c \{\psi\} & \quad \vdash \{\varphi \land \neg b\} \ d \{\psi\} & \quad \text{(IF)} \\
\vdash \{\varphi\} \ \text{if} \ b \ \text{then} \ c \ \text{else} \ d \{\psi\} & \quad \vdash \{\phi \land b\} \ c \{\phi\} & \quad \text{(WHILE)} \\
\vdash \{\phi\} \ \text{while} \ b \ \text{do} \ c \ \{\phi \land \neg b\} & \quad \vdash \{\varphi\} \ c \{\psi\} \quad \varphi' \Rightarrow \varphi \quad \psi \Rightarrow \psi' & \quad \text{(SUB)} \\
\end{align*}
\]

Figure 5.1: Hoare logic

5.1 Preliminary: Relational Hoare logic vs Hoare Logic

There is a striking similarity between the proof rules of Hoare logic and those of relational Hoare logic. Yet it appears that no formal connection has been established between their derivations. This section presents a remarkably simple embedding from derivations in relational Hoare logics to derivations in Hoare logics. The embedding relies on the ability to construct for any two programs \(c_1\) and \(c_2\) s.t. \(\vdash \{\varphi\} c_1 \sim c_2 \{\psi\}\) their synchronized product program \(c\) which performs the execution steps of \(c_1\) and \(c_2\) in lockstep and verifies \(\vdash \{\varphi\} c \{\psi\}\). In preparation for the general definition of product, we further provide a more liberal construction allowing products to perform execution steps of the programs \(c_1\) and \(c_2\) asynchronously.

5.1.1 A review of (Relational) Hoare logic

For illustrative purposes, we consider a simple imperative language with conditional and loops. We let \(x\) range over a set \(V\) of variables, \(e\) and \(b\) range over a set of integer and boolean expressions (\(AExp\) and \(BExp\) respectively); commands and basic blocks are defined by the following grammar:

- **basic blocks**
  
  \[i ::= x := e \mid \text{skip} \mid i; i\]

- **commands**
  
  \[c ::= i \mid c; c \mid \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \mid \text{while} \ b \ \text{do} \ c\]

States are partial functions from variables to integer values, i.e. \(S = V \rightarrow \mathbb{Z}\). The semantics of a command \(c\) is modeled by a relation \([c] \subseteq S \times S\), defined in the usual way from the semantics of expressions, and are given by \([e]_{e \in AExp} : S \rightarrow \mathbb{Z}\), and \([b]_{b \in BExp} \subseteq S\).

**Hoare Logic** Program correctness is expressed by triples of the form \(\{\varphi\} c \{\psi\}\), where \(c\) is a command, and \(\varphi\) and \(\psi\) are first-order formulae with variables in \(V\). We let \([\varnothing]\) denote the set of states satisfying \(\varnothing\).

**Definition 5.1.1** (Valid triple). \(\{\varphi\} c \{\psi\}\) is valid, written \(\vdash \{\varphi\} c \{\psi\}\), iff for all states \(\sigma, \sigma' \in S\) such that \(\sigma \in [\varphi]\) and \((\sigma, \sigma') \in [c]\) then \(\sigma' \in [\psi]\).

Triples can be proved valid using the rules of Hoare logic (Fig. 5.1). Hoare logic is sound: if \(\vdash \{\varphi\} c \{\psi\}\) is derivable, then \(\vdash \{\varphi\} c \{\psi\}\).

**Relational Hoare Logic** Relational Hoare logic is a variant of Hoare logic to reason about two programs. Its judgments are quadruples \(\{\varphi\} c_1 \sim c_2 \{\psi\}\), where \(c_1\) and \(c_2\) are separable commands, i.e., commands that
do not have variables in common, and \( \varphi \) and \( \psi \) are assertions. Under the separability assumption, one can identify assertions as relations on states: \( (\sigma_1, \sigma_2) \in [\varphi] \) iff \( \sigma_1 \uplus \sigma_2 \in [\varphi] \), where \( \sigma_1 \uplus \sigma_2 \) denotes the union of finite maps \( \sigma_1 \) and \( \sigma_2 \).

**Definition 5.1.2** (Valid quadruple). \( \{ \varphi \} c_1 \sim c_2 \{ \psi \} \) is valid, written \( \models \{ \varphi \} c_1 \sim c_2 \{ \psi \} \), iff for all states \( \sigma_1, \sigma_2, \sigma_1', \sigma_2' \in S \) such that \( \sigma_1 \uplus \sigma_2 \in [\varphi] \), \( (\sigma_1, \sigma_1') \in [c_1] \) and \( (\sigma_2, \sigma_2') \in [c_2] \) then \( \sigma_1' \uplus \sigma_2' \in [\psi] \).

