Thesis for The Degree of Doctor of Philosophy

Combined Static and Dynamic Verification of Object Oriented Software Through Partial Proofs

Jesús Mauricio Chimento

CHALMERS
UNIVERSITY OF TECHNOLOGY

Division of Formal Methods
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2019

Combined Static and Dynamic Verification of Object Oriented Software Through Partial Proofs

Jesús Mauricio Chimento

ISBN 978-91-7597-866-6
Doktorsavhandlingar vid Chalmers tekniska högskola,
Ny serie nr 4547.
ISSN 0346-718X

Technical Report 172D
Department of Computer Science & Engineering
Division of Formal Methods
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

Printed by Chalmers Reproservice,
Gothenburg, Sweden 2019.

Abstract

When verifying software one can make use of several verification techniques. These techniques mostly fall in one of two categories: *Static Verification* and *Dynamic Verification*. Static verification deals with the analysis of either concrete source code, or a model of it. These kinds of techniques can verify properties over all possible runs of a program. Dynamic verification is concerned with the monitoring of software, providing guarantees that observed runs comply with specified properties. It is strong in analysing systems of a complexity that is difficult to address by static verification, e.g., systems with numerous interacting sub-units, concrete (as opposed to abstract) data, etc. On the other hand, its major drawbacks are the impossibility to extrapolate correct observations to all possible runs, and that the monitoring of a property introduces runtime overheads.

It is quite clear that static and dynamic verification have largely disjoint strengths. Therefore, their combination can allow the verification process to deal with richer properties, with greater ease. The work presented in this thesis addresses this issue by introducing some manners to combine static and dynamic verification, where partial proofs are used as a means to accomplish the combination. The main novelty in these combinations consists in the fact that all of them consider the use of the partial proofs in the verification process, whereas, in general, other verification approaches discard them right away.

The main contributions of this thesis are: (i) *ppDATE*, an automata-based formalism to specify both data- and control-oriented properties; (ii) structural operational semantics for *ppDATE*; (iii) a translation of ppDATE to DATE together with a proof of correctness; (iv) StaRVOOrS, a tool for combining (static) deductive verification and runtime verification of object oriented software; (v) a testing focused development methodology which integrates deductive and runtime verification in its workflow; and (vi) a methodology to infer global trace conditions for a system, from partial proofs local to the transitions of a model, obtained by performing low effort verification attempts to properties.

ACKNOWLEDGMENTS

First of all, I would like to express my most sincere gratitude to both my supervisor Wolfgang Ahrendt, and my co-supervisor Gerardo Schneider, for their guidance, and their advice, during all of my PhD. It has been a pleasure working with them during these 5 years, and I am really looking forward to continue doing so in the near future.

Many thanks to Gordon Pace, Christian Colombo, Richard Bubel, and Martin Henschel for their collaboration. Completing this thesis without their insights and help would have been much harder for me.

I would also like to thank my colleagues, but most important my friends, at Chalmers. Coming every day to such an amazing work environment was always a great motivation to keep up the hard work.

Thanks to all of my friends from the dance floor, and from the Happy Wednesdays, for keeping life interesting outside work.

Ni que hablar de mis amigos en Argentina. Si bien no estamos tan en contacto como antes, siempre es bueno saber que ustedes van a estar ahí cuando uno los necesite.

Last but not the least, gracias totales a mi familia por apoyarme siempre incondicionalmente durante estos 5 largos años. Siempre van a estar conmigo, no importa en donde esté.

List of Publications

This thesis is based on the work disseminated in the following documents, each presented in a separate chapter.

Chapter 2. *Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools.* W. Ahrendt, J. M. Chimento, G. Pace, and G. Schneider. Formal Methods in System Design. 04 April 2017. Springer.

Chapter 3. *StaRVOOrS User Manual (release 1.7).* J. M. Chimento. 31 May 2018.

Chapter 4. *Testing Meets Static and Runtime Verification.* J. M. Chimento, W. Ahrendt, and G. Schneider. FormaliSE'18. June 02, 2018. Pages 30-39. ACM.

Chapter 5. *Inferring Global Trace Conditions From Local Partial Proofs.* J. M. Chimento. Technical report. November 2018.

Other publications

The following publications were published during my PhD studies. However, they are not included in this thesis as their content is subsumed by the publication: *Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools.*

1. *A Specification Language for Static and Runtime Verification of Data and Control Properties.* W. Ahrendt, J. M. Chimento, G. Pace, and G. Schneider. Formal Methods 2015 (FM'15), 20th International Symposium on Formal Methods. LNCS. June 24-26 2015. Springer.

2. *StaRVOORS: A Tool for Combined Static and Runtime Verification of Java.* J. M. Chimento, W. Ahrendt, G. Pace, and G. Schneider. Runtime Verification 2015 (RV'15). LNCS. September 23-25 2015. Springer.

CONTENTS

CHAPTER
ONE

INTRODUCTION

During the last decade, the integration of technology into our ordinary activities has been growing at a fast pace. Desktop computers, laptops, netbooks, tablets, smart phones, and smart watches represent just a short list of devices which are used on a daily basis all around the globe. In addition, the use of online services, e.g., home banking, instant messaging, TV streaming, tax payment, e-education, etc., have become a trend. Even though these devices and services are really different, all of them have one aspect in common: they all operate by running software.

In general, software developers provide as part of the documentation of their products an informal description regarding the intended behaviour of their programs. However, they do not usually offer any guarantees about the accomplishment of such a behaviour. In fact, it is a common practice for them to include in the installation mechanism of their programs a *Terms and Conditions* section, where they add disclaimers saying, for instance, that they do not take any responsibility if the use of their products leads to a malfunction of the devices running them.

Unexpected software behaviour may be a real headache for everyone. It is true that if a program which checks the weather forecast fails during its execution, it may not represent any harm. However, if the software in a self-driven car fails, the result of this failure may be a catastrophe as people's lives can be in danger. In addition, an error in a home banking system could cause the loss of all the savings of the customers of a bank.

Fortunately, efforts by developers to avoid unexpected behaviour on their programs are increasing. Among the different measures that developers are taking to avoid the presence of errors in their products, one can highlight the use of *formal verification*.

Formal verification consists on proving the correctness of the program under scrutiny by showing that it fulfils a formal specification of its intended behaviour through the use of verification techniques. These techniques may be divided into two categories: *static verification* techniques, and *dynamic verification* techniques.

Static verification techniques deal with the analysis of either concrete source code, or a model of it. These techniques can verify properties over

all possible executions of a program. However, their application may require a high computational effort. In addition, one may have to introduce special annotations, e.g., loop invariants, to deal with certain properties. This fact may increase the complexity of achieving a full automation of the whole verification process.

Regarding the dynamic verification techniques, they are concerned with the monitoring of software, i.e., the program has to be executed in order to check the properties. These techniques, which in general are fully automated, provide guarantees that observed executions of a program comply to the specification. However, it is impossible to extrapolate results of correct observations to all possible executions. In addition, monitoring introduces runtime overheads which may be prohibitive in certain systems.

It is quite clear that static and dynamic verification have largely disjoint strengths. Therefore, their combination can allow the verification process to deal with richer properties, with a greater ease. This work presents a novel approach to address the combination of static and dynamic verification techniques, by using partial proofs, i.e., an unfinished proof of a property, as a means for accomplishing the combination. The novelty in this approach consists in the fact that we consider the use of partial proofs in the verification process, whereas, in general, other verification approaches discard them.

As a result, we were able to enhance both the verification of correctness properties, and the development of software through the use of techniques based on verification.

The structure of this chapter is as follows: Section 1.1 introduces several verification techniques of interest for this work. Section 1.2 introduces the verification tools used in this thesis, and their specification languages. Section 1.3 discusses the notion of partial proof. Section 1.4 briefly elaborates on the combination of static and dynamic verification through partial proofs. Section 1.5 describe possible continuations for the work developed within this thesis. Finally, section 1.6 describes the contributions of this thesis, which are properly reflected on its different chapters.

1.1 Verification Techniques

Verification techniques are mainly used to analyse whether a program satisfies certain properties. Such properties usually describe the behaviour of the *system under test* (SUT). In general, these techniques are divided into either static or dynamic verification techniques, depending on whether the SUT is run to be verified. Below, several of the most widely used static and dynamic verification techniques are briefly described. In addition, we make a (brief) general comparison between the use of both kind of techniques, and we discuss the benefits which may be obtained by combining them.

1.1.1 Static Verification Techniques

In this section we analyse two of the most widely used static verification techniques: *Deductive Verification*, and *Model Checking*.

Deductive Verification

Deductive verification [56] focuses on turning the correctness properties of a program into logical formulae, e.g., first-order logic, high-order logic, program logic, etc., and then verifying these formulae by deduction in a (logic) calculus.

In general, there are three main approaches that one may adopt to perform deductive verification. Let us call these three approaches *Proof Assistants*, *Program Logic*, and *Verification Condition Generation*.

Proof Assistants are interactive theorem provers which, in general, target some high-order logic. These provers are not language-oriented. Instead, they provide a language in which both the syntax and the semantics of the SUT have to be described. In addition, correctness properties have to be modelled within the logic handled by the proof assistant. Thereby, one can use the proof assistant to develop the proof of the properties. Note that, even though proof assistants are interactive, they may present a certain degree of automation. As an example, the *Coq* [28] proof assistant targets *intuitionistic logic*, introduces the language *Gallina* to describe the syntax and the semantics of the SUT, and uses a *sequent calculus* to verify the correctness properties.

In relation to *Program Logic*, *Hoare Logic* [66] may be the most well-known program logic to analyse programs. Hoare logic offers both a clear notation to describe programs and their properties, and a set of axioms and inference rules which may be used to verify the properties [81]. In this logic, properties are described by using *Hoare triples*. A Hoare triple is an expression of the form $\{P\}\ S\ \{Q\}$, where S is (a unit of) the SUT, P is the precondition of S, and Q is the postcondition of S. In addition, one may consider the use of some language-oriented modal logic, e.g., *dynamic logic* [62], in order to reason about the correctness properties of the SUT. As an example, the *KeY* [9] tool uses *Java dynamic logic* to deal with the correctness properties, and a *sequent calculus* to verify them.

On the *Verification Condition Generation* approach, programs are annotated with assertions representing the correctness properties. Then, this assertions are used to generate first-order logic conditions which later may be discharged by using some automatic theorem prover. As an example, one may refer to *Dafny* [71] or *Why3* [29]. Dafny is a programming language which natively supports specification constructs to describe the specification of procedures. Such specifications are used to generate first-order conditions which are discharged by using *Z3* [50]. Regarding Why3, it is a deductive program verification platform which provides its own language for specification and programming. On this platform one can use both automated theorem provers, e.g., *Z3*, and interactive theorem provers, e.g., *Coq*, to discharge verification conditions.

Model Checking

Model Checking focuses on verifying properties about a system by analysing a finite state abstraction of it, which is usually referred as the *model* [40]. This technique determines whether a model fulfils the specified property by performing an exhaustive search over the entire state space of the system, aiming at finding an execution trace which violates it. If no such a trace is found, then the property is satisfied by the model.

As this technique deals with finite state systems, the tools implementing it, a.k.a. *model checkers*, automatically analyse all of the possible executions of the system, always terminating with a yes or no answer depending on whether the specification is fulfilled or not (if enough resources are available of course). However, when dealing with a real world problem the model checkers have to deal with the *state explosion problem* [41]. In short, when the amount of state variables in the system increases, the size of the system state space grows exponentially.

Whenever the specification is not fulfilled, i.e., a property is violated, model checkers return the execution trace which has violated the property, usually referred as *error trace*. Such a trace serves as a counter-example for the property.

An example of a model checker is SPIN [67], a popular model checker which uses the language *PROMELA* for the description of the models, and *Temporal Logic* as the specification language for the properties.

1.1.2 Dynamic Verification Techniques

In this section we analyse three dynamic verification techniques: *unit testing*, *model-based testing*, and *runtime verification*. It is important to remark that in this work we use the term dynamic verification to refer to any of the previous techniques, and not as a synonym for runtime verification.

Unit Testing

Unit Testing [65] aims at analysing particular executions of a program to determine whether they produce certain expected values, i.e., the program behaves as expected. Such executions are based on *test cases*. A test case represents a particular initial state of the SUT, and the expected state of the program once its execution is complete. It is defined by assigning particular values to the different variables and parameters associated to the program.

Two traditional manners of applying unit testing are *Black-box testing* and *White-box testing*. Below, we elaborate on them.

Black-box testing focuses on the analysis of the functionality of the SUT, treating it as a '*black box*', i.e., without looking into its source code. Instead, it is enough having only some idea of what the program is supposed to do. For instance, if the program is sorting an array, then one only has to know that the array has to be sorted after executing the program, without the need of knowing which sorting algorithm the program is implementing. In addition, this kind of testing is quite useful whenever (part of) the source code of the program is not available, e.g., the code belongs to a third party library.

Among the different black-box testing techniques, *Input Space Partitioning* is usually one of the most highlighted ones. This technique consists in, first, defining the *input space* of the program, i.e., the set of all possible inputs that may be fed to the program. Next, the input space is divided into several *disjoint partitions*, such that the values on each partition test different functionalities of the SUT. Then, test cases are generated by selecting a value from each partition. Finally, these test cases are used to execute the program, and the obtained results are analysed.

Regarding *White-box testing*, it analyses the structure of a program to trace possible execution paths through its code. Therefore, one needs to have access to the complete source code of the SUT to perform this kind of testing.

There are many criteria which can be followed when using white-box testing such as, for instance, *Code Coverage* criteria. Within these criteria, one can highlight the *Statement Coverage* criterion, where all the test cases are generated in such a manner that all the statements of the SUT are executed, or *Condition Coverage* criterion, where all the test cases are generated in such a way that every condition of the SUT is evaluated both to true and false.

Model-Based Testing

While unit testing focuses on writing tests cases in order to analyse the computation performed by a unit of the SUT on the *data*, *model-based testing* (MBT) [91] provides better support for testing *control*-oriented aspects, e.g., the flow of execution through the procedures in the SUT.

In general, most of the models used to generate tests for control-oriented aspects are based on variants of *finite-state machines*. From these models, MBT tools can automatically generate test cases, which might also contain the expected output in order to automate the decision on whether the test is successful or not [6, 90]. Coverage criteria to generate test cases in MBT include *Transition Coverage*, where the test cases have to traverse all of the transitions in the model, and *State Coverage*, where the test cases have to visit all the states in the model. In addition, these tools may generate failing traces which simplify the detection of errors in the SUT.

More concretely, MBT involves performing the following steps:

(i) Writing an abstract model, which may be annotated to capture the relation between tests and requirements;
(ii) Generating *abstract* tests from the model. This implies defining both a test selection and coverage criteria;
(iii) Generating *concrete* test cases. This implies creating an *adaptor* which converts abstract tests into concrete test cases;
(iv) Executing the tests on the SUT and assigning verdicts;
(v) Analysing the results of the tests and taking a corrective action (if necessary).

Note that a failing test case might not necessarily mean that there is an issue with the implementation. A test case may fail due to a fault in the adaptor, or in the model as well.

Among the benefits of using MBT it is usually mentioned [91] that this technique increases the possibility of finding errors in the SUT; it reduces testing cost and time, as programmers spend less time and effort writing tests, and analysing their results; it improves the quality of the tests by considering coverage of the model and the SUT; it gives traceability between requirements and the model, and between informal requirements and generated test cases; and it helps to update the test suites when requirements evolve.

On the other hand, MBT cannot guarantee finding all of the differences between the model and the SUT, and it needs skilled model designers, among other things. In addition, unless a table relating requirements with the model is kept up-to-date, one might get the wrong model from outdated requirements.

Finally, it is indeed an overhead writing the model (which might be wrong) and developing the adaptor (which might also introduce errors).

Runtime Verification

Runtime verification [25, 63, 74] is concerned with the monitoring of software executions. This technique detects violations of properties which occur during the execution of the SUT. Due to this fact, runtime verification gives the possibility of reacting to incorrect behaviour of a program whenever an error is detected.

Properties to be verified using this technique are usually described in two possible manners. One possibility is annotating the source code of the SUT with *assertions*. An assertion is a logical formula which is expected to be true whenever the execution of the annotated program reaches it. The other possibility is using a high-level specification language. For instance, *Linear Temporal Logic* (LTL) [78] is one of the most popular formalisms in use. In addition, another approach which is increasing in popularity is writing properties using automaton-based specification languages [10, 45].

To verify the properties runtime verification introduces the use of *monitors*. A monitor is a piece of software which runs in parallel to the SUT, controlling that the execution of the program does not violate any property. In addition, monitors usually create a log file where they add entries reflecting the results which are obtained whenever they attempt to verify a property.

In general, monitors are automatically generated from either the annotated assertions, or the (high-level) specification of the properties. Regarding monitor generation from annotated assertions, one may refer to *openJML* [42], or *jml4c* [82], which are tools that allow to perform runtime verification over Java programs annotated with assertions written in the Java Modelling Language [72]. Regarding monitor generation from specifications, one can refer either to LARVA [46] and MarQ [80] as examples of tools which apply runtime verification by generating monitors from the high-level specification languages *DATE* [45] and *Quantified Event Automata* [23], respectively.

One downside of the use of runtime verification is that it introduces some overhead to the execution of the system. Thus, one of the main objectives of the developers of tools which use this technique is to reduce as much as possible such overhead.

1.1.3 Comparing Static and Dynamic Verification Techniques

Static verification techniques are good to analyse the correctness of software. These techniques come with strong guarantees to verify properties over all possible executions of a program, or a model of it. In addition, as these techniques are used pre-deployment, they do not affect the behaviour of the programs at runtime.

However, full static verification may be hard to achieve automatically (not to say impossible). Among other things, loop invariants usually have to be provided by the developers. Therefore, one has to rely on code annotations, or interactive proof construction. In addition, the use of libraries generally

represents a challenge for static techniques, as source code of the libraries is not always accessible. Thus, either some data abstractions may have to be introduced in many of the properties, or some specification has to be provided for the library, in order to verify the properties using static verification. These constraints may require highly trained experts to handle them, they may increase the computational effort required to apply the techniques, and the techniques working with abstractions from the real code may loose accuracy when they analyse a property involving the use of the concrete data of the system.

Regarding the dynamic verification techniques, they are lightweight techniques which are usually strong in analysing programs of a complexity which is difficult to address by the static verification ones, like programs interacting with several other systems, heavy usage of mainstream (external) libraries, and concrete (as opposed to abstract) data. For instance, dynamic verification techniques can directly access the results of the calls to procedures of a library, whereas the source code of the libraries is not usually accessible to the static techniques.

Nonetheless, dynamic techniques cannot be used to guarantee the correctness of a program, mainly due to the fact that they cannot extrapolate the results of correct observations to all possible executions, as oppose to static techniques which verify properties over all possible executions of a program. In addition, dynamic techniques may induce overheads to the system which can be a problem in certain settings.

Static and dynamic verification have largely been applied to disjoint areas. For instance, deductive verification has been extensively used to verify properties focusing on a system's data, e.g., [9, 54, 69, 73], whereas runtime verification has been extensively used to verify control-flow properties with reasonable overheads [24, 35, 45, 79]. Still, from the comparisons above one can realise that these techniques are complementary. They both have advantages and disadvantages, and a natural question which can arise is whether they may be combined in a manner where one can get the advantages of both kinds of techniques, but without inheriting too much from their disadvantages. This work addresses this challenge, providing a positive answer to it (see Sec.1.4).

1.2 Selected Tools

This section briefly introduces the main tools used in this thesis, and their specification languages. Sec. 1.2.2 introduces the deductive verifier KeY and the Java Modelling Language (specification language). Sec. 1.2.3 introduces the runtime verifier LARVA and *DATE*, its specification language. Both tools are used to verify Java programs.

1.2.1 Preliminaries: Dynamic Logic

Before introducing the deductive verifier used in this thesis, we need to introduce the notion of dynamic logic. *Dynamic Logic* (DL) [62] is a modal logic which is used to reason about programs. Due to the many differences between different programming languages, it is not possible to have one single version of DL to analyse them all. A DL is therefore specific for the programming language at

hand. For instance, the version of DL used by KeY to analyse Java programs is referred as *Java DL*.

In particular, DL includes two modalities [ ], a.k.a. box, and $\langle\rangle$, a.k.a. diamond. Given a DL formula ϕ and a program p, $[p]\phi$ means that if p terminates its execution, then it is in a state where ϕ holds; and $\langle p\rangle\phi$ means that p terminates its execution and ϕ holds in the final state reached by p. For deterministic programs p, the only difference between these two modalities is that termination is *stated* in $\langle p\rangle\phi$, and *assumed* in $[p]\phi$.

In addition, DL formulae are written using the traditional logical operators $\wedge, \vee, \rightarrow$, and $\neg$, and both universally and existentially quantifications over *logic variables*.[1] Note that as DL is specific for the programming language under scrutiny, the syntax used for describing programs will depend on the programming language in use. For instance, assuming that p_1 and p_2 are sequences of program statements, the following expressions are all examples of Java DL formulae:

- `[x=x*x;y=x+20;y=y+1;] x` $\geqslant 0$
- $\langle$`x=y-x;if(y>0){`p_1`}else{foo()};`$p_2\rangle$ `x` $\neq$ `y`
- $\forall\, l \cdot l > 0 \rightarrow$ `[x=`l`;`p_1`;] x` > 0
- `x` $>$ `y` $\wedge$ `y` $> 0 \rightarrow \langle$`x=y*x;`$\rangle$ `[y=y+x;] y` $>$ `x`

Note that the DL formula $\phi \rightarrow [p]\psi$ is valid if whenever formula ϕ holds, and the execution of p terminates, the formula ψ is fulfilled afterwards. Therefore, the previous formula could be regarded as the Hoare triple $\{\phi\}p\{\psi\}$.

1.2.2 Deductive Verification using KeY

In this thesis we heavily use the deductive verifier KeY [9]. KeY is a tool for data-centric functional correctness properties of Java programs which, given a Java program annotated with *Java Modelling Language* (*JML*) [72], generates proof obligations (i.e., formulae) in Java DL, and attempts to prove them.

JML is a specification language which primarily focuses on the description of pre/post-conditions of methods and class invariants. This language is compatible with Java expression syntax, a fact that simplifies its use. Fig. 1.1 illustrates a JML specification (from line 1 to 5) for Java a method named `foo`. Line 1 describes which one is the behaviour expected for method `foo` (either normal as in this example, or exceptional); line 2 describes the precondition of `foo`; line 3 describes the postcondition of `foo`; and line 4 lists the variables of the class which are modified by executing `foo`.

Coming back to KeY, Fig. 1.2 roughly illustrates how this tool would generate the dynamic logic proof obligation in the column of the right, from the Java method `foo` which is annotated with JML in the column of the left.

Similarly to many other verification tools, KeY has a few restrictions: it does not support concurrency and floating-point arithmetic, and the generic types are expected to be compiled away. However, it is also worth mentioning that KeY fully covers Java Card, and that Java integer types, exceptions, and static initialization are accurately modelled on it [9].

At the core of KeY is a prover using a *sequent calculus* to construct proof trees for the generated proof obligations, by following the *symbolic execution* paradigm [64, 70]. Here, we will not introduce this calculus. However, we

[1] Logic variables never occur in programs.

```
/*@ public normal_behaviour
  @ requires n > 0 && y > 0;
  @ ensures x == n && y > x;
  @*/
public void foo (int n) {
   x = n;
   y = y + x;
}
```

Figure 1.1: JML specification for a particular Java method.

```
/*@ public normal_behaviour
  @ requires n > 0 && y > 0;
  @ ensures x == n && y > x;
  @*/                                   n > 0 ∧ y > 0 →
public void foo (int n) {                 ⟨x := n ; y := y + x;⟩ x = n ∧ y > x
   x = n;
   y = y + x;
}
```

Figure 1.2: Rough example of a DL proof obligation generated by KeY.

will show a simple example on how its sequent look like. Given a set of formulae Γ, the sequent $\Gamma \vdash \langle p \rangle \phi$ holds if p, when starting in a state fulfilling all formulae in Γ, terminates in a state fulfilling ϕ. In addition, due to the use of symbolic execution, DL has to be extended by *explicit substitutions*, e.g., $\{\texttt{x:=x*x}\}[\texttt{y=x+20;y=y+1;}]\ \texttt{x} \geqslant 0$. While symbolically executing p, its effects are gradually, starting from the front, turned into explicit substitutions. Thereby, after some proof steps, a certain prefix of p has turned into a substitution σ, representing the effects so far, while a remaining program p' is yet to be run. During the verification of p, an intermediate proof node may look like $\Gamma \vdash \sigma\langle p' \rangle \phi$. Such a node tells us that, if Γ was true before the original program p, and σ is the accumulated effect up to now, then ϕ will be true after the execution of the remaining program p'.

1.2.3 Runtime Verification using LARVA

In this thesis we also use the runtime verification tool LARVA [46]. This tool automatically generates a runtime monitor from a property written in the automaton-based specification language *DATE* [45]. In order to do so, LARVA transforms the set of properties into monitoring code together with AspectJ code, to link the system with the monitors.

Regarding *DATE*, it is a specification language to describe properties as finite state automata. Transitions in this language are tagged with labels of the form $e \mid cond \mapsto act$, where e represents a system event (primarily either an entry point $\texttt{e}^{\downarrow}$ or an exit point $\texttt{e}^{\uparrow}$ of a method), *cond* is a condition that must be true in order for the transition to take place, and *act* is is a code snippet to be performed when the transition is taken. Note that this description does not

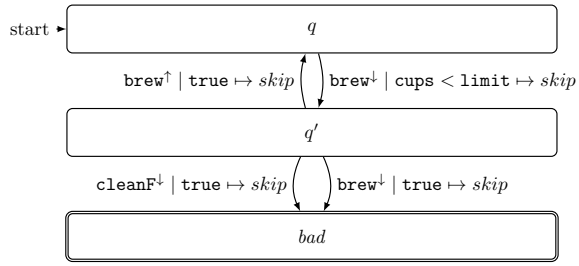


Figure 1.3: A *DATE* controlling the brew of coffee

mention many of the features offered by *DATE*. We refer to [45] for a more detailed introduction to this language.

Fig. 1.3 illustrates an example of *DATE* describing a property about the system of a coffee machine. This *DATE* ensures that whenever the coffee machine is not active (i.e., is not brewing coffee) and the method `brew` starts the coffee brewing process, then it is not possible either to execute this method again or to execute the method `cleanF`, which initialises the task of cleaning the filter, until the current brewing process terminates.

1.3 Partial Proofs and the Power Behind Them

In general, whenever a verification attempt of a property fails, its proofs is immediately discarded. However, having a *partial proof*, i.e., a proof which is obtained from a failed verification attempt and does not fully verify the property under scrutiny, can still be worthy.

Partial proofs are one of the most fundamental concepts for this thesis, as they will be used as a means for combining static and dynamic verification. Thanks to the use of partial proofs one can obtain significant results towards the verification of software correctness, even from *low effort* verification attempts, e.g., failing at automatically proving a property, and then not performing any interactive step in an attempt to finish the proof. Most of the contributions of this work are examples which can be used to justify this claim.

Below, we elaborate in more detail on the notion of partial proof, and the power behind their use.

1.3.1 Partial Proofs

To elaborate on the notion of partial proof, let us consider a proof for the Java DL sequent below, using the the sequent calculus which is part of the deductive verifier KeY (see Sec. 1.2.2):

$$\texttt{x} > 0 \vdash \langle \texttt{x=y;y=x+42;x=y-x;if(y>0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \qquad (1.1)$$

(where p_1, p_2, and q are (sequences of) program statements and ϕ is some postcondition). Sequent (1.1) says that in each state where `x` is positive, the program provided in the modality will terminate, and result in a state where ϕ holds.

The proof for the sequent above would start by, in some of steps, turning the three leading assignments into explicit substitutions, apply the first to second one, apply the result of this substitution to the third, and then perform some simplifications, arriving at

$$\texttt{x} > 0 \vdash (\texttt{x} \leftarrow \texttt{y} \parallel \texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow \texttt{y-x})\langle \texttt{if(y>0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (1.2)$$

where $(\texttt{x} \leftarrow \texttt{y} \parallel \texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow \texttt{y-x})$ represents the explicit (parallel) substitution obtained as a result from the symbolic execution of the first three statements. When clashes occurs in parallel substitutions, as it is the case for x in (1.2), a 'right-win' semantics is adopted in order to resolve them. Thereby, sequent 1.2) is reduced to:

$$\texttt{x} > 0 \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)\langle \texttt{if(y>0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (1.3)$$

In general, most proofs branch over case distinctions, often triggered by Boolean decisions in the source code. This branching occurs when applying rules like the following, simplified,[2] if rule:

$$\textsf{if}\ \frac{\Gamma, \sigma(b) \vdash \sigma\langle s_1\ \ \omega\rangle\phi \qquad \Gamma, \sigma(\neg b) \vdash \sigma\langle s_2\ \ \omega\rangle\phi}{\Gamma \vdash \sigma\langle \texttt{if}\ b\ s_1\ \texttt{else}\ s_2\ \ \omega\rangle\phi}$$

In our example, applying the if rule to sequent (1.3) results in splitting the proof into two branches, with the following sequents, respectively:

$$\texttt{x} > 0, (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)(\texttt{y>0}) \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)\langle p_1\texttt{;}q\rangle\phi$$
$$\texttt{x} > 0, (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)(\neg\texttt{(y>0)}) \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)\langle p_2\texttt{;}q\rangle\phi$$

Applying the substitution on the left side of either sequent results in:

$$\texttt{x} > 0, \texttt{(y+42)>0} \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)\langle p_1\texttt{;}q\rangle\phi \quad (1.4)$$
$$\texttt{x} > 0, \neg\texttt{((y+42)>0)} \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow 42)\langle p_2\texttt{;}q\rangle\phi \quad (1.5)$$

Once all proof branches are closed, i.e., they are verified, we have a *complete proof* of the root sequent. However, a proof attempt may result into a proof object where not all the branches are closed. We refer to this kind of proof as a *partial proof*. In addition, we refer to the set of formulae in a sequent, e.g., '$\texttt{x} > 0, \texttt{(y+42)>0}$' in sequent (1.4), as *branch condition*. In other words, a branch condition is a set of formulae leading the proof to one of its branches.

1.3.2 Partial Proof Capabilities

In the example above, consider a partial proof where sequent (1.4) is fully proved, but sequent (1.5) is only partially proved. From this partial proof, we can conclude that the following modification of the root sequent (1.1) is valid:

$$\texttt{x} > 0, \texttt{(y+42)>0} \vdash \langle \texttt{x=y;y=x+42;x=y-x;if(y>0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (1.6)$$

(We added $\texttt{(y+42)>0}$ to the left side of (1.1), as additional assumption). This sequent can be proven by replaying the original proof, where now both branches

[2] The simplified rule assumes the absence of side effects or exceptions which might be caused by b.

would close. The path leading to the closed proof of sequent (1.4) replays identically. In addition, sequent (1.5) now can be proved because the following variant of sequent (6) is closed immediately, due to contradicting assumptions:

$$\texttt{x} > \texttt{0}, (\texttt{y+42})\texttt{>0}, \neg((\texttt{y+42})\texttt{>0}) \vdash (\texttt{y} \leftarrow \texttt{y+42} \parallel \texttt{x} \leftarrow \texttt{42})\langle p_2\,;q\rangle\phi$$

An interesting remark is that if we were using a *perfect* prover, i.e., a prover that only returns *false* whenever the program under scrutiny does not fulfil the property being analysed, then in sequent (1.6) the set of formulae '`x`>`0`, (`y+42`)>`0`', would correspond to the *weakest precondition* of the program in the modality. However, in an ordinary set up the prover may fail to close a proof branch even if the sub-goal on that branch is valid. This may happen because of, for instance, lack of 'proving power', the strategies being used, or lack of code annotations (e.g., loop invariants). Still, even if the proof attempt fails, one can get close branches out of it. Thus, the branch condition of a closed branch forms a *sufficient precondition*, which is not necessarily the weakest precondition. Thereby, the true power behind partial proofs resides in the possibility of finding sufficient preconditions by extracting branch conditions from closed branches.

1.4 Combining Static and Dynamic Verification Through Partial Proofs

Enhancing verification techniques by combining them with other verification techniques is a practice that is getting more and more attention. In this work, we are mainly interested in the combination of static and dynamic verification techniques.

Combining static and dynamic verification techniques can allow the verification process to deal with properties with a greater ease. For instance, instead of possibly adding complicated abstractions to a property in order to statically handle the result of a call to a procedure belonging to an external library, one can attempt to verify such properties by using a dynamic verification technique that directly checks the results of such procedure calls at runtime.

In addition, such combinations can introduce benefits regarding the verification performance. For instance, by using static verification techniques one can improve the performance of the dynamic ones by ignoring at runtime the verification of all the properties which were proved correct statically.

1.4.1 Combining Static and Dynamic Verification Techniques

There are several possible technique combinations which can be analysed. The combination of testing and static verification techniques is one of the most explored ones, e.g., [13, 19, 47, 57, 59, 77, 89]. Here, static verification can be used, for instance, to limit the dynamic efforts by filtering test cases, or to accomplish high coverage of the test cases.

Another possibility is the combination of runtime verification and static verification techniques. For instance, in [30] a static verification technique which reduces runtime instrumentation is used to improve the efficiency of

runtime monitoring based on tracematches, and in [83] runtime verification is integrated with static code analysis in order to generate monitors which will allow to both check for possible faults in the system under scrutiny, and eliminate false positives obtained statically.

