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 D2.6
Refinement and Abstraction

Due date of deliverable: (T0+42)

Actual submission date: 1st September 2012

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

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

Revision 16391M

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Refinement and Abstraction

This document summarises deliverable D2.6 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`.

The deliverable describes various forms of refinement and abstraction in the context of the ABS modeling framework. As main result, the deliverable presents:

- an approach to show behavior-preserving refinements at the level of class libraries,
- a deductive compilation approach that refines an ABS model to Java bytecode while provably preserving the ABS model's properties, and
- a compositional notion of refinement for timed I/O automata with deadlines applied to the ABS setting.

List of Authors

Richard Bubel (TUD)
Mohammad Mahdi Jaghoori (CWI)
Ran Ji (TUD)
Yannick Welsch (UKL)

Contents

Chapter 1

Introduction

This deliverable reports on the HATS activities done as part of task 2.6 "Refinement and Abstraction". The two central goals of this task were to develop (1) a foundational notion for refinement and implementation correctness in the context of the ABS modeling framework, and (2) an automatic technique to check consistency properties during system derivation. According to the discussion at the last review meeting and the suggestions of the reviewers, we focused the activities in this task on specific aspects of refinement. In particular, we considered refinement from ABS models to code implementations of the models (Chapters 3 and 4) as well as comparing ABS models to other ABS models, or code to other code, respectively (Chapter 2). The task was organized into three lines of work, each described in one chapter.

- Chapter 2 is about behavior-preserving refinements, called backward compatibility, at the level of class libraries.
- Chapter 3 considers deductive compilation that refines an ABS model to Java bytecode while provably preserving the ABS model properties.
- Chapter 4 presents a compositional notion of refinement for timed I/O automata with deadlines applied to the ABS setting. In particular, this part addresses refinement in the setting of software families.

As main results in these three research threads, we developed a formal technique for checking backward compatibility for sets of classes based on a fully abstract semantics, a new method for program specialization exploiting symbolic evaluation, as well as a technique to apply the theory of timed automata families for compositional schedulability analysis of real-time software families. The results have been published in the ten papers that we briefly summarize in the section below.

1.1 List of Papers Comprising Deliverable D2.6

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. The papers are not directly attached to Deliverable **D2.6**, but are made available on the HATS web site at the following url: `http://www.hats-project.eu/sites/default/files/D2.6`. Direct links are also provided for each paper listed below.

Paper 1: A Type System for Checking Specialization of Packages in Object-Oriented Programming

This paper [16] provides a type system to enable large-scale development using clean interfaces that promote encapsulation and information hiding for object-oriented programs. A prerequisite for API compatibility is that all client code that compiles against the old API version also compiles against the new version. Explicit package signatures are proposed that allow for modular type-checking of such properties. It is then shown how the signatures can be derived from packages and a checkable specialization relation for package signatures is given. This paper is tied to the work described in Chapter 2.

This paper was written by Ferruccio Damiani, Arnd Poetzsch-Heffter, and Yannick Welsch, and it will be published in the proceedings of SAC 2012.

(Download Paper 1.)

Paper 2: Full Abstraction at Package Boundaries of Object-Oriented Languages

In this paper [53], we develop a fully abstract trace-based semantics for sets of classes in object-oriented languages, in particular for Java-like sealed packages. Our approach enhances a standard operational semantics such that the change of control between the package and the client context is made explicit in terms of interaction labels. By using traces over these labels, we abstract from the data representation in the heap, support class hiding, and provide fully abstract package denotations. This paper is tied to the work described in Chapter 2.

This paper was written by Yannick Welsch and Arnd Poetzsch-Heffter, and it was published in the proceedings of SBMF 2011.

(Download Paper 2.)

Paper 3: A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries

This paper [54] is an extended version of the previous one. Here we give a proof of soundness and completeness of the trace semantics by using specialized simulation relations on the enhanced operational semantics. The simulation relations also provide a proof method for reasoning about backward compatibility. This paper is tied to the work described in Chapter 2.

This paper was written by Yannick Welsch and Arnd Poetzsch-Heffter, and has been submitted for journal publication.

(Download Paper 3.)

Paper 4: Verifying Backwards Compatibility of Object-Oriented Libraries using Boogie

This paper [55] presents a technique to verify the backwards compatibility or equivalence of class libraries in the setting of unknown program contexts. For a number of textbook examples we have formulated the verification conditions as input to the Boogie program verification system and validated the approach. This paper is tied to the work described in Chapter 2.

This paper was written by Yannick Welsch and Arnd Poetzsch-Heffter, and was published in the proceedings of FTfJP 2012.

(Download Paper 4.)

Paper 5: Model-based Compatibility Checking of System Modifications

This paper [47] generalizes the approach of the previous papers from class libraries to system components. This paper is tied to the work described in Chapter 2.

This paper was written by Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch, and will be published in the proceedings of ISoLA 2012.

(Download Paper 5.)

Paper 6: Program Specialization Via a Software Verification Tool

This paper [14] proposes a new approach to generate specialized programs for a Java-like language via the software verification tool KeY. This is achieved by symbolically executing source programs interleaved with calls to a simple partial evaluator. In a second phase the specialized programs are synthesized from the

symbolic execution tree. The correctness of this approach is guaranteed by a bisimulation relation on the source and specialized programs. This paper is tied to the work described in Chapter 3.

This paper was written by Reiner Hähnle, Richard Bubel and Ran Ji, and it was published in the proceedings of FMCO 2010.

(Download Paper 6.)

Paper 7: PE-KeY: A Partial Evaluator for Java Programs

This paper [39] presents a prototypical implementation of a partial evaluator for Java programs based on the verification system KeY. We argue that using a program verifier as technological basis provides potential benefits leading to a higher degree of specialization. The first results achieved with the presented tool is provided. This paper is tied to the work described in Chapter 3.

This paper was written by Ran Ji and Richard Bubel, and it was published in the proceedings of iFM 2012.

(Download Paper 7.)

Paper 8: Composing Real-Time Concurrent Objects

This paper [33] extends the usual notion of refinement to real-time input-output automata where a deadline on an output specifies when the task is required to finish, and a deadline on an input specifies the guaranteed time before which the task is finished. This paper is tied to the work described in Chapter 4.

This paper was written by Mohammad Mahdi Jaghoori, and it was published in the proceedings of FSEN 2011.

(Download Paper 8.)

Paper 9: An Automata-Theoretic Framework for Real-Time Actors

This paper [34] presents a theoretical framework for schedulability analysis of real-time concurrent objects. The analysis is done compositionally in which a counter-example oriented test case generation is proposed based on the notion of refinement between the objects' definitions and their behavioral interfaces. This paper is tied to the work described in Chapter 4.

This paper was written by Mohammad Mahdi Jaghoori, Delphine Longuet, Frank de Boer, Tom Chothia and Marjan Sirjani and it is submitted to ACM TOPLAS.

(Download Paper 9.)

Paper 10: Scheduling and Analysis of Real-Time Software Families

This paper [49] presents a formal model of real-time software product lines which supports variability in scheduling policies and rigorous and efficient techniques for modular schedulability analysis. This paper is tied to the work described in Chapter 4.

This paper was written by Hamideh Sabouri, Mohammad Mahdi Jaghoori, Frank de Boer and Ramtin Khosravi and it was published in the proceedings of COMPSAC 2012.

(Download Paper 10.)

Chapter 2

Verifying Backward Compatibility of Class Libraries

2.1 Introduction

Object-oriented libraries are usually implemented by the complex interplay of different classes. As libraries evolve over time, adaptations have to be made to their implementations. Sometimes such evolution steps do not preserve backward compatibility with existing clients (called *breaking API changes* in [22]), but often libraries should be modified, extended, or refactored in such a way that client code is not affected. Software developers can use informal guidelines (e.g., [21]) and special tools (e.g., [24]) to check compatibility aspects.

Reasoning about the behavioral equivalence of two library implementations is not only useful when no specifications are available; we conjecture (similar to Godlin and Strichman [28], and Banerjee and Naumann [5]) that it may often be simpler to verify the equivalence of two similar library implementations than to build a specification of the full behavior of the library and verify conformance to the specification. We further conjecture that, in the setting of class libraries, such verification tasks should be feasible by employing similar automated tools as in the case of specification conformance checking (e.g., [7, 10, 26, 32]). Our goal is to leverage the power of existing verification tools for object-oriented programs to safely check that two different OO library implementations exhibit the same behavior in the setting of unknown program contexts. In the rest of this chapter, we illustrate our goals using an example, introduce the developed reasoning method and the resulting verification system, and discuss the relation to the ABS framework.

2.2 Example

To illustrate our goals, let us consider a very simple library at the top of Fig. 2.1, which provides a Cell class to store and retrieve references to objects. In a more refined version of the Cell library at the bottom of Fig. 2.1, a library developer might now want the possibility to not only retrieve the last value that was stored, but also the previous value. In the new implementation of the class, he therefore introduces two fields to store values and a boolean flag to determine which of the two fields stores the last value that was set. This second representation allows to add a method to retrieve the previous value, e.g., **public** Object previous() { **return** f ? c2 : c1; }.

The developer might now wonder whether the old version of the library can be safely replaced with the new version, i.e., whether the new version of the Cell library still retains the behavior of the old version when used in program contexts of the old version. Intuitively, the developer might argue in the following way why he believes that both library implementations are equivalent: If the boolean flag in the second library version is true, then the value that is stored in the field c1 corresponds to the value that is stored in the field c in the first library version. Similarly, if the boolean flag is false, then the value that is stored in c2 corresponds to the value that is stored in c. In our work, we have developed a method to give a formal underpinning to this intuition. We have shown how to capture the relation (called *coupling invariant*) between the two

```
public class Cell { // old version
  private Object c;
  public void set(Object o) { c = o; }
  public Object get() { return c; }
}


public class Cell { // new version
  private Object c1, c2;
  private boolean f;
  public void set(Object o) {
    f = !f;
    if (f) c1 = o; else c2 = o; }
  public Object get() { return f ? c1 : c2; }
}
```

Figure 2.1: Cell example

library implementations formally and how to automatically verify the equivalence of such libraries using the program verifier Boogie [7].