The rules of relational Hoare logic appear in Fig. 5.2. All rules, except (R-INST) and (R-COMPOSE), are standard and self-explanatory. The rule (R-INST) allows relating basic blocks of different length, and by extension commands that have the same control flow but differ in their basic blocks, and subsumes the following rules for assignment (R-ASSIGN) and skip (R-SKIP)

\[
\vdash \{ \phi \} \text{skip} \sim \text{skip}\{ \phi \} \quad \text{(R-SKIP)}
\]

\[
\vdash \{ \phi[e_1/x_1, e_2/x_2] \} x_1 := e_1 \sim x_2 := e_2 \{ \phi \} \quad \text{(R-ASSIGN)}
\]

The rule (R-COMPOSE) allows intermediate commands in reasonings. The assertion \( \exists X_2. \ \varphi \land \phi' \) existentially quantifies over the variables of \( c_2 \), simulates the composition of relations in the setting of assertions. Note that soundness of (R-COMPOSE) requires that \( c_2 \) terminates as \( \models \{ \varphi \} c_1 \sim \text{while true do skip}\{ \psi \} \) is valid for every command \( c_1 \) and assertions \( \varphi \) and \( \psi \). Relational Hoare logic is sound: if \( \vdash \{ \varphi \} c_1 \sim c_2 \{ \psi \} \) is derivable, then \( \models \{ \varphi \} c_1 \sim c_2 \{ \psi \} \).

### 5.1.2 Embedding

The embedding of relational Hoare logic into Hoare logic is based on the notion of synchronized product of two commands. Synchronized products are closely related to cross-products, introduced by Zaks and Pnueli [38] for translation validation of structure-preserving optimizations.

**Definition 5.1.3** (Synchronized product). The synchronized product \( c_1 \oplus c_2 \) of two commands \( c_1 \) and \( c_2 \) is, if it exists, any command \( c \) s.t. \( c_1 \sim c_2 \rightarrow c \) is derivable using the rules of Fig. 5.3.
Proposition 5.1.4. If \( c_1 \sim c_2 \rightarrow c \) then \( c_1 \bowtie c_2 \) is defined and moreover \( \vdash \{ \varphi \} c_1 \bowtie c_2 \{ \psi \} \).

The embedding is blatantly incomplete, since the (R-WHILE) rule enforces that related while loops perform the same number of iterations. A possible solution to accommodate commands that are not structurally equal is to extend the construction of the synchronized product with a self-composition rule that allows transforming \( c_1 \) and \( c_2 \) into \( c_1 ; c_2 \). Unfortunately, the approach has two drawbacks: first, in absence of a general criterion for applying the self-composition rule, the product construction is not deterministic. Second, it is often difficult to find appropriate annotations to verify programs that are built using self-composition. Therefore, we develop an alternative method for constructing verifiable product programs that allow validating relational Hoare judgments.

5.2 Product Programs for 2-Properties

In order to support effective verification of relational judgments that involve two programs that are not structurally equivalent, we extend the cross-product construction with structural rules. The resulting verification method combines product construction with standard program verification. For technical reasons, we introduce \texttt{assert} statements that are used to verify that the product program simulates precisely the behavior of its components. These assertions are discharged during the program verification phase.

Figure 5.4 defines a set of structural rules defining judgments of the form \( \vdash c \bowtie c' \). From the rules given in the figure, one can see that the executions of \( c \) and \( c' \) coincide for every initial state that makes the introduced
assert statements valid. One can prove that the judgment \( \vdash c \trianglerighteq c' \) establishes a refinement relation: if \( \vdash c \trianglerighteq c' \) then \( c \trianglerighteq c' \). Then, we enrich the set of rules defining the construction of cross-products by adding an extra rule that introduces a preliminary refinement transformation over the product components:

\[
\frac{c_1 \trianglerighteq c'_1 \quad c_2 \trianglerighteq c'_2 \quad c'_1 \times c'_2 \rightarrow c}{c_1 \times c_2 \rightarrow c}
\]

The following proposition reduces the problem of proving the validity of a relational judgment into two steps: the construction of the corresponding program product plus a standard verification over the program product.

**Proposition 5.2.1.** For all statements \( c_1 \) and \( c_2 \) and pre and post-relations \( \varphi \) and \( \psi \), if \( c_1 \times c_2 \rightarrow c \) and \( \vdash \{ \varphi \} c_1 \sim c_2 \{ \psi \} \), then \( \vdash \{ \varphi \} c_1 \sim c_2 \{ \psi \} \).

We have implemented a proof of concept verification plugin in the Frama-C environment, and used this plugin to validate several examples of relational verification. The plugin receives as input a file with two programs and a predicate that describes the relation between the abstract and concrete states, using the ANSI C Specification Language (ACSL). The plugin builds a product of the supplied programs, adding when required additional program annotations. The final annotated product program is fed into the Frama-C Jessie plugin, which translates the C product program into Why’s intermediate language and discharges the verification conditions using the available SMT solvers (AltErgo, Simplify, Z3, etc.). Figure 5.5 depicts the interaction of the plugin with other components of the framework.

### 5.3 Application to Program Vectorization

Program vectorization is an optimization technique that exploits the data parallelism provided by vector instructions. Despite its success, the applicability of program vectorization has been limited to a relatively small set of applications, and it remains a significant challenge to leverage its benefits to a wider range of programs. In general, program vectorization has been mostly confined to loops that operate on large blocks of contiguous scalar data and that contain minimal and relatively simple control structures. In particular, vectorizing compilers generally do not effectively handle programs with extensive, non-trivial, control flow dependencies. Additionally, manual program vectorization is error-prone and specific to a particular instruction set, making the task of testing and maintaining bug fixes among multiple implementations unmanageable.