In particular, in this work we pay special attention to the combination of runtime verification and deductive verification, the combination of testing and deductive verification, and the combination of testing and runtime verification. For instance, before using runtime verification, one may attempt to prove some of the properties in the specification of the SUT by using deductive verification. Then, all the properties proved correct statically can be removed from the specification. This would result in an improvement on the performance of the runtime monitoring, as the monitor generated using runtime verification would only focus on the properties which were not proved correct statically. In addition, whenever a deductive verification proof attempt for a certain property in the specification does not succeed, i.e., the proof is not close, one can use testing to analyse why it was not possible to statically verify that property, e.g., one can test whether a procedure is actually returning the result expected in its specification. As result one may get hints towards finding either issues in the code of the SUT, or issues in the specification.

1.4.2 Partial Proofs as a Means for Combining the Techniques

The previously mentioned examples for combining verification techniques can be extended by considering the use of partial proofs in the verification process. If the deductive verification of a property results in a partial proof, then one can extract sufficient preconditions from them and use their negation to strengthen the properties. Then, in the case of the first example, one can strengthen the properties in such a manner that the runtime monitor will verify (at runtime) only the parts of the properties which were not proved correct statically, instead of verifying the whole property (task which may be more expensive to accomplish). Regarding the second example, one can strengthen the properties in such a manner that the test cases will be focused only on analysing the parts of the code in the SUT which do not comply with the property in the specification.

In this work we study how one can use partial proofs as a means for the combination of verification techniques. The ideas described above are, in fact, two examples of combinations which we have explored. Below, we briefly describe three results achieved as an outcome of this study. Note that we elaborate on these results in the different chapters of this work.

Combining Static and Runtime Verification

This thesis has been mainly developed in the context of the research project *Unified **St**atic and **R**untime **V**erification of **O**bject-**Or**iented **S**oftware*, or StaR-VOoRS for short. This project, which was funded by *The Swedish Research Council* (Vetenskapsrdet), had as a main purpose the development of a methodology for specifying and verifying both data- and control-oriented properties of object-oriented software systems, in a unified manner.

As a starting point to address this objective, Ahrendt et al. proposed in [14] a verification framework which combines the use of runtime verification with deductive verification. In short,

(i) Low effort deductive verification is used to verify those parts of the properties which may be confirmed statically;

(ii) the previous results, even if they are only partial, are used to refine the original specification of the properties such that the monitors generated by using runtime verification will not have to check at runtime the statically verified parts of the properties.

Furthermore, [14] presents initial ideas about an automaton-based specification language, called *ppDATE*, which captures both the description of control-oriented and data-oriented properties. Basically, this language consists of a transition system alike the *DATE* formalism, whose states may include Hoare triples describing properties about the methods of the SUT.

One of the main contributions of this thesis consists in the full development of the syntax (both abstract and concrete), the grammar, and the formal semantics of *ppDATE*. Moreover, as a first approach on the use of the framework described above, this thesis introduces the combination of the deductive verifier KeY, and the runtime verifier LARVA, in order to verify Java programs. Such combination is accomplished with the implementation of a verification tool which automatically combines the use both of the previous tools. Regarding the use of partial proofs, they play a fundamental role in this work, as they work as a means to combine KeY and LARVA in a manner that the monitor generated by LARVA only has to check the parts of the properties which were not statically verified by KeY. This reduces the overhead added to the system by the monitors.

Combining Testing with Static and Runtime Verification

Test-driven development (TDD) [20] is a development technique where test cases are used in order to guide the development of a system. Considering that the properties of a system capture both data- and control-oriented aspects, the use of TDD can be extended by including MBT (model-based testing) as part of its workflow.

In this work we integrate the use of deductive and runtime verification into the workflow of a testing focused development methodology based on TDD and MBT, with the help of partial proofs. In particular, TDD is integrated with deductive verification as an aid in the development of the data-oriented aspects, whereas model-based testing is integrated with runtime verification as an aid in the development of the control-oriented aspects.

As a result of such integration, the extended testing development methodology features the benefits of TDD and model-based testing, but enhanced. For instance, thanks to the use of deductive verification, one has an early detection of bugs which may be missed by the traditional used of TDD. In addition, thanks to the use of runtime verification, one can validate the overall system with respect to the model.

Regarding the role of partial proofs in these combinations, they are used to (automatically) generate test cases for the parts of the proofs which were

not closed by performing deductive verification in the properties featured for developing the system while considering data aspects. These test cases are used to enhance the (traditional) application of TDD.

Combining Deductive Verification with Dynamic Verification

This works also contributes in the combination of verification techniques by presenting a new methodology which uses partial proofs local to the transitions of a provided model, in order to infer global trace conditions which impose restrictions over the execution of the SUT.

Given a property associated to a state of the model, this methodology starts by performing reachability analysis, i.e., analysing which transitions can be taken to reach this state.

Next, the methods associated to the incoming transitions are used to statically verify that these transitions have the desired property as a postcondition. This results in local (possibly) partial proofs for them.

Then, closed-path conditions are extracted from the partial proofs, and are backwards propagated through the transitions of the model. Such conditions guarantee that, if any of these transitions is taken, a system trace fulfilling them will lead the system towards the desired state in the model, and the provided property will hold when that state is reached. Therefore, closed-path conditions come as a generalisation of the idea of weakest precondition for the methods. These steps are repeated until the initial state is reached.

Finally, the property is backwards propagated to the initial state, together with a system trace going from the initial state of the model to the desired state, which is created using the results of the reachability analysis previously performed, represent a trace condition for the system.

Applications for the use of trace conditions include (global) test case generation, state invariant verification, and runtime verification.

1.5 Perspectives

Section 1.4 presents different achievements of this work. Here, we discuss possible manners in which one can move forward in the same direction.

It is true that considering the use of partial proofs in the verification process has given us many interesting results. Still, one may wonder to what degree the property is covered by such a proof. In the context of StaRVOOrS, understanding such a coverage degree can give us a notion of the optimality of the monitor generated by this tool. Regarding the inference of global trace conditions, when using the trace conditions for testing, having some coverage metrics can allow us to introduce the notion of, for instance, *global trace coverage*, i.e., how much of the overall system is covered by the trace conditions. To estimate how much of a property is covered by a partial proof, we consider the possibility of integrating into our work the results obtained by Beckert et al in [27]. That work introduces the notion of *state space coverage* for partial proofs and uses it to compute an estimation of the percentage of a property covered by a partial proof of it. In addition, the fact that these ideas were implemented in a prototype tool which uses the same deductive verifier as we

do in this thesis, i.e., KeY, makes this integration more appealing to extend our work.

In addition, one can also focus on individual extensions. For instance, in the case of StaRVOOrS, one can opt to extend the workflow of its framework by integrating it with other verification techniques, e.g., testing, model checking. Another possibility is extending the semantics of *ppDATE* by including the use of timers on them. This would require to extend the semantics rules, and to adapt the proof of correctness for the translation of *ppDATE*s to *DATE*s.

Regarding the proposed testing focus development technique, one can look for systems developed using TDD and analyse them using our technique instead. This can give us more experience on how to use our technique, and further evidence on its benefits.

Regarding the inference of global trace conditions, as at the moment the different tools used to develop our case studies are manually connected, implementing a complete tool chain which automatically applies our methodology would be a great step forward. This would make the use of our methodology more appealing. In addition, one can elaborate in the analysis of the applications to fully explore the potential of the usage of our methodology. For instance, one can expand the theory about coverage by proposing and analysing global coverage methodologies. Moreover, one can perform experiments to compare, in terms of coverage, how the test cases generated from global trace conditions perform w.r.t. test cases generated with more standard testing techniques. Finally, we consider the possibility of introducing code annotations to modularise the inference of trace conditions. For instance, given a system divided into several layers of application, if we already know some sufficient preconditions for certain methods in a particular layer, we can annotate them in the source code to lift those results to another layer. This can be of a great aid to infer sufficient preconditions describing necessary conditions to go from one layer of the system to another one.

1.6 Contributions of the Thesis

The contribuitions of this thesis have been disseminated in the following documents: three peer reviewed conference papers [10,37,38], one peer reviewed journal article [11], one user manual [5], and one technical report [36]. This section presents a brief description of each one of these works, and outlines the contributions of Mauricio Chimento in all of them.

Note that [11], [5], [38], and [36], correspond to the chapters 2, 3, 4, and 5 of this thesis, respectively. Regarding [10] and [37], as they are subsumed by [11], we decided not to include them as part of this thesis. Anyhow, below we describe the contributions of Mauricio Chimento in these works. In addition, the format of all [5, 11, 38] were adapted to suit the required format for this thesis, all their references were unified and moved to the bibliography of the thesis, some typos were fixed, and some minor changes were introduced to improve the readability of the text and to clarify some explanations (but without affecting the contributions of the material that is already published).

A Specification Language for Static and Runtime Verification of Data and Control Properties

Paper [10] presents the development and formalisation of a notation language, called *ppDATE*, as an extension of the control-flow property language *DATE*, which is used in the runtime verification tool LARVA, and shows how specifications written in this notation can be analysed both using the deductive theorem prover KeY and the runtime verification tool LARVA. In addition, by using *ppDATE* to describe the corresponding specification, the STARVOORS verification framework is applied to Mondex, an electronic purse application.

Statement of contribution: The contributions of Mauricio Chimento on this paper are: (i) collaborating on the formalisation of the *ppDATE* notation to describe the Hoare triples associated to the states of a *ppDATE*; (ii) formalising the Hoare triples which are part of the *ppDATE* specification for the Mondex case study; (iii) applying the STARVOORS verification framework to the Mondex case study; and (iv) performing some experiments to analyse (and compare) the overhead added to Mondex by the monitor generated using STARVOORS, and by the monitor that would be generated without using static verification to analyse the Hoare triples.

StaRVOOrS - A Tool for Combined Static and Runtime Verification of Java

Paper [37] presents the tool STARVOORS, which aims at both the specification and verification of properties by combining the use of runtime verification and static verification. This tool is fed with a Java program and a ppDATE specification of the program, and automatically generates a monitor in order to runtime verify the provided program. In order to do so, STARVOORS combines the deductive theorem prover KeY and the runtime verifier LARVA. In addition, the effectiveness of this tool is demonstrated by applying it to Mondex, an electronic purse application.

Statement of contribution: The contributions of Mauricio Chimento on this paper are: (i) a refinement of the original verification framework proposed in [14]; (ii) full development and implementation of the tool STARVOORS; (iii) and demonstrating the effectiveness of the tool by applying it to the Mondex case study.

Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools

The journal article [11], which corresponds to the second chapter of this thesis, is an extension of the material presented in both [10] and [37]. In this paper, *ppDATE* is introduced as a proper specification language, and not just a simple notation. This is accomplished by introducing its syntax, grammar, and formal semantics. In addition, in order to cover new features of the *ppDATE* specification language, this paper introduces minor modifications into the algorithm used to translate a *ppDATE* specification into a *DATE* one, and provides the proof of correctness of such algorithm. Moreover, it

demonstrates the advantages of using the StaRVOOrS tool in two case studies.

Statement of contribution: The contributions of Mauricio Chimento on this paper are: (i) defining the syntax, the grammar (except the grammar of the templates), and the semantics (except the semantics of actions) of *ppDATE*; (ii) upgrading the translation algorithm to cover new features of the *ppDATE* specification language; (iii) proving the correctness of the translation algorithm; (iv) and using the StaRVOOrS tool to verify the SoftSlate case study. Note that in (i), Mauricio Chimento proposed initial versions for the definitions, and refined then with input from the co-authors. In addition, regarding (iii), Mauricio Chimento developed the whole proof of correctness on his own. However, some high level ideas related to the formalisation of the correctness theorem and its proof, in particular the introduction of the coupling invariants, were proposed by the co-authors.

StaRVOOrS User Manual (release 1.7)

The StaRVOOrS user manual (release 1.7) [5], which corresponds to the third chapter of this thesis, gives a high level explanation about how the StaRVOOrS tool works, provides an intuitive description of the *ppDATE* specification language, shows how to write a *ppDATE* specification in the input language of this tool by introducing its concrete grammar, and provides a complete example on how to use the tool.

Statement of contribution: The contributions of Mauricio Chimento in this work are: (i) the introduction of the concrete grammar to write *ppDATE* specifications as a script, fact which is essential to use the tool; (ii) the introduction of timers in *ppDATE*; (iii) the introduction of AspectJ features in *ppDATE*, which allows the use of StaRVOOrS in the presence of active objects.

Testing Meets Static and Runtime Verification

Paper [38], which corresponds to the fourth chapter of this thesis, introduces a testing focused development methodology which is based on a combination of *test-driven development* (TDD) and *model-based testing*. This technique is enhanced by integrating static and runtime verification into its workflow. In particular, TDD is integrated with (static) deductive verification, and model-based testing is integrated with runtime verification. As a result of this integration, the proposed methodology features the benefits of TDD and model-based testing, but enhanced with better test coverage in the former, and a validation analysis for the overall system with respect to the model in the latter.

Statement of contribution: Mauricio Chimento is the main author of this work. His principal contributions to it are: (i) full analysis and development of a testing focused development methodology combining TDD and model-based testing; (ii) enhancement of this development technique by integrating deductive and runtime verification on its workflow; and (iii) application of this technique in all its extent to develop a concrete example consisting of a small bank system.

Inferring Global Trace Conditions From Local Partial Proofs

The technical report [36], which corresponds to the fifth chapter of this thesis, presents a methodology which uses the power of local (partial) proof attempts to infer global trace conditions for a system. Given a model of the system, a state of the model, and a property associated to this state, this methodology is applied by following the next three stages: *Reaching Transitions Analysis*, *Backwards Reachability Tree Computation*, and *Trace Condition Inference*. In short, in the first stage one extracts information from the model about the reachability of its states, i.e., which transitions can be taken to reach each state. That information is used in the following stage, in addition to sufficient preconditions extracted from (partial) deductive proofs local to the transitions of the model, to compute a backwards reachability tree. This tree represents the manners in which the sufficient preconditions are backwards propagated through the transitions of the model. In addition, such preconditions guarantee that, if any of the transitions is taken, a system trace fulfilling them will lead the system towards the desired state in the model, and the provided property will hold when that state is reached. Therefore, sufficient preconditions come as a generalisation of the idea of weakest precondition for the methods. Finally, in the last stage the backwards reachability tree is analysed to infer trace conditions for the system. Such trace conditions consists of a property backwards propagated to the initial state through the transitions of the model, together with a system trace going from the initial state of the model to the provided state. Applications for the use of trace conditions include test case generation, runtime verification, and state invariants verification.

Statement of contribution: Mauricio Chimento is the single author of this work. The contributions in this work are: (i) full development and analysis of a new methodology for inferring trace conditions from local partial proofs; (ii) proposing concrete applications for the use of the methodology; and (iii) applying the methodology to two case studies. As a remark, some of the ideas presented in this work evolved from discussions with Wolfgang Ahrendt. In addition, Wolfgang Ahrendt provided valuable input to improve the presentation of this work.

CHAPTER
TWO

VERIFYING DATA- AND CONTROL-ORIENTED PROPERTIES COMBINING STATIC AND RUNTIME VERIFICATION: THEORY AND TOOLS

W. Ahrendt, J. M. Chimento, G. Pace, and G. Schneider

Abstract

Static verification techniques are used to analyse and prove properties about programs before they are executed. Many of these techniques work directly on the source code and are used to verify data-oriented properties over all possible executions. The analysis is necessarily an over-approximation as the real executions of the program are not available at analysis time. In contrast, runtime verification techniques have been extensively used for control-oriented properties, analysing the current execution path of the program in a fully automatic manner. In this article, we present a novel approach in which data-oriented and control-oriented properties may be stated in a single formalism amenable to both static and dynamic verification techniques. The specification language we present to achieve this that of ppDATE, which enhances the control-oriented property language of DATE, with data-oriented pre/postconditions. For runtime verification of ppDATE specifications, the language is translated into DATE. We give a formal semantics to ppDATE, which we use to prove the correctness of our translation from ppDATE to DATE. We show how ppDATE specifications can be analysed using a combination of the deductive theorem prover KeY and the runtime verification tool LARVA. Verification is performed in two steps: KeY first partially proves the data-oriented part of the specification, simplifying the specification which is then passed on to LARVA to check at runtime for the remaining parts of the specification including the control-oriented aspects. We show the applicability of our approach on two case studies.

2.1 Introduction

Runtime verification has been touted as a practical verification technique, and although it does not provide program analysis before deployment, it can check correct behaviour post-deployment by observing whether actual execution paths at runtime conform to the specification. Runtime verification scales up much more effectively than static analysis both in terms of performance and in terms of applicability to diverse contexts in which a program may interact with various other systems, services, and libraries.

Despite the fact that overheads induced by runtime verification might be regarded as small when compared to the computational effort required for static analysis, the fact that it is done while the software is live can be problematic and prohibitive for certain systems. In this paper we present an approach to address the issue of runtime overheads through the use of static, deductive verification — an approach which also has the benefit of being able to verify parts of the specification a priori for all potential execution paths, leaving only parts which could not be proved before deployment to be checked dynamically.

Apart from the computational power required to perform the analysis, deductive and runtime verification have largely been applied to disjoint areas — whereas deductive analysis has been extensively used to verify properties focusing on a system's data, e.g., [9, 54, 69, 73], runtime verification has been extensively used to verify control-flow properties with reasonable overheads [24, 35, 45, 79]. Combining the two approaches has the additional benefit that static analysis might be more effective in proving the parts of a specification which dynamic analysis might struggle most with. The challenge is thus to design a specification language which allows the expression of combined data- and control-flow properties in such a manner that they can be effectively decomposed for the application of different verification techniques.

The StaRVOOrS framework [14] addresses these issues by identifying a specification notation for such properties and a verification methodology combining static and dynamic analysis to verify combined control- and data-oriented properties. Although one may envisage different ways to combine static and dynamic analysis tools, a crucial requirement is that the specification languages used in the tools chosen are either identical, or can be somehow combined to allow for rich specifications getting the best of both approaches. Similar to *mode automata* [75] we have chosen to adopt an automata-based specification language (for the control-flow properties) but extended with data-flow properties encoded in the different states of the formalism.

This article is a significantly extended and revised version of two papers. In [10] we introduced the formalism *ppDATE*, where parts of the syntax where left underspecified, and we gave a high-level description of the algorithm to translate *ppDATE* into *DATE* [45], the formalism used in the runtime verification tool Larva [46]. In [37] we presented the tool StaRVOOrS, a full implementation of the framework introduced in [10, 14].

The novel contributions of this paper, going beyond the results reported in [10] and [37] are the following: i) We present a complete formal definition of *ppDATE* automata, including a formal semantics for the formalism (Sec. 2.5); ii) A proof of soundness of the algorithm to translate from *ppDATE* specifications into *DATE* ones (Sec. 2.7). iii) The application of our approach

to SoftSlate Commerce, an open-source Java shopping cart web application (Sec. 2.9); iv) A description of the results of the case study including an analysis of the verification process providing evidence that our approach reduces the overhead of the runtime monitoring (Sec. 2.9).

Structure of the paper Sec. 2.2 provides background information regarding the verification techniques used on this paper. Sec. 2.3 introduces informally the specification language *ppDATE*. Sec. 2.4 introduces the StaRVOOrS framework and provides a description of its workflow. Sec. 2.5 presents formally the specification language *ppDATE*, and Sec. 2.6 provides its operational semantics. Sec. 2.7 gives a translation algorithm from *ppDATE*s into *DATE*s, and provides a proof of correctness. Sec. 2.8 presents a fully automated tool which implements the StaRVOOrS framework. Sec. 2.9 and 2.10 discuss two case studies which illustrate the benefits of using StaRVOOrS for verifying software. Sec. 2.11 discusses related work. We conclude this paper in Sec. 2.12.

2.2 Preliminaries

The work presented in this article is centred around static and runtime verification of Java systems. To implement these verification techniques, we use the deductive verifier KeY and the runtime verifier Larva. In this section, we introduce these tools at a high level of abstraction, but with sufficient detail to enable the understanding of the rest of the paper.

2.2.1 The deductive verifier KeY

KeY [9] is a deductive verification tool for data-centric *functional correctness* properties of Java source code. KeY generates proof obligations in *dynamic logic* (DL), a modal logic for reasoning about programs. DL extends first-order logic with two modalities, $\langle p\rangle\phi$ and $[p]\phi$, where p is a program and ϕ is another DL formula. The formula $\langle p\rangle\phi$ is true in a state s if there *exists* a terminating run of p, starting in s, resulting in a state where ϕ holds. The formula $[p]\phi$ holds in a state s if *all* terminating runs of p, starting in s, result in a state in which ϕ holds. For deterministic programs p, the only difference between the two modalities is that termination is *stated* in $\langle p\rangle\phi$, and *assumed* in $[p]\phi$.

KeY features (static) verification of Java source code annotated with specifications written in the *Java Modelling Language* (JML) [72]. JML allows for the specification of pre- and postconditions of method calls, and class/interface invariants. The main features of KeY are the translation of JML annotated Java programs to Java DL, and a theorem prover for validity of Java DL formulae, using a *sequent calculus*, covering almost all features of sequential Java (with the exception of generics and floating-point types currently). Given a set of formulae Γ, the sequent $\Gamma \vdash \langle p\rangle\phi$ holds if p, when starting in a state fulfilling all formulae in Γ, terminates in a state fulfilling ϕ. The calculus uses the *symbolic execution* paradigm. For that, DL is extended by *explicit substitutions*. During the symbolic execution of p, the effects of p are gradually, starting from the front, turned into explicit substitutions. Thereby, after some proof steps, a certain prefix of p has turned into a substitution σ, representing the effects so far, while a remaining program p' is yet to be executed. While verifying p, an intermediate proof node may look like $\Gamma \vdash \sigma\langle p'\rangle\phi$. It tells us that, if Γ was

true before the original program p, and σ is the accumulated effect up to now, then ϕ will be true after executing the remaining program p'.

As an example, consider a proof of the following DL sequent:

$$\texttt{x} > 0,\ \texttt{y} > 0 \ \vdash \langle \texttt{x=x+y;y=x-y;x=x-y;if(x\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (2.1)$$

(where p_1, p_2, and q are Java fragments and ϕ is some postcondition). The sequent says that in each state where `x` and `y` are positive, the program given in the modality (which first swaps `x` and `y` using arithmetics) will terminate and result in a state where ϕ holds. When proving this sequent, the KeY prover will first, in a number of steps, turn the three leading assignments into explicit substitutions, apply the first to the second, the result to the third, and perform arithmetic simplification, arriving at

$$\texttt{x} > 0,\ \texttt{y} > 0 \ \vdash (\texttt{x} \leftarrow \texttt{x+y} \parallel \texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle \texttt{if(x\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi$$

where $(\texttt{x} \leftarrow \texttt{x+y} \parallel \texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})$ denotes the explicit (parallel) substitution resulting from symbolic execution of the first three statements. A 'right-win' semantics is adopted to resolve clashes in substitutions, such that the above simplifies to:

$$\texttt{x} > 0,\ \texttt{y} > 0 \ \vdash (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle \texttt{if(x\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi$$

In general, most proofs branch over case distinctions, often triggered by Boolean decisions in the source code. The branching happens by applying rules like the following, simplified[1] `if` rule:

$$\texttt{if}\ \frac{\Gamma, \sigma(b) \vdash \sigma\langle s_1 \ \ \omega\rangle\phi \qquad \Gamma, \sigma(\neg b) \vdash \sigma\langle s_2 \ \ \omega\rangle\phi}{\Gamma \vdash \sigma\langle \texttt{if}\ b\ s_1\ \texttt{else}\ s_2 \ \ \omega\rangle\phi}$$

In our example, applying the `if` rule to the latest sequent results in splitting the proof into two branches, with the following sequents, respectively:

$$\texttt{x} > 0,\ \texttt{y} > 0,\ (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})(\texttt{x\%2} = 0) \ \vdash (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle p_1\,\texttt{;}\,q\rangle\phi$$
$$\texttt{x} > 0,\ \texttt{y} > 0,\ (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})(\neg(\texttt{x\%2} = 0)) \ \vdash (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle p_2\,\texttt{;}\,q\rangle\phi$$

Applying the substitution on the left side of either sequent results in:

$$\texttt{x} > 0,\ \texttt{y} > 0,\ \texttt{y\%2} = 0 \ \vdash (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle p_1\,\texttt{;}\,q\rangle\phi \quad (2.2)$$
$$\texttt{x} > 0,\ \texttt{y} > 0,\ \neg(\texttt{y\%2} = 0) \ \vdash (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle p_2\,\texttt{;}\,q\rangle\phi \quad (2.3)$$

Note that in this step, by applying the swapping substitution, the branching condition (`x` being even or odd) on the *state after swapping* got translated into a condition on the *prestate* of the original program p, before the swapping. The resulting sequents tell us, among other things, that if `y` is even (respectively odd) in the prestate of p, then path p_1 (respectively p_2) is taken in the execution of p. In general, when building a proof in such a symbolic manner, the left side of sequents accumulate conditions on the original prestate through a particular execution path.

[1] The simplified rule ignores side effects or exceptions possibly caused by b.

Once all proof branches are closed, we have a *complete proof* of the root sequent. However, a proof attempt may result in a *partial proof*, only, where some proof branches are closed and others are not. Such partial proofs are important for the work presented in this article. In the above example, consider a partial proof where the left branch, i.e., the sub-proof for sequent (2.2), is closed, whereas the right branch, i.e., the sub-proof for sequent (2.3), is *not* closed. From this partial proof, we can conclude that the following modification of the root sequent (2.1) is valid:

$$\texttt{x} > 0, \texttt{y} > 0, \texttt{y\%2} = 0 \;\vdash\; \langle \texttt{x=x+y;y=x-y;x=x-y;if(x\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \tag{2.4}$$

(We added $\texttt{y\%2} = 0$ to the left side of (2.1), as additional assumption.) This sequent can be proven by replaying the original proof, where now both branches would close. The left branch closes as the sub-proof for (2.2) will replay identically. The right branch closes because the following variant of (2.3) can be closed immediately, due to contradicting assumptions:

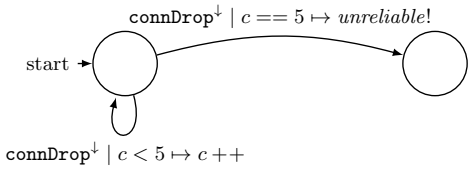
$$\texttt{x} > 0,\ \texttt{y} > 0,\ \texttt{y\%2} = 0,\ \neg(\texttt{y\%2} = 0) \;\vdash\; (\texttt{y} \leftarrow \texttt{x} \parallel \texttt{x} \leftarrow \texttt{y})\langle p_2\,\texttt{;}\,q\rangle\phi$$

2.2.2 The runtime verifier LARVA

LARVA[2] [46] is an automata-based runtime verification tool for Java programs. As with many other runtime verifiers, LARVA automatically generates a runtime monitor from a property written in a formal language, in its case using *Dynamic Automata with Timers and Events* (DATEs) [45]. Transitions in a *DATE* are of the form: *event* | *condition* $\mapsto$ *action*, where *event* is what triggers the transition, the *condition* is checked and must hold in order for the transition to take place, and the *action* is a code snippet to be performed when taking the transition (after checking the condition). DATEs are an extension of timed automata — they are effectively finite state automata, whose transitions are triggered by system events (primarily entry points $\texttt{f}^{\downarrow}$ and exit points $\texttt{f}^{\uparrow}$ of methods) and timers, but augmented with: (i) A symbolic state which may be used as conditions to guard transitions and can be modified via actions also specified on the transition; (ii) replication of automata, through which a new automaton is created for each discovered instance of an object; (iii) communication between automata using standard CCS-like channels with $c!$ acting as a broadcast on channel c and which can be read by another automaton matching on event $c?$. Full details of the formalisation of DATEs can be found in [46].

The automata illustrated in Fig. 2.1 represent an example of *DATE* automata describing a property which should hold during a connection. The first automaton ensures that if the connection drops (event $\texttt{connDrop}^{\downarrow}$) occurs five times, a message is broadcast (over channel *unreliable*) to highlight the fact that the connection port is unreliable. The second automaton (with the *foreach* keyword) ensures that every time a file transfer is initiated, an automaton is created to monitor that transfer. If during the transfer (i.e. between the events $\texttt{start}^{\downarrow}$ and $\texttt{end}^{\downarrow}$) one receives event *unreliable*?, no further transfers may occur.

[2] *Logical Automata for Runtime Verification and Analysis.*



foreach transfer :

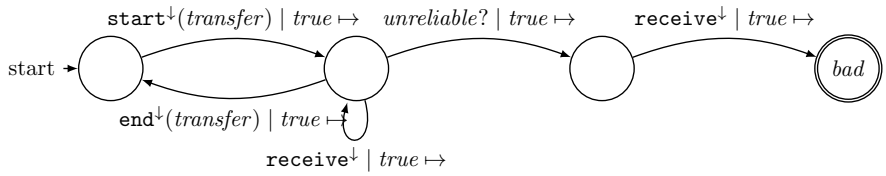


Figure 2.1: Example of a *DATE* specification.

In order to monitor a system using LARVA, the user must provide the system to be monitored (a Java program) and a set of properties in the form of a LARVA script (a textual representation of DATEs). LARVA transforms the set of properties into monitoring code together with AspectJ code to link the system with the monitors. Since the Java byte code is used for instrumentation, it is possible to monitor third-party software with LARVA, though knowledge of methods names is still required.

2.3 *ppDATE*: A Specification Language for Data- and Control-oriented Properties

In many cases, verification tools perform more effectively on a particular style of specification. In combining two different verification tools which use very different analysis techniques, one challenge is that if we adopt an off-the-shelf language, we cannot expect to derive useful verification results from both tools. Given that deductive verification tools like KeY perform much better on data-centric properties, while runtime verification tools like LARVA perform better on control-flow properties, we have defined a specification language to combine the two types of properties. In real scenarios, there is often a need to specify both, rich data constraints and legal execution sequences.

Data-oriented properties are typically written in expressive formalisms (like first-order logic), but typically give invariants about specific points in the execution of a system, rather than properties across traces of execution. JML is one such language, which focuses primarily on pre/postconditions of method calls and class invariants, but is not well suited for specifying which sequences of events or states are correct. In contrast, *control-oriented* specification languages specialise primarily on identifying legal sequences of events or states, for instance using automata or temporal logics. Although constraints about the data are possible, they are usually cumbersome and greatly increase the computational complexity required to verify them. *DATE*

is one such specification language.

Coding control-flow into data-centric languages, like coding legal execution traces via model/ghost fields in JML, or including data-flow information in control-centric languages, like considering variable updates as events in *DATE* specification, can lead to substantial increase in the complexity of the specification from an understandability and/or verification perspective.

In order to address this, we propose *ppDATE*, a formalism to deal with both types of properties ensuring understandability and tractability of analysis using the StaRVOOrS verification framework. *ppDATE* [10] is an automata-based formalism to specify both control- and data-oriented properties. *ppDATE*s are basically transition systems with states and transitions between states. Transitions are labelled by a trigger (tr), a condition (c), and an action (a). Together, the label is written $tr \mid c \mapsto a$. A transition is *enabled* to be taken whenever its trigger is active and its condition holds. A trigger is activated by the occurrence of either a visible system event such as the invocation or termination of a method execution, or a *ppDATE* internal event generated by certain actions labelling other transitions. If a transition is taken, we will say that it *fires*. The conditions may depend on the values of *system variables* (i.e., variables of the system under scrutiny) and the values of *ppDATE variables*. The latter can be modified via actions in the transitions. *ppDATE* states represent the status of an *observer* of a system (rather than, directly, the status of a system itself). Note that each state essentially represents the set of observed system traces leading to that state. The language also offers parallelism on the specification side, in the sense that different *ppDATE*s run in parallel, possibly communicating which each other through events, and possibly creating new *ppDATE*s on demand. This parallelism allows for a strong separation of concerns in the specification.

In addition to the above, a particular feature of the *ppDATE* is that states may be tagged with any number of Hoare triples, to specify the computation of a method in a history-context sensitive way. For instance, assume that a *ppDATE* state q is tagged with the Hoare triple $\{\pi\}foo\{\pi'\}$. This means that, if *foo* is invoked after a system trace which led the observer to q, and if furthermore π holds at the time of the invocation, then π' should be satisfied upon termination of this execution of *foo*. This allows for data-centric specification of individual methods' behaviour (Hoare triple), however in a control sensitive manner (state).

Compared to usual automata based (or temporal logic based) specification approaches, *ppDATE* is more expressive concerning the computation on data. Compared to data-centric pre/post-specification (like, e.g., JML), *ppDATE* can avoid the coding of some notion of status into additional data and additional constraints in the pre/postconditions.

To write a *ppDATE*, a good approach may be to, first, define the control-oriented properties, i.e., the automata. Next, one shall proceed to define the different Hoare triples. Finally, one places the Hoare triples on the appropriate states of the *ppDATE*.

Below, we provide a few examples of *ppDATE* specifications. On these examples, $tr^{\downarrow}$ and $tr^{\uparrow}$ have the same meaning as it was explained above for *DATE*.

Example 1. Let us consider a *coffee machine system* where after a certain

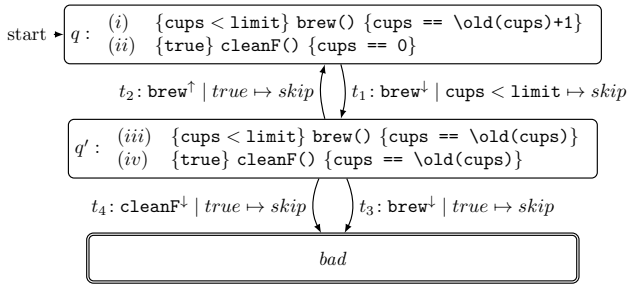


Figure 2.2: A *ppDATE* controlling the brew of coffee

amount of coffee cups are brewed, its filters have to be cleaned. If the limit of coffee cups is reached, the machine should not be able to brew any more coffee. In addition, while the coffee machine is active (a coffee cup is being brewed), it is not possible to start brewing another coffee, or to clean the filters.