2.3 Results

As described in the introduction, the main research goal of this line of work was to develop a tool-supported method to verify backward compatibility for object-oriented class libraries. Reasoning about the equivalence of class library implementations is challenging for the following reasons: 1) the number of possible contexts is infinite and contexts are complex and 2) the states and heaps can be significantly different between the different library implementations. In this section, we report on the progress made so far. Our contributions are as follows:

- We studied a type system to support the interface evolution of class libraries [16].
- We developed a sound and complete method for reasoning about backwards compatibility of class libraries [53, 54].
- We manually formulated verification conditions for a number of textbook examples as input to the Boogie program verification system to validate the reasoning approach [55].
- We are developing a verification condition generator that automates this task [9]. To the best of our knowledge, we present the first tool-supported formal verification approach for (modular) equivalence checking of OO libraries.
- We generalized the reasoning approach from class libraries to system components [47].

Related work There is a large body of work on studying the equivalence of programs and program parts. In this presentation, we focus on such studies in the setting of object-oriented programs. The two most popular ways to reason about the behavior equivalence of class implementations is to use denotational methods or bisimulations.

Denotational methods have been successfully used to investigate properties of object-oriented programs [15]. The denotational semantics corresponding to these methods provide representations of program

parts (e.g., classes) as mathematical objects describing how program parts modify the stack and heap. However, these denotations are often not *abstract* enough, i.e., they differentiate between classes that have the same behavior. Banerjee and Naumann [4] presented a method to reason about whole-program equivalence in a Java subset. Under a notion of confinement for class tables, they prove equivalence between different implementations of a class by relating their (classical, fixpoint-based) denotations by simulations. In subsequent work [5], they use a discipline employing assertions and ghost fields to specify invariants and heap encapsulation (by ownership techniques) and to deal with reentrant callbacks. Jeffrey and Rathke [38] give a fully abstract trace semantics for a Java subset with a package-like construct. However, they do not consider inheritance, down-casting and cross-border instantiation. It remains unclear how their trace semantics can be applied for verification purposes. Using similar techniques, Steffen [50] and Ábrahám et al. [1] give a fully abstract semantics for a concurrent class-based language (without inheritance and subtyping). In the setting of concurrent data structures, Gotsman and Yang [29] use traces and a most general client to check whether one library linearizes another. Filipovic et al. [25] use a trace abstraction to study whether sequential consistency or linearizability implies observational refinement.

Bisimulations were first used by Hennessy and Milner [30] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [51] and with type abstraction and recursion [52]. Koutavas and Wand, building on their earlier work [40] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [41] in different Java subsets. The subset they considered includes inheritance and down-casting. Their language, however, neither considers interfaces nor accessibility of types.

Our approach We have given a fully abstract trace-based semantics for class libraries of a *sequential* object-oriented programming language with the typical OO features, namely interfaces, classes, inheritance and subtyping [53]. To model encapsulation aspects, we considered a package system which allows package-local types. As our semantics is geared towards practical application, the language is a more faithful subset of Java than Jeffrey and Ratkhe's JavaJr [38]. The main idea behind our fully abstract semantics was to combine characteristics from

- denotational, trace-based semantics, i.e., the mental model of traces which characterize the behavior of a library in terms of its input and output, as well as
- bisimulation approaches, i.e., by providing a strong link to a standard operational semantics.

The idea behind our verification method is based on the states where control is outside of the library, which we call the *observable states*[1]. The (coupling) invariant that relates the configurations of both library implementations must hold at corresponding observable states in the execution. As libraries are comprised of multiple classes, the observable states are, in contrast to [23, 45, 46], not statically bound to program points like start and end of methods.

Our approach is modular in the sense that we do not need knowledge about the contexts of the library. In contrast to Banerjee and Naumann [5], we consider multiple classes and we do not use a special *inv/own* discipline as described in [6], i.e., an explicit representation of when object invariants are known to hold. In our case, it is clear when the invariant must hold, namely in observable states which result directly from the language and module system. On the one hand, we have a less parametric framework than Banerjee and Naumann as the notion of confinement (that results from the encapsulation offered by the module system, in our case Java packages) is fixed by the language semantics. On the other hand, our framework is more powerful as it is not tied to an external confinement discipline. In particular, our reasoning method is complete with respect to contextual equivalence. We allow for example different boundary objects to share their representation (e.g., a list with iterators), which is not possible in [5].

All of the related work that we are aware of present no embedding of their reasoning framework into a mechanized verification framework. The closest we could find was the mechanized bisimulation for the

[1] We use the term *observable states* in allusion to the *visible states* based techniques [23, 45].

nu-calculus by Benton and Koutavas [11] in Coq. We believe that due to the OO setting with nominal type systems, object identity etc., a high degree of automation is achievable. As a proof of concept, we have implemented a fully automatic verifier to check interesting equivalences [9].

Relation to the ABS framework The formal language that served as a basis for developing the reasoning method is closer to the Java language than the ABS language. This is due to the following reasons:

- This research was initiated already at the start of the project, when the ABS language was not yet in a stable state.
- More importantly, building a verifier requires a complex tool chain. As our goal was to target an automatic verifier (Boogie), we were limited by the choices of technologies. More tools were readily available for Java, e.g., the tool B2BPL for translating Java bytecode into Boogie code. Targeting Boogie directly from ABS would have prevented us from getting similar results before the end of the task.

However, we are certain that the results carry directly over to sequential ABS. Although inheritance is not available in ABS, we believe that its effect on the reasoning method is highly relevant when developing a similar reasoning method for ABS with deltas. Similar to inheritance, deltas allow part of the fields and methods of an object to be defined in the library and another part to be defined outside of the library.

We are currently investigating how to connect our tool with the existing ABS tool chain. With the help of the ABS to Java compiler, which translates ABS code in a fairly straightforward way to Java code, we believe that verification of ABS libraries with the given tool is feasible.

Verification conditions In the following, we illustrate the verification conditions that are needed to prove backward compatibility between two implementations of a library using the Cell example. For a more detailed account, we refer to the papers cited at the beginning of this section.

In order to prove backward compatibility, the library developer needs to provide a *coupling relation* that relates the internal state of the first library implementation with the second library implementation, and then prove that this relation has the simulation property. In the following, we illustrate coupling relations using the Cell example in Fig. 2.1. The definition of a coupling relation is also called a *coupling invariant*. An intuitive description of the coupling invariant was already given in Section 2.2. To formally talk about objects which are corresponding to each other in runs of program contexts with the first library implementation and the second library implementation, i.e., objects that assume the same roles in both program runs, we use a *correspondence relation* ρ which is bijective renaming between the objects of the first and the second program configuration that are known to the program context.

The coupling invariant for the Cell example then looks as follows. For all corresponding objects $(o_1, o_2) \in \rho$ that have the dynamic type Cell or a subtype thereof and where the value of the field $o_2.\mathsf{f}$ is true, the values that are stored in the fields $o_1.\mathsf{c}$ and $o_2.\mathsf{c1}$ are either both null or corresponding objects, i.e. $(o_1.\mathsf{c}, o_2.\mathsf{c1}) \in \rho$. Similarly, if the value of the field $o_2.\mathsf{f}$ is false, then $(o_1.\mathsf{c}, o_2.\mathsf{c2}) \in \rho$ or these fields are both null.

We then need to show that the relation induced by the previously described coupling invariant has the simulation property. This means that 1) the property holds between the initial states of the programs, that 2) the property is preserved by corresponding steps in program contexts of the library and that 3) for similar inputs to the library, the libraries react in similar ways and the execution reaches again corresponding states of the program context. As the coupling invariant talks only about the library parts of the configurations, the proof of 1) and 2) is trivial.

The main proof obligation is to show that the simulation property is preserved in the cases where control of execution jumps from code of program contexts to the library. Such a change in control can only happen if program contexts call the methods get or set, or if they create a new object of type Cell or a subtype thereof and the constructor of Cell is executed. We consider similar changes in control in states which are coupled (i.e., where the coupling invariant holds). This means, for example, that we consider a call by program contexts to the method get of the first library implementation and a similar call to the method get of the

second library implementation on corresponding Cell objects (i.e., o_1.get and o_2.get where $(o_1, o_2) \in \rho$). We then have to prove that the states right after the next change in control (i.e., where the control returns from code of the library implementation to code of the program context) are also coupled and the results of both method calls are coupled.

2.4 Outlook

The backward compatibility verifier is still in its early stage of development. Having a tool at hand, however, it is easy to spot the limitations (at the usage as well as at the methodological level of the approach). In the future, we want to address the following topics (for the first two points we want to have results within the timeline of the HATS project):

- Currently the coupling invariant needs to be specified in the input language of the verifier (Boogie). To provide a better user experience, we are currently designing a JML [44] -like specification language to relate two library implementations.
- At the moment, the verifier cannot deal yet with diverging behavior or unbounded internal execution of a library. In order to verify, for example, that a library that uses a while loop to internally iterate over a list of items and another library that uses a recursive method to recurse of the list elements have the same behavior, we need auxiliary relations to connect intermediary states of both library implementations. We are currently generalizing the concept of loop invariants [27] to talk about the states of two libraries.
- We would also like to explore more general notions of refinement, by relating not only class libraries to class libraries, but also more general library models to class library implementations. As library models, we would like to consider class libraries, where we introduce additional statements into the language for specification purposes, for example a non-deterministic choice operator. We could then, for example, try to prove refinement between a library model of the subject-observer pattern allowing the subject to notify the observers in an arbitrary order and a library implementation that notifies the observers in a fixed order.
- We would like extend our method from the sequential to a concurrent setting. Here, the ABS concurrency model of *concurrent object groups* seems to fit perfectly with the existing reasoning method based on observable states. We would also like to study how to connect the given approach with more traditional verification approaches based on specifications (which we believe to be complementary).

Chapter 3

Deductive Compilation of ABS

3.1 Introduction

Most of the techniques for analysis and verification of behavioral properties developed within HATS focus on the ABS model level. ABS models can be compiled to executable code for a number of backends such as Java [19], Maude [19], and Scala. For trustworthy code one has to ensure that the properties that have been established on the ABS modeling level are preserved when compiling or refining the model to executable code.

In this chapter we investigate a deductive compilation approach that refines an ABS model to Java bytecode while provably preserving the ABS model properties. In a first phase the compilation process executes the ABS code *symbolically*. During this phase a logic representation of the symbolic state changes is computed in a form that is basically a single static assignment like representation of the program (relative to a symbolic path condition). The second phase synthesizes the executable code by a bottom-to-top traversal of the expanded symbolic execution tree. Both phases are described uniformly in a program logic framework.

The used program logic is an orthogonal extension of the ABS DL program logic [20] used to specify and verify ABS models. ABS DL features a sequent calculus based on symbolic execution. This allows us to

- keep a high precision description of the symbolic state at each intermediate state of the symbolic execution. We use this information (and additional gathered analysis results) to perform a number of optimizations.