We use relational program verification to build sound and practical vectorization methods that are applicable to large, general-purpose software. Informally, our method uses product programs to justify a sequence of program transformations that combines semantic-preserving loop optimizations, including loop range splitting and loop tiling. Moreover, we combine relational verification with program synthesis as follows: rather than proving the validity of a relational judgment with two complete programs, we build a partial proof of the source program, and of an incomplete program that contains a hole corresponding to the code fragment to be optimized. Then, our methodology delivers a specification, consisting of a pre-condition and
```c
int find(int a[], int n, int val ){
  int i;
  for(i = 0; i<n; ++i){
    if (a[i]==val) return (i);
  }
  return (-1);
}
```

Figure 5.6: Find algorithm implementation.

a post-condition, for program synthesis; this specification is given to a synthesizer, that produces a code snippet that matches the specification and uses vector instructions. By compositionality, it follows that the optimized program, consisting of the partial program with the generated snippet, is semantically equivalent to the source program. We have used this approach for building verified vectorized versions of representative algorithms for common data structures. We observe that the optimized code exhibits significant performance gains.

To illustrate our approach, consider the `find` algorithm, which returns the first occurrence of value `val` in array `a` of size `n`. Figure 5.6 shows a reference implementation of `find`. Unlike the programs typically considered for vectorization, which do not feature abrupt termination statements or complex control-flow dependencies, there is a strong loop-carried control flow dependency based on the conditional control flow and `return` statement in the loop body. Moreover, this control flow dependency cannot be converted to a data flow dependency in a way that preserves the semantics of the loop without incurring a large performance penalty. We provide a vectorized implementation of the algorithm, and a sketch of the method for proving that the vectorized and the original implementation have the same functional behavior. We describe the method step-by-step. The first step towards vectorizing the `find` algorithm is to identify the part of the loop that can be mirrored with vector instructions and isolate it from the control part. In this case, the sequence of assignments is

```c
bool a = a[i]==val;
bool b = a[i+1]==val;
bool c = a[i+2]==val;
bool d = a[i+3]==val;
```

These operations can be expressed in terms of vector instructions that loads four copies of `val`, then loads the values of positions `i` through `i+3`, and then compares these two vectors for pairwise equality.

The goal of the vectorization method is to exploit the semantic relationship between these two pieces of code. We perform stepwise semantic preserving code transformations. First, we perform loop range splitting to divide the loop into two loops, such that the first performs `distance ÷ 4` iterations and the second one the remaining `distance mod 4` iterations. Afterwards, we perform loop tiling on the first loop to increase its stride to 4 so its body captures the effect of 4 iterations. The resulting code is shown in Fig. 5.7.

Next, we modify the body of the first loop so that it takes advantages of the vector instructions. Informally, the new loop body is constructed by moving the assignments to the top of the loop and by replacing them with the instruction block as described above. The conditional return statements are modified accordingly.

Hence, the last step of the transformation is to replace the sequence of conditional return statements by a more efficient piece of code. Informally, the loop compares the whole result vector by performing bitwise `and` of its arguments and compares the result to 0. The result obtained indicates whether one of the indexes must hold the value `val` so the result is refined accordingly.

Now, we want to provide formal evidence supporting the fact that the reference implementation `find` and the optimized implementation `sse_find` are functionally equivalent, i.e., they produce equal results when
int find(int a[], int n, int val) {
    int i;
    for (i=0; i<n-3; i+=4){
        bool b0 = a[i]==val;
        if (b0) return (i);
        bool b1 = a[i+1]==val;
        if (b1) return (i+1);
        bool b2 = a[i+2]==val;
        if (b2) return (i+2);
        bool b3 = a[i+3]==val;
        if (b3) return (i+3);
    }
    for (; i<n; ++i){
        if (a[i]==val) return (i);
    }
}

Figure 5.7: Loop range splitting and tiling.

executed with equal arguments. We capture equivalence as a relational judgment:

\{\texttt{val1 = val2}} \texttt{find} \sim \texttt{sse_find} \{\texttt{res1 = res2}\}

This means that if the programs are executed with arguments that satisfy the relational precondition and finish, they return values satisfying the relational postcondition. To prove the validity of the judgment, we use the product program of the original and optimized versions of \texttt{find}. The product program threads the instructions of the original and of the optimized programs, and is intended to reproduce their combined effects. To verify that the product program satisfies the pre- and post-conditions derived from the relational judgment, we decorate the product program with suitable loop invariants. Fortunately, loop invariants for establishing the correspondence between the two implementations can be obtained systematically as a conjunction of linear equalities between program variables of the left and right programs.
Chapter 6

Non-Terminating Behaviour

Prior to the work described in this chapter, the formalised semantics of ABS did not capture non-terminating behaviour. In the setting of local concurrency and global distribution, as featured by the ABS language, this is a serious shortcoming. First of all, certain non-terminating behaviours are indeed intended in our setting, where each object (group) can be seen as a reactive system, accepting tasks infinitely. Second, and related to the first point, non-terminating behaviour carries information in the distributed setting, and that information must be captured semantically. This is different from the purely sequential setting, where the bottom element of Scott domains carries precisely zero information, and different non-terminating runs are indistinguishable.