Fig. 2.2 illustrates a *ppDATE* describing this part of the system. In other words, whenever the coffee machine is not active, i.e., the machine is not brewing a cup of coffee, and the method `brew` starts the coffee brewing process, then it is not possible either to execute this method again, or to execute the method `cleanF` (which initialises the task of cleaning the filter), until the initialised brewing process finishes.

The previous property can be interpreted as follows: initially being in state q, state which represents that the coffee machine is not active, whenever method `brew` is invoked and it is possible to brew a cup of coffee (i.e., the limit of coffee cups was not reached yet), then transition t_1 shifts the *ppDATE* from state q to state q'. While in q', state which represents that the coffee machine is active, if either method `brew` or method `cleanF` are invoked, then transitions t_3 or transition t_4 shift the *ppDATE* to state *bad*, respectively. This indicates that the property was violated. On the contrary, if method `brew` terminates its execution, then transition t_2 shifts the *ppDATE* from state q' to state q. Note that the names used on the transitions, e.g. t_1, t_2, etc, are not part of the specification language. They are included to simplify the description of how the *ppDATE* works.

In addition to this, the Hoare triples in state q ensure the properties: (i) if the amount of brewed coffee cups has not reached its limit yet, then a coffee cup is brewed; (ii) cleaning the filters sets the amount of brewed coffee cups to 0. Property (i) has to be verified if, while the *ppDATE* is on state q, the method `brew` is executed and its precondition holds. A similar situation stands for the property (ii) with respect to the method `cleanF`. Regarding state q', the Hoare triples in this state ensure the properties: (iii) no coffee cups are brewed; (iv) filters are not cleaned. Property (iii) and (iv) are verified if either method `brew` and method `cleanF` are executed, and their preconditions hold, respectively. Here, remember that this state represents that the coffee machine is active. Thus, if it occurs that either the method `brew` or the method `cleanF` are executed while the *ppDATE* is on this state, then, as this would move the *ppDATE* to state `bad`, one would expect the value of the variable `cup` to remain unchanged. This is precisely what is verified when either property (iii)

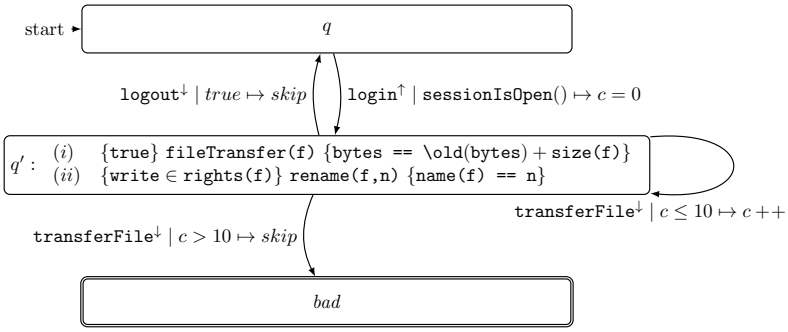


Figure 2.3: A *ppDATE* limiting file transfers

or (iv) are analysed.

Note that none of the Hoare triples makes reference to the state of the coffee machine, i.e. there is no information about whether the machine is active or not. This is due to fact that the state of the machine is implicitly defined by the states of the *ppDATE*. If the *ppDATE* is in state q, the coffee machine is not active. However, if it is in state q', then the machine is active. Therefore, it is possible to assume that on each state the Hoare triples are *context dependent* and thus contain such information. This is the reason why, we can describe properties with the same precondition, but with different postconditions depending on the state of the *ppDATE* in which they are placed. □

Example 2. In this example let us consider a *file system* where only 10 file transfers can be performed between a log in and log out of a user.

Fig. 2.3 illustrates a *ppDATE* describing part of the behaviour of this system. This *ppDATE* ensures the property: *no more than 10 file transfers take place in a single login session.* In other words, once a user logs in the system (`login`), she can only perform 10 file transfers (`transferFile`) before logging out (`logout`). This fact is tracked using the *ppDATE* variable c. This variable keeps count of the number of files transferred in a single session. Whenever a user logs in, the *ppDATE* moves to state q' and c is set to 0 (zero). While in q', this variable is increased by one every time a file transfer is performed. If at some point the user transfers a file but the value of c is bigger than 10, then the *ppDATE* moves to state *bad*, i.e., the property was violated.

In addition to this, the Hoare triples in state q' ensure the properties: (i) the number of bytes transferred increases when a file transfer is done; (ii) renaming a file works as expected if the user has the sufficient rights.

□

2.4 The StaRVOOrS Framework

The StaRVOOrS framework (STAtic and Runtime Verification of Object-ORiented Software), originally proposed in [14], combines the use of the deductive source code verifier KeY [9] with that of the runtime monitoring tool

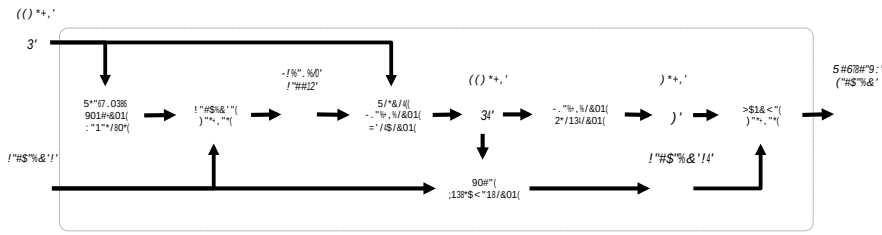


Figure 2.4: High-level description of the StaRVOOrS framework workflow

Larva [46], to analyse and monitor systems with respect to a *ppDATE* specification. Note that the definition of the specification language *ppDATE*, which enables the effective combination of the results from the two verification approaches, is a major contribution of StaRVOOrS. *ppDATE* allows our framework to naturally address the intrinsic differences between the verification tools — whereas one typically verifies data-centric properties in deductive verifiers like KeY, one typically focuses on control-flow properties using runtime verifiers like Larva.

The abstract workflow of the use of StaRVOOrS is given in Fig. 2.4. This workflow is applied fully automatically in four consecutive stages: *Deductive Verification*, *Specification Refinement*, *Translation and Instrumentation*, and *Monitor Generation*.

In the ***Deductive Verification*** stage, given a Java program P and a *ppDATE* specification S, the module Pre/post-Condition Generator transforms all the Hoare triples—assigned to the various states of S—into JML contracts, which are textually added to P as annotations of the respective methods. In this step, the association of pre/postcondition pairs to *ppDATE* states in S is lost, which is intentional and natural. Note that each *ppDATE* state represents the set of event histories leading to that state. The deductive verifier, however, offers analysis of the effect of methods *in terms of system data*, and has no notion of the history of events preceding a method call.[3] Once all JML contracts are generated, the Deductive Verifier module uses KeY in an attempt to statically verify each of them. The result is either a complete proof, or a partial proof where some branches are closed and others are not (see Sec. 2.2.1), or an entirely open proof, where no branches are closed. In our setting, partial proofs are the most common case. One reason is that we use KeY only fully automatically, not employing its interactive features. Also, we do not assume users to provide loop invariants, or similar annotations which support the prover. Finally, KeY has no knowledge of the *context* (*ppDATE* state) in which the Hoare triple at hand should hold. To illustrate this point, consider the Hoare triples (i) and (iii) from our (deliberately primitive) example in Fig. 2.2. The implementation of `brew()` is given by:

[3]There exist approaches to deductive verification which are history-aware, including a KeY version for the compositional verification of distributed systems [12]. These approaches are however much more heavyweight, both in terms of specification as well as verification, than what we are aiming at in this work. The same holds for approaches based on refinement.

```
public void brew() {
    if (!active && cups < limit)
       cups++;
}
```

KeY will produce partial proofs for these Hoare triples because the specification does not provide any information on how q and q' relate to the field `active`. In general, the missing information can be an arbitrary condition on the system state, more than just a Boolean as is the case here.

In the ***Specification Refinement*** stage,[4] the Partial Specification Evaluation module evaluates the results produced by KeY in order to refine S. This refinement is performed in two steps. In the first step, all fully verified Hoare triples are deleted, resulting in a *ppDATE* S'. Any Hoare triple related to a contract which is not fully verified by KeY is left in the states of S' to be verified at runtime. In the second step, S' is refined into a *ppDATE* S'' by strengthening the preconditions of those Hoare triples in S' which were *partially verified* by KeY. For that, the partial KeY proofs are analysed, to extract branch conditions corresponding to the closed branches of the proof. In the example in Sec. 2.2.1, that 'closed branch condition' is `y%2 = 0` in sequent (2.4). Note again that the branch condition is a condition *on the prestate* of the code being verified. Let us abbreviate the 'closed branch(es) condition' as *cbc* for now. A Hoare triple $\{\pi\}foo\{\pi'\}$ that was partially verified by KeY is clearly equivalent to having two Hoare triples $\{\pi \wedge cbc\}foo\{\pi'\}$ and $\{\pi \wedge \neg cbc\}foo\{\pi'\}$. However, as we know that the first one is valid (by the proof replay argument from Sec. 2.2.1), only the second one needs to be checked at runtime. For this reason, every Hoare triple $\{\pi\}foo\{\pi'\}$ in S' that was partially verified by KeY is replaced by $\{\pi \wedge \neg cbc\}foo\{\pi'\}$, resulting in S''. At runtime, checking such an optimised Hoare triple is trivial whenever π is false or *cbc* is true, as the postcondition does not need to be checked then. For instance, analysis of the partial proof of Hoare triple (i) in Fig. 2.2 will result in the closed branch condition $\neg$`active`. Therefore, (i) is replaced by {`cups < limit` $\wedge$ `active`} `brew()` {`cups == \old(cups)+1`} (we simplified away double negation). Note that, in cases where the history context, i.e., *ppDATE* state, is the *only* information that was missing to close a partial proof, *cbc* actually represents a refinement of the according *ppDATE* state to a condition on internal system data, which will always be true when *foo* is called in that state. We can remark already here that this is the phenomenon which made the monitoring speedup particularly dramatic in the Mondex case study, see Sec. 2.10.

In the ***Translation and Instrumentation*** stage, the Specification Translation module translates S'' into an equivalent specification in *DATE* format (D), which can be used by the runtime verifier LARVA (see the next stage). The most significant change of this translation is that the Hoare triples are translated away, using notions native to *DATE* (see Sec. 2.7.2). This change also requires to instrument P, through the Code Instrumentation module, in order to (i) distinguish between different executions of the same code unit, and to (ii) evaluate Hoare triples in the states of S'' at runtime. Regarding (i), method declarations get a new argument which is used as a counter for invocations

[4] For readability, we use $\wedge$ and $\neg$ in this paragraph, instead of the *ppDATE* syntax `&&` and `!`.

of this method. Regarding (ii), not every condition in a pre/postcondition of a Hoare triple can be directly written as a Java Boolean Expression, e.g., quantified expressions. Thus, methods which operationalise the evaluation of those conditions are added to P.

Finally, in the ***Monitor Generation*** stage, the instrumented version of P (P') and the *DATE* specification D are used by the Runtime Verifier module to generate a monitor M. For this, LARVA generates M from D by using aspect-oriented programming techniques to capture relevant system events. Such events allow to link P' with M. Later, once deployed, M and P' are executed together. If M identifies any violation at runtime, it will report an error trace for further analysis.

2.5 Formal Definition of *ppDATE*s

2.5.1 Notation

We will use the following notation to write quantified formulae, based on the notation used by Gries [61].

$$\forall\ x \cdot R(x) \cdot B(x)$$
$$\exists\ x \cdot R(x) \cdot B(x)$$

These formulae mean "for all x satisfying R, B is fulfilled" and "there exists x satisfying R for which B is fulfilled", respectively. Both R and B are formulae potentially containing x as a free variable. We will refer to R and B as the *range* and *body* of the quantified formula, respectively. This notation relates to standard (un-ranged) quantified formulae in the following way:

$$\begin{array}{lcl} \forall\ x \cdot R(x) \cdot B(x) & \equiv & \forall\ x \cdot (R(x) \rightarrow B(x)) \\ \exists\ x \cdot R(x) \cdot B(x) & \equiv & \exists\ x \cdot (R(x) \wedge B(x)) \end{array}$$

2.5.2 *ppDATE*

In this section we formally define the notion of *ppDATE* previously introduced in Sec. 2.3. In order to do so, we first introduce formal definitions for triggers, conditions and actions.

Definition 1. Given a set of method names Σ, the syntactic category of triggers is defined as follows:

$$\begin{array}{lcl} \mathit{trigger} & ::= & \mathit{systemtrigger} \\ & & \mid \mathsf{actevent?} \end{array}$$

$$\mathit{systemtrigger} ::= \mathsf{methodname}^{\downarrow} \mid \mathsf{methodname}^{\uparrow}$$

where $\mathsf{methodname} \in \Sigma$.

□

In the previous definition, *systemtrigger* matches a visible system event, such as the point of entry into a method or the termination of a method execution. Given a method name $\sigma \in \Sigma$, $\sigma^{\downarrow}$ represents entering method σ and $\sigma^{\uparrow}$ represents the termination of the execution of σ.

In addition, actevent represents an event generated by the execution of an action in a transition of a *ppDATE*, which we will call *action events*. This kind of events can only be generated by bang ("!") actions (see Def. 2). An action $h!$ generates the action event h, which in the next step can activate the trigger $h?$. This way, action events enable communication among *ppDATE*s, where $h!$ and $h?$ mean sending and receiving a message, respectively.

As we have mentioned before, whenever a transition is fired an action can be executed. The following shows the definition of actions.

Definition 2. Actions are syntactically defined as follows:

$$
\begin{aligned}
\mathit{action} ::= \ & \mathsf{skip} \\
| \ & v = e \\
| \ & \mathsf{actevent!} \\
| \ & \mathsf{create}(\mathit{template}, \overline{\mathit{args}}) \\
| \ & \mathit{action} \ ; \mathit{action} \\
| \ & \mathsf{if} \ \mathit{cond}_{\mathit{Sys} \cup V} \ \mathsf{then} \ \mathit{action} \\
| \ & \mathsf{Program}
\end{aligned}
$$

□

skip is the effect-less action. The '=' is an assignment operator, v is a *ppDATE* variable and e is a (side-effect free) expression that may depend on system variables and *ppDATE* variables; actevent! represents the generation of action event actevent; create represents the creation of a *ppDATE*, where *template* is a *ppDATE* template to be instantiated (see Def. 8), and $\overline{\mathit{args}}$ are the values which the formal parameters of *template* are instantiated with; the ';' is the sequence operator for actions; if-then is a conditional whose branching condition depends on the valuations of system variables (*Sys*) and *ppDATE* variables (V); and Program represents a *side-effect free* program (see Def. 3), i.e., it is restricted to not have any effect on the system which could in turn be observed by the (*ppDATE* generated) monitor. For instance, a Program could perform logging of system/monitor behaviour. More powerful Programs, which would for instance allow error recovery, are relevant, but left for future work.

Definition 3. A *side-effect free program* has the properties that

- its execution always terminates,
- the method calls on its body do not generate any observable system event,
- it does not interfere with the system under scrutiny, i.e., it does not modify the values of system variables.

□

Boolean expressions are used in different contexts: (i) conditions (c) of transitions; (ii) conditions of if-then actions, and (iii) pre- and postconditions (π, π') in Hoare triples. As a syntactic category for such Boolean expressions, we chose *Boolean JML expressions*. They extend *Boolean Java expressions*, and thereby allow Java methods as sub-expressions (like in '`m.get(k) == o`'). Additional features of *Boolean JML expressions* include universal and existential quantification, which are frequently used in Hoare triples, the ability

to refer in a postcondition to a) the return value (with '`\result`'), and b) the preexecution value of an expression (like in '`x == \old(x + y)`').

Definition 4. *Boolean JML expressions* (BJMLE) are recursively defined as follows:

- any side-effect free Boolean *Java* expression is a BJMLE,
- if `a` and `b` are BJMLEs, and `x` is a variable of type `t`, the following expressions are BJMLEs:
  - `!a`, `a&&b`, and `a||b`
  - `a ==> b` ("`a` implies `b`")
  - `a <==> b` ("`a` is equivalent to `b`")
  - `(\forall t x; a)`
    ("for all `x` of type `t`, `a` holds")
  - `(\exists t x; a)`
    ("there exists `x` of type `t` such that `a`")
  - `(\forall t x; a; b)`
    ("for all `x` of type `t` fulfilling `a`, `b` holds")
  - `(\exists t x; a; b)`
    ("there exists an `x` of type `t` fulfilling `a`,
    such that `b`")
- replacing any sub-expression `e` in a BJMLE with `\old(e)` gives a BJMLE,
- replacing any sub-expression in a BJMLE with `\result` gives a BJMLE, (well-typedness is context dependent, see Def.5)

□

We do not give a formal definition of the semantics of BJMLE here, just the following comments. The meaning of negation, conjunction, disjunction, implication, and equivalence are standard. The same is true for the first two forms of quantification. Concerning the other two forms, "... `a; b)`", they relate to standard quantification in exactly the same way as was explained in Sec. 2.5.1. (The only difference is that there we discussed meta-level notation, whereas BJMLE is part of *ppDATE*.) The constructs `\old` and `\result` are only allowed in postconditions of Hoare-triples (i.e., in π'). `\result` refers to the return value of a (non-`void`) method. `\old` allows to evaluate sub-expressions not in the post-state (which is the default), but in the prestate of a method's execution. For instance, '`x == \old(x + y)`' in a postcondition of method `m` says that the difference between the values of `x` before and after the execution of `m` is the value which `y` had *before* `m`'s execution.

In order to allow or disallow `\old` and `\result`, in the following, we provide one syntactic category for postconditions, and one for all other conditions.

Definition 5. The syntactic category of postconditions over variables in *Var*, $postcond_{Var}$, is given by Boolean JML expressions over *Var*. (Well-typedness of postconditions is context dependent, assuming that `\result` has the same type as the specified method.) The syntactic category $cond_{Var}$ is given by Boolean JML expressions over *Var* containing neither `\result` nor `\old`.

□

Now we can formally define *ppDATE*. As a *ppDATE* describes properties about a particular system, we assume that every time we make reference to the set of system variables, these variables belong to the system under scrutiny.

Definition 6. Given a set of system variables Sys and a set of *ppDATE* variables V, a *ppDATE* m is a tuple (Q, t, B, q_0, Π) such that:

- Q is the finite set of states.
- t is the transition relation among states in Q, where each transition is tagged with (i) a trigger; (ii) a condition; (iii) an action which may change the valuation of *ppDATE* variables: $t \subseteq Q \times trigger \times cond_{Sys \cup V} \times action \times Q$.
- $B \subseteq Q$ is the set of bad states.
- $q_0 \in Q$ is the initial state.
- Π is a function which tags each state of m with Hoare triples for particular method names in Σ: $\Pi \in Q \longrightarrow \mathcal{P}(cond_{Sys} \times \Sigma \times postcond_{Sys})$. □

We will write $q \xrightarrow{tr|c \mapsto a}_m q'$ to mean that, given a *ppDATE* m whose transition relation is t, $(q, tr, c, a, q') \in t$. The subscript m is omitted if it is clear from the context. In addition, we will use the usual Hoare triple notation $\{\pi\}\, \sigma\, \{\pi'\} \in \Pi(q)$ to denote $(\pi, \sigma, \pi') \in \Pi(q)$.

Example 3. Consider once again, the *ppDATE* shown in Fig. 3.1. It can be formalised as follows: $m = (Q, t, B, q_0, \Pi)$, where,

- $Q = \{q, q', \texttt{bad}\}$,
- $V = \{c\}$,
- $\Sigma = \{\texttt{fileTransfer}, \texttt{login}, \texttt{logout}\}$,
- $B = \{\texttt{bad}\}$,
- $q_0 = q$.

Furthermore, the transition relation t consists of four elements, including: $q' \xrightarrow{\texttt{fileTransfer}^{\downarrow}|c \leq 10 \mapsto c{+}{+}} q'$ and $q' \xrightarrow{\texttt{fileTransfer}^{\downarrow}|c > 10 \mapsto \texttt{skip}} \texttt{bad}$. In addition, relation Π is defined as follows:

$$\begin{array}{ll} \Pi(q) & = \{\ \{\texttt{true}\}\ \texttt{fileTransfer(f)}\ \{\texttt{bytes} == \texttt{\textbackslash old(bytes)}\}\ \} \\ \Pi(q') & = \{\ \{\texttt{true}\}\ \texttt{fileTransfer(f)}\ \{\texttt{bytes} == \texttt{\textbackslash old(bytes)} + \texttt{size(f)}\}, \\ & \quad \{\texttt{write} \in \texttt{rights(f)}\}\ \texttt{rename(f,n)}\ \{\texttt{name(f)} == n\}\ \} \end{array}$$

□

In addition to *ppDATE*s which exist up-front, and 'run' from the beginning of a system's execution, new *ppDATE*s can be created by existing ones. For instance, one may want to create a separate 'observer' for each new user logged into a system. For that, one needs to be able to define parameterised *ppDATE*s, which we call *templates*, and allow *ppDATE*s to create new instantiations of

templates. Given a *ppDATE* m, the creation of a new *ppDATE*, which will run in parallel to m, can be achieved by using action create on a transition of m. This action receives as arguments a *ppDATE* template describing the *ppDATE* to be created and a list of arguments to instantiate the quantified variables on the template. Below, we formally define *ppDATE* templates.

Definition 7. *ppDATE templates of order n* are recursively defined as follows:

- The set of *ppDATE templates of order 0* is exactly the set of *ppDATE*s.
- Assume C is a syntactic sub-category of *ppDATE* (Def. 6), i.e., a syntactic (sub-)category of Q, t, B, q_0, or Π, respectively. If m is a *ppDATE* template of order n, then $\lambda X{:}C.m'$ is a *ppDATE template of order* $n+1$, where m' is the result of replacing, in m, some (sub-)term *trm* of category C by X. We call X the template variable of $\lambda X{:}C.m'$. □

In the above definition, a template of order $n+1$ is defined by 'abstracting' over templates of order n, annotating the abstracted 'hole' X by the right category, such that template instantiation (see below) can be guaranteed to result in a well-typed *ppDATE*. When constructing a *ppDATE* template, the choice of *trm* in Def. 7 does not matter. Its only role is to carry well-typedness of *ppDATE*s over to *ppDATE* templates. Informally, the above definition says that, within $\lambda X{:}C.m'$, the X can appear anywhere in m' where a term of category C is expected.

We will refer to *ppDATE* templates without referring to an order to mean templates that are not of order greater than 0. Formally:

Definition 8. The set of *ppDATE templates* T_{ppd}, is defined as the union of *ppDATE* templates of order $n \geq 1$.

If $\overline{X}$ is a vector of template variables $X_1, \ldots, X_n$ and $\overline{C}$ is a vector of syntactic categories $C_1, \ldots, C_n$, then we can write $\lambda\overline{X}{:}\overline{C}.m$ to mean $\lambda X_1{:}C_1 \ldots \lambda X_n{:}C_n.m$.

Finally, we define what it means to instantiate a *ppDATE* template:

Definition 9. Given a term *trm* of syntactic category C, the *instantiation of a ppDATE template with term trm*, denoted $inst(m, trm)$, is defined by:

$$inst(\lambda X{:}C.m,\ trm) = m[X/trm]$$

where $m[X/trm]$ denotes the result of substituting all occurrences of X in m by *trm*.

We can expand template instantiation to multiple arguments in the following way. Given $n \geq 2$, assume $\overline{X} = X_1, \ldots, X_n$, and $\overline{C} = C_1, \ldots, C_n$, and $\overline{trm} = trm_1, \ldots, trm_n$ (with $trm_i \in C_i$). We extend the instantiation function *inst* to an arbitrary number of arguments in the following way:

$$\begin{aligned}
&inst(\lambda\overline{X}{:}\overline{C}.m,\ \overline{trm})\\
&= \textit{(by syntactic convention)}\\
&inst(\lambda X_1{:}C_1 \ldots \lambda X_n{:}C_n.m,\ trm_1, \ldots, trm_n)\\
&\stackrel{df}{=}\\
&inst(inst(\lambda X_1{:}C_1 \ldots \lambda X_n{:}C_n.m,\ trm_1),\ trm_2, \ldots, trm_n)
\end{aligned}$$

$\mathit{one\text{-}at\text{-}a\text{-}time} = \lambda\ C, S : \mathit{cond}, \mathit{trigger}.$

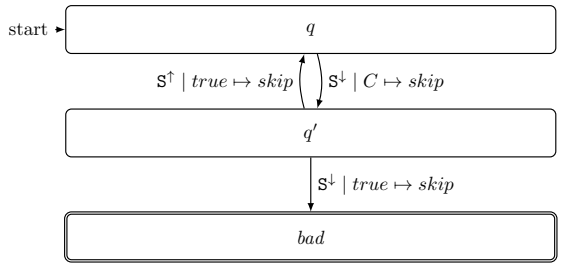


Figure 2.5: *ppDATE* template example.

$\mathit{inst}(\mathit{one\text{-}at\text{-}a\text{-}time}, \mathit{cups} < \mathit{limit}, \texttt{brew}) =$

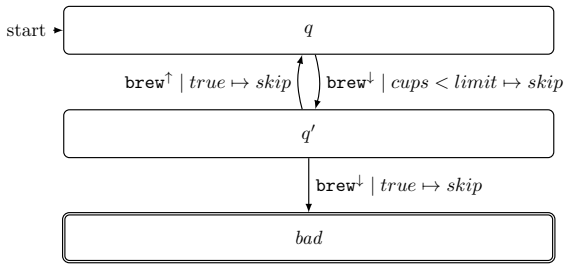


Figure 2.6: *ppDATE* created using the template illustrated in Fig. 2.5.

Example 4. Fig.2.5 illustrates a *ppDATE* template, based on the *ppDATE* depicted in Fig. 2.2. Let us call it *one-at-a-time*. This template has two parameters: C, which represents a condition, and S, which represents a method name. Then, by executing the action $\texttt{create}(\mathit{one\text{-}at\text{-}a\text{-}time}, \mathit{cups} < \mathit{limit}, \texttt{brew}^{\downarrow})$, it would instantiate the *ppDATE* depicted in Fig.2.6, i.e., C is instantiated with $\mathit{cups} < \mathit{limit}$ and S is instantiated with `brew`. This *ppDATE* specifies the property: *it is not possible to brew one more coffee cup until the brewing process is done.*

□

In the rest of this work we will only consider the use of deterministic *ppDATE*s. Formally:

Definition 10. We say that a *ppDATE* m is *deterministic* if, for any two transitions of m with same trigger tr which go from a state q to a different state, their conditions are mutually exclusive:

$$\forall\ tr, c, c', a, a', q, q', q'' \cdot \\ q \xrightarrow{tr|c \mapsto a}_m q' \ \mathit{and}\ q \xrightarrow{tr|c' \mapsto a'}_m q'' \cdot \mathit{not}(c\ \mathit{and}\ c')$$

In addition, although determinism on the Hoare triples' preconditions is not problematic in itself, we choose to extend the determinism condition to ensure that any two Hoare triples in a single state over the same function have

disjoint precondition so as to have a more effective monitoring algorithm of these triples: for any $\{\pi_1\}\,\sigma\,\{\pi_1'\}$ and $\{\pi_2\}\,\sigma\,\{\pi_2'\}$ in $\Pi(q)$, $not(\pi_1\ and\ \pi_2)$. □

After having defined (individual) *ppDATE*s, we can now define a *network* of *ppDATE*s.

Definition 11. Given a set of system variables *Sys*, a *ppDATE network pn* is represented with a tuple (M, V, ν_0, T_{ppd}):

- M is a set of *ppDATE*s. If $m \in M$, then we say that $m = (Q_m, t_m, B_m, q_{0m}, \Pi_m)$.
- V is a set of *ppDATE* variables.
- ν_0 is the initial valuation[5] of variables in V.
- T_{ppd} is a set of *ppDATE* templates. □

Note that on a network, whenever a trigger is activated, several *ppDATE*s can have an enabled transition ready to be fired, i.e., a transition whose trigger is active and whose condition holds. Whenever this happens all these enabled transitions are fired in parallel. Also note that the set of *ppDATE* variables V is global to the network of *ppDATE*s, rather than local to individual *ppDATE*s. Thereby, V is effectively the 'shared memory' of the network.

Finally, we extend the notion of deterministic *ppDATE* to a *ppDATE* network.

Definition 12. A *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$ is *deterministic* whenever every *ppDATE* in M is deterministic and every *ppDATE* which can be created when executing action `create` is deterministic.

□

2.6 *ppDATE* semantics

In this section we present the semantics of a network of *ppDATE*s by introducing *Structural Operational Semantics* (SOS) rules. These rules will show how a global configuration is shifted to a new one by considering events and system variables valuations in a system trace.

Informally, a global configuration (L, ν) (of a *ppDATE* network) consists of a set L of local configurations (one for each *ppDATE* in the set of *ppDATE*s of the network and one for each generated instance of a *ppDATE* template), and a valuation ν of the set of *ppDATE* variables V (associated to the *ppDATE* network). The local configurations store the current state, and record, for each ongoing method execution whose precondition was fulfilled at call time, the postcondition to be checked on exit.

Every time the system under scrutiny generates an event, e.g., by entering or leaving a method, all local configurations in L with enabled transitions will replace their current state value by the state indicated in the fired transition, and execute the action of this transition, all simultaneously. For instance, given a *ppDATE* m whose current state is q, and with a transition t_1 of the form

[5] A valuation is a mapping from variables to values of adequate types.

$q \xrightarrow{tr|c\mapsto a}_m q'$, when a system event triggers tr (and condition c holds), then t_1 is fired, state q is replaced by q' in the appropriate local configuration in L, and a is executed. If the executed actions contain *ppDATE* variables assignments, the valuation ν is updated. In addition, any action event generated by these executions will be stored in a buffer.

Once all the previous enabled transitions are fired, every transition that becomes enabled by the events in the buffer will be fired as well. For instance, let us assume that action a in transition t_1 (only) generates the action event h, i.e., $a = h!$, and that a *ppDATE* m' running in parallel to m is in state q'', and has a transition t_2 of the form $q'' \xrightarrow{h?|true\mapsto a'}_{m'} q'''$. Then, whenever t_1 is fired, execution of $h!$ will add to the buffer an event which will enable t_2, due to the fact that trigger $h?$ is activated by h and its condition (trivially) holds. Therefore, after firing t_1, t_2 will be also fired.

Note that the buffer will be emptied before firing the transitions enabled by the events consumed from the buffer. Therefore, the buffer only contains events generated by the recent action executions, and no events from previous ones. This procedure is repeated until no new action event is generated, i.e., the buffer is empty. In general, the process may not terminate, however if we want to guarantee termination, we can adopt an approach which ensures that there is no transitive mutual communication dependencies over the set of automata as explained in the original semantics of LARVA [45].

The rest of this section goes as follows: Sec. 2.6.1 and Sec. 2.6.2 introduce necessary ground concepts for defining the semantics of *ppDATE*. Sec. 2.6.3 presents denotational semantics for the actions in the transitions of *ppDATE*. Sec. 2.6.4 introduces structural operational semantics for *ppDATE*. Finally, Sec. 2.6.5 introduces the notion of valid, and violating, trace.

2.6.1 Events, Valuations, and Traces

ppDATE networks describe which system behaviours are allowed, and which are not. Here, we consider as behaviour basically a series of system events, where each event also comes with a 'snapshot' of the values of (visible) system variables, taken at the time where the event occurs. Formally, these snapshots are *valuations*, i.e., mappings from variables to values (of adequate types). Apart from the observed system, the *ppDATE* networks themselves may create new events.

An *event* may therefore either be a *system event* (i.e., generated by the system under scrutiny due to entering or leaving a method) or an *action event* (i.e., generated by the execution of an action ! in a *ppDATE* transition). Formally:

Definition 13. Given a set of method names Σ, the syntactic category of events is defined as follows:

$$\xi ::= systemevent \mid \mathsf{actevent}$$

$$systemevent ::= systemtrigger_{\mathbb{N}}$$ □

A *systemevent* consists of a *systemtrigger* which is indexed with a natural number representing the *nth* execution of the method associated to the trigger.

Such an index will be considered an identifier[6] unique to each execution of the method.

We distinguish the set of system variables valuations Θ_{Sys}, with typical element θ, and the set of *ppDATE* variable valuations N, with typical element ν. We represent valuations both as functions and (functional) relations[7], i.e., sets of pairs. This means that the notation $\beta(v) = val$ is equivalent to the notation $(v, val) \in \beta$. The *union of valuations* is therefore a set union such that, for any two valuations β and β', $\beta \cup \beta' = \{(v, val) \mid (v, val) \in \beta \text{ or } (v, val) \in \beta'\}$. In the presentation of examples, we limit the valuations to those variables which matter for the example at hand, for simplicity.