- we can reuse the soundness proof of the program logic to ensure that our approach produces property preserving bytecode (w.r.t. to an underlying runtime environment/library).

The latter property is established by enriching the soundness proof of the logic by proving a bisimulation property between the ABS model and the produced bytecode.

We need only to focus on the sequential core ABS. For the mapping of ABS concepts like concurrent object group (COGs) to the Java world we rely on a Java library providing the necessary runtime environment. Proving that the Java library is correctly implemented is a complementary problem that needs to be addressed.

3.2 Background

3.2.1 ABS Dynamic Logic

ABS dynamic logic (ABS DL) is a sorted first-order logic (plus arithmetic) extended with modalities. For our purpose we need only the box modality $[\cdot]\cdot$ which corresponds to partial correctness.

Let p denote an executable sequence of ABS statements and ϕ any formula in ABS DL, then $[\mathsf{p}]\phi$ is an ABS DL formula. The intuitive meaning of the formula is that *if* p is executed and terminates *then* formula ϕ holds in the reached final state.

Example 1 *An example of an ABS DL formula is*

$$\mathsf{i} \doteq i_0 \land \mathsf{j} \doteq j_0 \rightarrow [\mathsf{i} = \mathsf{i} + \mathsf{j}; \mathsf{j} = \mathsf{i} - \mathsf{j}; \mathsf{i} = \mathsf{j} - \mathsf{i};](\mathsf{i} \doteq j_0 \land \mathsf{j} \doteq i_0)$$

The formula expresses that the enclosed program swaps the content of the program variables i *and* j.

ABS DL denotes a collection of logics where each logic is defined with respect to a context program, i.e., for each ABS program an instance of the logic is created defining the required sorts, etc. Further, the context program declares all available interfaces, classes and methods. Formally, an ABS DL formula is evaluated with respect to an ABS DL Kripke structure. A Kripke structure $K = (\mathcal{D}, S, \rho)$ consists of a first-order structure $\mathcal{D}$ defining the domain D and the interpretation I of state independent function and predicate symbols. A set of states S (a state coincides roughly an assignment of program variables and fields to values of the domain), and a state transition relation ρ define the program semantics. A formula ϕ is satisfiable if there is an ABS DL Kripke structure $\mathcal{K}$, an initial state s_0 and a variable assignment β such that ϕ is evaluated to true ($\mathcal{K}, s, \beta \models \phi$). A formula is valid if it is true for all ABS DL Kripke structures, states and variable assignments. A central concept in ABS DL is the notion of a history which can intuitively be seen as a view on system events like object creation, asynchronous method invocations, etc. The ABS DL calculus is designed to be inherently compositional which intuitively requires the accessible history to be restricted to a local projection. The only way to express certain properties of the environment is by specifying interface invariants which have to be ensured before each suspending event, but can then also be relied upon when resuming the execution. For our purpose this aspect of the logic is only of importance for the overall correctness of the compilation process, but not for the compilation process itself.

The final syntactic construct of ABS DL to be introduced here are *updates* [48]. Updates capture state changes and can be seen as explicit generalized substitutions. An *elementary update*

$$l := r$$

is a pair of a location l and a term r. Its meaning is the same as an assignment. *Elementary updates* can be combined to parallel updates

$$u_1 \parallel u_2$$

There might be a conflict between u_1 and u_2, i.e., the same location is assigned different values. Conflict resolution follows the last-win semantics, i.e., the later assignment (here u_2) overrides the former (here u_1). Updates are applied to formulas or terms, i.e., let u denote an update and ξ a term/formula in ABS DL then $\{u\}\xi$ is a term/formula in ABS DL.

3.2.2 Sequent Calculus

We use a sequent calculus to check an ABS DL formula for validity. In this section we define the basic notions tailored to the needs of the presented work. A *sequent* $\Gamma \Longrightarrow \Delta$ relates two sets of formulas Γ and Δ. Its meaning is equivalent to

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$$

The formulas on the left side of the sequent arrow are called *antecedent* of a sequent, the formulas on the right side are called *succedent* of a sequent.

A *sequent proof* is a tree of which each node is labelled with a sequent and there is a (sequent) rule

$$\mathsf{rule}\ \frac{seq_1 \mid \ldots \mid seq_n}{seq}$$

for each node such that the parent matches the rule's conclusion and the children its premises using the same instantiation like the parent. First-order sequent rules are applied from bottom to top, which means matching its conclusion with the sequent of one of the leaves of a proof tree and then adding the instantiated premises as new children.

The sequent calculus also provides rules for ABS programs. The rule assignment

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathsf{l} := r\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathsf{l} = r; \omega]\phi, \Delta}$$

rewrites the assignment statement between two local variables into an update, $\mathsf{l} := r$, where $\mathcal{U}$ stands for an update that may have been accumulated in previous rule applications. Other assignment rules treat more complex cases like assigning the sum of two expressions to a program variable or field. In these cases, the addition has to be rewritten into its logic counterpart.

The rule for conditional statements

$$\text{conditionalSplit}\ \frac{\Gamma, \{\mathcal{U}\}\mathsf{b} \Longrightarrow \{\mathcal{U}\}[\mathsf{p}; \omega]\phi, \Delta \qquad \Gamma, \{\mathcal{U}\}\neg\mathsf{b} \Longrightarrow \{\mathcal{U}\}[\mathsf{q}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathsf{if}\ (\mathsf{b})\ \{\mathsf{p}\}\ \mathsf{else}\ \{\mathsf{q}\}\ \omega]\phi, \Delta}$$

splits the proof into two branches. The left branch considering the case where the condition is satisfied and the then branch executed. While the right branch treats the complementary case where the symbolic execution continues with the else branch assuming the condition does not hold.

The calculus provides two different kinds of rules to treat loops. The first one realizes—as one would expect from a program interpreter—a simple unwinding of the loop:

$$\text{loopUnwind}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathsf{if}\ (\mathsf{b})\ \{\mathsf{p}'; \mathsf{while}\ (\mathsf{b})\ \{\mathsf{p}\}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathsf{while}\ (\mathsf{b})\ \{\mathsf{p}\}\ \omega]\phi, \Delta}$$

where p' is identical to p except for renaming of the newly declared variables in p to avoid name collisions.

Another way to treat the loop is the loop invariant rule whileInv:

$$\text{whileInv}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta & \text{(init)}\\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\mathsf{b} = \mathsf{TRUE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\mathsf{p}]inv, \Delta & \text{(preserves)}\\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\mathsf{b} = \mathsf{FALSE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\omega]\phi, \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathsf{while}\ (\mathsf{b})\ \{\mathsf{p}\}\ \omega]\phi, \Delta}$$

The loop invariant rule requires the user to provide a sufficiently strong formula *inv* capturing the functionality of the loop. The formula needs to be valid before the loop is executed (init branch) and must not be invalidated by any loop iteration started from a state satisfying the loop condition (preserves branch). Finally, in the third branch the symbolic execution continues with the remaining program after the loop. The update $\{\mathcal{V}_{mod}\}$ is used to erase knowledge about the state which might have changed by the loop. Instead those variables/fields that may have been changed are set to new, unknown values. The only information we have about their current value is given by the loop invariant. This optimization allows us to keep the context information (Γ, Δ) and to minimize the size of the loop invariants considerably.

3.3 Deduction Compilation Rules

While the sequent calculus is applied analytically when used for verification, it is also possible to interpret a proof in a program construction manner. We use the second reading as motivation for our approach to generate Java bytecode on top of a proof attempt of ABS program.

The main idea is to extend the box modality

$$[\,\mathtt{p} \sim \mathtt{bc}_{\mathtt{p}}\,]@(obs, use)$$

to carry additional information. The extension consists of a second compartment for the Java bytecode bc_{p} of the source ABS program p and two additional annotations *obs* and *use* containing sets of locations (program variables, fields, etc.). The location set *obs* keeps track of observable locations by an "outside" entity, while the location set *use* contains those locations that are read in the continuation of the program. Without going into detail, these sets of locations are used to detect unused variable assignments and to eliminate them as soon as possible. We call these modalities, *bisimulation* modalities as p and $\mathsf{bc_p}$ have to be in a bisimulation relation with respect to the postcondition and the set of observable variables.

The general sequent calculus rules for the bisimulation modality are of the following form:

$$\mathsf{ruleName}\ \frac{\Gamma_1 \Longrightarrow \{\mathcal{U}_1\}[\ \mathsf{p_1}\ \sim\ \mathsf{bc_1}\ ]@(obs_1, use_1)\phi_1, \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \{\mathcal{U}_n\}[\ \mathsf{p_n}\ \sim\ \mathsf{bc_n}\ ]@(obs_n, use_n)\phi_n, \Delta_n}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathsf{p}\ \sim\ \mathsf{bc}\ ]@(obs, use)\phi, \Delta}$$

The application of sequent calculus rules for the bisimulation modality consists of two phases.

1. Symbolic execution of ABS source program p. It is performed bottom-up as usual in sequent calculus rules. In addition, the observable location sets obs_i are also propagated since they contain the locations observable by $\mathsf{p_i}$ and ϕ_i, which will be used in the second phase to synthesize the Java bytecode. Normally *obs* could contain the return variables of a method and the locations used in the continuation of the program.

2. We synthesize the target Java bytecode $\mathsf{bc_i}$ and use_i by applying the rules in a top-down manner.

To compile a method `m(args)` (to Java bytecode) we start the proof with a sequent of the form

$$\Longrightarrow pre \rightarrow [\ \texttt{r=m(args)}\ \sim\ \mathsf{bc}\ ]@(\{\mathsf{r}\}, use)\texttt{POST}$$

where *pre* is the precondition of method `m` and `POST` is an unspecified predicate which can neither be proven nor disproved. This allows the easy identification of closed proof branches with infeasible paths and thus sound elimination of dead-code. The variables *bc* and *use* are placeholders for the Java bytecode and the used variable set which is computed in the second phase.

In the second phase, when the program is fully symbolically executed, the Java bytecode is synthesized by "applying" the rules in the opposite direction. This phase starts effectively with nodes where the emptyBox rule has been applied:

$$\mathsf{emptyBox}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}@(obs, use)\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathsf{NOP}\ \sim\ \mathsf{NOP}\ ]@(obs, obs)\phi, \Delta}$$

with NOP denoting the empty program. The rule initializes the variable set *use* to the set of observable variables *obs*. The idea is that *use* keeps track of all variables whose value has (potentially) been read.