In this work, we developed a framework for big-step semantics for a sub-set of ABS, with interactive input-output in combination with divergence. The results were presented at SOS 2010 [84]. The framework is based on coinductive and mixed inductive-coinductive notions of resumptions, evaluation and termination-sensitive weak bisimilarity. In contrast to standard inductively defined big-step semantics, this framework handles divergence properly; in particular, runs that produce some observable effects and then diverge, are not “lost”. Here we scale this approach for shared-variable concurrency on a simple example language. We develop the metatheory of this semantics in a constructive logic.

As future work, beyond the lifetime of the HATS project, we will lift this framework to capturing the full ABS language.

6.1 Introduction

This work advocates two ideas. First, big-step operational semantics can handle divergence as well as small-step semantics, so that both terminating and diverging behaviors can be reasoned about uniformly. Big-step semantics that account for divergence properly are achieved by working with coinductive semantic entities (transcripts of possible infinite computation paths or non-well-founded computation trees) and coinductive evaluation. Second, contrary to what is so often stated, concurrency is not inherently small-step, or at least not more inherently than any kind of effect produced incrementally during a program’s run (e.g., interactive output). Big-step semantics for concurrency can be built by borrowing the suitable denotational machinery, except that we do not want to use domains and fixpoints to deal with partiality, but coinductively defined sets and corecursion. In this chapter we use resumptions, more specifically coinductive resumptions. This datatype is a monad that accommodates both concurrency and divergence.

We build on our previous work [84] and develop a resumption-based big-step semantics with several variations for a simple imperative language with shared-variable concurrency. The metatheory of this semantics (e.g., the equivalence of evaluation in the big-step semantics to maximal multi-step reduction in a reference small-step semantics) is entirely constructive, which means that we can compute big-step evaluations from maximal multi-step reductions and vice versa. Moreover, deterministic versions of evaluation are also computable functions.

The idea that divergence can be properly handled by switching to coinductively defined semantic entities such as possibly infinitely delayed states or possibly infinite traces, is due to Capretta [20]. The deeper
underlying theory is based on completely iterative monads and has been treated in detail by Goncharov and Schröder [60].

Leroy [76] attempted to use coinductive big-step semantics to reason about both terminating and diverging program runs in the Compcert project on a formally certified compiler, but ran into certain semantic anomalies. Proving the big-step and small-step semantics equivalent required the use of excluded middle, which should not be needed. Infinite loops were not specifically arranged to be productive, with the effect that infinite loops with no observable effects led to finite traces, to which other traces could be appended. (Cf. also the simultaneous work by Cousot and Cousot [23]). Nakata and Uustalu [82] fixed the anomalies and arrived at a systematic account of trace-based big-step semantics for divergence in a purely sequential, side-effect-free setting (in relational and also functional styles). Further [83, 84], they also developed a matching Hoare logic and scaled the approach to a combination of interactive input/output with divergence. Danielsson [29] has strongly promoted especially functional-style coinductive big-step semantics. Ancona [7] used a coinductive big-step semantics of Java to show it type-sound in a positive rather than negative sense: if a program is type-sound, it produces a trace.

An inductive trace-based big-step semantics for a concurrent language (not handling divergence) has appeared in the work of Mitchell [81].

6.2 Language and Semantics

6.2.1 Language

The ABS subset we consider is the minimal language with shared-variable concurrency (cf. Amadio [5]) whose statements are given inductively by the grammar:

\[ s ::= x := e \mid \text{skip} \mid s_0 ; s_1 \mid \text{if } e \text{ then } s_t \text{ else } s_f \mid \text{while } e \text{ do } s_t \mid s_0 || s_1 \mid \text{atomic } s \mid \text{await } e \text{ do } s \]

The statement \( s_0 || s_1 \) stands for the parallel composition of \( s_0 \) and \( s_1 \) (in particular, it terminates when both branches have terminated). The statement \( \text{atomic } s \) is executed by running \( s \) atomically; the statement \( \text{await } e \text{ do } s \) is executed by waiting until \( e \) is true (other computations can have their chance in the meantime) and then running \( s \) atomically. For most of the text, we assume that scheduling is preemptive, only assignments and boolean guards are atomic implicitly. But we also comment on the option of cooperative scheduling. This way we capture in a minimalistic setting the essential technical subtleties of both concurrency models of ABS: the message-passing concurrency model between different concurrent object groups (i.e., preemptive and pure) and the shared-memory cooperative concurrency model within one concurrent object group (i.e., cooperative and imperative).