In our semantic rules, we will use union over valuations only when the domain of valuations do not overlap, as for instance in $\theta \cup \nu$. Another operation on valuations is the *modification* of a valuation β at variable x by value val, written $\beta[x \leftarrow val]$. It is defined as:

$$\beta[x \leftarrow val](v) = \begin{cases} val & \text{iff } v = x \\ \beta(v) & \text{otherwise} \end{cases}$$

Given a set of variables S, a valuation β for S, and condition $c \in cond_S$, we will write $\beta \models c$ to denote that c is satisfied by β. This is however not sufficient for postconditions as they can refer to two valuations, after and before ("`\old`") a method's execution. For that, $\models$ will be overloaded. Given a set of system variables *Sys*, valuations θ and θ', and a postcondition $c \in postcond_{Sys}$, we will write $\theta, \theta' \models c$ to denote that c is satisfied by θ and θ'. When this is used, θ' will be the current valuation of *Sys* when exiting a certain method execution, whereas θ holds the valuation from before that method execution. We only sketch the definition of $\models$ here as it follows the standard of first-order logic semantics. We use the two semantic truth values T and F. For $c \in cond_S$, we define $\beta \models c$ *iff* $eval_\beta(c) = T$, where $eval_\beta$ is recursively defined over the structure of c as standard in first-order logic[8], with the base case $eval_\beta(\mathtt{x}) = \beta(\mathtt{x})$ for variables `x`. For $c \in postcond_{Sys}$, we define $\theta, \theta' \models c$ *iff* $eval_{\theta,\theta'}(c) = T$. The definition of $eval_{\theta,\theta'}$ is almost identical to the definition $eval_\beta$, with the base case $eval_{\theta,\theta'}(\mathtt{x}) = \theta'(\mathtt{x})$ for program variables `x`. The only case in the definition where the pre-valuation θ matters is the evaluation of `\old`-expressions: $eval_{\theta,\theta'}(\mathtt{\backslash old(}e\mathtt{)}) = eval_\theta(e)$. This means that, in postconditions, the post-valuation θ' acts as the default, however not inside `\old`-expressions, where instead the pre-valuation θ counts. The other additional operator in postconditions is `\result`. To handle its evaluation properly, we assume a special system variable named `\result`. Whenever a non-void method returns, its return value, say *val*, is assigned to `\result`, such that, in the post-valuation θ', we have $\theta'(\mathtt{\backslash result}) = val$.

A *system trace* is a sequence of tuples consisting of an *event* and a 'system snapshot', i.e., a valuation of the system variables taken at the time when that event occurs.

[6] These identifiers can be created automatically using techniques as those presented in [55] or through stack frame references.

[7] A (binary) relation R is *functional* if $\{(x, y), (x, y')\} \subseteq R$ implies $y = y'$.

[8] To be precise, *eval* has one extra parameter, which is a *logical* variable assignment, needed to define the evaluation of quantified formulas. We omit that parameter since it is unimportant for our discussion here.

Definition 14. A *system trace* w is a sequence of tuples in $systemevent \times \Theta_{Sys}$, i.e. $w \in (systemevent \times \Theta_{Sys})^*$. □

2.6.2 Configurations

Given a system trace w, each tuple in w will shift a *global configuration* of a *ppDATE* network to another. Global configurations are defined with the help of local ones, so we start there.

Definition 15. Given a set of method names Σ, a *local configuration* is a tuple (m, q, ρ) where m is a *ppDATE*, $q \in Q_m$, and $\rho \subseteq \mathcal{P}(systemevent \times postcond_{Sys} \times \Theta_{Sys})$. □

The tuple (m, q, ρ) is a configuration of *ppDATE* m — where q represents the current state, and ρ allows to monitor potential violations of Hoare triples. For that, ρ stores which exit event ($\in$ *systemevent*) should cause a checking of which postcondition ($\in$ *postcond*). The semantic rules described below (Sec. 2.6.4) will guarantee that only method exit events (of the form $\sigma_i^{\uparrow}$) will appear in ρ. During the processing of a trace, the appearance of $(\sigma_i^{\downarrow}, \theta)$ at the same time that the current state has a Hoare-triple with a fulfilled precondition, e.g., $\theta \models \pi$, will lead to associate the corresponding postcondition, e.g., π', with $\sigma_i^{\uparrow}$ in ρ, together with θ. Later, the appearance of $(\sigma_i^{\uparrow}, \theta')$ will cause a look-up of $(\sigma_i^{\uparrow}, \pi', \theta)$ in ρ, in order to check $\theta, \theta' \models \pi'$.

Example 5. Recall the *ppDATE* illustrated in Fig. 2.2, here called m. Its initial local configuration is $(m, q, \emptyset)$. Then, after firing transition t_1 whenever certain system event $\texttt{brew}_{id}^{\downarrow}$ (with $id \in \mathbb{N}$) occurs, assuming that the field *cups* is valuated to zero, the next local configuration is $(m, q', \{(\texttt{brew}_{id}^{\uparrow}, cups == \texttt{\textbackslash old}(cups) + 1, \{(cups, 0)\})\})$. □

Definition 16. Given *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, a *global configuration* for pn is a tuple (L, ν) such that:

- L is a set of local configurations. For each $m \in M$, there is exactly one q and one ρ, such that $(m, q, \rho) \in L$. For each $(m, q, \rho) \in L$, we have $q \in Q_m$ and either $m \in M$ or $m = inst(t, \overline{args})$, for some $t \in T_{ppd}$.
- ν is *ppDATE* variable valuation with domain V. □

Before giving an example, we define the notion of *initial global configuration* for a *ppDATE* network.

Definition 17. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$ where $m \in M$ is the tuple $(Q_m, t_m, B_m, q_{0m}, \Pi_m)$, the *initial global configuration* $C_{init}(pn)$ is defined as the tuple (L_0, ν_0), where $L_0 = \{(m, q_{0m}, \emptyset) \mid m \in M\}$ is the set of initial local configurations. □

Example 6. Let us assume a *ppDATE* network $pn = (\{m, m'\}, \{v\}, \{(v, 0)\}, \emptyset)$, such that $q_{0m'} \xrightarrow{tr|true \mapsto v=v+1}_{m'} q_{1m'}$. The initial global configuration for pn is $C_{init}(pn) = (L_0, \{(v, 0)\})$, where $L_0 = \{(m, q_{0m}, \emptyset), (m', q_{0m'}, \emptyset)\}$. Then, if the given transition is fired, the new global configuration is $(L', \{(v, 1)\})$, where $L' = \{(m, q_{0m}, \emptyset), (m', q_{1m'}, \emptyset)\}$. □

In the above example, the action $v = v + 1$, does not generate any event. In general, however, actions may generate events. For storing action events (and process them in the next step), we introduce the concept of *extended global configuration*.

Definition 18. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, and a set of system variables *Sys*, an *extended global configuration* for pn is a tuple (L, ν, E, θ) such that:

- (L, ν) is a global configuration for pn,
- $E \subseteq \mathcal{P}(\xi)$ is a set of events,
- $\theta \in \Theta_{Sys}$ is a system variables valuation. □

E contains the events to be processed in the next (small) step. In the operational semantics to be described below, E will either be a singleton set containing a system event, or a set of action events generated by the executions of actions in the latest transition.

Example 7. Let us assume a *ppDATE* network $pn = (\{m, m'\}, \{v\}, \{(v, 0)\}, \emptyset)$, such that $q_1 \xrightarrow{\texttt{foo}^{\downarrow}|true \mapsto h!}_m q_2$, $q_1' \xrightarrow{h?|true \mapsto v=v+1}_{m'} q_2'$, $\Pi_m(q_1) = \{\{\pi\}\texttt{foo}\{\pi'\}\}$, with q_1 and q_1' the initial states of m and m', respectively. In addition, let us assume that $C_1 = (L_1, \{(v, 0)\}, \{\texttt{foo}_{id}^{\downarrow}\}, \emptyset)$ is an extended global configuration for pn (for some index $id \in \mathbb{N}$), where $L_1 = \{(m, q_1, \emptyset), (m', q_1', \emptyset)\}$. Then, when the given transition of m is fired, given that π holds and the current system variables valuation is θ, the next extended global configuration for pn is $C_2 = (L_2, \{(v, 0)\}, \{h\}, \emptyset)$, where $L_2 = \{(m, q_2, \{(\texttt{foo}_{id}^{\uparrow}, \pi', \theta)\}), (m', q_1', \emptyset)\}$. After that, event h in C_1 triggers the given transition of m', leading to the extended global configuration $C_3 = (L_3, \{(v, 1)\}, \emptyset, \emptyset)$, where $L_3 = \{(m, q_2, \{(\texttt{foo}_{id}^{\uparrow}, \pi', \theta)\}), (m', q_2', \emptyset)\}$. □

The Structural Operational Semantics given in Sec. 2.6.4 formalises such behaviour.

2.6.3 Semantics of Actions

When assigning meaning to actions, there are two levels to consider. One is the level of the local actions, executed when an individual *ppDATE* takes a transition. The semantics of those is sequential, as defined below. On top of the assignments changing the *ppDATE* variable valuation, the local actions may generate events, and create new instances of *ppDATE* templates.

The other level is parallel actions, where we compose simultaneous actions of transitions taken in parallel by different *ppDATE*s. Here, we need to devote special care to exclude conflicting writes to, as well as race conditions between reads and writes from/to, the same variable. Also, we need to make sure that if only one *ppDATE* writes to x, then the parallel composition propagates this effect. All this makes it necessary to keep track of all reads and writes at the local level, prior to execute the parallel composition. However, the treatment of the local effects and newly created *ppDATE*s is simpler: we just take the union of those when doing the parallel composition.

Definition 19. For each $a \in action$, its *meaning* $[\![a]\!]_{\theta,\nu}$ (relative to system/*ppDATE* variable valuations θ and ν) is given by a tuple (ν', W, R, E, New), where:

- $\nu' \in N$ is a *ppDATE* variable valuation computed (locally) in a,
- $W \subseteq V$ is a set of *ppDATE* variables written to in a,
- $R \subseteq V$ is a set of *ppDATE* variables read from in a,
- $E \subseteq \mathsf{actevent}$ is a set of action events generated in a,
- $New \subseteq ppDATE$ is a set of *ppDATE*s newly created in a.

Given that *pvars* returns the *ppDATE* variables appearing in its argument(s), $[\![a]\!]_{\theta,\nu} = (\nu', W, R, E, New)$ is defined as follows

$$
\begin{array}{rcl}
[\![\mathsf{skip}]\!]_{\theta,\nu} & = & (\nu, \emptyset, \emptyset, \emptyset, \emptyset) \\
[\![v = e]\!]_{\theta,\nu} & = & (\nu[v \leftarrow eval_{\theta\cup\nu}(e)], \{v\}, pvars(e), \emptyset, \emptyset) \\
[\![h!]\!]_{\theta,\nu} & = & (\nu, \emptyset, \emptyset, \{h\}, \emptyset) \\
[\![\mathsf{create}(t, \overline{args})]\!]_{\theta,\nu} & = & (\nu, \emptyset, pvars(\overline{args}), \emptyset, inst(t, \overline{args})) \\
[\![a_1 \ ; \ a_2]\!]_{\theta,\nu} & = & \left\{ \begin{array}{l} (\nu_2, W_1 \cup W_2, R_1 \cup R_2, E_1 \cup E_2, New_1 \cup New_2) \\ \quad \text{where} \\ \quad [\![a_1]\!]_{\theta,\nu} = (\nu_1, W_1, R_1, E_1, New_1) \\ \quad \text{and} \\ \quad [\![a_2]\!]_{\theta,\nu_1} = (\nu_2, W_2, R_2, E_2, New_2) \end{array} \right. \\
[\![\mathsf{if}\ c\ \mathsf{then}\ a]\!]_{\theta,\nu} & = & \left\{ \begin{array}{l} (\nu', W, R \cup pvars(c), E, New) \\ \quad \text{if } \theta \cup \nu \models c \text{ and } [\![a]\!]_{\theta,\nu} = (\nu', W, R, E, New) \\ (\nu, \emptyset, pvars(c), \emptyset, \emptyset) \\ \quad \text{otherwise} \end{array} \right. \\
[\![prog]\!]_{\theta,\nu} & = & [\![\mathsf{skip}]\!]_{\theta,\nu}
\end{array}
$$

□

Following the definition of actions (Def. 2), the *prog* in the last line above is a side-effect free program, i.e., it has no effect which could be noticed in the current formalism, which is why we can simulate it with skip. *prog* will have purposes orthogonal to our formalisation, like logging.

We are now in the position to define the parallel composition of actions. Imagine we have a configuration with 5 parallel *ppDATE*s, 3 of which have enabled transitions, with actions a_1, a_2, and a_3, respectively. Assume moreover that the current *ppDATE* variable valuation is ν. The parallel composition of the meaning of a_1, a_2, and a_3, is performed by $mergeParalActs_\nu(\{[\![a_1]\!], [\![a_2]\!], [\![a_3]\!]\}) = (\nu', E', New')$. The function *mergeParalActs* takes a set of semantic actions as input, and computes a resulting valuation ν', a resulting set of events E', and a resulting set of newly generated *ppDATE*s, New'. The sets E' and New' will simply be the union of the corresponding sets from $[\![a_1]\!]$, $[\![a_2]\!]$, and $[\![a_3]\!]$. But the resulting valuation is slightly more involved. Actions may conflict (e.g., we write to the same variable in different actions), or have race conditions (i.e., we read from a variable and write to it in different actions). In those cases, we leave the result of *mergeParalActs* deliberately *undefined*. In all other cases, the different effects of the actions are merged. The index of the merging function, ν, serves as a fall back for those variables which have not been written to. In particular, $\nu' = \nu$ in case the set of actions to be merged is empty.

These explanations are formalised in the following function, merging a set of action meanings (Def. 19):

Definition 20.

$$mergeParalActs_\nu(\{(\nu_1, W_1, R_1, E_1, New_1), \ldots, (\nu_n, W_n, R_n, E_n, New_n)\})$$
$$= \begin{cases} \text{undefined} \\ \quad \text{if } \exists\, i, j \cdot (i, j \in [1, \ldots, n] \text{ and } i \neq j) \cdot (W_i \cap W_j \neq \emptyset \text{ or } W_i \cap R_j \neq \emptyset) \\ (\nu', E', New') \text{ otherwise, where} \\ \quad E' = \bigcup_{i=1}^{n} E_i, \quad New' = \bigcup_{i=1}^{n} New_i, \quad \nu'(v) = \begin{cases} \nu_i(v) \text{ if } v \in W_i \\ \nu(v) \text{ if } v \notin \bigcup_{i=1}^{n} W_i \end{cases} \end{cases}$$

□

Note that if there are no actions to merge, we have $mergeParalActs_\nu(\emptyset) = (\nu, \emptyset, \emptyset)$.

2.6.4 Structural Operational Semantics

In this section we give structural operational semantics rules (SOS) for *ppDATE*s. These rules will have the following generic form:

$$name \frac{\begin{array}{c} H_1 \\ \cdots \\ H_n \end{array}}{Goal}$$

where *name* is a label used to identify the rule, *Goal* is the property enforced by the rule and the premises $H_1, \cdots, H_n$ are assumptions over the values of the *Goal*.

2.6.4.1 Auxiliary Predicates

In the semantic definitions given below, we use the following predicates.

activatedBy Given a (transition) trigger tr and an event e, predicate $activatedBy(tr, e)$ holds if tr and e match, in the following way:

$$activatedBy(tr, e) \stackrel{df}{=} \begin{cases} \exists\, i \cdot i \in \mathbb{N} \cdot e = tr_i & \text{iff } e \in systemevent \\ tr = e? & \text{iff } e \in \mathsf{actevent} \end{cases}$$

For instance, the trigger $\sigma^{\downarrow}$ is activated by the *systemevent* $\sigma_3^{\downarrow}$, and the trigger $h?$ is activated by $\mathsf{actevent}\ h$ (generated before by the execution of action $h!$).

□

nextState Given a local configuration (m, q, ρ), a state q', an event e, a system variables valuation θ and a *ppDATE* variables valuation ν, predicate *nextState* holds whenever there exists an enabled transition on m going from q to q'. We formally write this as follows,

$$nextState((m, q, \rho), e, \theta, \nu, q') \stackrel{df}{=} \exists\, tr, c, a \cdot q \xrightarrow{tr|c \mapsto a}_m q' \text{ and } activatedBy(tr, e) \text{ and } \theta \cup \nu \models c$$

□

checkOnExit Given a local configuration (m, q, ρ), a system event $\sigma_{id}^{\downarrow}$, a system variables valuation θ, and a postcondition π', predicate *checkOnExit* holds if there exists a condition π such that the Hoare-triple $\{\pi\}\, \sigma\, \{\pi'\}$ is associated to state q, and π holds. We formally write this as follows,

$$checkOnExit((m, q, \rho), \sigma_{id}^{\downarrow}, \theta, \pi') \stackrel{df}{=} \exists\, \pi \cdot \{\pi\}\, \sigma\, \{\pi'\} \in \Pi_m(q) \text{ and } \theta \models \pi$$

□

enabled Given a local configuration l, an event e, a system variables valuation θ, and a *ppDATE* variables valuation ν, predicate *enabled* holds if either l has an enabled transition or it has a Hoare triple associated to q which has to be memorised. Formally,

$$\begin{aligned} &enabled(l, e, \theta, \nu) \stackrel{df}{=} \\ &\quad \exists\, q' \cdot nextState(l, e, \theta, \nu, q') \\ &\quad \text{or} \\ &\quad \exists\, \pi' \cdot checkOnExit(l, e, \theta, \pi') \end{aligned}$$

□

toBeExecuted Given a local configuration (m, q, ρ), an event e, a system variables valuation θ, a *ppDATE* variables valuation ν, and an action a, predicate *toBeExecuted* holds if there exists an enabled transition such that a is its action. Formally,

$$\begin{aligned} &toBeExecuted((m, q, \rho), e, \theta, \nu, a) \stackrel{df}{=} \\ &\quad \exists\, tr, c, q' \cdot activatedBy(tr, e) \text{ and} \\ &\qquad q \xrightarrow{tr|c \mapsto a}_m q' \text{ and } \theta \cup \nu \models c \end{aligned}$$

□

2.6.4.2 Small Steps for Local Configurations

The first step to define SOS rules describing the behaviour of a *ppDATE* network is to introduce rules showing how a local configuration performs a small step.

Given an event e, a system variables valuation θ, and a *ppDATE* variables valuation ν, a *small local configuration step* (or simply *small step local*), written $\xrightarrow{(e,\theta,\nu)}$, takes a local configuration (m, q, ρ) to some other local configuration (m, q', ρ'). This step relation is defined by the rules shown in Fig. 2.7. If e is an entry event of the form $\sigma^{\downarrow}_{id}$, there are three different possibilities: (i) there is an enabled transition in m going from state q to state q', and there is a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ associated to q such that π holds ($entry_1$); (ii) there is an enabled transition in m going from state q to q', but no Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ associated to q such that π holds ($entry_2$); or (iii) there are no enabled transitions in m, but there is a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ associated to q such that π holds ($entry_3$).

In case of ($entry_1$), the next state reached by the enabled transition is q', and ρ gets extended by the tuple $(\sigma^{\uparrow}_{id}, \pi', \theta)$, in order to track the information about the postcondition which has to be checked upon the exit of method σ. Entry event identifiers are assumed to be unique in traces, and thereby, $\sigma^{\uparrow}_{id}$ is unique in ρ. In case of ($entry_2$) and ($entry_3$), only one of these two effects takes place. Then, apart from entry events, whenever e is either an exit event, i.e., it has the form $\sigma^{\uparrow}_{id}$, or an action event, by the rules *exit* and *act*, respectively, $\xrightarrow{(e,\theta,\nu)}$ results in the local configuration (m, q', ρ), where q' is the next state reached by the enabled transition.

2.6.4.3 Small Steps for Extended Global Configurations

Given an extended global configuration $EC = (L, \nu, E, \theta)$, the relation *small step for extended global configurations* (or simply *small step global*), written as $\rightarrowtail$, takes EC to some extended global configuration (L', ν', E', θ) by following rule *iter* (depicted in Fig. 2.8). Note that in the rule's premises we define the set L_{en} of all the local configurations $(m, q, \rho) \in L$ such that m has an enabled transition whose triggers are activated by the events in E. L_{en} is used to define both the set L_{nch} of local configurations in L that will *n*ot *ch*ange, and the set L_{ch} of the local configurations obtained after performing a small step on the local configurations in L_{en}. These two

$$entry_1 \frac{checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \quad nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q',\rho \cup \{(\sigma^{\uparrow}_{id},\pi',\theta)\})}$$

$$entry_2 \frac{\nexists\, \pi' \cdot checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \quad nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q',\rho)}$$

$$entry_3 \frac{checkOnExit((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\pi') \quad \nexists\, q' \cdot nextState((m,q,\rho),\sigma^{\downarrow}_{id},\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(\sigma^{\downarrow}_{id},\theta,\nu)} (m,q,\rho \cup \{(\sigma^{\uparrow}_{id},\pi',\theta)\})}$$

$$exit \frac{nextState((m,q,\rho),\sigma^{\uparrow}_{id},\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(\sigma^{\uparrow}_{id},\theta,\nu)} (m,q',\rho)}$$

$$act \frac{e \in \mathsf{actevent} \quad nextState((m,q,\rho),e,\theta,\nu,q')}{(m,q,\rho) \xhookrightarrow{(e,\theta,\nu)} (m,q',\rho)}$$

Figure 2.7: Small Step Rules for Local Configurations

$$iter \frac{\begin{array}{c} L_{en} = \{l \mid l \in L, enabled(l,e,\theta,\nu), e \in E\} \\ L_{nch} = L \backslash L_{en} \\ L_{ch} = \{l' \mid l \in L_{en}, l \xhookrightarrow{(e,\theta,\nu)} l', e \in E\} \\ Acts = \{a \mid l \in L_{en}, toBeExecuted(l,e,\theta,\nu,a), e \in E\} \\ mergeParalActs_{\nu}(\{[\![a]\!]_{\theta,\nu} | a \in Acts\}) = (\nu', E', New') \\ L_{new} = \{(m, q_{0m}, \emptyset) \mid m \in New'\} \\ L' = L_{ch} \cup L_{nch} \cup L_{new} \end{array}}{(L,\nu,E,\theta) \rightarrowtail (L',\nu',E',\theta)}$$

Figure 2.8: Small Step Rule for Extended Global Configurations

sets are used to define L'. Next, we define the set *Acts* of all the actions which label the 'firing' transitions, and merge the meaning of those actions, which results in the valuation ν' and events E' of the new extended global configuration. We also initialise local configurations L_{new} for the newly created *ppDATE*s from New'. Finally, L' is the union of L_{ch}, L_{nch} and L_{new}.

Note that if *mergeParalActs* is undefined, due to conflicts in parallel variable assignments (see Def. 20), then no global small step is defined, i.e., the execution aborts.

2.6.4.4 Big Steps for Global Configurations

Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, a global configuration (L, ν) such that for all $(m, q, \rho) \in L$, $m \in M$ and $q \in Q_m$, and ν a valuation of the *ppDATE* variables V, a system event e and the system variables valuation θ, the relation *big step rules for global configurations* (or simply *big step global*), written $\xRightarrow{(e,\theta)}$, shifts

$$shift\frac{(L,\nu,\{e\},\theta) \rightarrowtail^* (L',\nu',\emptyset,\theta)}{(L,\nu) \xrightarrow{(e,\theta)} (L',\nu')}$$

Figure 2.9: Big Step Rules for Global Configurations

(L,ν) to some global configuration (L',ν'), written $(L,\nu) \xrightarrow{(e,\theta)} (L',\nu')$, by rule *shift* given in Fig. 2.9. Note that here e and θ are external to the global configuration of the *ppDATE* network: they come from the system acting as input to each step of the global configuration.

This rule means that whenever e occurs while the current system variables valuation is θ, (L,ν) shifts to (L',ν') if the transitive closure of the relation *small step global* ($\rightarrowtail$, Fig. 2.8) takes the extended global configuration $(L,\nu,\{e\},\theta)$ to the extended global configuration $(L',\nu',\emptyset,\theta)$. We need the transitive closure because the execution of actions may generate action events which also have to be consumed, meaning that we iterate using *small step global* until the set obtained by applying rule *iter* is the empty set. After having reached $(L',\nu',\emptyset,\theta)$, the small steps are saturated, because any configuration $(\_,\_,\emptyset,\_)$ is a fixed-point of $\rightarrowtail$.

Lemma 1. For each set of local configurations L, *ppDATE* variable valuation ν, and system variables valuation θ, the extended global configuration $(L,\nu,\emptyset,\theta)$ is a fixed-point of the relation small step global, i.e.,

$$(L,\nu,\emptyset,\theta) \rightarrowtail (L,\nu,\emptyset,\theta)$$

Proof. In rule *iter* (Fig. 2.8), if $E = \emptyset$, then $L_{en} = L_{ch} = Acts = \emptyset$, and $L_{nch} = L$. From the note below Def. 20, we deduce that $(\nu', E', New') = (\nu,\emptyset,\emptyset)$, such that $L_{new} = \emptyset$, and $L' = L_{nch} = L$. Therefore, $(L',\nu',E',\theta) = (L,\nu,\emptyset,\theta)$. □

We can now define the semantics of *ppDATE*s by identifying how a system trace changes the global configuration associated to a network of *ppDATE*s.

Definition 21. We define how a system trace $w \in (systemevent \times \Theta_{Sys})^*$ shifts a *ppDATE* from the global configuration (L,ν) to the global configuration (L',ν'), written $(L,\nu) \overset{w}{\Rightarrow} (L',\nu')$, by induction over w:

$(L,\nu) \overset{\varepsilon}{\Rightarrow} (L',\nu') \overset{df}{=} L = L' \text{ and } \nu = \nu'$;
$(L,\nu) \xrightarrow{w:(e,\theta)} (L',\nu') \overset{df}{=} \exists\, L'',\nu'' \cdot (L,\nu) \overset{w}{\Rightarrow} (L'',\nu'') \text{ and } (L'',\nu'') \xrightarrow{(e,\theta)} (L',\nu')$;

□

For this definition we will overload the operator we previously introduced to represent the relation *big step global*, i.e., $\Rightarrow$ since it is straightforward to distinguish between the two from the context.

2.6.5 Valid Traces and Violating Traces

Before defining *violating system traces*, we have to introduce the notion of *counter-example*.

Definition 22. Given a network of *ppDATE*s $pn = (M,V,\nu_0,T_{ppd})$, a system trace $w \in (systemevent \times \Theta_{Sys})^*$ is called a *counter-example* if $C_{init}(pn) \overset{w}{\Rightarrow} (L,\nu)$, and (i) $\exists\, m,q,\rho \cdot (m,q,\rho) \in L \cdot q \in B_m$; or (ii) $w = w_1 +\!\!+ \langle(\sigma^{\uparrow}_{id},\theta')\rangle$, $C_{init}(pn) \overset{w_1}{\Rightarrow} (L',\nu')$ and $\exists\, m,q,\rho,\pi',\theta \cdot ((m,q,\rho) \in L' \text{ and } (\sigma^{\uparrow}_{id},\pi',\theta) \in \rho) \cdot \theta,\theta' \not\models \pi'$. □

(The symbol $+\!\!+$ represents the concatenation of traces.) This means that a counter-example either (i) ends in a bad state (in one of the local configurations), or (ii) ends with the exiting of a method execution whose postcondition (stored in ρ) is currently violated. Note that (i) and (ii) are not exclusive, so a counter-example may have both properties at once. Also note that violations of *pre*conditions when entering methods is not mentioned here. In our semantics, the violation of preconditions does not as such result in a counter example. It only means that the postcondition of the corresponding Hoare triple does not need to be checked further on (see $entry_2$, Fig. 2.7).

Example 8. Recall the *ppDATE* m shown in Fig. 2.2, and let us assume that it is in state q. Then, for any system variables valuation θ, $w = \langle(\texttt{brew}_1^{\downarrow}, \theta), (\texttt{brew}_2^{\downarrow}, \theta)\rangle$ is a counter-example corresponding to the case (i) of Def. 22.

In addition, if the trigger $\texttt{cleanF}_1^{\downarrow}$ is activated and the postcondition of the Hoare triple `{true} cleanF() {cups == 0}` is violated when method `cleanF` terminates, then $w' = \langle(\texttt{brew}_1^{\downarrow}, \theta), (\texttt{brew}_1^{\uparrow}, \theta), (\texttt{cleanF}_1^{\downarrow}, \theta), (\texttt{cleanF}_1^{\uparrow}, \theta)\rangle$ is a counter-example corresponding to the case (ii) of Def. 22. □

Definition 23. The set of *violating system traces* of a *ppDATE* network pn, written $\mathcal{VT}(pn)$, is defined to be system traces which have a counter-example of pn as a prefix. □

Definition 24. The set of *valid system traces* of a *ppDATE* network pn, written $\mathcal{VAT}(pn)$, is defined to be the system traces which are not violating. □

Example 9. The following system traces, for the coffee machine system of Fig. 2.2, are all valid:

$w = \langle(\texttt{brew}_1^{\downarrow}, \theta), (\texttt{brew}_1^{\uparrow}, \theta), (\texttt{brew}_2^{\downarrow}, \theta), (\texttt{brew}_2^{\uparrow}, \theta)\rangle$
$w' = \langle(\texttt{brew}_5^{\downarrow}, \theta), (\texttt{brew}_5^{\uparrow}, \theta), (\texttt{cleanF}_2^{\downarrow}, \theta), (\texttt{cleanF}_2^{\uparrow}, \theta)\rangle$
$w'' = \langle(\texttt{cleanF}_4^{\downarrow}, \theta), (\texttt{cleanF}_4^{\uparrow}, \theta), (\texttt{brew}_2^{\downarrow}, \theta), (\texttt{brew}_2^{\uparrow}, \theta)\rangle$ □

2.7 From *ppDATE* to *DATE*

In our framework, KeY first tries to prove all Hoare-triples of a *ppDATE* m, and then the partial proofs are used to get an optimised *ppDATE* m'. To make the property m' runtime-checkable, we further translate away the (remaining/optimised) Hoare triples, to arrive at a set of parallel (pure) *DATE*s that can be processed by LARVA.

In this section, we formally define *DATE*s, we present the algorithm used by STARVOORS to translate *ppDATE*s into *DATE*s, finally, after introducing the semantics of *DATE*s, we prove soundness of the translation.

2.7.1 *DATE*

DATE [45] is a formalism similar to *ppDATE*, except that the automata do not include Hoare triples in the states. *DATE*s also include support for timers, which are not in *ppDATE*s. However, since the work we present here does not use timers, we leave them out from the formalisation. Formally:[9]

Definition 25. A *DATE* is a *ppDATE* of the form $(Q, t, B, q_0, \Pi_\emptyset)$, where relation $\Pi_\emptyset$ represents that there are no Hoare triples assigned to any of the states in Q, i.e., $\Pi_\emptyset(q) = \emptyset$, $\forall q \in Q$.

□

[9] Note that the definition of *DATE* given here is different from the one given in [45] as $\Pi_\emptyset$ was not defined in the original formulation. It is easy to see that the formulations are equivalent (modulo the differences mentioned above).

$exit\_cond\_checker = \lambda\ S, A : \Sigma, cond.$

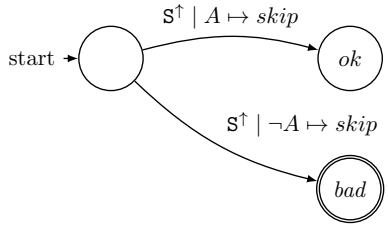


Figure 2.10: *DATE* template for verifying postconditions of Hoare triples.

Note that since a *DATE* is effectively a *ppDATE*, the semantics for *DATE*s are already covered by the semantics of *ppDATE*s. We will also refer to a (deterministic) network of *ppDATE*s where each *ppDATE* in the network is a *DATE*, as a network of *DATE*s and similarly *DATE* templates.

2.7.2 Translation from *ppDATE*s to *DATE*s

Here we present how to translate a *ppDATE* (network) into a *DATE* (network). However, first, let us intuitively analyse how the *ppDATE* depicted in Fig. 2.2, which we will refer to as m, can be translated into a *DATE* m'.

For simplicity, we assign the following names to the different Hoare triples in the states of m.

- h_1: $\{\texttt{cups} < \texttt{limit}\}\,\texttt{brew()}\,\{\texttt{cups == \old(cups)+1}\}$
- h_2: $\{\texttt{true}\}\,\texttt{cleanF()}\,\{\texttt{cups == 0}\}$
- h_3: $\{\texttt{cups} < \texttt{limit}\}\,\texttt{brew()}\,\{\texttt{cups == \old(cups)}\}$
- h_4: $\{\texttt{true}\}\,\texttt{cleanF()}\,\{\texttt{cups == \old(cups)}\}$

Then, we begin the translation by generating the *DATE* template illustrated in Fig. 2.10, which will be used to create *DATE*s in charge of controlling the postconditions of the previous Hoare triples.

Next, we start dealing with the translation of the transitions of m. m' will have exactly the same set of states as m, and it will have similar transitions to the ones of m. The only difference is that the transitions in m' will also have to address the verification of the Hoare triples. For instance, while being in state q, if the method `brew()` is executed and the precondition of h_1 holds, then its postcondition will have to be verified whenever method `brew()` finishes its execution.