We show only some of the bisimulation rules. We use $\overline{p}$ to denote the Java bytecode of p and omit for space reasons the context variables Γ, Δ. The rule

$$\mathsf{assignment}\ \frac{\Longrightarrow \{\mathcal{U}\}\{\mathsf{l} := \mathsf{r}\}[\ \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi}{\begin{array}{ll} \Longrightarrow \{\mathcal{U}\}[\ \mathsf{l} = \mathsf{r}; \omega\ \sim\ \mathsf{load\ r}; \mathsf{store\ l}; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi & \\ \qquad \text{if } l \in use\ \&\ \mathsf{r} := \mathsf{r}' \notin \mathcal{U} & \\ \Longrightarrow \{\mathcal{U}\}[\ \mathsf{l} = \mathsf{r}; \omega\ \sim\ \mathsf{load\ r}'; \mathsf{store\ l}; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r'\})\phi & \\ \qquad \text{if } l \in use\ \&\ \mathsf{r} := \mathsf{r}' \in \mathcal{U} & \\ \Longrightarrow \{\mathcal{U}\}[\ \mathsf{l} = \mathsf{r}; \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi \qquad otherwise & \end{array}}$$

describes the specialization of an assignment statement where one local variable is assigned to another one. The rule has three conclusions of which only one is taken. They differ only in the synthesized program compartments, i.e., in the analyzing (first) phase no ambiguity arises.

If the left side l of the assignment is a location with a (potential) read access before its next re-definition, i.e., $l \in use$, an assignment statement is generated. The *use* set is updated by removing the now re-defined program variable l and adding the program variable r which is read by the assignment. This explains the first of the three conclusions. But we can do better: in case the preceding update contains an elementary update with r as left-hand side and r' as right-hand side, we inline the actual value directly and generate as assignment $l = r'$. The *use* set is updated accordingly[1].

We give a brief example motivating the existence of the second conclusion. Assume we encounter the following sequent:

$$\Longrightarrow \{\ldots \| \mathsf{y} := \mathsf{z}\}\{\mathsf{x} := \mathsf{y}\}[\ \omega \ \sim\ \overline{\omega}\ ]@(obs, \{x, \ldots\})\phi$$

Since x is in the *use* set, an assignment statement has to be generated. Notice that $\mathsf{y} := \mathsf{z}$ occurs in the update u, therefore the assignment is synthesized as $\mathsf{load\ z; store\ x;}$ according to the second case of the assignment rule. The variable x is removed and the variable z is added to the *use* set. We get as result:

$$\Longrightarrow \{\ldots \| \mathsf{y} := \mathsf{z}\}[\mathsf{x} = \mathsf{y}; \omega\ \sim\ \mathsf{load\ z; store\ x;}\overline{\omega}\ ]@(obs,\{z, \ldots\})\phi$$

The third conclusion of the assignment rule is used if the value of l has not been accessed by the remaining program ω and does not generate an assignment at all.

On encountering a conditional statement, symbolic execution splits into two branches, namely the then-branch and else-branch. The compilation of the conditional statement will result in a conditional construct in Java bytecode. The guard is the same as used in the source program, then-branch is the Java bytecode of the source ABS then-branch continued with the rest of the program after the conditional, and the else-branch is analogous to the then-branch.

$$\mathsf{conditionalSplit}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}b \Longrightarrow \{\mathcal{U}\}[\ p;\omega\ \sim\ \overline{p;\omega}\ ]@(obs, use_{p;\omega})\phi, \Delta \\ \Gamma, \{\mathcal{U}\}\neg b \Longrightarrow \{\mathcal{U}\}[\ q;\omega\ \sim\ \overline{q;\omega}\ ]@(obs, use_{q;\omega})\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathsf{if}\ (b)\ \{p\}\ \mathsf{else}\ \{q\};\omega\ \sim\ \begin{array}{l}\texttt{iload b}\\ \texttt{ifeq lelse}\\ \texttt{lthen :}\ \overline{\texttt{p;}\omega}\\ \texttt{lelse :}\ \overline{\texttt{q;}\omega}\end{array}\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi, \Delta}$$

(with b boolean variable.)

Note that the statements following the conditional statement are symbolically executed on both branches. This leads to duplicated code in the Java bytecode, and, potentially to code size duplication at each occurrence of a conditional statement. One note in advance: code duplication can be avoided when applying a similar technique as presented later in connection with the loop translation rule. However, it is noteworthy that the application of this rule might have also advantages: as discussed in [13], symbolic execution and partial evaluation can be interleaved resulting in (considerably) smaller execution trace. Interleaving symbolic execution and partial evaluation is orthogonal to the approach presented here and can be combined easily. In several cases this can lead to different and drastically specialized and therefore smaller versions of the remainder program ω and its specialization $\overline{\omega}$. The *use* set is extended canonically by joining the *use* sets of the different branches and the guard variable.

Synthesizing loops is achieved using one (or a combination) of two approaches: (i) loop unwinding to execute a fixed number of loop iterations and (ii) using the loop invariant rule for loops with no fixed bound:

[1] if r' is an expression all variables occurring in r' have to be added to *use*

$$\text{whileInv}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\mathsf{b} = \mathsf{TRUE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\ \mathsf{p}\ \sim\ \overline{\mathsf{p}}\ ]@(obs \cup use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\mathsf{b} = \mathsf{FALSE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\ \omega\ \sim\ \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathsf{while(b)\{p\}}\,\omega\ \sim \\ \qquad\end{array}\begin{array}{l}\texttt{iload b} \\ \texttt{guard}:\texttt{ifeq exit} \\ \texttt{body}:\ \overline{\mathsf{p}} \\ \texttt{goto guard} \\ \texttt{exit}:\ \overline{\omega}\end{array}\begin{array}{l}\\ \\ \\ \\ \\ ]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta\end{array}}$$

to generate Java bytecode in finite time.

On the logical side the loop invariant rule is as expected and has three premises. Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use *true* as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onwards (if *true* is used as invariant then it holds trivially).

Two things are of importance: the third premise executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, we must add to the variable set *obs* of the second premise the used variables of the third premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the *use case* premise of the loop invariant rule and then using the resulting use_1 set in the second premise.

Using the loop invariant rule here allows us to compile the loop body and the program after the loop in separation. In particular, we can keep some information about the symbolic state which improves our static analysis capabilities after a loop and allows us more optimizations as when using other solutions.

For asynchronous method calls, creation of new COGs or await statements, we rely on the runtime library to faithfully implement all ABS concepts in terms of Java. The extension to the ABS rules for these cases are rather simple and can be reduced to the production of the corresponding method invocation primitives of the Java bytecode language for the library methods. The complications with anonymizing the history on points of control releases and similar are already treated by ABS DL. The produced additional branches to ensure that interface and class invariants have been established are necessary for the actual verification, but do not add any additional complication to the compilation as they are first-order only proof goals not involving any programs that need symbolic execution.

3.4 Example

We demonstrate the deductive compilation approach on a small example. The ABS method to be compiled is shown in Fig. 3.1. The program basically computes the total price of a collection of items. The price per item is discounted when buying more than two of them and the buyer is in possession of a coupon. It consists of a loop iterating over the items and the loop body itself contains a conditional statement checking if the reduced or normal price of the item should be applied.

The first phase of our approach starts symbolically executing the ABS code with the return value tot as the only observable location, i.e., $obs = \{\mathsf{tot}\}$. The first statements of the method declare and initialize variables. These statements are executed similar to assignments. Altogether the assignment rule is applied three times, where each assignment rule application is immediatley followed by a partial evaluation step. We end up with

```
Int i = 0;
Int count = n;
Int tot = 0;
while(i <= count) {
 if (i>=2 && cpn)
        tot = tot + m * 9 / 10;
 else
        tot = tot + m;
 i++;
}
return tot;
```

Figure 3.1: ABS source program (cpn is field of type Bool).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Longrightarrow \{\ldots||\mathsf{tot} := 0\}[\ \mathsf{while(i <= n)}\ldots \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)}}{\Longrightarrow \{\ldots||\mathsf{count} := \mathsf{n}\}[\ \mathsf{tot} = 0; \mathsf{while(i <= n)}\ldots \sim bc_2\ ]@(\{\mathsf{tot}\}, use_2)}}{\Longrightarrow \{\mathsf{i} := 0\}[\ \mathsf{count} = \mathsf{n}; \ldots \sim bc_1\ ]@(\{\mathsf{tot}\}, use_1)}}{\Longrightarrow [\ \mathsf{i} = 0; \ldots \sim bc_0\ ]@(\{\mathsf{tot}\}, use_0)}$$

where bc_i denotes the corresponding specialized program.

The next statement to be symbolically executed is the while loop computing the total sum. Instead of immediately applying the loop invariant rule, we unwind the loop once using the loopUnwind rule. Partial evaluation allows to simplify the guard i <= n and i >= 2 && cpn of the introduced conditional to i <= 2 and 0 >= 2 && cpn by applying constant propagation. Furthermore, the then-branch is eliminated because the guard 0 >= 2 && cpn can be evaluated to *false*. The result is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\Longrightarrow \{\mathsf{i} := 0||\ldots||\mathsf{tot} := 0\} \\ \quad [\ \mathsf{if(0 <= n)\{tot = m; i = 1; while\ldots\}} \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)\end{array}}}{\begin{array}{l}\Longrightarrow \{\mathsf{i} := 0||\ldots; \mathsf{tot} := 0\} \\ \quad [\ \mathsf{if(0 <= n)\{\ldots tot = 0 + m; i = 1; while\ldots\}} \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)\end{array}}}{\begin{array}{l}\Longrightarrow \{\mathsf{i} := 0||\ldots\} \\ \quad [\ \mathsf{if(0 <= n)\{\ldots if(0 >= 2\ \&\&\ cpn)\ldots; i = 0 + 1; while\ldots\}} \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)\end{array}}}{\begin{array}{l}\Longrightarrow \{\mathsf{i} := 0||\ldots\} \\ \quad [\ \mathsf{if(i <= n)\{\ldots if(i >= 2\ \&\&\ cpn)\ldots; i + +; while\ldots\}} \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)\end{array}}}{\Longrightarrow \{\mathsf{i} := 0||\ldots||\mathsf{tot} := 0\}[\ \mathsf{while(i <= n)}\ldots \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)}$$

Application of the conditionalSplit rule creates two branches. The else-branch contains no program so it is synthesized right away by applying the emptyBox rule. Symbolic execution of the then-branch, applies the assignment rule three times until we reach the while loop again. We decide to unwind the loop a second time. The symbolic execution follows then the same pattern as before until we reach the loop for a third time.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1 \leq \mathsf{n} \Longrightarrow \{\ldots \| \mathsf{i} := 2\}[\ \mathsf{while(i <= n)} \ldots \sim bc_6\ ]@(\{\mathsf{tot}\}, use_6)}{\ldots}}{0 \leq \mathsf{n} \Longrightarrow \{\ldots\}[\ \mathsf{if(i <= n)} \ldots ; \mathsf{while} \ldots \sim bc_5\ ]@(\{\mathsf{tot}\}, use_5)}}{0 \leq \mathsf{n} \Longrightarrow \{\ldots \| \mathsf{tot} := \mathsf{m} \| \mathsf{i} := 1\}[\ \mathsf{while(i <= n)} \ldots \sim bc_5\ ]@(\{\mathsf{tot}\}, use_5)} \quad \dfrac{}{\ldots}}{\neg(0 \leq \mathsf{n}) \Longrightarrow \{\ldots\}[\ \sim \mathsf{NOP}\ ]@(\{\mathsf{tot}\}, \{\mathsf{tot}\}) \qquad 0 \leq \mathsf{n} \Longrightarrow \{\ldots\}[\ \mathsf{tot} = \mathsf{m}; \ldots \sim bc_4\ ]@(\{\mathsf{tot}\}, use_4)}}{\Longrightarrow \{\ldots\}[\ \mathsf{if(0 <= n)\{tot = m; i = 1; while} \ldots\} \sim bc_3\ ]@(\{\mathsf{tot}\}, use_3)}$$

Instead of unwinding the loop once more, we apply the loop invariant rule whileInv. The rule creates three new goals. The goal for the init premise is not of importance for the specialization itself, hence, we ignore it in the following.