6.2.2 A Resumption-Based Big-Step Semantics

The central semantic entities in our semantics are resumptions (computation trees). Resumptions are defined coinductively by the following rules (here and in the following, coinductive definitions are indicated by double rule-lines).

\[
\begin{align*}
\sigma : \text{state} & \quad r : \text{res} \\
\text{ret } \sigma : \text{res} & \quad \delta r : \text{res} \\
\frac{r_0 : \text{res} \quad r_1 : \text{res}}{r_0 + r_1 : \text{res}} & \quad \frac{s : \text{stmt} \quad \sigma : \text{state}}{\text{yield } s \sigma : \text{res}}
\end{align*}
\]

The resumption \( \text{ret } \sigma \) denotes a computation that terminated in a state \( \sigma \). The resumption \( \delta r \) is a computation that first produces a unit delay (makes an internal small step) and continues then as \( r \). The resumption \( r_0 + r_1 \) is a choice between two resumptions \( r_0 \) and \( r_1 \). The resumption \( \text{yield } s \sigma \) is a computation that has released control in a state \( \sigma \) and will further execute a statement \( s \) when (and if) it regains control. The definition being coinductive has the effect that resumptions can be non-well-founded, i.e., computations can go on forever.

Evaluation of a statement \( s \) relates a (pre-)state to a (post-)resumption and is defined coinductively by the rules in Fig. 6.1.
Deliverable D4.3 Correctness

coinductively as follows: statement to start when (and if) the resumption terminates or releases control. Also this relation is defined given statement in parallel with a given resumption. The idea is to create an opportunity for the given scheduling.

of the second statement of a sequential composition. [Again, in brackets, we show the rule for cooperative addition, the base case inserts a control release between the termination of the first statement and the start essentially, sequential extension of evaluation is a form of coinductive prefix closure of evaluation. But, in extension of evaluation

is evaluated) to a (post-)resumption (the total resumption after). It is defined coinductively by the rules:

\[
\begin{align*}
\text{while } e \text{ do } s_t, \sigma &\rightarrow \delta (\text{yield } (s_t; \text{while } e \text{ do } s_t) \sigma) \\
\text{await } e \text{ do } s, \sigma &\rightarrow \delta \, r' \\
s_0, \sigma &\rightarrow r_0 \\
\text{if } e \text{ then } s_t \text{ else } s_f, \sigma &\rightarrow \delta (\text{yield } s_f \sigma) \\
\text{if } e \text{ then } s_t \text{ else } s_f, \sigma &\rightarrow \delta (\text{yield } s_f \sigma) \\
\end{align*}
\]

Finally, \textit{closing a resumption} makes sure it does not release control. This is done by (repeatedly) “stitching” a resumption at every control release point by evaluating the residual statement from the state at this point. The corresponding relation between two resumptions is defined coinductively by:

\[
\begin{align*}
\text{yield } s &\sigma \rightarrow \delta \, r' \\
\text{yield } s &\sigma \rightarrow \delta \, r' \\
\end{align*}
\]

We have made sure that internal small steps take their time by inserting unit delays at all places where assignments or boolean guards are evaluated. The \textit{yields} in the rules for \textit{if} and \textit{while} signify control release points. Control release also occurs at the “midpoint” of evaluation of any sequential composition. This is handled by the base rule for sequential extension of evaluation (see below). [In brackets, we show the rules for cooperative scheduling where control is not released at these points.]

\textit{Sequential extension of evaluation} relates a (pre-)resumption (the resumption present before the statement is evaluated) to a (post-)resumption (the total resumption after). It is defined coinductively by the rules:

\[
\begin{align*}
\text{await } e \text{ do } s, \sigma &\rightarrow \delta \, r' \\
s_0, \sigma &\rightarrow r_0 \\
\text{if } e \text{ then } s_t \text{ else } s_f, \sigma &\rightarrow \delta (\text{yield } s_f \sigma) \\
\text{if } e \text{ then } s_t \text{ else } s_f, \sigma &\rightarrow \delta (\text{yield } s_f \sigma) \\
\end{align*}
\]

Figure 6.1: Evaluation of statements
To give only two small examples, for $s = x := 1 \parallel (x := x + 2; x := x + 2)$, $\sigma = [x \mapsto 0]$, we have $s, \sigma \Rightarrow \delta(yield (x := x + 2; x := x + 2) [x \mapsto 1]) + \delta(yield (x := x + 2 \parallel x := 1) [x \mapsto 2])$ while atomic $s, \sigma \Rightarrow \delta^3(ret [x \mapsto 5]) + \delta^3(\delta^3(ret [x \mapsto 1]) + \delta^3(ret [x \mapsto 3]))$. For $s = (await x = 0 do x := 1) \parallel x := 2$, $\sigma = [x \mapsto 0]$, we have $s, \sigma \Rightarrow \delta^3(yield x := 2 [x \mapsto 1]) + \delta^4(yield (await x = 0 do x := 1) [x \mapsto 2])$ whereas atomic $s, \sigma \Rightarrow \delta^4(ret [x \mapsto 2]) + \delta^\infty$.