Therefore, for every transition of the form $q \xrightarrow{\sigma^{\downarrow} \mid c \mapsto a}_{m} q'$, such that a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ is in q, m' will include a modified version of this transition in such a way that whenever this transition is fired, if π holds, then the execution of its action will have to create an instance of template *exit_cond_checker*. Thus, transitions t_1, t_3 and t_4 (recall Fig. 2.2) are modified as follows:

- $t'_1 : q \xrightarrow{\texttt{brew}^{\downarrow} \mid \texttt{cups} < \texttt{limit} \mapsto \texttt{skip}\ ;\ a_1}_{m'} q'$
- $t'_3 : q' \xrightarrow{\texttt{brew}^{\downarrow} \mid \texttt{true} \mapsto \texttt{skip}\ ;\ a_2}_{m'} bad$
- $t'_4 : q' \xrightarrow{\texttt{cleanF}^{\downarrow} \mid \texttt{true} \mapsto \texttt{skip}\ ;\ a_3}_{m'} bad$

where,

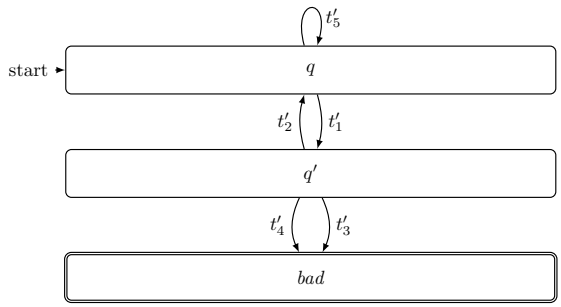


Figure 2.11: Translation to *DATE* of the *ppDATE* illustrated in Fig. 2.2.

- a_1: if (`cups` < `limit`)
  then *create(exit_cond_checker,*`brew`*,*`part_eval(cups==\old(cups)+1)`*)*
- a_2: if (`cups` < `limit`)
  then *create(exit_cond_checker,*`brew`*,*`part_eval(cups==\old(cups))`*)*;
- a_3: if (`true`)
  then *create(exit_cond_checker,*`cleanF`*,*`part_eval(cups == \old(cups))`*)*.

In the previous transitions we have used as the conditions of the if-expressions in actions a_1, a_2 and a_3, the preconditions of the different Hoare triples to be verified in each case. Moreover, function `part_eval` partially evaluates its argument, replacing the expressions of the form `\old`(e) by the current value of e. If a postcondition does not include such operator, then `part_eval` is the identity. Note that even though the if-expression in the transitions t'_1 and t'_4 may seem unnecessary, we include it anyway in order to exactly reflect how the translation algorithm works.

In addition, if at a certain state, a Hoare triple has to be verified, but in that state there are no outgoing transitions with an event related to the method in the Hoare triple, then a new transition is added to m' in order to be able to control such Hoare triple. For instance, in state q the following *self*-transition has to be added in order to verify h_2 and h_3.

- t'_5: $q \xrightarrow{\texttt{cleanF}^{\downarrow}|\texttt{true}\mapsto a_4}_{m'} q$

where,

- a_4: *create(exit_cond_checker,*`cleanF`*,*`part_eval(cups == 0)`*)*

Again, we use the preconditions of the Hoare triples as conditions of the previous action.

Given a transition $q \xrightarrow{tr|c\mapsto a}_m q'$ such that (i) tr fires upon exiting a method, or (ii) tr fires upon entering a method but there is no Hoare triple associated to this method in q, these transitions remain untouched, i.e., it is translated as $q \xrightarrow{tr|c\mapsto a}_{m'} q'$. For instance, transition t_2 is translated as follows.

- t'_2 : $q' \xrightarrow{\texttt{brew}^{\uparrow}|\texttt{true}\mapsto skip}_{m'} q$

Fig 2.11 illustrates the *DATE* obtained when translating m following the previous steps. Note that whole translation would consist on the previous *DATE* and the generated template *exit_cond_checker*.

2.7.2.1 Translation Algorithm

For clarity of presentation we give two algorithms, one for the case when no Hoare triples clashes arise, and one for the full case. Intuitively, we call it a clash if the behaviour of a method σ, in a certain *ppDATE* state q, is defined by both, a Hoare triple in q, and an outgoing transition from q. Formally, we define a clashing Hoare triple as follows.

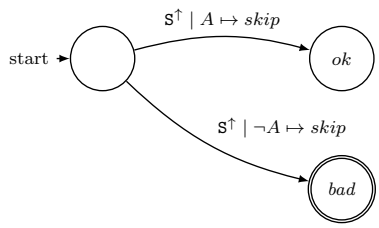
Definition 26. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$ such that every *ppDATE* $m \in M$ is defined as the tuple $(Q_m, t_m, B_m, q_{0m}, \Pi_m)$, a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\} \in \Pi_m(q)$, for some $q \in Q_m$, is called *clashing* if an outgoing transition from q is guarded by trigger $\sigma^{\downarrow}$ (i.e., $\exists\, c, a, q' \cdot q \xrightarrow{\sigma^{\downarrow}|c\mapsto a}_m q'$). A *clash-free ppDATE* is a *ppDATE* with no clashing Hoare triples. □

We now present the algorithm to translate a clash-free *ppDATE* network into a *DATE* network. The translation works by replacing each Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ in a state q of a *ppDATE* by a new reflexive transition (from q to q) triggered by an entry into function σ such that the precondition π holds, and a parallel *DATE* is created, checking the postcondition.

We assume a function `part_eval` $\in$ *postcond* $\mapsto$ *cond*, which removes `\old` constructs in postconditions. The function performs *partial evaluation* — replacing each `\old`(e) with the current value of e. Our algorithm syntactically places the `part_eval` function in an action that will be executed when the according method is entered, i.e., partial evaluation does not happen during the translation algorithm, but at runtime, when the method is entered.

Algorithm 1. Given a *clash-free ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, such that every *ppDATE* $m \in M$ is defined as the tuple $(Q_m, t_m, B_m, q_{0m}, \Pi_m)$, we can construct a *DATE* network equivalent to pn in the following manner:

1. With each Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ in a *ppDATE* state, replace in π' each instance of the `\result` by the variable `ret`. This variable will represent the value returned by the method associated to the Hoare triple.
2. Generate the following *DATE* template:
   $exit\_cond\_checker = \lambda\, S, A : \Sigma, cond.$

   This template will be used to create *DATE*s handling the verification of the postcondition of the method.
3. Transform M, the set of *ppDATE*s of pn, into the set of *DATE*s $M' = \{m' \mid m' = (Q_m, t'_m, B_m, q_{0m}, \Pi_\emptyset), m \in M\}$ such that t'_m follows the rules below:
   3a. each Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ in $\Pi_m(q)$ is replaced by $q \xrightarrow{\sigma^{\downarrow}|\pi\mapsto a}_{m'} q$, where $a = \mathsf{create}(exit\_cond\_checker, \sigma, part\_eval(\pi'))$;
   3b. each transition $q \xrightarrow{tr|c\mapsto a}_m q'$ remains unchanged, i.e. $q \xrightarrow{tr|c\mapsto a}_{m'} q'$
4. Translate T_{ppd} (the set of *ppDATE* templates in pn) into a set of *DATE* templates T_d by repeatedly applying steps 3a and 3b to the body of the templates.
5. Extend the set T_d by including the template generated in step 2. Let us call this extension T'_d.

6. Finally, the resulting *DATE* network is defined to be (M', V, ν_0, T'_d).

This translation works well except that it would introduce non-determinism when the *ppDATE* includes clashes. To extend the translation to work in the presence of clashes, we transform Hoare triples clashing with a transition into a family of disjoint transitions, each of which performs the transition but also checks whether the postcondition checker should be created.

Algorithm 2. Given a (possibly clashing) *ppDATE* network *pn*, we construct a network of *DATE*s equivalent to *pn* by using Algorithm 1 except that we replace steps 3a and 3b, by the following:

$3a_1$. Each non-clashing Hoare triple: $\{\pi\}\,\sigma\,\{\pi'\}$ in $\Pi_m(q)$ is turned into a transition $q \xrightarrow{\sigma^{\downarrow}|\pi \mapsto \mathsf{create}(exit\_cond\_checker, \sigma, part\_eval(\pi'))}_{m'} q$

$3a_2$. For each clashing Hoare triple: $\{\pi\}\,\sigma\,\{\pi'\} \in \Pi(q)$, clashing with n outgoing transitions, $q \xrightarrow{\sigma^{\downarrow}|c_k \mapsto a_k} q^k$ $(0 \leq k < n)$:

- Replace $q \xrightarrow{\sigma^{\downarrow}|c_k \mapsto a_k}_m q^k$ with: $q \xrightarrow{\sigma^{\downarrow}|c_k \mapsto (a_k;\, \mathtt{if}\ \pi\ \mathtt{then}\ a)}_{m'} q^k$;
- Add the following transition: $q \xrightarrow{\sigma^{\downarrow}|(!c_0 \&\& \ldots \&\& !c_n \&\& \pi) \mapsto a}_{m'} q$,

where, in both cases, $a = \mathsf{create}(exit\_cond\_checker, \sigma, part\_eval(\pi'))$

3b. each transition $q \xrightarrow{tr|c \mapsto a}_m q'$ such that either $\Pi_m(q) = \emptyset$, $\Pi_m(q) \neq \emptyset$ but there is no Hoare triple associated to trigger tr, or trigger tr is activated by an exit event, remains unchanged, i.e. $q \xrightarrow{tr|c \mapsto a}_{m'} q'$.

2.7.3 Proof of Soundness of the translation algorithm

In this section we will show that the translation algorithms introduced in the previous section are sound.

2.7.4 Coupling Invariant Lemmas

Here, we formally introduce two lemmas which together form the coupling invariant that is used to prove soundness. The proofs of these lemmas can be found in Appendix A.

Lemma 2 states that given a trace, both a *ppDATE* network *pn* and its translation to *DATE* will change their initial global configuration to global configurations (L, ν) and $(\tilde{L}, \nu')$, respectively, such that $\nu = \nu'$, and that for every $(m, q, \rho) \in L$ where m is in *pn*, there is a local configuration $(m', q', \emptyset) \in \tilde{L}$ such that m' is the translation of m and both m and m' are in the same state, and vice versa.

In this lemma we represent the translation of a single *ppDATE* to *DATE* with the function $\kappa \in ppDATE \mapsto DATE$.

Lemma 2. Given a network of *ppDATE*s $pn = (M, V, \nu_0, T_{ppd})$, its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$, a trace $w \in (systemevent \times \Theta_{Sys})^*$, and the global

configurations (L, ν) and $(\tilde{L}, \nu')$,

$$
\begin{array}{l}
C_{init}(pn) \stackrel{w}{\Rightarrow}_M (L, \nu) \ and \ C_{init}(ppd2DATE(pn)) \stackrel{w}{\Rightarrow}_{M'} (\tilde{L}, \nu') \\
implies \\
\quad \nu = \nu' \\
\quad and \\
\quad\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\quad\quad\quad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L} \ \cdot \kappa(m) = m' \ and \ q = q' \\
\quad and \\
\quad\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\quad\quad\quad \exists\, m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad and \\
\quad\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\quad\quad\quad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad and \\
\quad\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot \\
\quad\quad\quad \exists\, m, q, \rho \cdot (m, q, \rho), m \notin M \in L \cdot q = q'
\end{array}
$$

Lemma 3 states that given a trace, if this trace shifts a *ppDATE* network pn and its *DATE* translation from their respective initial global configuration to some global configurations (L, ν) and $(\tilde{L}, \nu')$, respectively, then for each entry $(\sigma^{\uparrow}_{id}, \pi', \theta)$ in a ρ component of a local configuration in L there is one local configuration in $\tilde{L}$ whose *DATE* component is an instance of the template *exit_cond_checker* in charge of controlling π', and vice versa.

Lemma 3. Given a network of *ppDATE*s $pn = (M, V, \nu_0, T_{ppd})$, its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$, a trace $w \in (systemevent \times \Theta_{Sys})^*$, and the global configurations (L, ν) and $(\tilde{L}, \nu')$,

$$C_{init}(pn) \stackrel{w}{\Rightarrow}_M (L, \nu) \ and \ C_{init}(ppd2DATE(pn)) \stackrel{w}{\Rightarrow}_{M'} (\tilde{L}, \nu') \ implies \ \psi(L, \tilde{L})$$

where,

$$
\begin{array}{l}
\psi(L, \tilde{L}) = \forall\, m, q, \rho \cdot (m, q, \rho) \in L \cdot \\
\qquad \forall\, \sigma^{\uparrow}_{id}, \pi', \theta \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho \cdot \\
\qquad\quad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot m' = inst\,(exit\_cond\_checker, \sigma, \pi') \\
\quad and \\
\qquad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot \\
\qquad\quad \exists\, \sigma^{\uparrow}_{id}, \pi' \cdot m' = inst\,(exit\_cond\_checker, \sigma, \pi') \\
\qquad\qquad\qquad implies \exists\, m, q, \rho, \theta \cdot (m, q, \rho) \in L \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho
\end{array}
$$

□

2.7.4.1 Proof of Soundness

We can now prove the soundness of the translation algorithm. Below we provide the formalisation of this property and an intuitive explanation for it. However, a rigorous proof of this theorem can be found in Appendix B.

Theorem 1. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, and its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$,

$$\mathcal{VT}(pn) = \mathcal{VT}(ppd2DATE(pn))$$

Proof. To prove the soundness of the translation algorithm we will show that both a *ppDATE* network pn and its translation to a *DATE* network have the same set of violating traces. Intuitively, we will prove that given a trace w which is violating for pn, i.e., $w \in \mathcal{VT}(pn)$, is also violating for pn's translation, i.e., $w \in \mathcal{VT}(ppd2DATE(pn))$, and vice versa.

In the case when $w \in \mathcal{VT}(pn)$, by definition of counter-examples of *ppDATE*s, w has a prefix w' such that either (i) w' takes the initial global configuration $C_{init}(pn)$ to a global configuration (L', ν') such that the state component of L' is a bad state; (ii) given a method σ and a system variables valuation θ', w' can be written as $w_1 +\!\!+ (\sigma^{\uparrow}_{id}, \theta')$ such that w_1 takes $C_{init}(pn)$ to a global configuration (L', ν') where there exists a local configuration in L' whose ρ component contains a tuple $(\sigma^{\uparrow}_{id}, \pi', \theta)$, such that π' fails to be satisfied in the 'moment' event $\sigma^{\uparrow}_{id}$ appears.

In the case of (i), we use the fact that (by Lemma 2), if w' takes the translation from the initial global configuration $C_{init}(ppd2DATE(pn))$ to a global configuration $(\tilde{L}, \nu)$, for every local configuration in L', there is a local configuration in $\tilde{L}$ such that its state component is the same. Thus, there is a local configuration in $\tilde{L}$ whose state component is a bad state, which means that w' is a counter-example of the translation as well.

In the case of (ii), due to the fact that a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ has to be verified, we know that some local configuration will have a ρ component such that $(\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho$. We can now use the fact that by Lemma 3, tuple is handled by a *DATE* in the translation (which verifies the postcondition). Thus, there exists a *DATE* controlling π' which fails moving to a bad state, i.e., w' is a counter-example of the translation as well.

In order to prove the opposite direction, we assume $w \in \mathcal{VT}(ppd2DATE(pn))$. Again, since this is a counter-example and this is a *DATE* (and thus cannot fail due to a violated postcondition), it can be only the case that w has a prefix w' such that this prefix takes the initial global configuration $C_{init}(ppd2DATE(pn))$ to a global configuration $(\tilde{L}, \nu)$ such that there is a local configuration in $\tilde{L}$ whose state component is a bad state. Then, assuming that w' takes pn from the initial global configuration $C_{init}(pn)$ to a global configuration (L', ν'), we proceed to do a case analyses depending whether the bad state belongs to a *DATE* which was controlling the postcondition of a Hoare triple or not. In the affirmative case, we will use this fact to show that, given certain method σ and a system variables valuation θ', w' can be selected to be a prefix which can be written as $w_1 +\!\!+ (\sigma^{\uparrow}_{id}, \theta')$ such that w_1 takes $C_{init}(pn)$ to a global configuration (L', ν') where the verification of the postcondition fails whenever event $\sigma^{\uparrow}_{id}$ occurs. Therefore, w' is a counter-example of pn. Finally, (by Lemma 2), there is a local configuration in L' such that its state component is the same as the bad state in $\tilde{L}$. Therefore, w' is a counter-example of pn. □

2.8 The StaRVOOrS Tool Implementation

In this section we present how the (fully automatic) verification tool StaRVOOrS [37] implements the framework presented in Sec. 2.4. To illustrate this, we use a running example of a *bank system* in which users log in to perform transactions[10]. The set of logged-in users is implemented as a `Hashtable` object, whose class represents an open addressing hashtable with linear probing as collision resolution. Method `add`, which is used to add objects into the hashtable, first attempts to put the corresponding object at the position of its computed hash code. However, if that index is occupied, then `add` searches for the nearest following index which is free. Fig. 2.12 depicts a

[10]Both the source code and the *ppDATE* specification for this example are available from [4].

```
1  public void add (Object o, int key) {
2      if (size < capacity) {
3         int i = hash_function(key);
4         if (h[i] == null) {
5            h[i] = o;
6            size++;
7            return;
8         }
9         else {
10           while (h[i] != null) {
11                 if (i == capacity-1) i = 0;
12                 else {i++;}
13           }
14           h[i] = o;
15           size++;
16           return;
17        }
18     }
19 }
```

Figure 2.12: Code snippet for method `add`.

code snippet for this method. Within the hashtable object, users are stored into an array `arr`. This means that the set of logged-in users has its capacity limited by the length of `arr`. In order to check in a straightforward manner whether the capacity of `arr` is reached or not, a field `size` keeps track of the amount of stored objects and a field `capacity` represents the (total) number of objects that can be added into the hash table. In addition, this system has to fulfil the properties described with the *ppDATE* template depicted in Fig. 2.13. This template specifies the following properties:

(i) *A user has to be logged-in in order to perform a deposit, i.e. a deposit should happen between a login and a logout.*

(ii) *Provided there is space in the hashtable, executing method* `add` *with object* `o` *and key* `k` *should add the object to the table.*

Property (i) is verified with the transitions of the *ppDATE* template, whereas property (ii) is represented by the Hoare triple in state q_1. If `size < capacity`, then there is room in the hashtable for one more element, and if method `add` places the object `o` in the hashtable, there exists an index in the array `arr` such that `o` is placed in that index, i.e., $\exists$ `int i; i>= 0 && i < capacity; arr[i] == o`. Note that the given Hoare triple is only included in state q_1 since only a successful login leads to the execution of the method `add`, i.e., this Hoare triple is context dependent; and that `login(f)`$^{\downarrow}$ means that method `login` associated to the trigger is the one defined within object f. In addition, we assume that the specification of the system has a *ppDATE* with a single state q and single transition of the form $q \xrightarrow{\texttt{new(o)}^{\downarrow}|\texttt{true}\mapsto create(prop-deposit-temp,\texttt{o})} q$, such that the trigger `new(o)`$^{\downarrow}$ is activated by the declaration of an object o of the class `UserInterface`. Thus, this *ppDATE* creates an instance of the template in Fig. 2.13 every time an object of the class `UserInterface` is declared.

prop-deposit-temp = λ *f* : `UserInterface`.

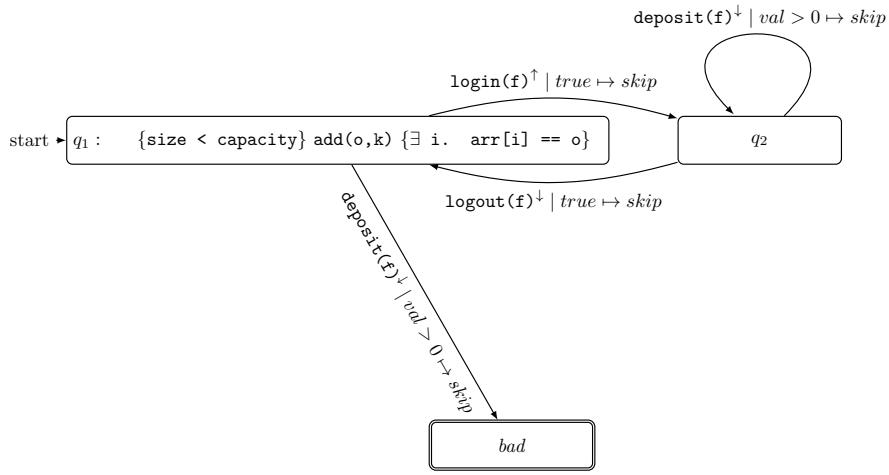


Figure 2.13: *ppDATE* specification of properties for a bank system.

2.8.1 *ppDATE* Specification as an Input Script for StaRVOOrS

Before describing how StaRVOOrS works, we need to introduce how a *ppDATE* specification is written as an input script for this tool. Below, we show the input script for the *ppDATE* template illustrated in Fig. 2.13, and the *ppDATE* which creates its instances. In addition, we give a brief description of each one of the sections this script. For a full description on how to write *ppDATE*s as an input script for our tool, one may refer to the StaRVOOrS *User Manual*[11].

```
IMPORTS { main.UserInterface ; main.Hashtable ; }

GLOBAL {
  PROPERTY prop-deposit {
      PINIT { (prop-deposit-temp, UserInterface) }
  }
}
TEMPLATES {
 TEMPLATE prop-deposit-temp (UserInterface uf) {
   TRIGGERS {
     login_exit(String un, int pwd)
        = {UserInterface f.login(un, pwd)exit()} where {uf = f}
     logout_entry()
        = {UserInterface f.logout()entry} where {uf = f}
     deposit_entry(int val)
        = {UserInterface f.deposit(val)entry} where {uf = f}
   }
   PROPERTY prop_deposit {
     STATES {
       ACCEPTING { q2 ; }
       BAD { bad ; }
       STARTING { q1 (add_ok) ; }
     }
```

[11] This document is available from [4], in the Downloads section.

```
    TRANSITIONS {
      q1 -> q2 [login_exit \ f.getUser() != null]
      q1 -> bad [deposit_entry]
      q2 -> q1 [logout_entry \ f.getUser() != null ]
      q2 -> q2 [deposit_entry \ f.getUser() != null]
    }
  }
 }
}
CINVARIANTS {
  HashTable {\typeof(h) == \type(Object[])}
  HashTable {arr.length == capacity}
  HashTable {arr != null}
  HashTable {size >= 0 && size <= capacity}
  HashTable {capacity >= 1}
}
HTRIPLES {
  HT add_ok {
    PRE {size < capacity}
    METHOD {Hashtable.add}
    POST  {(\exists int i; i>= 0 && i < capacity; arr[i] == o)}
    ASSIGNABLE {size, arr[*]}
  }
}
```

The section `IMPORTS` lists the Java packages which may be used in any of the other sections of the script, in this case `UserInterface` and `Hashtable`. The section `TEMPLATES` contains the description of the *ppDATE* templates (tagged by `TEMPLATE`). Here, the section `TRIGGERS` is used to declare the different triggers which may be used in the transitions of the *ppDATE*, i.e, `login_exit`, `logout_entry`, `deposit_entry`, and the section `PROPERTY` describes the different states, i.e., `q1`, `q2` and `bad`, and transitions of the *ppDATE*. Note that the syntax `q1 (add_ok)` associates the Hoare triple tagged as `add_ok` to state `q1`. This means that the Hoare triple `add_ok` has to be verified if the method associated to it, in this case method `add`, is executed whenever the *ppDATE* is in state `q1`. The section `GLOBAL` contains the description of the *ppDATE*. Here, *ppDATE*s are described in the same manner as in a `TEMPLATE` section. However, note that it is also possible, as it is the case in our example, to use the special section `PINIT` when describing the section `PROPERTY`. Section `PINIT` represents a *ppDATE* with single state, and a looping transition which is fired every time an object of the class listed within this section (`UserInterface` in our example) is declared, leading to the creation of an instance of the listed template for that object (*prop-deposit-temp* in our example). We have included this special case because it is quite common to have *ppDATE*s only focus on creating instances of a template upon declaration of a particular object. Regarding the section `CINVARIANTS`, class invariants are described by the syntax `class_name {invariant}`, meaning that `invariant` has to be fulfilled by all the methods in the class `class_name`. These invariants are only meant as a help for the deductive verification of the Hoare triples (see Sec. 2.8.2). If no invariants are needed, then this section can be omitted. Finally, the section `HTRIPLES` gives a list of named Hoare triples (tagged by `HT`). Here, `PRE` describes the precondition of the Hoare triple, `POST` describes the postcondition of the Hoare triple, `METHOD` indicates which one is the method associated to the Hoare triple, and `ASSIGNABLE` lists the (class) variables that might be modified when the method associated to the Hoare triple is executed. Note that the predicates in invariants, pre- and postconditions follows JML-like syntax and pragmatics. For instance, in the Hoare triple `add_ok` the second semicolon separates the range predicate (`i>=0 && i<capacity`) from the desired property over integers in that range, (`arr[i]==o`).

2.8.2 Running StaRVOOrS

StaRVOOrS is a fully automatic verification tool which takes the Java source code of the system under scrutiny and a file with the *ppDATE* specification for this system and produces (i) a runtime monitor, (ii) an instrumented version of the system given as input with the required event generation and additional code infrastructure, (iii) a report summarising the results of the deductive verification of the Hoare triples, and (iv) a refined version (if any) of the provided *ppDATE* specification.

This tool implements the framework described in Sec. 2.4 with each stage of the framework, i.e., *Deductive Verification*, *Specification Refinement*, *Translation and Instrumentation*, and *Monitor Generation*, being performed automatically by the tool. Below, we describe the implementation of these stages through the use of the working example.

2.8.2.1 Deductive Verification

The first step performed by StaRVOOrS is the deductive verification of the Hoare triples associated to the states of the *ppDATE* (template) using KeY. To accomplish this, StaRVOOrS extracts the Hoare triples specified in the *ppDATE* script, converts them into JML contracts, and then annotates these contracts in the Java sources, before the corresponding method declaration. For instance, the following JML contract associated to method `add` is extracted from the Hoare triple `add_ok`:

```
requires size < capacity;
ensures (\exists int i; i>= 0 && i < capacity ; arr[i] == o);
assignable size, arr[*];
```

Note that the `requires` clause describes the precondition of `add`, the `ensures` clause describes the postcondition of `add`, and the `assignable` clause lists the (class) variables that might be modified when `add` is executed.

Once all the JML contracts are in place, i.e., they are annotated in the code, StaRVOOrS uses KeY to verify them. First, KeY generates proof obligations in Java Dynamic Logic for each JML contract. Next, it attempts to prove the contracts automatically. Finally, it stores the results of all the verification attempts in a XML file. Here, note that even though it could be possible to allow for user interaction (using KeY's elaborate support for interactive theorem proving), we chose to use KeY in automatic mode, since StaRVOOrS targets users untrained in theorem proving. StaRVOOrS generates a report summarising the results produced by KeY in an easy to understand format.

Using our running example, when KeY tries to verify the previous JML contract, it will result in a partial proof. This analysis is shown in the following fragment of the generated XML file:

```
<executionPath
  pathCondition="arr[hash_function(key)] = null"
  verified="true"/>
<executionPath
  pathCondition="!arr[hash_function(key)] = null"
  verified="false"/>
```

This indicates that while KeY was symbolically executing method `add`, there was a branching in the condition `arr[hash_function(key)] = null`, leading to two possible execution paths (depending on its truth value). Recalling the code snippet in Fig. 2.12, this condition corresponds to the condition on the if-expression in line 4. Thus, the execution path for the condition `arr[hash_function(key)] = null` corresponds to the case where the array `arr` has a free slot at the hash code of `key`, whereas the

execution path for the condition `!arr[hash_function(key)] = null` corresponds to the case where the program enters the while-loop in line 10, searching for the next free slot in `arr`. In addition, in the XML, the component `verified` represents whether KeY was able to prove the branch of the proof (`verified=true`), or not (`verified=false`). Therefore, from the previous fragment of the XML file we know that KeY was able to close the branch where the array `arr` has a free slot (`= null`) at the hash code of `key`, but it was not able to verify the other case (where the program enters a loop searching for the next free slot). The main reason why KeY was not able to prove the latter case is the lack of loop invariants to deal with the while-loop.

2.8.2.2 Specification Refinement

The output of KeY is then used to refine the Hoare triples in the specification based on what was (partially) proved. The Hoare triples associated to JML contracts which were fully verified by KeY are entirely removed from the specification, while the precondition of the Hoare triples associated to partially proved JML contracts are refined based on what KeY managed to prove. The new precondition is the conjunction of the original precondition with the disjunction of new preconditions corresponding to open proof goals, i.e., the path condition on each different execution paths. Note that StaRVOOrS generates a new *ppDATE* specification script based on such refinements, instead of modifying the provided *ppDATE* script.

In the example, the precondition of the Hoare triple `add_ok` will be refined with the condition for the one goal not closed by KeY, i.e., `!(arr[hash_function(key)] == null)`. The Hoare triple will thus be strengthened as follows:

```
HT add_ok {
  PRE {size < capacity && !(h[hash_function(key)] == null)}
  METHOD {Hashtable.add}
  POST {(\exists int i; i>=0 && i<capacity; arr[i]==o)}
  ASSIGNABLE {size, arr[*]}
}
```

2.8.2.3 Translation and Instrumentation

Once the refined *ppDATE* specification is ready, StaRVOOrS translates it into (pure) *DATE* formalism using the algorithm from Sec.2.7.2. This enables the monitor generation by Larva (explained in the next stage). In addition, in order to properly address the refined *ppDATE*, our tool operationalises the conditions and instruments the code, as described below.

Pre/Postcondition Operationalisation

In this step, the tool syntactically analyses the specification for expressions in pre- and postconditions of the Hoare triples which may have to be operationalised, i.e., transformed into algorithmic procedures. For instance, transforming either existential or universal quantifications into loops.

During the operationalisation process, the tool creates Java code containing the implementation of all necessary methods for runtime verification, including those generated to algorithmically check the pre/postconditions.

In our example, as the postcondition of the Hoare triple `add_ok` has an existential quantifier, it has to be operationalised, producing the following method:

```
1 public static boolean add_ok_post_opE_1(Hashtable hasht,
2       Object o, int key) {
```

```
3   boolean r = false;
4   for (int i = 0 ; i < hasht.capacity ; i++) {
5     if (hasht.arr[i] == o) { r = true ; break; }
6   }
7   return r;
8 }
```

The for-loop declaration in line 3 is created from the conditions in the range of the existential quantification, i.e., `i>=0 && i<capacity`, and the condition of the if-expression in line 4 is created from the condition in the body of the existential quantification, i.e., `arr[i]==o`. Thus, if any value on the range of the existential quantification fulfils its body, then this method returns `true`, i.e., exists a value that fulfils the existential quantification. Otherwise, it returns `false`, i.e., it does not exist a value fulfilling the existential quantification.

Code Instrumentation

Next, StaRVOOrS instruments the Java source code of the system adding identifiers to each method associated to a Hoare triple in the refined *ppDATE* specification script, and additional code to get fresh identifiers. As mentioned in Sec. 2.4, these identifiers will be used to distinguish different executions of the same method. However, in order to avoid modifying all the calls to these methods in the entire system, we have opted to introduce this instrumentation in the form of auxiliary methods. For instance, in our working example the method `add` has to be instrumented, resulting in:

```
public void add (Object o, int key) {
   addAux(o,key,fid.getNewId());
}
public void addAux (Object o, int key, Integer id) {...}
```

The method `addAux` implementation corresponds to the body of method `add` in Fig. 2.12. This method represents the instrumentation of method `add` with the extra argument `Integer id`, which is used as identifier. In addition, method `add` now simply calls `addAux`, but generating a fresh identifier for the call using function `fid.getNewId`.

Moreover, the previously generated *DATE* specification is modified accordingly, to refer to the instrumented version of the methods. In our example, the *DATE* specification would be modified to refer to method `addAux` instead of method `add`.

2.8.2.4 Monitor Generation

Finally, StaRVOOrS uses Larva to automatically generate a monitor from the *DATE* specification obtained in the previous stage. Larva takes this *DATE* and generates the monitoring system and aspects instrumenting the communication between the system and the monitor [46].

2.9 Case Study: SoftSlate Commerce

SoftSlate Commerce (or simply SoftSlate) [3] is an open-source Java shopping cart web application designed following a *Model-View-Controller* architecture. A user of SoftSlate sends a request to a server hosting the application via a web browser. Then, the server processes the received request and executes an action associated to it (*Controller layer*). Such action may require to interact with and/or modify the information in the database (*Model layer*), e.g., information about users, products, orders, etc. Finally, once the request is fully processed, the server sends back a response to the user. The information in this response will be reflected on a web page

loaded on the browser (*View layer*). The administrator of the application interacts with it in a similar fashion.

SoftSlate offers a basic implementation of a shopping cart web application featuring outer space related pictures, whose server is set up by using *Apache Tomcat* [1]. This implementation is meant to be used by developers to start building their own web applications.

In this case study we analyse an extension of the SoftSlate basic implementation. This extension increases modularity of parts of the implementation, to better link it to the required properties. Basically, we have created a few helper methods in order to better observe the various steps performed by a user to checkout a purchase. In addition, we have modified a few methods to receive an entire object instead of some of its components, and to properly access the components.

As our main focus is to verify the source code offered by SoftSlate, in our extension we are not adding any new feature to the ones already provided in the basic implementation, i.e., the functionality of the basic implementation and our extension is the same.

Note that when we started developing this case study there was an open source version of SoftSlate available online. However, later, this version was not available any more. Thus we cannot distribute the sources we have used. However, in [4] one may find the files for the *ppDATE* specifications described below.

2.9.1 *ppDATE* specification

Here we introduce two *ppDATE*s specifications, one describing a property related to the log in and log out of users in the web application, and one describing a property related to the checkout of the purchases performed by the users of the application. These properties address basic functionalities which we consider that a web cart application should offer.

Note that even though we could have either described more properties or specified more control- and data-oriented behaviour in the properties we are depicting in this section, the *ppDATE*s introduced here are sufficient to highlight the benefits of using StaRVOOrS in a real application. In addition, for readability reasons, Hoare triples are not going to be included on the figures depicting the *ppDATE*s. Moreover, as the application is placed in a server, the monitor generated by our tool is placed in the server as well.