The *used variables* set *use* of the preserves premise depends on the instantiation of the *use* set in the use case premise. To resolve the dependency we continue with the latter. In this case, the use case premise contains no program, so it is trivially synthesized by applying the emptyBox rule which results in nop as the specialized program and the only element tot in *obs* becomes the *use* set. Based on this, the *use* set of the preserves premise is the union of *obs*, {tot} and the locations used in the loop guard: {tot, i}. The program in the preserves premise is then symbolically executed by applying suitable rules until it is empty. This process is similar to that when executing the program in the then-branch of the conditional generated by loopUnwind. The proof tree resulting from the application of the loop invariant rule is shown as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots, \mathsf{cpn} \Longrightarrow \ldots[\ \ldots \sim bc_8\ ]@(\{\mathsf{tot}\} \cup \{\mathsf{i}\}, use_8)} \quad \dfrac{}{\ldots, \neg\mathsf{cpn} \Longrightarrow \{\ldots\}[\ \mathsf{tot} = \mathsf{tot} + \mathsf{m}; \ldots \sim bc_9\ ]@(\{\mathsf{tot}\} \cup \{\mathsf{i}\}, use_9)}}{\ldots \Longrightarrow \{\ldots \| \mathsf{tot} := \mathsf{tot} + \mathsf{m}\}[\ \mathsf{i}{+}{+}; \sim bc_{10}\ ]@(\{\mathsf{tot}\} \cup \{\mathsf{i}\}, use_{10})}}{\ldots \Longrightarrow \{\ldots \| \mathsf{i} := \mathsf{i} + 1\}[\ \sim bc_{11}\ ]@(\{\mathsf{tot}\} \cup \{\mathsf{i}\}, use_{11})}}{\ldots, \neg(\mathsf{i} \leq \mathsf{n}) \Longrightarrow [\ \sim \mathsf{nop}\ ]@(\{\mathsf{tot}\}, \{\mathsf{tot}\}) \qquad \ldots, \mathsf{i} \leq \mathsf{n} \Longrightarrow [\ \mathsf{if(cpn)}\ \ldots \sim bc_7\ ]@(\{\mathsf{tot}\} \cup \{\mathsf{i}\} \cup \{\mathsf{tot}\}, use_7)}}{1 \leq \mathsf{n} \Longrightarrow \{\ldots \| \mathsf{i} := 2\}[\ \mathsf{while(i <= n)} \ldots \sim bc_6\ ]@(\{\mathsf{tot}\}, use_6)}$$

After symbolic execution we enter the second phase of our approach in which the Java bytecode is synthesized. Recall that when applying the whileInv rule, the procedure of synthesizing the loop starts with the use case branch. In our example, we have already performed this step and could already determine the instantiation of the observable location set *obs* of the preserves premise.

Following the symbolic execution tree backwards and applying the corresponding rules, we finally synthesize the ABS bytecode as follows:

```
iconst_0    istore i
iconst_0    istore tot
iconst_0    iload  n    if_icmpgt goto ret
iload m     istore tot
iconst_1    iload  n    if_icmpgt goto ret
iload tot   iload  m    iadd    istore tot
iconst_2    istore i
loop:
iload i     iload n     if_icmpgt goto ret
aload cpn   getfield #2; //Field cpn:Z
ifeq goto lelse
lthen:
 iload tot   iload m  bipush 9 imul
 bipush 10   idiv
 iadd        istore tot goto ifexit
lelse:
```

```
 iload tot    iload m  iadd  istore tot
ifexit:
 iload i      iload_1  iadd  istore i
 goto loop
ret:
 iload tot
 ireturn
```

The compiled program is optimized as it simplifies the condition of the **if** statement in the loop body removing the check for the value of i which is at that time known to hold. The number of loop unwindings can be obtained heuristically by analyzing comparisons of variables modified by the loop with constants inside the loop body.

3.5 Conclusion

The presented deductive compilation approach guarantees a sound refinement from the ABS program into executable Java bytecode. Extending the approach to include concurrent constructs naïvely should be straightforward. Basically, the translation needs to map asynchronous method calls, await statements, etc., to corresponding Java library methods. Correct blocking of threads and prevention of reentrance in the asynchronous case have to be ensured by introducing and acquiring locks in the Java bytecode translation.

Our previous work [14, 39] focus on generating Java source code from Java source code, thus realized a Java program specializer. Although the working languages are different, both approaches are based on the same theoretical foundation. Our approach is closely related to rule-based compilation [3, 12]. Our approach differs in the sense that to the best of our knowledge their inference machine is by far not as powerful as here. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [8]) which also use a theorem prover to discharge proof obligations. These work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates the actual strategy from the translation rules. Our work is also related to the *Verifying Compiler* [31] project which aims at the development of a compiler that verifies the program during compilation.

The close connection between the program logic and the compilation allows to ensure the correctness of the compilation process as such. We see a great potential of our approach when encoding security or safety properties in terms of postconditions. This should allow to identify unsafe or unsecured execution paths during compilation and either to abort compilation or to wrap the undesired execution paths in a wrapper that at least ensures the safety or security property of interest. For example, execution paths that may leak information can be secured by omitting the assignments which violate secure information flow. Another possibility would be to ensure that if the program enters an unsecured execution path, then the program will not terminate. Exploring these avenues is future work.

Chapter 4

Real-Time Concurrent Objects

4.1 Introduction

The work described in this chapter aims at compositional schedulability analysis of multiple-processor distributed systems specified with concurrent objects in ABS; a real-time system is schedulable if it can finish all of its tasks within their deadlines. We use timed I/O automata to model the real-time behavior of concurrent objects at an abstract level, as in our previous work [35]. Automata theory provides a rich basis for analysis; nevertheless, we need compositional techniques to overcome the complexity of large asynchronous distributed systems.

While an object comprises a queue, a scheduling policy, and several methods and is thus modeled in several automata, the abstract behavior of the object is given in one automaton, called the behavioral interface. A behavioral interface specifies at a high level and in the most general terms how an object may be used; thus it is used as the key to compositional analysis. Each object is analyzed individually for schedulability with respect to its behavioral interface. As in modular verification [43], which is based on assume-guarantee reasoning, individually schedulable objects can be used in systems *compatible* with their behavioral interfaces. The schedulability of such systems is then guaranteed [36]. This is illustrated in Fig. 4.1.

In interface-based design, refinement is usually used as the means for compositional analysis. Given a set of components C_j with interfaces I_j, C_j is considered a correct implementation if it refines I_j. Then ideally, when the interfaces are *compatible* their implementations should also be able to work together. To capture all incompatibilities, any behavior not allowed in the interfaces should lead to an error (e.g., [17], cf. related work); however, this is too restrictive in practice because interfaces are abstract and easily produce spurious counterexamples to compatibility. An optimistic approach (e.g., [18]) considers two interfaces compatible if there exists a common behavior that allows them to work together. This is useful if we can make sure the implementation of evey components indeed follows this common behavior. We formalized this last step in [36] by requiring the composition of the components $C = \|_j C_j$ to be a refinement of $I = \|_j I_j$.

First in this chapter, as reported in [33], we give a compositional solution to checking the refinement between $C = \|_j C_j$ and $I = \|_j I_j$. The idea is that the outputs of each component C_j should be expected as an input by the interface of the receiving component; this is formalized as every C_j being a refinement of I. Traditional views on refinement do not allow this relation because C_j and I have incompatible sets of inputs and outputs. We generalize refinement such that it considers the common set of actions as the observable behavior. Thus, I is comparable to each C_j with respect to the inputs and outputs of C_j.

Next, we describe a counter-example oriented method based on refinement for testing compatibility of real-time concurrent objects (reported in the submitted article [34]). The above compositional verification is much more practical than the pessimistic approach of [17], but still it may miss some cases. Next to verifying the refinement relation, we introduced in [34] a novel method for *counter-example oriented* testing. Consider the system illustrated in Fig. 4.1. In this method, we generate a test case from B as follows. We take a trace from B and complete it into a test case for S by adding transitions that capture all possible

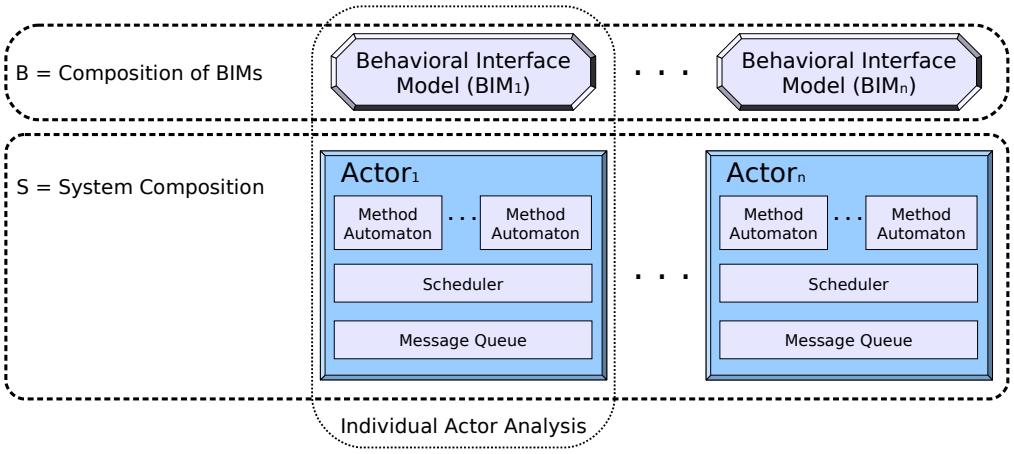

Figure 4.1: An off-the-shelf actor component is guaranteed to be schedulable if it is used as expected in its behavioral interface. This correct usage in a system S can be tested, called the compatibility check.

one-step deviations from the original trace. Among these transitions, those not allowed in B produce a counter-example, i.e., *a trace of S which does not belong to B*. This technique is much more effective than generating test cases from S to be checked against B, because it allows for automated generation of test cases from B (note that B does not involve queues) and a reduction of the overall system behavior S by the test case.