For example, the first judgement above is derivable as follows:

$$
\begin{array}{c}
\vdash 1, [0] \Rightarrow \delta(ret [1]) \\
\vdash +2, ; +2, ret [1] \Rightarrow \text{par} \ delta(yield (\vdash +2, ; +2) [1]) \\
\vdash +2, ; +2, \delta(ret [1]) \Rightarrow \text{par} \ delta(yield (\vdash +2, ; +2) [1]) \\
\vdash +2, ret [2] \Rightarrow yield (\vdash +2) [2] \\
\vdash 1, yield (\vdash +2) [2] \Rightarrow \text{par} yield (\vdash +2 || 1) [2] \\
\vdash 1, \delta(yield (\vdash +2) [2]) \Rightarrow \text{par} \delta(yield (\vdash +2 || 1) [2]) \\
\end{array}
$$

(We have abbreviated $x := n$ as $:= n$, $x := x + n$ as $:= +n$, $[x \mapsto n]$ as $[n]$.)

### 6.2.3 Small-Step Semantics

To validate our big-step semantics, we can relate it to a small-step semantics. In this semantics, single-step reduction associates to a statement and a state an extended configuration in a deterministic and total way (if the step amounts to making a choice, both alternatives are recorded). With maximal multi-step reduction, a statement’s all runs are developed into a resumption.

Extended configurations (we use the notation of an inductive definition, but in fact this datatype is a simple disjoint union) are defined by the following rules:

$$
\begin{array}{l}
\begin{array}{l}
s : \text{stmt} \\
s_0 : \text{stmt} \\
\sigma_0 : \text{state} \\
\sigma_1 : \text{state} \\
s : \text{stmt} \\
\delta(s, \sigma) : xcfg \\
yield s : \sigma : xcfg \\
\end{array}
\end{array}
$$

Single-step reduction relates a state to an extended configuration and is defined inductively (not coinductively!) in Fig. 6.2.

Notice that skip $s$ differs from $s$ by allowing a control release before $s$ is started. We have also used an auxiliary statement of the form $s_0 \parallel s_1$ (parallel composition, but $s_0$ makes the first small step).

In the rules for sequential composition, the small step of $s_0; s_1$ is determined by the small step of $s_0$. If this terminates $s_0$ in state $\sigma'$, then $s_0; s_1$ is suspended in $\sigma'$ the with $s_1$ as the residual statement. The rules for $\parallel$ are similar.

Maximal multi-step reduction relates a state to a resumption and is defined coinductively:

$$
\begin{array}{l}
\begin{array}{l}
s, \sigma \Rightarrow s' \\
s, \sigma \Rightarrow \text{par} s' \\
s, \sigma \Rightarrow \delta(s', \sigma') \\
s, \sigma \Rightarrow \delta^m(s', \sigma') \\
\end{array}
\end{array}
$$

Evaluation agrees with maximal multi-step reduction: $s, \sigma \Rightarrow r$ iff $s, \sigma \Rightarrow^m r$. 

HATS Deliverable D4.3

Correctness

6.2.4 Functional Version of the Big-Step Semantics

In the big-step semantics we have shown, we collect the possible executions of a statement into a single computation tree and divergence is represented by infinite delays. Evaluation is deterministic and total. This means that it can be turned into a function (so from the constructive point of view, evaluations can not only be checked, but also computed—which is very positive). Evaluation is defined as a function given in Figures 6.3, 6.4, 6.5, 6.6.

Functional and relational evaluation agree: $eval \ s \ \sigma \sim r$ iff $s, \sigma \Rightarrow r$.

6.2.5 Giant-Step Semantics

It is also possible to work with a different (purely semantic) form of resumptions, requiring that the residual programs at control release points are evaluated. This leads to a form of giant-step semantics. In the giant-step semantics, resumptions are defined as follows:

$$
\begin{array}{ll}
\sigma : state
& r : Res \\
ret \sigma : Res
& \delta r : Res \\
r_0 : Res
& r_1 : Res \\
k : state \rightarrow Res
& \sigma : state \\
yield k \sigma : Res
\end{array}
$$

Functional-style evaluation is defined as given in Figures 6.7, 6.8, 6.9, 6.10.

One might, of course argue, that what we have called the “big-step” semantics here should be called “medium-step”, and the “giant-step” semantics should be called “big-step”. Our choice of terminology here was motivated by the intuition that “big-step” evaluation should run a statement to its completion. When a statement’s run has reached a control release point, it is complete in the sense that it cannot run further on its own and it cannot be told what the scheduler will do with it (it might even be unfair and not return
control to it at all). Note, however, that the big-step and giant-step evaluation agree fully for statements of the form atomic s.

### 6.2.6 Equivalences of Resumptions

When are two resumptions to be considered equivalent? This depends on the purpose at hand. The finest sensible notion is the "very strong" bisimilarity defined coinductively by the rules:

\[
\begin{align*}
\text{ret } \sigma & \sim \text{ret } \sigma \\
\delta \ r & \sim \delta \ r \\
(\text{yield } s_0 \ σ) & \sim (\text{yield } s_1 \ σ) \\
(r_0 + r_1) & \sim (\text{yield } s_0 \ σ + \text{yield } s_1 \ σ) \\
\text{yield } s \ σ & \sim \text{yield } s \ σ
\end{align*}
\]

Classically, it is just equality of resumptions; but equality in intensional type theory is yet stronger. Our big-step semantics is deterministic and agrees with the functional version in exactly this strongest meaningful sense.