Login — Logout

Users can freely browse through the web site of the application. However, if they want to buy products (i.e., pictures), they have to be logged in the application, to be able to proceed to the checkout section.

Fig. 2.14 and Fig. 2.15 illustrate the specification. The *ppDATE* in Fig. 2.14 creates instances of the *ppDATE* template *login-logout* whenever an object of class `User` is created, and the *ppDATE* template *login-logout* (Fig. 2.15) describes the following properties:

(i) *A user has to be logged in the application in order to perform a purchase, i.e., the checkout of a purchase can only happen between a login and a logout.*

(ii) *If a user is logged in, then that user cannot successfully log in again in the application until she logs out from it.*

(iii) *If a user is not logged-in, then that user cannot successfully log out from the application.*

(iv) *A user can only proceed to the checkout section if her status is a valid one.*

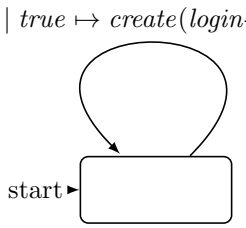


Figure 2.14: *ppDATE* in charge of creating instances of the template *login-logout*.

login-logout = λ *u* : `User`.

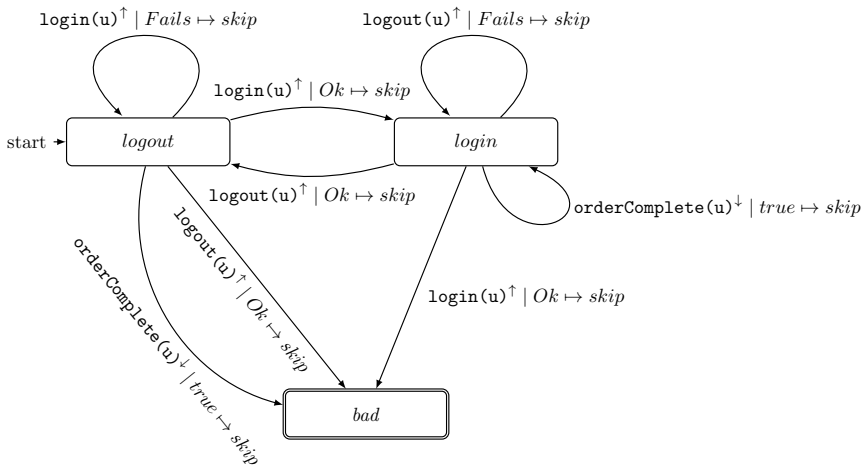


Figure 2.15: *ppDATE* template describing properties about the log in and log out of users.

(v) *A user who is not a costumer cannot proceed to the checkout section.*

The transitions of the *ppDATE* described by the template control properties (i)–(iii). Initially, this *ppDATE* is in state *logout*. Then, whenever there is a successful login, the *ppDATE* moves to state *login*. Later, once the user logs out, the *ppDATE* returns to state *logout*. Therefore, if a purchase is performed (i.e., an order is checkout) while the *ppDATE* is in state *login*, then the *ppDATE* remains in that state. However, if a purchase is performed while the *ppDATE* is in state *logout*, then it shifts to state *bad*.[12] In addition, while being at state *logout*, if an attempt to log in is not successful, then the *ppDATE* stays in that state; and if there is a successful logout, then the *ppDATE* shifts to state *bad* due to the fact the user is considered to be logged out while the *ppDATE* is in that particular state. Something similar happens when the *ppDATE* is in state *login*. (In Fig. 2.15, *Fails* and *Ok* are abbreviations, for presentation purpose, of real Java expression checking the failure or success of the respective operations.)

[12]Shifting to state *bad* means that a property was violated.

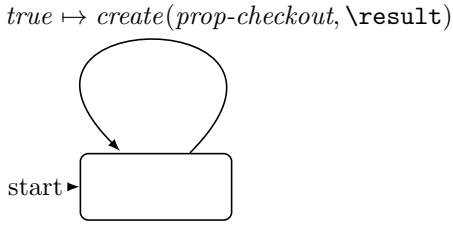


Figure 2.16: *ppDATE* in charge of creating instances of the template *prop-checkout*.

Regarding properties (iv) and (v), they are addressed using Hoare triples. For instance, property (iv) is represented as follows:

```
{ !baseForm.getUserStatus().equals("Registered")
  && !baseForm.getUserStatus().equals("Unapproved"); }
prepareCheckout(baseForm)
{ \result.equals("success"); }
```

As the only non valid statuses are "Registered" and "Unapproved", if the status of the user is not one of these values, then starting a purchase, i.e., using method `prepareCheckout`, should return "success". Regarding property (v), a user is only considered to be a costumer if she has logged-in into the application. Even though this property seems to be similar to property (i), this similarity is only apparent. Property (i) only addresses the proper order in which the methods should be executed, whereas property (v) focuses on controlling how the data related to a user is modified during such executions. Finally, both properties (iv) and (v) are only placed in state *login* because that is the only state in which a successful purchase can occur, i.e., (iv) and (v) are context dependent data-oriented properties.

Purchases Checkout

We consider that a purchase starts whenever an item (i.e., a product) is added to the cart. A user can continue either by adding other items to the cart or by removing some of the items from the cart. We refer to all the items in a cart as the *order*.

Once the user finishes the creation of her order, she may proceed to the checkout page. In SoftSlate, a checkout is realised in four steps. First, the user enters the contact information and delivery address. Then, the shipping method is selected (either ground transport or air transport), after which the user enters her credit card details. Finally, a confirmation for the order is requested. If accepted, the order is settled. Later, when the user receives the items, the order is considered to be completed.

Note that a user can modify her order as long as she has not yet confirmed it. If so, whenever she proceeds to the checkout section again, all its required steps have to be performed one more time. In addition, if the user removes all the items in an order, clears the cart or logs out[13], then the order is considered to be removed.

Fig. 2.16 and Fig. 2.17 illustrate a *ppDATE* specification where the *ppDATE* in Fig. 2.16 creates instances of the *ppDATE* template *prop-checkout* whenever an object of class `User` is created, and the *ppDATE* template *prop-checkout* (Fig. 2.17) describes the following properties:

(1) *The checkout of a purchase should be performed following the four required steps.*

[13]Logging out clears the cart.

prop-checkout = $\lambda\ u$: `User`.

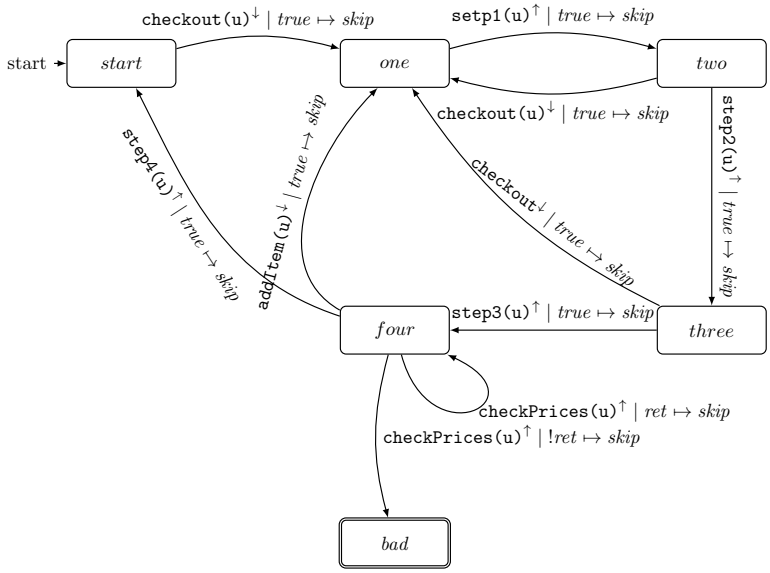


Figure 2.17: *ppDATE* template describing properties related to checkout of purchases.

(2) *It should not be possible to buy zero or less items.*

(3) *The expiration date of the credit card should not earlier than the current date.*

(4) *The price of a product should be positive.*

(5) *Before a purchase is completed, taxes should be processed.*

(6) *The total cost of a purchase should be equal to the sum of the prices of all the products to be purchased.*

(7) *If the price of an item changes, then its price in the order of the user should be updated.*

Again, consider the transitions of the *ppDATE* described by the template. When the first item is added to the cart, the *ppDATE* shifts to state *one*. In this state, once the first step of the checkout is completed, the *ppDATE* shifts to state *two*, and so on until reaching state *four*. In state *four*, once the order is settled, the *ppDATE* shifts back to state *start* in order to wait for a possible new purchase. Moreover, while being at either state *one*, *two*, *three* or *four*, if there is any change in the order, then the *ppDATE* shifts to state *one*, meaning that all the steps of the checkout have to be performed again. This is enough to control property (1).

Note that for readability reasons, in states *one*, *two*, *three* and *four* we have not included transitions going to state *start* whenever the user logs out, the cart is cleared or all the items in the cart are removed. In addition, we have not included transitions going to state *bad* from either state *one*, *two*, *three* or *four* if a step of the checkout was performed in a wrong way. For instance, if while being at state *one* either a second step, a third step or a fourth step of a purchase occurs instead of the first step, then the *ppDATE* shifts to state *bad*.

Regarding property (7), since the method in charge of updating the orders whenever the price of an item changes in the database is fully implemented using

different Java libraries, writing an appropriate Hoare triple for it would require introducing several work-arounds, e.g., defining stub versions of the methods associated to the different libraries and introducing invariants (or properties) regarding what would be expected as their behaviour. Instead, we implemented a method which compares the prices of the items in the order with their prices in the database, and include it as part of the information validation process corresponding to the fourth step of the purchase. Thereby, in state *four* there are two transitions controlling the result of this method. (Most real world applications of this kind would guarantee prices for some defined duration, and adjust it when that time has passed. For simplicity, we only model the latter in (7).)

Properties (2)–(6) are addressed with Hoare triples. Properties (2)–(4) are related to the integrity of the information introduced by either the users, in the case of (2) and (3), or the administrator, in the case of (4), on their requests to the server. Property (5) is related to the proper processing of taxes associated to the items in the current order. Property (6) enforces that the total amount that the user has to pay for her order should be equal to the sum of the totals of all the items included in the order.

As items could be added to the cart at any time during a purchase, property (2) is included in all the states of the *ppDATE*, with exception of the state *bad*.

On the other hand, property (3) is context dependent. This property should only be enforced on state *three*, which represents the step of a purchase where a user enters her credit card details. Note that, as it is in this case, a single property might be associated to several Hoare triples. For instance, below we introduce two of the four Hoare triples which describe property (3),

```
{ cardYear > actualYear; }
checkDate(cardMonth,cardYear, actualMonth,actualYear)
{ \result; }
```

```
{ cardYear < actualYear; }
checkDate(cardMonth,cardYear, actualMonth,actualYear)
{ !\result; }
```

Regarding property (4), we assume that initially all the data in the database is properly set. Therefore, this property should only be enforced every time that the administrator modifies the price of an item. As this may happen at any time during a purchase, this property is included in all the states of the *ppDATE*, with exception of the state *bad*.

In relation to property (5), in SoftSlate whenever the taxes of items are processed, the status of the order changes to "Tax processed". This change is done by using the following method,

```
public void setStatus(String s) { status = s;}
```

This method might be simply specified as follows:

```
{ true; }  setStatus(s)  { status.equals(s); }
```

However, due to the fact that taxes are processed while the *ppDATE* is in state *four*, that we know which particular value should be written when updating the status of the order, i.e., "Tax processed", and that *ppDATE* allows us to write context dependent properties, we include in *four* the following Hoare triple:

```
{ true; }  setStatus(s)  { status.equals("Tax␣processed"); }
```

Regarding property (6), it is represented by the following Hoare triple:

```
{ true; }
updateOrderAndDeliveryTotals(user,order,item)
{ user.getOrder().getSubtotal().doubleValue() ==
    (\old(user).getOrder().getSubtotal().doubleValue()
```

```
    + item.getTotal().doubleValue());}
```

In short, the new total amount is equal to the old total amount plus the amount of the newly added item.

2.9.2 Using StaRVOOrS

The previous specifications were analysed on a PC Pentium Core i7 using a single core. A similar setup was used to perform the experiments in the following section (2.9.3).

Since SoftSlate uses many Java libraries, to perform static analysis on its source code it was necessary to generate stub files for some of these libraries in order to allow KeY to find information about their method declarations.

Login — Logout

When feeding StaRVOOrS with this property and the source code of SoftSlate, it automatically generates a runtime monitored version of the application and a report which summarises the results obtained from the static analysis.

Regarding the result of the translation, it consisted of a *DATE* specification which looks exactly like the original *ppDATE* specification. The static analysis and instrumentation process takes 11 seconds, where most time is used by KeY to statically analyse the Hoare triples (approximately 7 seconds). By inspecting the report we notice that KeY successfully verified all the Hoare triples in the *ppDATE* specification. Thus, the refined *ppDATE* specification to be translated was already a *DATE*, .i.e, the translation process did not have add any new transitions to the specification.

Purchases Checkout

When feeding StaRVOOrS with this property and the source code of SoftSlate, it automatically generates a runtime monitored version of the application and a report which summarises the results obtained from the static analysis. The static analysis and instrumentation process takes 23 seconds, where most time is used by KeY to statically analyse the Hoare triples (approximately 20 seconds). By inspecting the report we can see that properties (2) and (3) are fully proved, properties (4) and (5) are not proved, and that property (6) and (7) are partially proved.

Regarding property (7), as KeY does not have any information about the state of purchases, and this property is context dependent, obviously, it is not able to prove it. However, thanks to the use of StaRVOOrS we can include this property in an appropriate state of the *ppDATE*, fact which guaranties that whenever a purchase reaches such state, this property is going to be verified at runtime by the generated monitor.

Regarding property (6), the report shows that this property postcondition is going to be checked upon entering method `updateOrderAndDeliveryTotals` only if the condition `user.getOrder() != null` holds. Thereby, this property is refined by StaRVOOrS as follows:

```
{ user.getOrder() != null; }
updateOrderAndDeliveryTotals(user,order,item)
{ user.getOrder().getSubtotal().doubleValue() ==
    (\old(user).getOrder().getSubtotal().doubleValue()
      + item.getTotal().doubleValue());}
```

This refined version of property (6) is the one verified by the generated monitor at runtime.

Finally, the result of the translation consisted on one *DATE* to create instances of the obtained *DATE* template *prop-checkout* (the translation of its homonymous *ppDATE* template), and three generated *DATE* templates whose instances verify properties (4)–(6). Note that the instances of the generated *DATE* templates are created by actions on the transitions of the *DATE* template *prop-checkout*.

2.9.3 Experimentation

2.9.3.1 Properties Analysis

Login — Logout

Although this property may appear to be simple, by verifying it we discovered unexpected behaviour in SoftSlate when a user logs in, performs a purchase, and logs out. In spite of the fact that the user was logged in the application, the monitor flagged a violation of property (iii). It turned out that after performing the purchase, SoftSlate replaced the object representing the logged-in user by a new one.

More concretely, the log file generated by the monitor showed that a new monitor, corresponding to a new instance of the template *login-logout*, was generated for the 'new' user. So, we got two different user objects, the one who originally logged in into the system (let's call it u_{logged}) and the new generated one (let's call it u_{new}). The new monitor (corresponding to the user u_{new}) would then be in its initial state, that is in the state *logout*. Thus, when the (real) user tried to log out, the monitor corresponding to user u_{new} shifted to a *bad* state, while the monitor corresponding to user u_{logged} remained in state *login*. As a consequence, property (iii) was violated.

In order to understand whether this is an error in the implementation we inspected the source code to better understand how the login and purchase were implemented. We found that each instance of class `User` was associated to a session, whose information was unique for each different execution of the application. Though the relation between (real) users and the session is bijective (for each real user there is a unique session, and vice versa), there were (at least) two instances of the class `User`, u_{logged} and u_{new}, associated with each session.

We were not sure what were the real reasons behind this design decision, but the implementation seemed correct, and our specification did not capture this situation. So, we decided to change our *ppDATE* template to capture this by including a Boolean variable reflecting whether the (real) user was connected or not, which we refer to as *active*. The updated *ppDATE* template is shown in Fig. 2.18. Further executions of the system (reproducing the previous executions and providing new ones) did not violate this property.

Purchases Checkout

We also run the system many times in order to analyse whether the execution of SoftSlate fulfils the properties described by the provided *ppDATE* specification.

First, we performed several purchases to analyse if property (1) was fulfilled. We added some items to the cart, bought them, and added and removed items at any stage of the checkout of a purchase, and then completed the purchase. None of these operations violated this property. We re-run the system executing the same steps as above to check property (5), which was not violated.

Next, we continued performing purchases, but this time the administrator of the application introduced modifications in the price of some items during the purchases. By doing so we were able to analyse whether properties (4), (6) and (7) were violated.[14]

[14] Remember that properties (2) and (3) were fully proved statically.

login-logout = $\lambda\ u : \texttt{User}.$

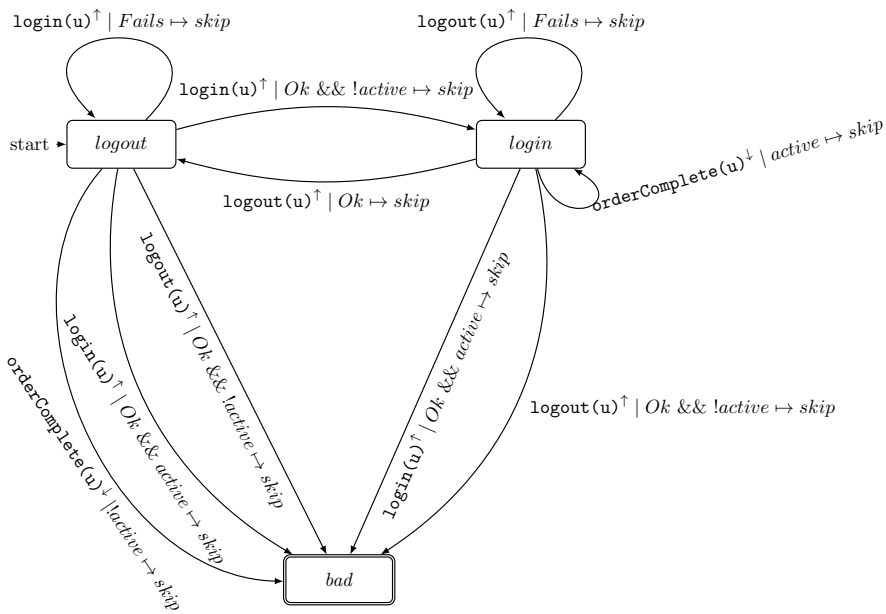


Figure 2.18: Extension on the *ppDATE* describing properties related to the log in and log out of users illustrated in Fig. 2.15.

In order to check whether property (4) held, we executed the system logged in as administrator and as a normal user (in parallel). The user performed a purchase (and thus the item was added to the cart), and as administrator we modified the price of the item introducing a negative value as its new price. At this moment the monitor reported that property (4) was violated. By inspecting the price of the modified item in the database, we could confirm that the negative value provided by the administrator was actually assigned to the item. This clearly was an error. We corrected this by not allowing to input negative numbers, and thus property (4) was finally satisfied.

On the other hand, when the administrator modified the price of an item introducing a positive value as its new price, then property (4) was fulfilled as expected. However, we noticed that property (7) was violated: some of the prices of the items in the order did not match with the prices in the database.[15] In particular, the mismatched values were those that were modified by the administrator: the new prices were propagated to the database but they were not updated in the visualisation of the cart (to the user). This was an error, and when inspecting the code we realised that there was a method implementing the propagation of the update, but it was not called when the change (done by the administrator) was performed. We have not yet corrected this error in the original code.

Property (6) was not violated by any of the previous executions.

[15]This also happened when entering negative numbers, but we only found out this when focusing on checking property (7) after correcting the issue with negative inputs.

Purchases	(a) no monitoring	(b) monitoring S	(c) monitoring S'
1	800 ms	1,300 ms	1,100 ms
10	10,500 ms	15,500 ms	13,000 ms
100	120,000 ms	190,000 ms	150,000 ms

Table 2.1: Performance of different purchases.

2.9.3.2 Runtime Verification Overhead Analysis

In this section we analyse the overhead added to SoftSlate by the monitor generated using StaRVOOrS. To perform this analysis, we considered three scenarios: several users performed one purchase, 10 purchases in a row, and 100 purchases in a row.

Table 2.1 shows the average execution time of: (a) an unmonitored execution of SoftSlate; (b) a monitored execution of SoftSlate using the original *ppDATE* specification S, and (c) a monitored execution of SoftSlate using specification S', obtained from S via static (partial) proof analysis using StaRVOOrS. In all three scenarios, the users and the server hosting SoftSlate were run in different computers with identical specifications (a PC Intel Core i7 using a single core). Note that as SoftSlate is an interactive application, in order to perform these experiments we have implemented a program which uses url connections to access the application and perform a purchase[16]. Therefore, our experiments consist on executing this program repeatedly and measuring its execution time.

As expected, adding a monitor to SoftSlate introduced overhead on its execution time. However, when we compared the overhead added by the monitor which uses the original *ppDATE* specification (without optimisations) (b), with the one added by the monitor which was generated using StaRVOOrS (c), one could notice a reduction in overheads gained by using our tool.

Through optimisations introduced by StaRVOOrS, we obtained a version of the monitor which, in relation to the times in (a), introduced in average a 25% of overhead to the execution time of the system. On the contrary, the monitor without the optimisations of StaRVOOrS introduced a 50% of overhead to the execution time.

Even though these results are not as impressive as the one we obtained on the case study analysed in [10] (Mondex, also reported here in Sec. 2.10), the monitor generated by our tool for SoftSlate still has a better performance than the one which uses the original *ppDATE* specification. The main difference lies in the amount of Hoare triples which have to be runtime verified in each case study. Every time an experiment is performed to analyse SoftSlate, the optimised monitor generated by StaRVOOrS verifies 3 Hoare triples, whereas the monitor using the original *ppDATE* specification (without optimisations) verifies 5. However, each experiment performed on Mondex requires the verification of 7 Hoare triples when using the unoptimised version of the monitor, whereas the optimised one does not have to verify any Hoare triples at all (cf. Sec. 2.10).

2.10 Case study: Mondex

Mondex is an electronic purse application which is used by smart cards products [2], and has been considered as a verification benchmark problem since 2006, originally appearing as case study as part of the Verified Software Grand Challenge [95].

[16]The package java.net is used here to handle the communication between our program and SoftSlate.

Mondex's original sanitised specification can be found in [85]. It consists of a Z specification [84], together with hand-written proofs of several properties.

Mondex essentially provides a financial transaction system supporting transferring of funds between accounts, or *purses*. Whenever a person has to make a transaction, electronic money is taken from their electronic purse and transferred to the target electronic purse. Such transactions are performed following a multi-step message exchange protocol: (1) the source and destination purses should (independently) register with the central fund transferring manager; (2) await a request to deduct funds from the source purse; (3) await a request to add the funds to the destination purse; and finally (4) an acknowledgement is sent to indicate that the transfer took place before the transaction ends.

In our version of this case study we consider a Java implementation running on a desktop computer instead of a Java Card implementation running on smart cards. The principal difference in the implementation is that in our version some methods return values to indicate whether their output is normal or erroneous, instead of raising Java Card exceptions. Our specification is strongly inspired by the JML formalisation presented in [88]. The full specification and source code of our case study can be found in [4]. The specification (see Fig. 2.19) consists of a *ppDATE* with 10 states, 25 transitions and a total of 26 different Hoare triples. The implementation of Mondex consists on 514 lines of code (without comments) which are distributed over 8 files.

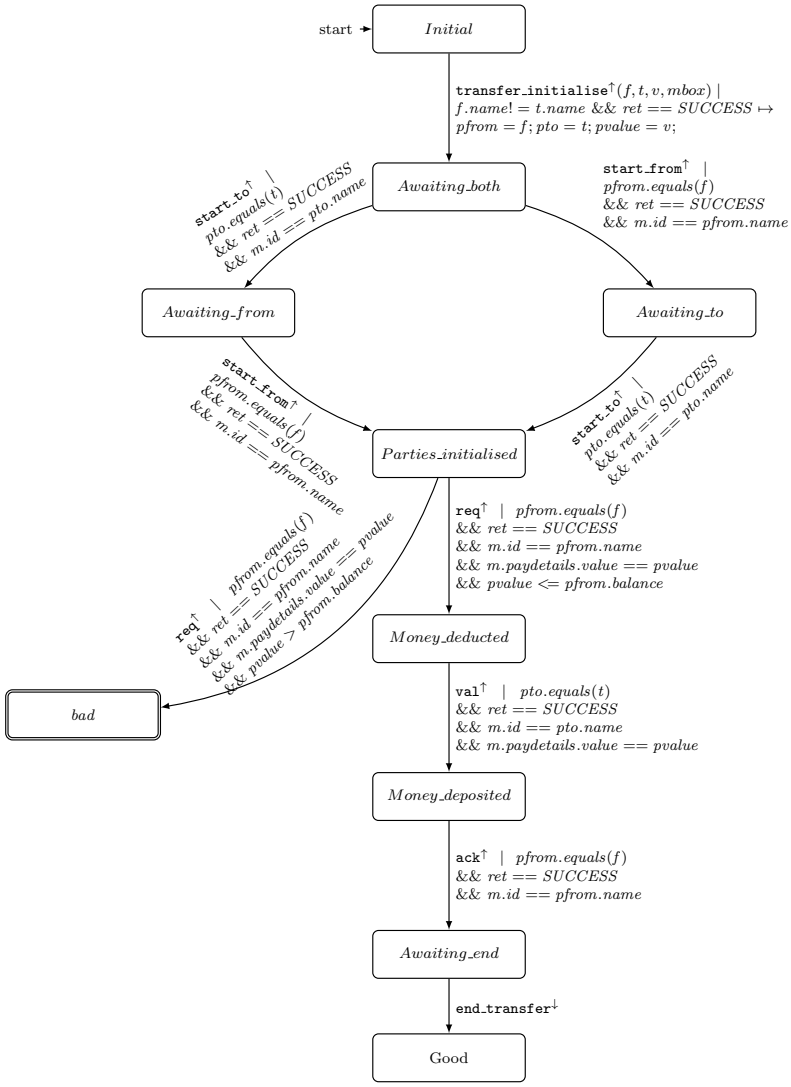
Note that *ppDATE* allows us to represent the overall status of the observer using *ppDATE* states. In other pre/post-style specification approaches, one would instead introduce additional data, and corresponding additional constraints, as is indeed done in [88] when specifying Mondex with JML. Such additional data implies a certain complexity of the specification, which somehow lacks the structure of the problem. We believe that specifications of this kind are sometimes developed with an automaton in mind. In *ppDATE*, we can make that automaton explicit. This being said, we want to stress again that we took great advantage of the JML specification of Mondex in [88].

2.10.1 *ppDATE* Property

Fig. 2.19 illustrates a *ppDATE* describing the top-level specification of Mondex. To keep the *ppDATE* readable, the description of the different Hoare triples are not included in the figure. (We will show some of them below.)

At the automaton level, the *ppDATE* specifies the control-oriented property which indicates how the multi-step message exchange protocol is suppose to work. For instance, after the parties are initialised (encoded in state Parties Initialised), a message requesting to transfer more money than the one available in the source purse should fail. Otherwise, such a message should take the *ppDATE* to a state in which the protocol now allows for the money to be transferred to the destination purse (named Money deducted). Note that the *ppDATE* will not take any explicit action whenever the state BAD STATE is reached. It will stay in this state until the whole monitor is restarted.

In contrast, the pre/postconditions properties placed on the states of the *ppDATE* ensure the well-behaviour of the methods involved in the individual steps of the protocol, behaviour which obviously changes together with the status of the protocol. For instance, once two purses agree on participating in a money transfer and the destination purse has requested for certain amount of money, (encoded in state Money Deducted), method `val_operation` which transfers money from the source purse to the destination one should succeed and increase the money of the destination purse by the sent amount (provided the limit of its account has not been reached), as shown in the Hoare triple below:



In addition:

- All states have outgoing transitions for `ret == SUCCESS && SENDER != party` (where party is the party from whom a message is not expected), going to a bad state.
- All states but `Awating_end` have outgoing transitions for `end_transfer`, going to a bad state.

Figure 2.19: *ppDATE* template describing the transaction protocol of Mondex to perform a transference.

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance); }
val_operation
{ \result == SUCCESS
```

```
  && (balance == \old(balance) + transaction.value); }
```

On the other hand, if the same method is accessed after the funds have already been transferred (encoded in state Money deposit), then the destination purse content should remain unchanged, and the request should be ignored:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance); }
val_operation
{ \result == IGNORED; }
```

Note that both Hoare triples above have the same precondition, but depending on the state of the *ppDATE* (i.e., the state of the protocol) different behaviours (i.e., postconditions) are expected for method `val_operation`.

2.10.2 Using StaRVOOrS

For this case study, we have used a setup identical to the one described in Sec. 2.10.2. Running StaRVOOrS on the source code of Mondex and the *ppDATE* depicted in Fig. 2.19 automatically produces a runtime monitored version of the application and a report summarising the results obtained from the static analysis. The static analysis and instrumentation process takes 1 minute 20 seconds, where most time is used by KeY to statically analyse the Hoare triples (approximately 1 minute 15 seconds).

The monitor generated by our tool consists one *DATE* to control the main property, and 24 *DATE*s templates to control the postconditions which were only partially verified by KeY, with 106 states and 196 transitions in total. By inspecting the report we can see that the two Hoare triples associated to the initialisation and termination of a transaction were fully proved, and that all the other 24 triples about the methods involved in the transaction protocol were the partially verified ones. For instance, let us consider the property already discussed in the previous section about method `val_operation`, which we will refer here to as *val_operation_ok*:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance); }
val_operation
{ \result == SUCCESS
  && (balance == \old(balance) + transaction.value); }
```

The report shows that the postcondition will have to be checked at runtime only when the condition `status != 2` holds upon entering `val_operation` (i.e., the destination purse is not waiting for the arrival of the requested money). Thus, the previous Hoare triple was refined by StaRVOOrS as follows:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance)
  && !(status == ProtocolStatus.Epv); }
val_operation
{ \result == SUCCESS
  && (balance == \old(balance) + transaction.value); }
```

This refined version of the property is the one which will be runtime verified by the generated monitor.

The size of the source code of the original implementation of Mondex was 23.5kB. After running the tool, the total size of all the generated files (i.e. instrumented version of the source code and the implementation of the monitor) grows to 277.4kB.

Transactions	(a) no monitoring	(b) monitoring S	(c) monitoring S'
10	8 ms	120 ms	15 ms
100	50 ms	3500 ms	90 ms
1000	250 ms	330000 ms	375 ms

Figure 2.20: Performance of different transactions which do not violate any of the specified properties

2.10.3 Experimentation

We now summarise the experimental results of applying our approach to the Mondex case study.

2.10.3.1 Normal Behaviour

The table 2.20 shows the execution time of: (a) an unmonitored implementation of Mondex; (b) a monitored implementation using the original *ppDATE* specification S, and (c) a monitored implementation using specification S', obtained from S via static (partial) proof analysis using StaRVOOrS. In all three scenarios, the system is run over a numbers of transactions which do not violate the specification. Note that in case (c), statically analysing all the Hoare triples took KeY around 1 minute, which however is done once and for all prior to deployment.

As one would expect, the addition of a monitor to the system introduces execution time overhead (b). However, if we compare this overhead to the one added by the monitor which was generated by StaRVOOrS (c), one can see a substantial overhead reduction, gained through the use of our tool. Through our optimisations we obtain a version which is at least 10 times faster for a low number of transactions, and this factor rises up to 900 when the number of transactions is increased. This significant reduction in execution time overheads is mainly due to the fact that monitoring data-centric properties may be prohibitively expensive. In fact, using S, each method invocation involved in the transfer protocol creates an additional *DATE* that will check the postcondition on exit. However, the postcondition checker is only created if the precondition holds on method invocation. In this case study, this causes large overheads when monitoring the unoptimised specification. Using the results from static verification, however, strengthens the preconditions by additional constraints, which in the Mondex case state were always falsified at invocation time, meaning that no postcondition checker is ever created. Apparently, in Mondex, the algorithmic complexity of the individual method implementations is limited enough such that KeY could fully prove the methods correct (automatically) *if only* the internal constraints corresponding to the *ppDATE* states were provided to KeY. But as they are not, KeY generates those constraints (closed branch conditions, see Sec. 2.4), and adds their negation to the preconditions. With that, the preconditions are never true at runtime. This phenomenon cannot be fully generalised to cases where KeY really lacks (automated) proving power for the code at hand, or where the code is faulty of course.

2.10.3.2 Faulty Behaviour

Usually, it is hard to get full proofs when using a static verifier like KeY without considering either user interaction with the prover or the use of special annotations, e.g., loop invariants, to help the prover on its task. However, it might be the case that the static verifier does not succeed in closing a branch in the proof due to the fact that

the remaining open goal was generated by an erroneous execution path. KeY cannot *per se* determine which one of these situations is dealing with. Fortunately, LARVA can detect the occurrence of the erroneous case whenever it appears at runtime.

We have intentionally injected errors into Mondex source to verify that the optimised monitor still detects them. Consider the case of a bug in the implementation of method `val_operation` — the value of variable `balance` is incremented with a different amount from the one given in the specification of the method. When analysing property `val_operation_ok`, KeY obviously does not manage to prove it. Therefore, the whole property will have to be runtime verified. The monitor spots this error reaching a bad state.