The final part of this chapter covers analysis of real-time software families. In order to generalize the above framework to actor-based real-time software product lines we introduce variability into both the detailed actor models and their behavioral interfaces by specifying behavior (including scheduling policies) at both levels conditionally on the set of selected features. As a general model of behavior, we introduce the notion of a *timed automata family (TAF)* as an extension of timed automata. Every action and edge in a timed automata family is supplied with an *application condition (AC)*, defined as a boolean formula over the set of features, determining under what conditions the action/edge is applicable. By (de)selecting each feature, some application conditions become unsatisfiable. One can (partially) configure a TAF by explicitly (de)selecting some features, formally resulting in the removal of the edges and actions with an unsatisfiable AC. Removal of an action causes the elimination of all corresponding edges. We provide in [49] a detailed account of how to use the TAF theory for compositional schedulability analysis of real-time software families.

4.2 Compositional Refinement Checking

Two timed I/O automata A and B are traditionally (e.g., in [17]) considered comparable if they have the same sets of input and output actions, i.e., $\Sigma_A^I = \Sigma_B^I$ and $\Sigma_A^O = \Sigma_B^O$. Since we want to consider refinement in the context of composition where inputs and outputs may need to be compared to internal actions (which are in turn the result of synchronization), we need to be more liberal with the relation of the action sets. We say the timed I/O automata A and B are comparable for the relation $A \sqsubseteq B$ if:

$$\Sigma_A^I \subseteq \Sigma_B^I \cup \Sigma_B^\tau \quad \wedge \quad \Sigma_A^O \subseteq \Sigma_B^O \cup \Sigma_B^\tau \quad \wedge \quad \Sigma_A^\tau \cap (\Sigma_B^I \cup \Sigma_B^O) = \emptyset$$

In refinement between timed I/O automata, inputs and outputs are treated differently, as in alternating refinement [2]. Intuitively, when A refines B, the refined model A must accept any input that is acceptable in B; and, A may produce an output only if it is allowed at the abstract level B (e.g., see Fig. 4.2).

As timed I/O automata are used in component-based design, we would like to be able to use them in compositional analysis, too. Given $A_1 \sqsubseteq B_1$ and $A_2 \sqsubseteq B_2$, B_1 and B_2 could be abstract interfaces for components A_1 and A_2, and one might expect that $A_1 \parallel A_2 \sqsubseteq B_1 \parallel B_2$. Such a compositional reasoning

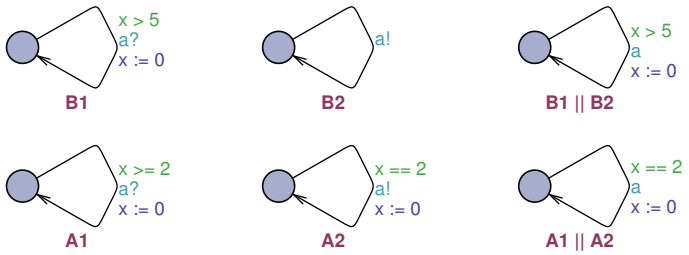

Figure 4.2: Composition and refinement: $A_1 \sqsubseteq B_1$ and $A_2 \sqsubseteq B_2$ but $A_1 \parallel A_2 \not\sqsubseteq B_1 \parallel B_2$. This result is expected because $A_1 \sqsubseteq B_1 \parallel B_2$ but $A_2 \not\sqsubseteq B_1 \parallel B_2$.

does not hold for timed I/O automata; this is illustrated in the counterexample in Fig. 4.2. In this example, A_1 admits the input $a?$ in a bigger time interval than B_1; it becomes an internal action after synchronization with $a!$ in A_2, this action would exist in $A_1 \parallel A_2$ but not necessarily in $B_1 \parallel B_2$. To compositionally infer $A_1 \parallel A_2 \sqsubseteq B_1 \parallel B_2$, we suggest an extra sufficient step $A_1 \sqsubseteq B_1 \parallel B_2$ and $A_2 \sqsubseteq B_1 \parallel B_2$. Here, we give a definition for refinement of timed I/O automata that supports such compositional analysis.

Definition 4.2.1 (TIOA Refinement) *Given two comparable timed I/O automata A and B, we say A refines B iff there is a relation R between their underlying transition systems such that $(s_0^A, s_0^B) \in R$, and if $(s,t) \in R$*

- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a}_B t'$, then $s \xrightarrow{a}_A s'$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^O \cup (\Sigma_A^\tau \cap \Sigma_B)$, if $s \xrightarrow{a}_A s'$, then $t \xrightarrow{a}_B t'$ and $(s',t') \in R$;*
- *if A can delay: $s \xrightarrow{d}_A s'$ then B can also delay: $t \xrightarrow{d}_B t'$ and $(s',t') \in R$.*

It is easy to see that when A and B are normal timed automata, i.e., the input and output action sets are empty. Furthermore, we do not require any direct relation between the inputs (resp. outputs) of A and B; we may compare inputs or outputs of A with internal actions of B. Thus we can compare arbitrary automata which helps us check refinement in a compositional way, described below.

Theorem 4.2.2 *Given the timed I/O automata A_1, A_2 and B, we have:*

$$A_1 \sqsubseteq B \ \wedge \ A_2 \sqsubseteq B \implies A_1 \parallel A_2 \sqsubseteq B$$

4.2.1 Deadlines in Refinement

In this section, we describe how to consider deadlines in refinement. We add deadlines as parameters to actions: A deadline on an output specifies when the task is required to finish. A deadline on an input specifies the guaranteed time before which the task is finished. Usually parameters are not included in the theory of refinement; instead, they are handled by expansion, i.e., an action is expanded to several actions considering different valuations of the parameter. Deadline parameters cannot be treated by expanding. A component may require weaker deadlines than its interface on the outputs and provide stronger guarantees for the inputs. We redefine refinement giving deadlines this special treatment.

A deadline specifies the time before which a task must be done. A common property to check for real-time systems is schedulability, i.e., whether all tasks finish within their deadlines. We associate a *relative* deadline $d \in \mathbb{N}$ to input and output actions, i.e., the deadline is d time units after the action is taken. The interpretation of a deadline depends on the action type:

- An automaton with an input action $a(d)?$ guarantees the deadline d; therefore, it naturally also guarantees $d+1$.

- An output action $a(d)!$ requires a deadline d; naturally, a deadline d is a stronger requirement than $d+1$.

At the lowest level of abstraction, the tasks are implemented and one needs to check whether they indeed meet their deadlines, as explained in [36, 34].

Composition In presence of deadlines, we restrict composition of timed I/O automata by allowing only compatible actions to synchronize; two actions $a(d)?$ and $a(d')!$ are compatible if $d \leq d'$, i.e., the required deadline is not stronger than the guaranteed one. As a result of this synchronization, the composed automaton will have an internal action $a(d, d')$. A deadline interval $[d..d']$ associated to an internal action a is stronger than $[\delta..\delta']$ if the interval $[d..d']$ is included in $[\delta..\delta']$, i.e., $\delta \leq d$ and $d' \leq \delta'$. When two transitions have a sync action as input and output with incompatible deadlines, they do not synchronize and they do not appear in the composed automaton.

Refinement When considering deadlines in refinement, the refined model must provide the same (or stronger) deadline guarantees on its inputs compared to the abstract model; the refined model may not require stronger deadlines on its outputs than the abstract model. A common internal action cannot have a stronger deadline interval than the abstract one. Below,we extend the usual definition of refinement to include deadlines with the abovementioned considerations. Taking deadline intervals for internal actions makes this definition of refinement transitive, i.e., given $A \sqsubseteq B$ and $B \sqsubseteq C$ we have $A \sqsubseteq C$.

Definition 4.2.3 (Refinement with Deadlines) *Given two comparable timed I/O automata, we say A refines B iff there is a relation R between their underlying transition systems such that $(s_0^A, s_0^B) \in R$, and if $(s,t) \in R$*

- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a(d)?}_B t'$ then $s \xrightarrow{a(\delta)?}_A s'$ with $d \geq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a(d,d')}_B t'$ then $s \xrightarrow{a(\delta)?}_A s'$ with $d \geq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^O$, if $s \xrightarrow{a(\delta)!}_A s'$ then*
  - *$t \xrightarrow{a(d)!}_B t'$ with $d \leq \delta$ and $(s',t') \in R$; or,*
  - *$t \xrightarrow{a(d,d')}_B t'$ with $d' \leq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^\tau \cap \Sigma_B$, if $s \xrightarrow{a(\delta,\delta')}_A s'$ then $t \xrightarrow{a(d,d')}_B t'$ and $\delta \leq d$ and $d' \leq \delta'$ and $(s',t') \in R$;*
- *if A can delay: $s \xrightarrow{d}_A s'$ then B can also delay: $t \xrightarrow{d}_B t'$ and $(s',t') \in R$.*

We have shown in [33] that Theorem 4.2.2 still holds when including deadlines in the definition of refinement.

4.3 Testing Refinement and Compatibility Checking

Consider the system of actors in Fig. 4.1. Once an actor is proved to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. A system composed of individually schedulable actors is itself schedulable if the actual use of the actors in this system is compatible with their behavioral interfaces. For each actor, the behavioral interface abstractly models its observable behavior in terms of the messages it may receive and the messages it sends. Assuming that B is deterministic, in theory one can prove that S is refinement of B by model-checking the synchronous product of S and B (restricted by the computed queue bounds). Due to the message queues (one for each actor) in S, this would lead however to an unmanageable state-space explosion.