The useful coarser notions evaluate residual statements of suspended resumptions and compare the results for bisimilarity; ignore order and multiplicity of choices (ordinary strong bisimilarity); or hide finite delays and/or choices altogether (termination-sensitive weak bisimilarity). The definition of termination-sensitive weak bisimilarity requires combining or mixing induction and coinduction, with several caveats to avoid. First, it is easy to misdefine weak bisimilarity so that it equates any resumption with the divergent resumption and therefore all resumptions. Second, a fairly attractive definition fails to give reflexivity without the use of excluded middle, which is a warning that the definition is not the “right one” from the constructive point of view. We refrain from giving the details here, but the key ideas are those from our SOS ’10 paper.

### 6.2.7 A Trace-Based Big-Step Semantics

A trace-based semantics is obtained from the resumption-based semantics by simply removing the + constructor of resumptions, splitting the evaluation rule for \( \parallel \) into two rules (thereby turning evaluation non-deterministic) and removing the rules for + in the definitions of extended evaluations and closing. Because of the non-determinism, trace-based evaluation cannot be turned into a function. But notice that it is still total. Any scheduling leads to a valid trace. Differently from standard inductive big-step semantics, divergence from endless work or waiting does not lead to a “lost trace”.

Differently from the case of trace-based big-step evaluation, trace-based giant-step evaluation must, in one way or another, use “guessing”. A valid giant-step trace of a statement consists in a trace from a given pre-state to a control release state, followed by a trace of the residual statement from an arbitrary (“guessed”) control grab state, etc. (unless the run terminates or diverges). When a statement is atomized, most of the traces constructed in this speculative way are thrown away.
\[
\text{evalpar } s \ (\text{ret } \sigma) = \text{yield } s \ \sigma \\
\text{evalpar } s \ (\delta \ r) = \delta \ (\text{evalpar } s \ r) \\
\text{evalpar } s \ (r_0 + r_1) = \text{evalpar } s \ r_0 + \text{evalpar } s \ r_1 \\
\text{evalpar } s \ (\text{yield } s_0 \ \sigma) = \text{yield } (s_0 \ || \ s) \ \sigma
\]

Figure 6.5: Big-step semantics: parallel extension of evaluation

\[
\text{close } (\text{ret } \sigma) = \text{ret } \sigma \\
\text{close } (\delta \ r) = \delta \ (\text{close } r) \\
\text{close } (r_0 + r_1) = \text{close } r_0 + \text{close } r_1 \\
\text{close } (\text{yield } s \ \sigma) = \delta \ (\text{close } (\text{eval } s \ \sigma))
\]

Figure 6.6: Big-step semantics: closing a resumption

6.3 Conclusion

We have shown that, with coinductive denotations and coinductive evaluation, it is possible to give simple and meaningful big-step descriptions of semantics of languages with concurrency. The key ideas remain the same as in the purely sequential case. Most importantly, due care must be taken of the possibilities of divergence. In particular, even diverging loops or await statements must be productive (by growing resumptions or traces by unit delays). Finite delays can then be equated by a suitable notion of weak bisimilarity.

Although we could not delve into this topic here, our definitions and proofs benefit heavily from the fact that the datatype of resumptions is a monad, in fact a completely iterative monad, and moreover a free one (as long as we equate only very strongly bisimilar resumptions).
Eval \( (x := e) \) \( \sigma \) = \( \delta (\text{ret } \sigma[x \mapsto [\llbracket e \rrbracket ]}) \)

Eval \( \text{skip} \) \( \sigma \) = \( \text{ret } \sigma \)

Eval \( (s_0; s_1) \) \( \sigma \) = \( \text{Evalseq } s_1 \ (\text{Eval } s_0 \ \sigma) \)

Eval \( (\text{if } e \ \text{then } s_t \ \text{else } s_f) \) \( \sigma \) = \( \text{if } [e] \ \sigma \ \text{then } \delta (\text{yield } (\text{Eval } s_t) \ \sigma) \ \text{else } \delta (\text{yield } (\text{Eval } s_f) \ \sigma) \)

Eval \( (\text{while } e \ \text{do } s_t) \) \( \sigma \) = \( \text{if } [e] \ \sigma \ \text{then } \delta (\text{yield } (\text{Evalseq } (\text{while } e \ \text{do } s_t) \circ \text{Eval } s_f) \sigma) \ \text{else } \delta (\text{ret } \sigma) \)

Eval \( (s_0 \parallel s_1) \) \( \sigma \) = \( \text{Merge } (\text{Eval } s_1) \ (\text{Eval } s_0 \ \sigma) + \text{Merge } (\text{Eval } s_0) \ (\text{Eval } s_1 \ \sigma) \)

Eval \( (\text{atomic } s) \) \( \sigma \) = \( \text{Close } (\text{Eval } s \ \sigma) \)

Eval \( (\text{await } e \ \text{do } s) \) \( \sigma \) = \( \text{if } [e] \ \sigma \ \text{then } \delta (\text{Close } (\text{Eval } s \ \sigma)) \ \text{else } \delta (\text{yield } (\text{Eval } (\text{await } e \ \text{do } s)) \ \sigma) \)