In addition, we have also considered incomplete and wrong specifications. In the case where the specification is too weak, the implementation may fulfil it for wrong reasons. As in all verification approaches, we may not catch this kind of problem. When using our verification approach there lies the possibility that the problem propagates to a state in which the specification is strong enough to identify it. For example, consider if the specification does not specify how the variables of a purse should be initialised by the `ConPurse` class constructor, and there is an implementation error where the variable `balance` is initialised to -1 instead of being initialised to 0. In spite of the error in the specification, KeY would proceed normally with the proofs and the previous particular situation would not be directly controlled on runtime. However, this erroneous initialisation leads to an erroneous initial charge of money in the purses (performed using the method `chargeMoney` in class `ConPurse`). As `balance` is negative, the previous method fails to update it with the new amount of money. Hence, after applying `chargeMoney` the value of `balance` is still -1. Thereby, whenever a purse tries to begin a transfer, either the method initialising the sender purse during a transaction or the method initialising the receiver purse during a transaction will fail its execution (the former due to insufficient funds and the latter due to a value overflow). This failure leads to an unsuccessful termination of the transfer, which is detected by the monitor controlling the transaction protocol and takes it to a bad state. This analysis can be easily concluded by inspecting the execution trace generated by the monitor. This trace allows one to backtrack through the execution of the different methods until reaching the one that was the cause the failure. In this scenario, it is important to note that in spite of the fact that we have not enforced any Hoare triple on the constructor of class `ConPurse`, it was specified and proved correct using KeY.

On the other hand, if a Hoare triple has an overly weak precondition or overly strong postcondition, then KeY will fail to prove the Hoare triple. STARVOORS thus ensures that the Hoare triple is checked at runtime, which allows us to realise the issue when expected results arise. Finally, another scenario is when the user uses erroneous data, not detected by the application. For instance, a user might request a transfer exceeding the amount of money in a purse. In this situation, the method initialising the sender purse during a transaction will fail its execution due to insufficient funds and this will lead to an unsuccessful termination of the transfer. This unsuccessful termination is detected by the runtime monitor controlling the transaction protocol.

2.11 Related Work

The combination of different verification techniques is gaining more and more popularity. One active area of research is the combination of testing and static analysis, e.g. [19, 32, 39, 47, 57, 59, 87]. Those works we have different objectives. We are not aiming at generating test cases, but at monitoring the actual post-deployment runs of the system. What we have in common is that static analysis/verification is used to limit the dynamic efforts, there by filtering test cases, here by filtering checks at

runtime.

Another line of research is the combination of testing and runtime verification. Decker *et al.* in [51] introduce an extension of the testing framework JUnit, which adds runtime verification artefacts to it. In this extension, during the execution of a test, a monitor is in charge of checking whether the actual executed test conforms with the property being monitored. In [18] Artho *et al.* present a framework where automated test case generation benefits from the use of runtime verification in a similar way to [51]. Falzon and Pace [55] study the combination of QuickCheck and LARVA by presenting a technique which extracts monitors from a QuickCheck testing specification. Even though this line of work has a different objective when compare to ours, it is worth mentioning that the QuickCheck automata used in [55] are quite similar to *ppDATE*s. QuickCheck automata employ pre/postconditions as part of their transitions, as opposed to *ppDATE*s which include them in the states of the automata. This similarity may suggest that it might be possible to extend our approach by also including the possibility of perform testing.

Another area worth mentioning is the combination of runtime assertion checks with runtime verification. In [48] de Boer *et al.* present SAGA, a framework which combines runtime assertion checking with monitoring. In contrast to our approach which targets data- and control-oriented properties in general, SAGA focuses only on the verification of data-flow and control-flow properties of Java classes and interfaces, e.g., interaction protocol among objects.

However, we are mainly interested in the combination of static verification and runtime verification such that static verification is used to reduce the overhead introduced to the system execution by monitoring properties. Wonisch *et al.* in [94] make use of program transformations in order to avoid unsafe program executions. In [30] the efficiency of runtime monitoring based on tracematches is improved by using a static analysis technique which reduces the runtime instrumentation needed. The technique consists on three stages: exclusion of some tracematches, elimination of inconsistent instrumentation points, and additionally refinement of this analysis considering the order of execution.

Other works use this kind of combination but with different goals. In [31] Bodden and Lam present CLARA, a framework which uses static techniques aiming to improve the monitors themselves, instead of verifying software. The work by Zee *et al.* in [96] investigates the combination of static and runtime verification, but aiming at a specification language whose specifications may be both statically and runtime checked. With this goal in mind, they extend the static verifier Jahob by adding techniques to verify specifications at runtime. In this approach, most of the properties which can be verified are data-oriented, as opposed to ours where control-oriented properties are covered as well. In [83] Sözer integrates static code analysis and runtime verification. On this approach, runtime verification statements are created from static code analysis alerts, in order to generate monitors which will allow to both check for possible faults in the system and eliminate false positives obtained in the static phase.

Many specification approaches, such as SPARK [21], JML [72] and SPEC# [22] are supported by both static and runtime verification tools. Nevertheless, to the best of our knowledge, static verification is not used to optimise the runtime verification of properties.

2.12 Conclusions

In this paper we have presented STARVOORS, a framework for verifying integrated data- and control-oriented properties for Java programs, using a combination of

static and runtime verification. The StaRVOOrS tool-chain uses KeY [9] for static verification, and Larva [46] for the verification performed at runtime.

We have presented the language *ppDATE* which is based on automata and pre/post conditions to describe properties of both, the control flow and the data computations. The basic structuring principle of the language is the composition of parallel automata, whose transitions fire simultaneously in reaction to events of the observed system, but also in reaction to events generated by some automata in the previous step. A distinguishing feature of the language is the inclusion of functional properties of computation units into the above, thereby capturing the dependency of functional properties on the history of previous events, by assigning Hoare triples to (automata-theoretic) states. Finally, the template concept allows to parameterise components in a great variety of ways, and create concrete instantiations dynamically.

We also presented here a semantics of *ppDATE*s, precisely describing the interplay of transitions, event consumption and generation, Hoare triple monitoring, creation of template instances. We then use the semantics to prove soundness of the algorithm our tool uses to translate *ppDATE* into *DATE*, allowing us to employ the *DATE* tool Larva as a back-end for runtime verifying *ppDATE* specifications.

This article also reports on the application of StaRVOOrS to SoftSlate, an open-source shopping cart web application. In this case study, we analyse *ppDATE*s describing properties about the proper behaviour of the system while users perform purchases. We selected this case study because verifying a real application is always quite challenging, and dealing with it would gave us a better perspective regarding the benefits which can be obtained when using our tool. We also report on the application of StaRVOOrS to the verification benchmark Mondex, an electronic purse application. We demonstrate how properties can be verified using combined static and runtime verification. This case study was selected because it is a usual benchmark in the static verification community, and we thought that it would be interesting to analyse what the use of runtime verification could bring into play.

As with all case studies, the empirical observations are difficult to generalise. However, our experimental results give an indication of what gains are possible with our technique. For SoftSlate, the overhead of pure runtime verification (without employing static verification) is roughly 50%, a penalty which we get down to roughly 25% when using StaRVOOrS, by facilitating static verification (cf. Sec. 2.9.3.2). These differences are much smaller compared to when we applied StaRVOOrS to the Mondex case study, where pure runtime verification created a much higher overhead. Compared to that, the monitor created by StaRVOOrS was 10 times faster for a low number of transactions, and up to 900 times faster as the number of transactions increase. 'When using the monitor generated from the original specification provided for Mondex, the execution of each method involved in a transaction (7 in total) creates an additional *DATE* to be traversed in parallel, which is in charge of checking the postcondition. This would lead to the large overheads obtained in that case study. However, when using the monitor generated by StaRVOOrS, thanks to the optimisations introduced in the specification by this tool, no additional *DATE*s are created when a transaction is performed, because the additional checks in the preconditions are false at runtime.

As a final remark, note that the efficiency gain for monitoring will benefit from any improvements in the used static and runtime verifiers. For instance, if KeY is improved in such a way that more branches are closed during the static proof, then this will have an immediate effect in StaRVOOrS thus reducing the runtime overhead. Similarly, any optimisation performed in Larva will only bring benefits to our tool.

We are currently looking at ways of pushing our techniques further. On one hand, we are looking at techniques to add control-flow static analysis to StaRVOOrS, thus

benefiting from further optimisation prior to deployment. We are also looking at extending the framework to deal with distributed systems [15], which brings in new challenges, and might require assume-guarantee reasoning to enable us to perform static analysis based optimisations.

A Proofs of Coupling Invariant Lemmas

In order to prove both Lemma 2 and Lemma 3, we introduce the following two propositions. Prop. 1 says that the translation algorithm only modifies the actions of the transitions in the translated *ppDATE* network. Prop. 2 says that for every transition in the translation either there is a similar transition in the original *ppDATE* network, or there is not such a transition, due to the fact that the transition is a new loop transition (added by the translation to control Hoare triples).

Remember that we represent the translation of a single *ppDATE* to *DATE* with the function $\kappa \in ppDATE \mapsto DATE$.

Proposition 1. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$ and its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$,

$$\begin{array}{l} \forall\ m, q, q', tr, c, a\ \cdot \\ \quad q \xrightarrow{tr|c \mapsto a}_m q'\ and\ m \in M\ and\ \kappa(m) \in M' \cdot \\ \quad\quad (\exists\ a' \cdot q \xrightarrow{tr|c \mapsto a'}_{\kappa(m)} q') \end{array}$$

Proof. Given a *ppDATE* $m \in M$ and a state $q \in Q_m$, whenever $\Pi_m(q) = \emptyset$, $\Pi_m(q) \neq \emptyset$ but there is no Hoare triple associated to the method related to trigger tr, or the trigger is associated to exiting a method, by Step 3b., transitions remain unchanged in the translation. Therefore, $a' = a$ in these cases.

On the other hand, for each clashing Hoare triple $\{\pi\}\,\sigma\,\{\pi'\} \in \Pi_m(q)$, by step $3a_2$., the transition $q \xrightarrow{tr|c \mapsto a}_m q'$ is replaced by one of the following transitions:
$q \xrightarrow{tr|c \mapsto \{a;\ \texttt{if}\ \pi\ \texttt{then create}(post\_checker, (\sigma^{\uparrow}_{id}, \pi'))\}}_{\kappa(m)} q'$, or
$q \xrightarrow{tr|c \mapsto \{a;\ \texttt{if}\ \pi\ \texttt{then create}(post\_checker\_h, (\sigma^{\uparrow}_{id}, val_i))\}}_{\kappa(m)} q'$.
Thereby, either $a' = a;\ \texttt{if}\ \pi\ \texttt{then create}(post\_checker, (e^{\uparrow}_{id}, \pi'))\}$, or
$a' = a;\ \texttt{if}\ \pi\ \texttt{then create}(post\_checker\_h, (e^{\uparrow}_{id}, val_i))\}$.

Finally, as in step $3a_1$ non-clashing Hoare triples add new transitions but do not modified existing ones, this case trivially holds. □

Proposition 2. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$ and its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$,

$$\begin{array}{l} \forall\ m', q, q', tr, c, a\ \cdot \\ \quad q \xrightarrow{tr|c \mapsto a}_{m'} q'\ and\ m' \in M' \cdot \\ \quad\quad (\exists\ m, a' \cdot m \in M, \kappa(m) = m' \cdot q \xrightarrow{tr|c \mapsto a'}_m q') \\ \quad\quad or \\ \quad\quad ((\nexists\ m, a' \cdot m \in M, \kappa(m) = m' \cdot q \xrightarrow{tr|c \mapsto a'}_m q')\ and\ (q = q')) \end{array}$$

Proof. Each transition $t' \in t'_m$ for any $m' \in M'$ is obtained by applying either step $3a_1$, $3a_2$ or 4b.

If t' was obtained by applying step $3a_1$, then it is a new loop transition added by the translation, i.e., its origin and destination states are the same, and given a *ppDATE* $m \in M$ such that $\kappa(m) = m'$, there not exists a transition associated to t' in m. Therefore, the right side of the disjunction holds.

If t' was obtained by applying step $3a_2$, then, given a *ppDATE* $m \in M$ such that $\kappa(m) = m'$, either there exists one transition on m with the same trigger, same condition, and similar action (but without including the if-expression checking the precondition), or t' is a new loop transition added by the translation. In the first case the left side of the disjunction holds, whereas in the the second case the right side of the disjunction holds.

Finally, if t' was obtained by applying step 3*b*, then, given a *ppDATE* $m \in M$ such that $\kappa(m) = m'$, m has exactly the same transition. Therefore, the left-hand side of the disjunction holds in these cases. □

Now, we proceed to prove the lemmas.

Lemma 2. Given a network of *ppDATE*s $pn = (M, V, \nu_0, T_{ppd})$, its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$, a trace $w \in (systemevent \times \Theta_{Sys})^*$, and the global configurations (L, ν) and $(\tilde{L}, \nu')$,

$$
\begin{array}{l}
C_{init}(pn) \stackrel{w}{\Rightarrow}_M (L, \nu) \text{ and } C_{init}(ppd2DATE(pn)) \stackrel{w}{\Rightarrow}_{M'} (\tilde{L}, \nu') \\
\text{implies} \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L} \ \cdot \kappa(m) = m' \text{ and } q = q' \\
\quad \text{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\qquad \exists\, m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad \text{and} \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad \text{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot \\
\qquad \exists\, m, q, \rho \cdot (m, q, \rho), m \notin M \in L \cdot q = q' \\
\quad \text{and} \\
\quad \nu = \nu'
\end{array}
$$

Proof. We proceed to prove this lemma by induction on the length of the trace w.

- Base case: $w = \varepsilon$ (empty trace)

$$
\begin{array}{l}
C_{init}(pn) \stackrel{\varepsilon}{\Rightarrow}_M (L, \nu) \text{ and } C_{init}(ppd2DATE(pn)) \stackrel{\varepsilon}{\Rightarrow}_{M'} (\tilde{L}, \nu') \\
\text{implies} \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot \kappa(m) = m' \text{and } q = q' \\
\quad \text{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\qquad \exists\, m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad \text{and} \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad \text{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot \\
\qquad \exists\, m, q, \rho \cdot (m, q, \rho), m \notin M \in L \cdot q = q' \\
\quad \text{and} \\
\quad \nu = \nu'
\end{array}
$$

By Def. 17 and Def. 21, we know that

$$
\begin{array}{l}
L_0 = L \textit{ and } \nu_0 = \nu \textit{ and } L_0' = \tilde{L} \textit{ and } \nu_0 = \nu' \\
\textit{implies} \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot \kappa(m) = m' \textit{and } q = q' \\
\quad \textit{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\qquad \exists\, m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad \forall\, m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad \textit{and} \\
\quad \forall\, m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot \\
\qquad \exists\, m, q, \rho \cdot (m, q, \rho), m \notin M \in L \cdot q = q' \\
\quad \textit{and} \\
\quad\ \nu = \nu'
\end{array}
$$

where $L_0 = \{(m, q_{0m}, \emptyset) \mid m \in M\}$, and $L_0' = \{(m', q_{0m'}, \emptyset) \mid m' \in M'\}$.

Next, by substitution with the antecedents we have to prove

$$
\begin{array}{l}
(1)\ \forall\, m, q, \rho \cdot (m, q, \rho) \in L_0, m \in M \cdot \\
\quad \exists\, m', q' \cdot (m', q', \emptyset) \in L_0' \cdot \kappa(m) = m' \textit{ and } q = q' \\
\textit{and} \\
(2)\ \forall\, m', q' \cdot (m', q', \emptyset) \in L_0', m' \in M' \cdot \\
\quad \exists\, m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L_0 \cdot q = q' \\
\textit{and} \\
(3)\ \forall\, m, q, \rho \cdot (m, q, \rho) \in L_0, m \notin M \cdot \\
\qquad \exists\, m', q' \cdot (m', q', \emptyset) \in L_0', m' \notin M' \cdot q = q' \\
\textit{and} \\
(4)\ \forall\, m', q' \cdot (m', q', \emptyset) \in L_0', m' \notin M' \cdot \\
\qquad \exists\, m, q, \rho \cdot (m, q, \rho), m \notin M \in L_0 \cdot q = q' \\
\textit{and} \\
(5)\ \nu_0 = \nu_0
\end{array}
$$

As in L_0' all the *DATE* components of the local configurations correspond to the translation of *ppDATE* in *pn*, both (1) and (2) are trivially fulfilled, and the ranges of both (3) and (4) are never fulfilled, meaning that, as these ranges are empty (i.e., *false*), both expressions are trivially evaluated to *true*. In addition, (5) is trivially fulfilled. Thereby, the base case holds.

- Inductive case: $w = w' : (e, \theta)$

IH: $\forall\ L, \tilde{L}, \nu, \nu' \cdot$

$$
\begin{array}{l}
C_{init}(pn) \stackrel{w'}{\Longrightarrow}_M (L, \nu)\ \mathit{and}\ C_{init}(\mathit{ppd2DATE}(pn)) \stackrel{w'}{\Longrightarrow}_{M'} (\tilde{L}, \nu') \\
\mathit{implies} \\
\quad \forall\ m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot \kappa(m) = m'\ \mathit{and}\ q = q' \\
\quad \mathit{and} \\
\quad \forall\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\qquad \exists\ m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad \mathit{and} \\
\quad \forall\ m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad \mathit{and} \\
\quad \forall\ m', q' \cdot (m', q', \emptyset) \in L', m' \notin M' \cdot \\
\qquad \exists\ m, q, \rho \cdot (m, q, \rho), m \notin M \in \tilde{L} \cdot q = q' \\
\quad \mathit{and} \\
\quad \nu = \nu'
\end{array}
$$

Given the previous inductive hypothesis IH, we have to prove,

$$
\begin{array}{l}
C_{init}(pn) \stackrel{w':(e,\theta)}{\Longrightarrow}_M (L, \nu)\ \mathit{and}\ C_{init}(\mathit{ppd2DATE}(pn)) \stackrel{w':(e,\theta)}{\Longrightarrow}_{M'} (\tilde{L}, \nu') \\
\mathit{implies} \\
\quad \forall\ m, q, \rho \cdot (m, q, \rho) \in L, m \in M \cdot \\
\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot \kappa(m) = m'\ \mathit{and}\ q = q' \\
\quad \mathit{and} \\
\quad \forall\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \in M' \cdot \\
\qquad \exists\ m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L \cdot q = q' \\
\quad \mathit{and} \\
\quad \forall\ m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}, m' \notin M' \cdot q = q' \\
\quad \mathit{and} \\
\quad \forall\ m', q' \cdot (m', q', \emptyset) \in L', m' \notin M' \cdot \\
\qquad \exists\ m, q, \rho \cdot (m, q, \rho), m \notin M \in \tilde{L} \cdot q = q' \\
\quad \mathit{and} \\
\quad \nu = \nu'
\end{array}
$$

By Def. 21 we have,

$$
\begin{array}{l}
\quad (i)\ \exists\ L'', \nu'' \cdot C_{init}(pn) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ \mathit{and}\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (L, \nu) \\
\mathit{and} \\
\quad (ii)\ \exists\ L'', \nu'' \cdot C_{init}(\mathit{ppd2DATE}(pn)) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ \mathit{and}\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (\tilde{L}, \nu')
\end{array}
$$

Then, we proceed with the proof by assuming the antecedent of the implication. This assumption allows us to remove the existential quantifiers in the antecedents by introducing the fresh values L'' and ν'' in (i), and the fresh values $\tilde{L}''$ and ν''' in (ii). Therefore, we have

$$
\begin{array}{l}
\quad (i')\ C_{init}(pn) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ \mathit{and}\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (L, \nu) \\
\mathit{and} \\
\quad (ii')\ C_{init}(\mathit{ppd2DATE}(pn)) \stackrel{w'}{\Longrightarrow} (\tilde{L}'', \nu''')\ \mathit{and}\ (\tilde{L}'', \nu''') \stackrel{(e,\theta)}{\Longrightarrow} (\tilde{L}, \nu')
\end{array}
$$

Next, by *IH* we know

$$
\begin{aligned}
&(iii)\ \forall\, m,q,\rho\cdot(m,q,\rho)\in L'', m\in M\cdot\\
&\qquad \exists\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}''\cdot \kappa(m)=m' \ and\ q=q'\\
&and\\
&(iv)\ \forall\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}'', m'\in M'\cdot\\
&\qquad \exists\, m,q,\rho\cdot m\in M, \kappa(m)=m', (m,q,\rho)\in L''\cdot q=q'\\
&and\\
&(v)\ \forall\, m,q,\rho\cdot(m,q,\rho)\in L, m\notin M\cdot\\
&\qquad \exists\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}, m'\notin M'\cdot q=q'\\
&and\\
&(vi)\ \forall\, m',q'\cdot(m',q',\emptyset)\in L', m'\notin M'\cdot\\
&\qquad \exists\, m,q,\rho\cdot(m,q,\rho), m\notin M\in \tilde{L}\cdot q=q'\\
&and\\
&(vii)\ \nu''=\nu'''
\end{aligned}
$$

In relation to L, by (i) we know it is obtained from L'' after performing a big step with (e,θ). Thereby, the local configurations on L are either the same as in L'', a modified version of the ones in L'', or new local configurations added to control a *DATE* which is a new instance of a template.

Let us introduce the sets L_{nc}, L_c and L_{new}, to represent the local configurations in each one of the previous categories, respectively. Then, we know that

$$(viii)\ L=L_{nc}\cup L_c\cup L_{new}$$

In addition, by using a similar approach with $\tilde{L}$ and (ii), we introduce the following sets.

$$(ix)\ \tilde{L}=\tilde{L}_{nc}\cup \tilde{L}_c\cup \tilde{L}_{new}$$

Let us come back now to the expression we want to prove.

$$
\begin{aligned}
&(x)\ \forall\, m,q,\rho\cdot(m,q,\rho)\in L, m\in M\cdot\\
&\qquad \exists\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}\cdot \kappa(m)=m' \ and\ q=q'\\
&and\\
&(xi)\ \forall\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}, m'\in M'\cdot\\
&\qquad \exists\, m,q,\rho\cdot m\in M, \kappa(m)=m', (m,q,\rho)\in L\cdot q=q'\\
&and\\
&(xii)\ \forall\, m,q,\rho\cdot(m,q,\rho)\in L, m\notin M\cdot\\
&\qquad \exists\, m',q'\cdot(m',q',\emptyset)\in \tilde{L}, m'\notin M'\cdot q=q'\\
&and\\
&(xiii)\ \forall\, m',q'\cdot(m',q',\emptyset)\in L', m'\notin M'\cdot\\
&\qquad \exists\, m,q,\rho\cdot(m,q,\rho), m\notin M\in \tilde{L}\cdot q=q'\\
&and\\
&(xiv)\ \nu=\nu'
\end{aligned}
$$

By (iii) and (iv), as the values in both L_{nc} and $\tilde{L}_{nc}$ are the same as in L'' and $\tilde{L}''$, respectively, we know that these values fulfil all the previous expressions. Thereby, we can reduce $(viii)$ and (ix) to

$$(viii')\ L=L_c\cup L_{new}\quad (ix')\ \tilde{L}=\tilde{L}_c\cup \tilde{L}_{new}$$

Regarding the newly created local configurations in both L_{new} and $\tilde{L}_{new}$, they do not fulfil the ranges of the universal quantifications in neither (x) nor (xi). In addition, by Prop. 1 and Prop. 2, we know that the only difference in the executed actions in the *ppDATE*s in *pn* and their translation is that the actions in the *DATE*s may include the creation of an instance of template *exit_cond_checker*. Besides, by

step 4 in the translation algorithm, we now that both the *ppDATE*s templates and their translations have similar transitions and are initialised in the same state. Thus, (xii) and (xii) are fulfilled for these values, and we can reduce $(viii)$ and (ix) to

$$(viii'')\ L = L_c \quad (ix'')\ \tilde{L} = \tilde{L}_c$$

Therefore, we have to prove,

$$\begin{aligned}
&(x')\ \forall\ m, q, \rho \cdot (m, q, \rho) \in L_c, m \in M \cdot \\
&\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c \cdot \kappa(m) = m'\ \textit{and}\ q = q' \\
&\textit{and} \\
&(xi')\ \forall\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c, m' \in M' \cdot \\
&\qquad \exists\ m, q, \rho \cdot m \in M, \kappa(m) = m', (m, q, \rho) \in L_c \cdot q = q' \\
&\textit{and} \\
&(xii')\ \forall\ m, q, \rho \cdot (m, q, \rho) \in L, m \notin M \cdot \\
&\qquad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c, m' \notin M' \cdot q = q' \\
&\textit{and} \\
&(xiii')\ \forall\ m', q' \cdot (m', q', \emptyset) \in L_c, m' \notin M' \cdot \\
&\qquad \exists\ m, q, \rho \cdot (m, q, \rho), m \notin M \in \tilde{L}_c \cdot q = q' \\
&\textit{and} \\
&(xiv)\ \nu = \nu'
\end{aligned}$$

By (iii) and Prop. 1 we know that for every enabled transition of a *ppDATE* $m \in M$, there is one enabled transition in $\kappa(m) \in M'$ performing the same change of state and, if any, generating the same action events. Thereby, both *pn* and its translation will shift the local configurations in L_c and $\tilde{L}_c$, respectively, in the same manner, i.e., (x') holds.

In addition, by (iv) and Prop. 2 we know that for every enabled transition in a *DATE* $m' \in M'$, there is either an enabled transition in a *ppDATE* $m \in M$, where $\kappa(m) = m'$, such that this transition performs the same change of state and, if any, generates the same action events, or the transition enabled in m' is a loop transition.

In the first case, both *pn* and its translation will shift the local configurations in L_c and $\tilde{L}_c$, respectively, in the same manner. Thus, (xi') holds.

In the second case, the local configuration obtained after the shift is in the same state as before the shift. Thus, by (iv), this (xi') holds.

Moreover, by *IH*, Prop. 1, Prop. 2 we know that whenever a *ppDATE* in *pn* creates an instance of a template, its translation will create an instance of the translation of such template, and vice versa. Besides, by the step 4 in the translation algorithm, as such instances have similar transitions, they will shift the local configuration associated to them in the same manner. Therefore, both (xii') and $(xiii')$ are fulfilled.

Finally, in relation to (xiv), by Prop. 1 and Prop. 2 we know that only difference in the executed actions in *pn* and its translation is that the actions of the latter may include the creation of an instance of template *exit_cond_checker* (whose actions do not modify *ppDATE* variables valuations). In addition, by step 4 in the translation algorithm we know that both an instance of a *ppDATE* template and a similar instance of the translation of the template will fire similar transitions (with the same actions). Therefore, they perform the same modifications in the valuations ν'' and ν'''. Thus, by (vii), (xiv) holds. □

Lemma 3. Given a network of *ppDATE*s $pn = (M, V, \nu_0, T_{ppd})$, its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$, a trace $w \in (systemevent \times \Theta_{Sys})^*$, and the global configurations (L, ν) and $(\tilde{L}, \nu')$,

$$C_{init}(pn) \overset{w}{\Rightarrow}_M (L, \nu)\ \textit{and}\ C_{init}(ppd2DATE(pn)) \overset{w}{\Rightarrow}_{M'} (\tilde{L}, \nu')\ \textit{implies}\ \psi(L, \tilde{L})$$

where,

$$\begin{aligned}\psi(L,\tilde{L}) = \forall\ & m,q,\rho\cdot(m,q,\rho)\in L\cdot\\ & \forall\ \sigma^{\uparrow}_{id},\pi',\theta\cdot(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho\cdot\\ & \exists\ m',q'\cdot(m',q',\emptyset)\in\tilde{L}\cdot inst\,(exit\_cond\_checker,\sigma,\pi') = m'\\ and\\ & \forall\ m',q'\cdot(m',q',\emptyset)\in\tilde{L}, m'\notin M'\cdot\\ & \exists\ \sigma^{\uparrow}_{id},\pi'\cdot inst\,(exit\_cond\_checker,\sigma,\pi') = m'\\ & implies\ \exists\ m,q,\rho,\theta\cdot(m,q,\rho)\in L\cdot(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho\end{aligned}$$

Proof. We proceed to prove this lemma by induction on the length of the trace w.

- Base case: $w = \varepsilon$ (empty trace)

$C_{init}(pn) \stackrel{\varepsilon}{\Rightarrow}_M (L,\nu)$ *and* $C_{init}(ppd2DATE(pn)) \stackrel{\varepsilon}{\Rightarrow}_{M'} (\tilde{L},\nu')$ *implies* $\psi(L,\tilde{L})$

By Def. 17 and Def. 21 we know that

$$L_0 = L\ and\ \nu_0 = \nu\ and\ L'_0 = \tilde{L}\ and\ \nu_0 = \nu'\ implies\ \psi(L,\tilde{L})$$

where $L_0 = \{(m,q_{0m},\emptyset)\mid m\in M\}$, and $L'_0 = \{(m',q_{0m'},\emptyset)\mid m'\in M'\}$.

Next, by substitution with the antecedents,

$$L_0 = L\ and\ \nu_0 = \nu\ and\ L'_0 = \tilde{L}\ and\ \nu_0 = \nu'\ implies\ \psi(L_0,L'_0)$$

Thus, by the definition of ψ we have to prove that,

$$\begin{aligned}& \forall\ m,q,\rho\cdot(m,q,\rho)\in\{(m,q_{0m},\emptyset)\mid m\in M\}\cdot\\ & \quad\forall\ \sigma^{\uparrow}_{id},\pi',\theta\cdot(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho\cdot\\ & \quad\quad\exists\ m',q'\cdot(m',q',\emptyset)\in\{(m',q_{0m'},\emptyset)\mid m'\in M'\}\cdot\\ & \quad\quad inst\,(exit\_cond\_checker,\sigma,\pi') = m'\\ & and\\ & \forall\ m',q'\cdot(m',q',\emptyset)\in\{(m',q_{0m'},\emptyset)\mid m'\in M'\}, m'\notin M'\cdot\\ & \quad\exists\ \sigma^{\uparrow}_{id},\pi'\cdot inst\,(exit\_cond\_checker,\sigma,\pi') = m'\\ & \quad\quad implies\ \exists\ m,q,\rho,\theta\cdot(m,q,\rho)\in\{(m,q_{0m},\emptyset)\mid m\in M\}\cdot\\ & \quad\quad\quad(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho\end{aligned}$$

First, let us analyse the expression,

$$\begin{aligned}& \forall\ m,q,\rho\cdot(m,q,\rho)\in\{(m,q_{0m},\emptyset)\mid m\in M\}\cdot\\ & \quad\forall\ \sigma^{\uparrow}_{id},\pi',\theta\cdot(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho\cdot\\ & \quad\quad\exists\ m',q'\cdot(m',q',\emptyset)\in\{(m',q_{0m'},\emptyset)\mid m'\in M'\}\cdot\\ & \quad\quad inst\,(exit\_cond\_checker,\sigma,\pi') = m'\end{aligned}$$

As ρ is always the empty set, the condition $(\sigma^{\uparrow}_{id},\pi',\theta)\in\rho$ will always evaluate to *false*. Therefore,

$$\begin{aligned}& \forall\ m,q,\rho\cdot(m,q,\rho)\in\{(m,q_{0m},\emptyset)\mid m\in M\}\cdot\\ & \quad\forall\ \sigma^{\uparrow}_{id},\pi',\theta\cdot false\cdot\\ & \quad\quad\exists\ m',q'\cdot(m',q',\emptyset)\in\{(m',q_{0m'},\emptyset)\mid m'\in M'\}\cdot\\ & \quad\quad inst\,(exit\_cond\_checker,\sigma,\pi') = m'\end{aligned}$$

Then, as the range of the inner universal quantification is empty (i.e., *false*), it is trivially evaluated to *true*.

$$\forall\ m,q,\rho\cdot(m,q,\rho)\in\{(m,q_{0m},\emptyset)\mid m\in M\}\cdot true$$

Finally, as the body of the previous universal quantification is simply the value *true* and its range is not empty, the whole expression is trivially evaluated to *true*.

Now, let us analyse the expression,

$$\begin{array}{l}\forall\ m', q' \cdot (m', q', \emptyset) \in \{(m', q_{0m'}, \emptyset) \mid m' \in M'\}, m' \notin M' \cdot \\ \quad \exists\ \sigma_{id}^{\uparrow}, \pi' \cdot inst\ (exit\_cond\_checker, \sigma, \pi') = m' \\ \qquad\qquad implies\ \exists\ m, q, \rho, \theta \cdot (m, q, \rho) \in \{(m, q_{0m}, \emptyset) \mid m \in M\} \cdot \\ \qquad\qquad\qquad (\sigma_{id}^{\uparrow}, \pi', \theta) \in \rho\end{array}$$

As in the initial configuration of the translation of *pn* there are no instances of *DATE* templates, the range of the universal quantification is always evaluated to *false*. Therefore,

$$\begin{array}{l}\forall\ m', q' \cdot false \cdot \\ \quad \exists\ \sigma_{id}^{\uparrow}, \pi' \cdot inst\ (exit\_cond\_checker, \sigma, \pi') = m' \\ \qquad\qquad implies\ \exists\ m, q, \rho, \theta \cdot (m, q, \rho) \in \{(m, q_{0m}, \emptyset) \mid m \in M\} \cdot \\ \qquad\qquad\qquad (\sigma_{id}^{\uparrow}, \pi', \theta) \in \rho\end{array}$$

Thus, as the range of the universal quantification is empty (i.e., *false*), the whole expression is trivially evaluated to *true*. Thereby, the base case holds.