Inputs $B = (L_B, l_{0_B}, E_B, I_B)$: A timed automaton specifying the abstract behavior
$T = (L_T, l_0, E_T, I_T)$ such that $L_T \subseteq L_B$ in linear form

Output $TC = (L_T \cup \{f\}, l_0, E_C, TR)$: A timed automaton representing the test case where TR is a function that always returns *true* as location invariant

Constructing the transitions:

$E_C := \{\ \}$
for each $i \in [0\,..\,n-1]$ **do**
  $h_i := I_B(l_i)$ // the invariant of l_i in B
  $E_C := E_C \cup \{l_i \xrightarrow{\neg h_i, \tau, \emptyset} f\}$
  $E_C := E_C \cup \{l_i \xrightarrow{g_i \wedge h_i \wedge (d \geq d_i), m_i(d), r_i} l_{i+1}\}$
  for each action $m \in \Sigma_B$ **do**
    $g_f := \mathit{false}$
    for each transition $l_i \xrightarrow{g, m(d'), r} l' \in E_B$ **do** $g_f := g_f \vee (g \wedge d \geq d')$ **endfor**
    $E_C := E_C \cup \{l_i \xrightarrow{h_i \wedge \neg g_f, m(d), \emptyset} f\}$
  endfor
endfor

Verdicts:

Label l_n with the verdict **Pass**
Label f with the verdict **Fail**

Figure 4.3: Test case generation algorithm

In [36], we have defined compatibility in terms of refinement: a closed system made up of individually schedulable objects is schedulable if it is a refinement of the composition of the behavioral interfaces. With our general definition of refinement in previous section, we can apply our method in *open systems* of multiple concurrent objects, too. The behavioral interface of the composite open system is the composition of the behavioral interfaces of individual objects.

Definition 4.3.1 (Compatibility) *The system S is compatible with the behavioral interfaces if and only if $S \sqsubseteq B$ where B is the synchronous product of the behavioral interfaces.*

The above compositional verification is much more practical than the pessimistic approach of [17], but still it may miss some cases. Next to verifying the refinement relation, we introduced in [34] a novel method for *counter-example oriented* testing.

4.3.1 Generating a test case

The idea is to take a timed trace from the abstract specification B and turn it into a test case. Since the exact timing of actions in a timed trace make the test case too restrictive, we take instead a linear timed automaton T. This consists of a sequence of transitions from B representing a set of timed traces, but correspondig to exactly one untimed trace. As shown in Fig. 4.3, T contains a sequence of transitions written as $l_{i-1} \xrightarrow{g_i, m_i, r_i} l_i$, $1 \leq i \leq n$. Such a sequence abstractly represents a desired system behavior (the test purpose).

The sequence of transitions of T corresponds to the behavior we want to test so the last location must be labeled **Pass**. All other locations are completed as follows, such that any forbidden behavior makes the test fail. If a location has an invariant h_i in B, violating this invariant must make the test fail; thus, a transition labeled with τ and with guard $\neg h_i$ leading to **Fail** is added. Furthermore, no other transition may be taken if the invariant is violated; this is ensured by conjunction of guards of all other transitions with h_i. Additionally, every behavior which is not allowed in B is forbidden, so for every action, a transition labeled by this action and whose guard is the complement of all the existing guards for this action leads

to a **Fail** location; this guard is computed in g_f. Any trace leading to **Fail** is an example of behavior not allowed in the abstract behavior specification.

A test case must drive the execution of the system such that actions happen in the specified order. Usually, this happens by feeding inputs to the system and observing the outputs. In our case, we deal with a closed system which has no inputs to be controlled. Instead, since we deal with a system *model*, we drive the system execution by making it synchronize with the test case. Formally, the execution of a test case on the system is defined as the parallel composition of the automata of the test case and the system, synchronizing on the same actions. We denote the product automaton by $TC \| MUT$.

The model under test *passes* the test, denoted by MUT passes TC, if and only if the **Fail** location is not reachable in the product $TC \| MUT$. A test set $\mathcal{T}$ being a set of test cases, the model under test passes $\mathcal{T}$, denoted by MUT passes $\mathcal{T}$, if and only if for all test cases TC in $\mathcal{T}$, MUT passes TC.

Soundness

The soundness requirement for a test set states that it must not reject a correct refinement. In other words, any counter-example reported by a test case (a trace leading to the **Fail** verdict) should indeed violate the refinement. A test case is formally defined to be *sound* (or unbiased) for the refinement relation $\sqsubseteq$ if and only if

$$MUT \sqsubseteq B \implies MUT \text{ passes } TC$$

A test set $\mathcal{T}$ is sound if and only if all test cases in $\mathcal{T}$ are sound.

Theorem 4.3.2 (Soundness) *Let B be a deterministic timed automaton and T be a linear timed automaton built from a sequence of transitions in B. The test case TC generated from T and B by the algorithm in Fig. 4.3 is* sound *for $\sqsubseteq$.*

Exhaustiveness

Soundness is not sufficient to ensure the relevance of test cases. A test case with no **Fail** location is sound but cannot reject any system. We also need to be sure that if the system is a wrong refinement, there exists a test case able to reject it. An exhaustive test set rejects any wrong refinement in the system. In other words, any system that passes the test set is a correct refinement of the specification. A test set is formally defined to be *exhaustive* for the refinement relation $\sqsubseteq$ if and only if

$$MUT \text{ passes } \mathcal{T} \implies MUT \sqsubseteq B$$

Theorem 4.3.3 (Exhaustiveness) *The set of all test cases for B that can be generated by the algorithm in Fig. 4.3 is* exhaustive *for $\sqsubseteq$.*

Rigidness

We are interested in test cases that reject models which behave in a wrong way *along the test case*: the test case should not say **Pass** if it is possible to detect something wrong during the test case execution. We show that any test case generated by our algorithm can detect every wrong behavior occurring along it. We can actually show that we can provide a counter-example for any incorrect refinement occurring along the sequence of transitions the test case is built from. Given a trace $\sigma \in \mathit{Traces}(B)$, suppose the action or delay $e \in \Sigma \cup \{\delta\} \cup \mathbb{R}_+$ is allowed after σ in MUT but not in B, i.e., $\sigma.e \in \mathit{Traces}_{obs}(MUT) \setminus \mathit{Traces}(B)$. A test case TC is rigid for the refinement relation $\sqsubseteq$ if and only if it rejects any incorrect refinement along the traces of the test case:

$$\sigma \in \mathit{Traces}(TC) \wedge l_0 \xrightarrow{\sigma} l_i, 1 \leq i < n \Rightarrow \sigma.e \in \mathit{Traces}(TC) \wedge l_0 \xrightarrow{\sigma.e} \textbf{Fail}$$

Intuitively, if σ ends in a non-leaf location in TC, the test case TC will observe any one-step divergence after σ. This notion is close to the notions of non-laxness in the untimed setting [37] and of strictness in the

timed setting [42] but it is stronger. These notions state that if the system behaves in a non-conforming way during the execution of the test case, it must be rejected. Also in our framework, every detected divergence leads to the rejection of the system, but we can add that every divergence is actually detected. This result directly follows from the construction of the test case.

Theorem 4.3.4 (Rigidness) *Let B be a deterministic timed automaton and T be a linear timed automaton built from a sequence of transitions in B. The test case for B generated from T by the algorithm in Fig. 4.3 is* rigid *for $\sqsubseteq$.*

4.4 Families of Timed Automata

Introducing the notion of variability in timed automata enables us to represent families of real-time objects. In this section, we describe the syntax and the semantics of *timed automata family* based on the notion of timed automata.

A timed automata family (TAF) is syntactically a timed automata in which each action and edge is given an application condition. An application condition of an action/edge indicates the set of products that include that action/edge. An action/edge that is included in all products is annotated with *true*.

Definition 4.4.1 (Timed Automata Family) *A timed automata family over feature set $\mathcal{F}$, actions Σ, and clocks X is a tuple $\langle L, l_0, \mathcal{T}, I, \Gamma \rangle$ where*

- *L is a finite set of locations*
- *$l_0 \in L$ is the initial location*
- *$\mathcal{T} \subseteq L \times \mathcal{B}(X) \times \Phi_F \times \Sigma \times 2^X \times L$ is a set of edges*
- *$I : L \mapsto \mathcal{B}(X)$ is a function that assigns an invariant to each location*
- *$\Gamma : \Sigma \mapsto \Phi_F$ is a function that associates an application condition to each action* □

As before, $\mathcal{B}(X)$ is the set of all clock constraints and Φ_F is the set of all application conditions. The function Γ associates application conditions to actions. For example, in an elevator with four floors, the action $req(5)$ does not exist. Every edge in a TAF also has an application condition $\varphi \in \Phi_F$. The idea is that such an edge exists in a given configuration c if $\Gamma(a)|_c \wedge \varphi|_c$ is satisfiable, where a is the action on this edge.

4.4.1 Projection and Refinement

Having a timed automata family T, we can project it on a (partial) configuration c and obtain another timed automata family $T|_c$. Projecting on a decided configuration results in a specific product. Projection can be done *statically* by checking the satisfiability of the application conditions of actions and edges in T based on configuration c. Then, we eliminate those actions and edges whose corresponding application condition is not satisfiable.

Definition 4.4.2 (TAF Projection) *Projection of a timed automata family T on a configuration c leads to another timed automata family with the following action set and transition set:*

- *$\Sigma' = \{a \mid a \in \Sigma \ \wedge \ \mathcal{S}(\Gamma(a)|_c) = true\}$*
- *$\mathcal{T}' = \{t : (l, g, \varphi, a, r, l') \mid t \in \mathcal{T} \wedge \mathcal{S}(\varphi(t)|_c \wedge \Gamma(a)|_c) = true\}$* □

As mentioned above, projection is a syntactic process. To preserve the feature selection by c at the semantic level, the semantics of the projection of T on c would be $[\![T|_c]\!]_{\psi|_c}$, where ψ represents the feature model of T.

By contrast to projection, we define a refinement process at the semantic level. We will show that these two notions coincide. In other words, a refined TAF corresponds to a smaller set of configurations.

Definition 4.4.3 (TAF Refinement) *Given two timed automata families T_A and T_B, we say T_A refines T_B, denoted $T_A \sqsubseteq T_B$, if and only if there is a relation $\mathcal{R}$ between the states of their underlying transition systems such that*

- $\mathcal{R}(s_{A_0}, s_{B_0})$*; and,*
- *if $\mathcal{R}(s_A, s_B)$ and $s_A \to s'_A$ then there exists $s_B \to s'_B$ and $\mathcal{R}(s'_A, s'_B)$; and,*
- *for every $\mathcal{R}((l, u, p), (l', u', p'))$ we have $p \implies p'$.*

□

In this definition, $s \to s'$ represents both action and delay transitions.