Figure 6.7: Giant-step semantics: evaluation

\[
\begin{align*}
\text{Evalseq } s \ (\text{ret } \sigma) & = \text{yield } (\text{Eval } s \ \sigma) \\
\text{Evalseq } s \ (\delta r) & = \delta (\text{Evalseq } s \ r) \\
\text{Evalseq } s \ (r_0 + r_1) & = \text{Evalseq } s \ r_0 + \text{Evalseq } s \ r_1 \\
\text{Evalseq } s \ (\text{yield } k \ \sigma) & = \text{yield } (\text{Evalseq } s \circ k) \ \sigma
\end{align*}
\]

Figure 6.8: Giant-step semantics: sequential extension of evaluation

\[
\begin{align*}
\text{Merge } k \ (\text{ret } \sigma) & = \text{yield } k \ \sigma \\
\text{Merge } k \ (\delta r) & = \delta (\text{Merge } k \ r) \\
\text{Merge } k \ (r_0 + r_1) & = \text{Merge } k \ r_0 + \text{Merge } k \ r_1 \\
\text{Merge } k \ (\text{yield } k_0 \ \sigma) & = \text{yield } (\lambda \sigma'. \text{Merge } k \ (k_0 \ \sigma')) \ \sigma
\end{align*}
\]

Figure 6.9: Giant-step semantics: merge of a continuation into a resumption

\[
\begin{align*}
\text{Close } (\text{ret } \sigma) & = \text{ret } \sigma \\
\text{Close } (\delta r) & = \delta (\text{Close } r) \\
\text{Close } (r_0 + r_1) & = \text{Close } r_0 + \text{Close } r_1 \\
\text{Close } (\text{yield } k \ \sigma) & = \delta (\text{Close } (k \ \sigma))
\end{align*}
\]

Figure 6.10: Giant-step semantics: closing a resumption

78
Chapter 7

Conclusion

This report describes the work done in Task 4.3 towards the correctness of systems modeled and realized with the HATS methodology.

Several approaches, distinguished by the kind of properties they consider, were developed to analyse correctness of systems. In particular they have been classified identifying five different lines of work, which also single out the different kinds of contributions provided in this deliverable:

- Compositional deductive verification of systems for ABS based on the translation of program properties expressed in a specification language; e.g., invariants expressed with attribute grammars are translated into dynamic logic proof-obligations. Dynamic logic formulas are then evaluated by the KeYABS verifier. The technique also encompasses object creation, dynamic binding and arrays. Recently it has also been extended to changes to target code, so to encompass verification of software families via delta modeling, by means of the new technique called abstract method calls.

- Verification of composition patterns and interaction policies at the system level with two different techniques: The former is based on compositional verification of system parts, by using behavioral specifications of their interaction and by identifying patterns of composition of objects at the abstraction level of COGs; the latter enforces data flow policies, by restricting the interleaved sequence of security relevant actions performed by the program, by using tree automata and an implementation inliner.

- Deadlock analysis by using two different approaches: one based on Petri nets and one on behavioural contracts. The former extracts from a program a finite Petri net via a particular deadlock preserving technique, which allows for a finite representation of futures originated from asynchronous method calls, and it detects deadlock by performing reachability analysis on the obtained Petri net. The latter extracts abstract descriptions of method’s behaviors in the form of behavioral contracts, which are then turned into finite state models by an evaluator computing a fixpoint semantics and they are then checked for circular dependency.

- Analysis based on relational logic to verify transformation correctness, non-interference and determinism. An automated construction is provided for transforming relational verification tasks into standard verification tasks that enables the use of existing standard tools for the verification of properties for which there has been poor tool support.

- Extension of verification techniques to properly account for nonterminating behaviors by usage of co-inductive big-step semantics. In contrast to standard inductively defined big-step semantics, this framework, which is based on coinductive and mixed inductive-coinductive notions of resumptions, handles divergence properly; in particular, runs that produce some observable effects and then diverge, are not “lost”.

79
Bibliography


Glossary

Terms and Abbreviations

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

ABS DL ABS Dynamic Logic a variant of dynamic logic used to reason about ABS models.

COG Concurrent Object Group, the unit of parallelism in ABS.

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.

Delta Synonymous with delta module

Contract Delta An extension of deltas which allows to add/modify/remove specifications.

Delta module A specification of modifications to core ABS language elements (classes, methods, interfaces, etc.)

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

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.

KeY A formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

Product program A program that combines the effects of two programs by interleaving their instructions.

Proof Obligation A formula expressing a property about a program.

Relational verification A method to reason about the relationship between two programs.

Scheduling The act of choosing one of a set of processes for execution.

Self-composition Technique used to establish non-interference properties through the use of standard program logics.

Software component A modelling abstraction reflecting the logical units of composition, which provides isolation, mobility, and data-flow reconfiguration capacities.

Symbolic Execution Execution of a model/program using symbolic values as input values.

Vectorization A optimization that transforms programs to take advantage of a vector processor. SSE is an example of instruction set that supports vectorized computations.