Theorem 4.4.4 *Given an initial constraint ψ, the projection of a timed automata family T over a configuration c refines T, formally written as: $[\![T|_c]\!]_{\psi|_c} \sqsubseteq [\![T]\!]_\psi$* □

4.4.2 More Semantical Properties

To be able to further argue about the semantical properties of timed automata families, we define a similar projection operator on the transition systems. Intuitively, given a configuration c, this operator removes those states that are not compatible with c.

Definition 4.4.5 (Transition System Projection) *Consider a timed transition system TS with states St and transitions T. The projection of TS over a configuration c is another transition system with the set of states $St' = \{(l, u, p) \mid (l, u, p) \in St \ \wedge \ \mathcal{S}(p|_c) = true\}$, and the set of transitions $T' = \{(s, s') \mid (s, s') \in T \ \wedge \ s, s' \in St'\}$* □

Intuitively, the set of states are projected on the configuration c and the corresponding transitions are kept. Given a timed automata family T, the following theorem shows the difference between applying projection on T itself, and on its semantics.

Theorem 4.4.6 *Given an initial constraint ψ, a timed automata family T and a configuration c, we have: $[\![T|_c]\!]_{\psi|_c} \sqsubseteq [\![T]\!]_\psi|_c$* □

4.4.3 An Object Family

An object family, formally denoted (R, Ξ), is defined as an instance of the class family R coupled with a set of schedulers Ξ. Each scheduler has an application condition, such that at each configuration one scheduler can be applied. For example, an elevator family has a normal scheduler plus another one supporting a VIP floor. To define the behavior of an object, we need to know how its methods are going to be called. Therefore, we consider its behavioral interface as an environment. It means that the behavioral interface sends messages to the object, and the object may only send out messages if they are accepted by the behavioral interface.

The behavior of an object family is then defined as a timed automata family. The locations of this TAF are written (b, s, Q) corresponding to the current location of the behavioral interface (shown as b), the current location of the currently running method (shown as s) and the contents of the queue (shown as Q). The initial location is $(b_0, \epsilon, [\,])$ which means that the behavioral interface is in its initial location b_0, no

$(b, Q) \xrightarrow[c_\nu=0]{a?} (b', Q :: a(d, c_\nu))$	if $b \xrightarrow{a(d)!}_B b'$, c_ν is a fresh clock
$(s) \to (s')$	if $s \to_M s'$
$(b, s) \xrightarrow{a(d)!} (b', s')$	if $s \xrightarrow{a(d)!}_M s'$, $b \xrightarrow{a?}_B b'$
$(s, Q) \xrightarrow{a!} (s', Q :: a(d_r, c_r))$	if $s \xrightarrow{a!}_M s'$, $a \in M_R$
$(s, [\,]) \to (\epsilon, [\,])$	if $s \not\to_M$
$(\epsilon, Q) \xrightarrow{\varphi(\xi)} (m_0, Q')$	if $Q \neq [\,]$, $(m, Q') = \xi(Q)$, $\forall \xi \in \Xi$

Assumptions:

- The clock and deadline of the currently running task are c_r and d_r, respectively, which are updated upon scheduling a new method.
- The initial location of method m is denoted m_0.
- φ returns the application condition of a scheduler ξ.
- M_R is the set of methods defined in class R, and Ξ is the set of schedulers in the object.
- Location s implies the currently running method, also identified by subscript M for transitions.

Figure 4.4: Simplified presentation of the timed automata family encoding the behavior of an object family w.r.t. its behavioral interface B.

method is currently running and the queue is empty. The transition relation of this TAF is to be computed as shown in Fig. 4.4. In this figure, only the relevant components of the state are shown. Furthermore, every guard, clock reset and application condition present on the transitions on the right (i.e., in the behavioral interface family or the method definition) will also be present in the transition on the left. These are dropped in the figure for simplicity in presentation.

We briefly explain each rule in Fig. 4.4 in one sentence: The behavioral interface can send a message a with deadline d which is added to the back to the queue. The currently running method can do one of the following: 1) it can take an invisible action; 2) it may send a message to the environment; 3) it can make a delegation, inheriting the deadline. The ϵ symbol shows that no method is currently running. The last rule (i.e., starting a new method) is applicable for all schedulers in Ξ resulting in a transition with a corresponding application condition. Note also that the scheduler function needs to look at the queue contents and therefore the generated state depends on the contents of the queue.

4.5 Conclusion

In this chapter, we provided a quick overview of the recent advancements in modeling and analysis of real-time concurrent objects, as in ABS, in the context of the HATS project. In this chapter, we mainly provided the theory at the level of timed automata. This theory can be easily applied to ABS objects by making an abstraction of a real-time ABS model into timed automata. In the attached articles, a complete description of these techniques is provided.

Bibliography

[1] Erika Abraham, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In *Theoretical Aspects of Computing – ICTAC 2004: First International Colloquium, LNCS*, volume 1, 2004.

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR '98*, volume 1466 of *LNCS*, pages 163–178, 1998.

[3] Lennart Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 218–227, New York, NY, USA, 1984. ACM.

[4] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *JACM: Journal of the ACM*, 52, 2005.

[5] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *LNCS*, pages 387–411. Springer, 2005.

[6] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[8] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. TVOC: a translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer-Verlag, 2005.

[9] BCVerifier: An automatic verifier for backward compatibility. `https://softech.informatik.uni-kl.de/bcverifier`.

[10] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, Berlin, 2007.

[11] Nick Benton and Vasileios Koutavas. A mechanized bisimulation for the nu-calculus, 2008.

[12] L.C. Breebaart. *Rule-based compilation of data parallel programs.* PhD thesis, Delft University of Technology, 2003.

[13] Richard Bubel, Reiner Hähnle, and Ran Ji. Interleaving symbolic execution and partial evaluation. In *Post Conf. Proc. FMCO2009*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[14] Richard Bubel, Reiner Hähnle, and Ran Ji. Program specialization via a software verification tool. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Post Conf. Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *LNCS*. Springer-Verlag, 2011.

[15] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[16] Ferruccio Damiani, Arnd Poetzsch-Heffter, and Yannick Welsch. A type system for checking specialization of packages in object-oriented programming. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1737–1742. ACM, 2012.

[17] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proc. Hybrid Systems: Computation and Control (HSCC'10)*, pages 91–100. ACM, 2010.

[18] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *Proc. Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 108–122, 2002.

[19] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[20] Verification of Behavioral Properties, March 2011. Deliverable 2.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[21] Jim des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`.

[22] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, pages 83–107, 2006.

[23] Sophia Drossopoulou, Adrian Francalanza, Peter MÃ$\frac{1}{4}$ller, and Alexander Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, LNCS, pages 412–437, 2008.

[24] Eclipse PDE API Tools. `http://www.eclipse.org/pde/pde-api-tools/`.

[25] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci*, 411(51-52):4379–4398, 2010.

[26] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[27] R. W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.

[28] Benny Godlin and Ofer Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.

[29] Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *LNCS*, pages 453–465. Springer, 2011.

[30] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.

[31] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50:63–69, January 2003.

[32] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In Kazunori Ueda, editor, *APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.

[33] Mohammad Mahdi Jaghoori. Composing real-time concurrent objects refinement, compatibility and schedulability. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 7141 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2011.

[34] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, Delphine Longuet, and Marjan Sirjani. An automata-theoretic framework for real-time actors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Submitted in 2012.

[35] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009.

[36] Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer, and Tom Chothia. Schedulability and compatibility of real time asynchronous objects. In *Proc. RTSS'08*, pages 70–79. IEEE CS, 2008.

[37] Claude Jard, Thierry Jéron, and Pierre Morel. Verification of test suites. In *Proc. Testing Communicating Systems (TestCom 2000)*, pages 3–18, 2000.

[38] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. *LNCS*, 3444:423–438, 2005.

[39] Ran Ji and Richard Bubel. PEKeY: A Partial Evaluator for Java Programs. Submitted to IFM'12, 2012.

[40] Vasileios Koutavas and Mitchell Wand. Bisimulations for untyped imperative objects. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *LNCS*, pages 146–161. Springer, 2006.

[41] Vasileios Koutavas and Mitchell Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL 2007*, January 2007.

[42] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of *LNCS*, pages 109–126, 2004.

[43] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.

[44] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
`ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlkluwer.ps.gz`.

[45] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[46] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[47] Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch. Model-based compatibility checking of system modifications. In *ISoLA*, LNCS. Springer, 2012. to appear.

[48] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer-Verlag, 2006.

[49] Hamideh Sabouri, Mohammad Mahdi Jaghoori, Frank de Boer, and Ramtin Khosravi. Scheduling and analysis of real-time software families. In *Proc. The 36th IEEE signature conference on Computer Software and Applications (COMPSAC)*, 2012. to appear.

[50] Martin Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages.* Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.

[51] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375, 2007.

[52] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54, 2007.

[53] Yannick Welsch and Arnd Poetzsch-Heffter. Full abstraction at package boundaries of object-oriented languages. In *SBMF 2011*, LNCS, pages 28–43. Springer, 2011.

[54] Yannick Welsch and Arnd Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. 2012. submitted for journal publication.

[55] Yannick Welsch and Arnd Poetzsch-Heffter. Verifying backwards compatibility of object-oriented libraries using boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41, New York, NY, USA, 2012. ACM.

Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification language developed within the HATS project.

Backward Compatibility A relation between two versions of a library which shows whether a usage context can observe a difference in behavior.

(Java) Bytecode Program representation that is directly executable by the Java virtual machine (similar to assembler languages).

Compilation Translation of a program into a representation executable by a (virtual) machine.

Coupling Invariant Specification of a relation that allows two prove that two library implementations are backward compatible.

Calculus A set of rules that allow to derive if a logic statement is valid.

ABS Dynamic Logic Dynamic logics are a subfamily of modal logics. In dynamic logics programs (here ABS programs) are first-class citizens and occur directly as part of formulas.

Execution Path List of statements representing one single *concrete* execution of a program.

Partial Evaluation Simplification/Optimization of a program under the assumption that a certain subset of input variables is static.

Program Context Set of classes that rely on a specific library and that contain a main method. Combining a program context with a library yields an executable program.

Sequent Calculus A logic calculus working on sequents (sets/sequences of formulas) originally introduced by Gentzen.

Symbolic Execution Execution of a program where the values of the input variables (and also the heap state) are represented symbolically.

Symbolic Execution Path List of statements representing one single symbolic execution of a program (captures possibly infinitely many concrete execution paths).