- Inductive case: $w = w' : (e, \theta)$

IH: $\forall\ L, \tilde{L}, \nu, \nu'\ \cdot$

$$\begin{array}{l}C_{init}(pn) \stackrel{w'}{\Longrightarrow}_M (L, \nu)\ and\ C_{init}(ppd2DATE(pn)) \stackrel{w'}{\Longrightarrow}_{M'} (\tilde{L}, \nu') \\ \quad implies\ \psi(L, \tilde{L})\end{array}$$

Given the previous inductive hypothesis *IH*, we have to prove,

$$\begin{array}{l}C_{init}(pn) \stackrel{w':(e,\theta)}{\Longrightarrow}_M (L, \nu)\ and\ C_{init}(ppd2DATE(pn)) \stackrel{w':(e,\theta)}{\Longrightarrow}_{M'} (\tilde{L}, \nu') \\ \quad implies\ \psi(L, \tilde{L})\end{array}$$

By Def. 21 we have,

$$\begin{array}{l}(i)\ \exists\ L'', \nu'' \cdot C_{init}(pn) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ and\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (L, \nu) \\ and \\ (ii)\ \exists\ L'', \nu'' \cdot C_{init}(ppd2DATE(pn)) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ and\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (\tilde{L}, \nu') \\ \quad implies\ \psi(L, \tilde{L})\end{array}$$

Then, we proceed with the proof by assuming the antecedent of the implication. This assumption allows us to remove the existential quantifiers in the antecedents by introducing the fresh values L'' and ν'' in (i), and the fresh values $\tilde{L}''$ and ν''' in (ii). Therefore, we have

$$\begin{array}{l}(i')\ C_{init}(pn) \stackrel{w'}{\Longrightarrow} (L'', \nu'')\ and\ (L'', \nu'') \stackrel{(e,\theta)}{\Longrightarrow} (L, \nu) \\ and \\ (ii')\ C_{init}(ppd2DATE(pn)) \stackrel{w'}{\Longrightarrow} (\tilde{L}'', \nu''')\ and\ (\tilde{L}'', \nu''') \stackrel{(e,\theta)}{\Longrightarrow} (\tilde{L}, \nu')\end{array}$$

Next, by *IH* we know that $\psi(L'', \tilde{L}'')$. Thus, we have

$$(iii)\ \psi(L'', \tilde{L}'')$$

In relation to L, by (i') we know it is obtained from L'' after performing a big step with (e, θ). Thereby, the local configurations on L are either the same as in L'',

a modified version of the ones in L'', or new local configurations added to control a *DATE* which is a new instance of a template.

Let us introduce the sets L_{nc}, L_c and L_{new}, to represent the local configurations in each one of the previous categories, respectively. Then, we know that

$$(iv)\ L = L_{nc} \cup L_c \cup L_{new}$$

In addition, by using a similar approach with $\tilde{L}$ and (ii'), we introduce the following sets.

$$(v)\ \tilde{L} = \tilde{L}_{nc} \cup \tilde{L}_c \cup \tilde{L}_{new}$$

As in the translation the set $\tilde{L}_{new}$ contains both the instances of ordinary templates and the instances of the templates about Hoare triples, we split $\tilde{L}_{new}$ into the sets $\tilde{L}'_{new}$ and $\tilde{L}_h$, to represent each one of the previous categories, respectively. Thus,

$$(v')\ \tilde{L} = \tilde{L}_{nc} \cup \tilde{L}_c \cup \tilde{L}'_{new} \cup \tilde{L}_h$$

Now, let us come back to the expression $\psi(L, \tilde{L})$. By (iv) and (v'), we replace it by

$$\psi(L_{nc} \cup L_c \cup L_{new}, \tilde{L}_{nc} \cup \tilde{L}_c \cup \tilde{L}'_{new} \cup \tilde{L}_h)$$

By (iii), as the values in both L_{nc} and $\tilde{L}_{nc}$ are the same as in L'' and $\tilde{L}''$, respectively, we know that the former fulfil ψ. Thereby, we can reduce the previous expression to

$$\psi(L_c \cup L_{new}, \tilde{L}_c \cup \tilde{L}'_{new} \cup \tilde{L}_h)$$

In addition, newly created local configurations in both L_{new} and $\tilde{L}'_{new}$ do not fulfil the ranges of the quantified expressions in ψ. Then, we can discard them.

$$\psi(L_c, \tilde{L}_c \cup \tilde{L}_h)$$

Next, by the definition of ψ, we have

$$\begin{array}{l}
(vi)\ \forall\ m, q, \rho \cdot (m, q, \rho) \in L_c \cdot \\
\qquad \forall\ \sigma^{\uparrow}_{id}, \pi', \theta \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho \cdot \\
\qquad\quad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c \cup \tilde{L}_h \cdot inst\,(exit_cond_checker, \sigma, \pi') = m' \\
and \\
(vii)\ \forall\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c \cup \tilde{L}_h, m' \notin M' \cdot \\
\qquad \exists\ \sigma^{\uparrow}_{id}, \pi' \cdot inst\,(exit_cond_checker, \sigma, \pi') = m' \\
\qquad\qquad implies\ \exists\ m, q, \rho, \theta \cdot (m, q, \rho) \in L_c \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho
\end{array}$$

In relation to the configurations in $\tilde{L}_c$, as they were obtained from configurations in $\tilde{L}''$, by (iii) we know they fulfil (vii) (same *DATE* component). Thereby, we only need to prove that

$$\begin{array}{l}
(vi)\ \forall\ m, q, \rho \cdot (m, q, \rho) \in L_c \cdot \\
\qquad \forall\ \sigma^{\uparrow}_{id}, \pi', \theta \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho \cdot \\
\qquad\quad \exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_c \cup \tilde{L}_h \cdot inst\,(exit_cond_checker, \sigma, \pi') = m' \\
and \\
(vii')\ \forall\ m', q' \cdot (m', q', \emptyset) \in \tilde{L}_h, m' \notin M' \cdot \\
\qquad \exists\ \sigma^{\uparrow}_{id}, \pi' \cdot inst\,(exit_cond_checker, \sigma, \pi') = m' \\
\qquad\qquad implies\ \exists\ m, q, \rho, \theta \cdot (m, q, \rho) \in L_c \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho
\end{array}$$

Now, let us focus on (vi). If event e is either an exit event, or an entry event which does not require to verify any Hoare triple, then it does not introduce any new values in ρ components of the local configurations in L_c. Thus, by (iii), (vi) is fulfilled in both cases.

If event e is an entry event which requires the check of Hoare triples, then by Lemma 2 and Prop. 1, we know that for every enabled transition which requires the verification of a Hoare triple in pn, a similar transition will be fired in its translation whose action will create a *DATE* in charge of controlling such Hoare triple. Thus, for every new entry in a ρ component in L_c, a new local configuration is added in $\tilde{L}_h$. Thereby, (vi) holds.

Regarding (vii'), if event e is either an exit event, or an entry event which does not require to verify any Hoare triple, then $\tilde{L}_h = \emptyset$. Thus, as the range of universal quantification is empty, (vii') is trivially fulfilled in both cases.

If event e is an entry event which requires the check of Hoare triples, then by the rules $entry_1$ and $entry_3$ in the relation *small step local*, we know that a new tuple is going to be added to the ρ component of the local configuration in L_c which are associated to the *ppDATE*s whose current state possess a Hoare triple that has to be verified. In addition, a local configuration is going to be included in $\tilde{L}_h$ for the *DATE* instantiated to control the corresponding Hoare triple. Thereby, (vii') holds. □

2.13 Proof of Soundness

Theorem 1. Given a *ppDATE* network $pn = (M, V, \nu_0, T_{ppd})$, and its translation $ppd2DATE(pn) = (M', V, \nu_0, T'_d)$,

$$\mathcal{VT}(pn) = \mathcal{VT}(ppd2DATE(pn))$$

Proof. To prove this theorem we will show that,

$$\forall\ w \cdot w \in (systemevent \times \Theta_{Sys})^* \cdot\ w \in \mathcal{VT}(pn)\ \textit{iff}\ w \in \mathcal{VT}(ppd2DATE(pn))$$

In the following, we abbreviate $ppd2DATE(pn)$ by dn.

- $w \in \mathcal{VT}(pn)$ *implies* $w \in \mathcal{VT}(dn)$

As $w \in \mathcal{VT}(pn)$, by Def. 22 we know that it has a prefix w' such that either,

(i) $C_{init}(pn) \overset{w'}{\Longrightarrow}_M (L', \nu')$ and $\exists\ (m, q, \rho) \cdot (m, q, \rho) \in L' \cdot q \in B_m$, or

(ii) $w' = w_1 +\!\!+ \langle(\sigma^{\uparrow}_{id}, \theta')\rangle$, $C_{init}(pn) \overset{w_1}{\Longrightarrow} (L', \nu')$ and $\exists\ m, q, \rho, \pi', \theta \cdot ((m, q, \rho) \in L' and\ (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho) \cdot \theta, \theta' \not\models \pi'$.

In relation to (i), let us assume that exists $(\tilde{L}, \nu)$ such that $C_{init}(dn) \overset{w'}{\Longrightarrow}_{M'} (\tilde{L}, \nu)$. Then, by Lemma 2 we know that for every local configuration in L', there is a local configuration in $\tilde{L}$ such that its state component is the same. Therefore, as in L' there is a local configuration in a bad state, there is a local configuration in $\tilde{L}$ in a bad state, i.e. w' is a counter-example of dn. Thereby, $w \in \mathcal{VT}(dn)$.

Regarding (ii), it corresponds to the case where (at least) one Hoare triple is not fulfilled when event $\sigma^{\uparrow}_{id}$ occurs. Here, by Lemma 3 we have

$$\psi(L', \tilde{L})$$

Therefore, by (ii) and $\psi(L', \tilde{L})$ we know that

$$\exists\ m', q' \cdot (m', q', \emptyset) \in \tilde{L} \cdot inst(exit\_cond\_checker, \sigma, \texttt{part\_eval}(\pi')) = m'$$

Let us assume that the local configuration $(m', q', \emptyset)$ is the one satisfying the previous existential quantification. In addition, let us assume $(\tilde{L}, \nu)$ to be given by $C_{init}(dn) \overset{w_1}{\Longrightarrow}_{M'} (\tilde{L}, \nu)$. Then, once $\sigma^{\uparrow}_{id}$ occurs, as by (ii) we know that the π' is not fulfilled, m' will shift to a bad state. Thereby, w' is a counter-example of dn, i.e. $w \in \mathcal{VT}(dn)$.

- $w \in \mathcal{VT}(dn)$ *implies* $w \in \mathcal{VT}(pn)$.

As $w \in \mathcal{VT}(dn)$, by Def. 22 and the fact that every *DATE* in dn has no Hoare triples associated to its states, we know that it has a prefix w' such that,

$$C_{init}(dn) \overset{w'}{\Longrightarrow}_{M'} (\tilde{L}, \nu) \text{ and } \exists\ (m, q, \rho) \cdot (m, q, \rho) \in \tilde{L} \cdot q \in B_m$$

Now let us assume that exists (L', ν') such that $C_{init}(pn) \overset{w'}{\Longrightarrow}_{M} (L', \nu')$. In addition, let us assume that the bad state in $\tilde{L}$ belongs to a local configuration associated to a *DATE* m', which is an instance of the template *exit_cond_checker*, i.e., m' was created to control a Hoare triple. Let us represent this Hoare triple as $\{\pi\}\,\sigma\,\{\pi'\}$. Then, by Lemma 3 we know that,

$$(1)\ \exists\ m, q, \rho, \theta \cdot (m, q, \rho) \in L' \cdot (\sigma^{\uparrow}_{id}, \pi', \theta) \in \rho$$

We will assume that the *ppDATE* m and the valuation θ are the ones fulfilling (1). Note that the index id is introduced by Lemma 3. Next, as m' is in a bad state we know that whenever $\sigma^{\uparrow}_{id}$ occurs, π' is not fulfilled. Thus, let us assume that the selected prefix is of the form $w' = w_1 +\!\!+ \langle(\sigma^{\uparrow}_{id}, \theta')\rangle$. Thereby, by Def. 22, w' is a counter-example of pn, i.e. $w \in \mathcal{VT}(pn)$.

On the other hand, if the bad state in $\tilde{L}$ does not belongs to a local configuration associated to a *DATE* m' which is an instance of the template *exit_cond_checker*, then by Lemma 2 we know that there is a local configuration in L' such that its state component is the same as the bad state in $\tilde{L}$. Therefore, w' is a counter-example of pn, i.e. $w \in \mathcal{VT}(pn)$. □

CHAPTER
THREE

STARVOORS USER MANUAL (RELEASE 1.7)

J. M. Chimento

3.1 Introduction

Day by day the use of formal verification techniques to verify the correctness of programs is increasing. In general, verification tools use either static verification techniques (i.e., the verification is performed prior to program execution), or dynamic verification techniques (i.e., the verification is performed during program execution), in order to verify whether a program fulfils certain properties.

Nowadays, a new trend focused on the combination of static and dynamic verification techniques is starting to emerge. StaRVOOrS (STAtic and Runtime Verification of Object-ORiented Software) is a tool which aims at both the specification and verification of properties by combining the use of *Static Verification* and *Runtime Verification*. On the whole, StaRVOOrS is fed with a Java program and a *ppDATE* specification [10] describing properties which the program under scrutiny must fulfil, and it automatically generates a runtime monitor which will verify the specified properties (at runtime) whenever the provided program is executed.

This document is the user manual of StaRVOOrS. Its structure is as follows. Section 3.2 provides an intuitive description of the *ppDATE* specification language used by this tool. Section 3.3 gives a high level explanation about how this tool works. Section 3.4 shows how to write a *ppDATE* specification in the input language of the tool. Finally, section 3.5 provides a complete example on how to run this tool.

3.2 *ppDATE* Specification Language

Here, we briefly introduce the *ppDATE* specification language. However, both its complete (formal) description and its semantics can be found in [11].

ppDATE is an automaton-based formalism which, basically, consists of a labelled transition system whose states may include Hoare triples describing properties about the methods of the system under scrutiny.

Transitions in a *ppDATE* are labelled by a trigger (tr), a condition (c) and an action (a). Together, the label is written $tr \mid c \mapsto a$. A transition is *enabled* to be taken whenever its trigger is active and the condition guarding it holds. In addition, if a transition is taken, we say that it is *fired*. Whenever a transition is fired, its action is executed.

Regarding the triggers, they are activated by the occurrence of either a visible *system event* such as entering or exiting a method, or an *action event* generated by certain actions labelling other transitions. We use the notation $\texttt{foo}^{\downarrow}$, $\texttt{foo}^{\uparrow}$, $e?$, to represent the trigger which is activated whenever the method `foo` is entered, the trigger which is activated whenever the method `foo` is exited, and the trigger which is activated whenever the action event e occurs, respectively.

Regarding the conditions, they are expressions written using JML boolean expression syntax [72]. Conditions may depend on the values of *system variables* (i.e., of the system under scrutiny) and the values of *ppDATE variables* (i.e., variables which belong to the *ppDATE*). The latter can be modified via actions in the transitions.

Regarding the actions, they consist on any number of the following: (i) assignments of the form $v = exp$, where v is a *ppDATE* variable and *exp* is an expression that may depend on system variables and *ppDATE* variables; (ii) an action *!* such that *e!* represents the generation of the action event e; (iii) an action `\create`, used to generate instances of a *ppDATE* template (see Sec. 3.4.3); (iv) IF-THEN conditional expressions whose branching condition depends on the valuations of system variables and *ppDATE* variables; (v) an action `\log` such that `\log`(*string*) adds *string* into the log file generated by the monitor; (vi) and (Java) programs. All the actions should end in a semicolon.

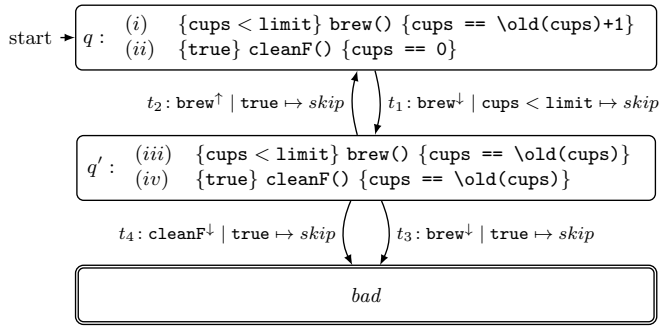


Figure 3.1: A *ppDATE* controlling the brew of coffee

In relation to the Hoare triples on the states of a *ppDATE*, intuitively, if a Hoare triple $\{\pi\}\, foo()\, \{\pi'\}$ is included in some state q, this property ensures that: if method `foo` is entered while the monitor is in state q, and pre-condition π holds, then upon reaching the corresponding exit from `foo`, post-condition π' should hold. Both pre-/post-conditions in the Hoare triples are expressed using *JML Boolean Expressions* syntax (see Sec. 3.4.5 for details about this syntax).

Now, let us introduce an example in order to give a better intuition on how a *ppDATE* is described.

ppDATE Specification Example

Let us consider a *coffee machine system* where, after a certain amount of coffee cups are brewed, its filters have to be cleaned. If the limit of coffee cups is reached, the machine should not be able to brew any more coffee. In addition, while the coffee machine is active (a coffee cup is being brewed), it is not possible to start brewing another coffee, or to clean the filters.

Fig. 3.1 illustrates a *ppDATE* describing this part of the system. In other words, whenever the coffee machine is not active, i.e., the machine is not brewing a cup of coffee, and the method `brew` starts the coffee brewing process, then it is not possible either to execute this method again, or to execute the method `cleanF` (which initialises the task of cleaning the filter), until the initialised brewing process finishes.

The previous property can be interpreted as follows: initially being in state q, the state which represents that the coffee machine is not active, whenever method `brew` is invoked and it is possible to brew a cup of coffee (i.e., the limit of coffee cups was not reached yet), then transition t_1 shifts the *ppDATE* from state q to state *q'*. While in *q'*, the state which represents that the coffee machine is active, if either method `brew` or method `cleanF` are invoked, then transitions t_3 or transition t_4 shift the *ppDATE* to state *bad*, respectively. This indicates that the property was violated. On the contrary, if method `brew` terminates its execution, then transition t_2 shifts the *ppDATE* from state *q'* to state q. Note that the names used on the transitions, e.g. t_1, t_2, etc., are not part of the specification language. They are included to simplify the description of how the *ppDATE* works.

In addition to this, the Hoare triples in state q ensure the properties: (i) if the amount of brewed coffee cups has not reached its limit yet, then a coffee cup is brewed; (ii) cleaning the filters sets the amount of brewed coffee cups to 0. Property (i) has to be verified if, while the *ppDATE* is on state q, the method `brew` is executed and its precondition holds; and property (ii) has to be verified if, while the *ppDATE* is on state q, the method `cleanF` is executed and its precondition holds. Regarding state

q', the Hoare triples in this state ensure the properties: (iii) no coffee cups are brewed; (iv) filters are not cleaned. Property (iii) and (iv) are verified if either method `brew` and method `cleanF` are executed, and their preconditions hold, respectively. Here, remember that this state represents that the coffee machine is active. Thus, if it occurs that either the method `brew` or the method `cleanF` are executed while the *ppDATE* is on this state, then, as this would move the *ppDATE* to state `bad`, one would expect the value of the variable `cup` to remain unchanged. This is precisely what is verified when either property (iii) or (iv) are analysed.

Note that none of the Hoare triples makes reference to the state of the coffee machine, i.e., there is no information about whether the machine is active or not. This is due to fact that the state of the machine is implicitly defined by the states of the *ppDATE*. If the *ppDATE* is in state *q*, the coffee machine is not active. However, if it is in state *q'*, then the machine is active. Therefore, the Hoare triples are *context dependent*. This is the reason why we can describe properties with the same precondition, but with different postconditions depending on the state of the *ppDATE* in which they are placed.

3.3 High-level Description of StaRVOOrS

StaRVOOrS takes three arguments: (i) the path to the main folder of the Java files to be verified; (ii) a description (as input language) of the *ppDATE* specification for the provided program; and (iii) the path of the output folder (the generated files are stored in this folder). Then, it automatically generates (1) a runtime monitor; (2) an instrumented version of the Java files in (i); (3) a report summarising the results obtained by statically verifying the Hoare triples described in (ii); (4) and a refined version of (ii), when possible.

To generate such output, StaRVOOrS combines the use of the deductive source code verifier KeY [9] with the runtime monitoring tool Larva [46]. KeY is a deductive verification system for data-centric *functional correctness* properties of Java programs, which generates, from JML [72] and Java, proof obligations in *Dynamic Logic* (a modal logic for reasoning about programs) [62], and attempts to prove them by using a *sequent calculus* which follows the *symbolic execution* paradigm. Larva is an automata-based Runtime Verification tool for Java programs which automatically generates a runtime monitor from a property using the automaton-based specification language *DATE* [45]. Larva transforms such specification into monitoring code together with AspectJ code to link the system under scrutiny with the generated monitor.

In a nutshell, StaRVOOrS output is generated by following the steps enumerated below.

(a) The Hoare triples described in (ii) are translated into JML contracts, which are textually added to to the Java files in (i) as annotations of the respective methods;

(b) KeY attempts to (statically) verify all the JML contracts automatically. The result obtained for each contract is either a complete proof, or a partial proof where some parts of the contract are proved and others are not, or that KeY cannot prove any of the parts the contract. These results are stored in a XML file. In addition, a report summarising the content of this file, i.e., (3), is generated. Here, note that our tool does not support user interaction with KeY. It uses this prover in fully automatic mode;

(c) The *ppDATE* specification is refined based on the XML file, i.e., (4). Fully verified Hoare triples are removed from the specification, but those Hoare triples which are not fully verified, are left in the specification to be verified at runtime. However,

the original pre-conditions of the remaining Hoare triples may be strengthen with the (path) conditions resulting from partial proofs, thus covering at runtime only executions that are not closed in the static verification step;

(d) The refined *ppDATE* specification is encoded into a *DATE* specification. In particular, the *DATE* specification language does not support pre/post-conditions which thus have to be translated to use notions native to this specification language. This also requires a number of changes to the system (through code instrumentation), in order to be able to distinguish different executions of the same code unit, and to evaluate the Hoare triples in the states of the refined *ppDATE* at runtime. i.e, (2). Regarding the former, method declarations get a new argument which is used as a counter for invocations of this method. Regarding the latter, not every condition in a pre/postcondition of a Hoare triple can be directly written as a Java Boolean Expression, e.g., quantified expressions. Thus, methods which operationalise the evaluation of those conditions are added to the Java files in (i);

(e) The LARVA compiler generates a runtime monitor using aspect-oriented programming techniques, i.e., (1).

Once deployed, the runtime monitor and the instrumented version of the Java files are executed together, thus effectively running the monitor in parallel with the program. The runtime monitor identifies violations at runtime, reporting error traces to be analysed.

3.4 Composing a *ppDATE* Specification in the Input Language of StaRVOOrS

In this section we explain in detail how to write a *ppDATE* specification using the input language of StaRVOOrS. The files written in such language have extension *.ppd*, and their content may consist on 6 sections which are ordered as follows: `IMPORTS`, `GLOBAL`, `TEMPLATES`, `CINVARIANTS`, `HTRIPLES` and `METHODS`. Below, we describe the content of each one of these sections, show their syntax, and provide examples illustrating how to write them.

3.4.1 IMPORTS

Section `IMPORTS` lists the packages included in the system under scrutiny which are related to the properties to be verified (both the Hoare triples and the automata). Its syntax is described as follows:

```
IMPORTS { import package ; }
```

Each package listed in this section follows the usual Java syntax for imports. For instance,

```
IMPORTS {
  import main.Foo ;
  import other.sub.Goo ;
  import other.Hoo ;
}
```

3.4.2 GLOBAL

Section `GLOBAL` contains the description of the *ppDATE* specification. Its syntax, which is described below, is written as follows:

```
GLOBAL {
 VARIABLES { -- definition of the variables -- }

 ACTEVENTS  { -- definition of the action events -- }

 TRIGGERS { -- definition of the triggers -- }

 PROPERTY property_name1 {
  STATES {  -- definition of the states of the ppDATE -- }
  TRANSITIONS {  -- definition of the transitions of the ppDATE -- }
 }

 PROPERTY property_name2 {
   -- definition of states and transitions --
 }
 ...
}
```

Note that one may describe more than one `PROPERTY`. This would be the case when one is describing several *ppDATE*s in one single specification file, i.e., each property represents a *ppDATE*.

3.4.2.1 VARIABLES

Subsection `VARIABLES` allows to include as part of the specification the declaration of variables. These variables, which are referred to as *ppDATE* variables, may be freely used in the transitions of a *ppDATE*, both in their conditions and actions. For instance, one may use an integer variable as a counter to keep track of how many times a method is executed. Below, we illustrate how variables may be defined within this subsection.

```
VARIABLES {
  type var ;
  type var = initial_value ;
}
```

Such syntax follows the usual Java syntax for the declaration of variables. For instance,

```
VARIABLES {
  String s;
  int i = 0;
}
```

Note that whenever a variable is not initialised when it is defined, its initialisation has to be performed by the execution of an action. Otherwise, there is going to be an exception at runtime whenever the monitor attempts to manipulate such variable.

```
public class Foo {
   public void foo();
}
```

```
public class Goo {
   public void goo(int x,boolean b);
   public int hoo(int x, int y, int z);
}
```

Figure 3.2: Example of Java classes.

3.4.2.2 ACTEVENTS

Subsection `ACTEVENTS` includes the declaration of the different *action events* which may be generated by using the action *!*. Here, it is only necessary to list the names of these events, as illustrated in the example below for the action events `e1`, `e2`, and `e3`.

```
ACTEVENTS {
  e1 ; e2 ; e3 ;
}
```

3.4.2.3 TRIGGERS

Subsection `TRIGGERS` includes the declaration of the different triggers which may be used in the transitions of a *ppDATE*.

Triggers Associated to System Events

The triggers which are activated by the occurrence of a visible *system event*, i.e., entering or exiting a method, have the following signature:

```
name(args) = {varDecl.method(args')sysevent }
```

Here, `name` is a label which works as an identifier for the trigger; `method` is the name of the method generating the system event which activates the trigger; and `sysevent` represents whether the trigger is activated by a system event produced by entering or exiting a method, represented with the notation `entry` or `exit`, respectively.

In addition, each trigger may have a number of arguments `args` which act as binds for `args'` (i.e., `args'` are the arguments in `args`, but without their types). This allows the access at runtime to the arguments which are being provided to the method, and to the value returned by a method (see examples below).

Regarding `varDecl`, it is a variable declaration which has the following signature:

```
varDecl = * | identifier | type identifier
```

The symbol `*` means that the triggers can be activated by an event associated to `method`, no matter what class it belongs to; `identifier` is the target object (instance of the class `type`) on which `method` is being called. Note that `identifier` should be always associated to a class. Thus, if one uses only `identifier` as variable declaration, then `args` should include an argument of the form `type identifier`. In addition, one may use `identifier` to access at runtime the target object.

Below, by considering the Java classes depicted in Fig. 3.2, we give several examples illustrating the different manners in which triggers might be defined.

```
TRIGGERS {
  foo1()                = {Foo f.foo()entry}
  foo2()                = {*.foo()entry}
```

```
  goo1(int x, boolean b) = {Goo g.goo(x,b)entry}
  goo2(int x)            = {Goo g.goo(x,*)entry}
  foo3()                 = {*.foo()exit()}
  goo4(int x,boolean b)  = {Goo g.goo(x,b)exit()}
  hoo(int x, int ret)    = {Goo g.hoo(x,*)exit(int ret)}
}
```

On these definitions, whenever either the target object of the method or any of the arguments of the method are not necessary for the definition of a trigger, e.g., the definition of a trigger which is activated by the execution of a method `foo` where several classes have an implementation for this method, they can simply be omitted by replacing them with the symbol '$*$', which is used as a place holder. Triggers `foo2`, `goo2`, `foo3`, and `hoo` are examples illustrating these situations. In addition, in the definition of a trigger which is activated by a system event produced by exiting a method, it is possible to refer to (and later to access) the value (or object) returned by the method, by including in the arguments of the trigger an argument with an appropriate type to represent such value, and then including this argument in the notation `exit`, as it is illustrated in the definition of trigger `hoo`.

Triggers Associated to Constructors

It is possible to define a special exit trigger which is activated when an object of a certain class is created. These triggers have the following signature:

```
name(type obj) = {type.new()exit(obj)}
```

where `obj` represents the created object for the class `type`. If the constructor has arguments, then they have to be included as part of the arguments of the trigger and the call to `new`. For instance, let us assume that the following signature for the constructor of a class `Foo`: `Foo(int n, boolean b)`. Then, one should define this kind of triggers as follows:

```
foo_new(int n, boolean b, Foo obj) = {type.new(n,b)exit(obj)}
```

3.4.2.4 PROPERTY

The core of this section is the subsection `PROPERTY`. It consists of the actual description of a *ppDATE*. This subsection is divided in two parts: `STATES` and `TRANSITIONS`. Note that there should be defined at least one property here.

STATES

Section `STATES` lists all of the states in a *ppDATE*. There are four kind of states: starting states, accepting states, bad states, and normal states. `STARTING` list the initial state of the *ppDATE*. There should be only one starting state listed. The accepting states, which are listed in `ACCEPTING`, represent the states in which it is desirable for the monitor to be in whenever the program under scrutiny terminates its execution. The bad states, which are listed in `BAD`, represent states which a monitor reaches whenever a property which is described with the transitions of the *ppDATE* is violated at runtime. Finally, normal states, which are listed in `NORMAL`, are neither accepting nor bad states, but simply possible states where a monitor may be in during the execution of a program.

In relation to the list of states, each entry consists on the name of a state, and a list of the names of the Hoare triples which have to be verified in that state (this is properly explained in Sec. 3.4.5). Entries in a list of states terminate in a semicolon.

Below you can see an example of a `STATES` subsection.

```
STATES {
 STARTING { q0 (h1) ; }
 ACCEPTING { q4 ; q5; }
 BAD { bad ; }
 NORMAL { q1 ; q2 (h2,h3) ; q3; }
}
```

Note that the order previously illustrated in the syntax (starting, accepting, bad, normal) should be preserved. In addition, it is mandatory to always include a starting state on a *ppDATE*. Otherwise, the monitor will not know from which state it should start. Finally, it is important to remark that the accepting states are considered *sink states*, i.e. they should not have outgoing transitions. Therefore, once a *ppDATE* reaches one of such states, it is deactivated (i.e. it stops running).

TRANSITIONS

Section `TRANSITIONS` contains the description of all the transitions in the *ppDATE*. For each *ppDATE* transition going from state `q` to state `q'` with trigger `tr`, (optional) condition `c`, and (optional) action `a`, this section includes a line of the form

```
q -> q' [tr \ c \ a]
```

Here, where `tr` should be either the name of a trigger defined in a `TRIGGERS` subsection (see Sec. 3.4.2.3), or an expression of the form `e?`, where `e` is an action event defined in the `ACTEVENTS` section (see Sec. 3.4.2.2); `c` is a boolean expression following JML syntax [72], which may depend on both system and *ppDATE* variables; and `a` is an action consisting on sequence of the following:

- assignments of the form $v = exp$, where v is a *ppDATE* variable and *exp* is an expression that may depend on system variables and *ppDATE* variables;
- an action `\gen(e)` which generates the action event `e` that activates the trigger `e?`;
- an action `\create`, used to generate instances of *ppDATE* templates (see Sec. 3.4.3);
- IF-THEN conditional expressions whose branching condition depends on the valuations of system variables and *ppDATE* variables;
- an action `\log` such that, `\log`(*string*) adds *string* into the log file generated by the monitor;
- a block of actions delimited by curly brackets;
- actions $++v$, $--v$, $++v$, $v--$, $v+=e$, and $v-=e$, with their standard (Java) meaning;
- (Java) programs.

All the actions should terminate in a semicolon. Regarding action `\gen(e)`, both in the theory ([10]) and in Sec. 3.2, this action is represented with the symbol !, i.e., `e!` generates the action event `e`. The main reason why we decided not to use the same notation in our input language as the one used in the theory is that both JML and Java use the symbol ! as the boolean negation. Thus, we consider that by introducing action `\gen` instead, we are avoiding the possible confusion which may arise in relation to whether `v!` refers to the negation of the boolean variable `v`, or the generation of the action event `v`. In addition, the trigger `e?` in the fourth transition represents the trigger which is activated whenever the action event `e` occurs.

login-logout = λ `u : User, tr : Trigger`.

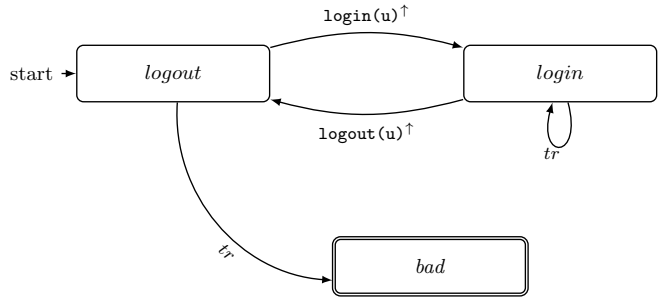


Figure 3.3: *ppDATE* template describing properties about the log in and log out of users.

Regarding the use of (Java) programs in the actions, the tool only supports the use of method calls, e.g., `inc(v)` in the first transition. However, it is possible to use the section METHODS (see Sec. 3.4.6) to define programs as (Java) methods. Then, one simply has to make a method call to them.

Below we list all the syntactically valid expressions which may be used within this section.

(1) `q -> q' [tr\c\a]`

(2) `q -> q' [tr\\]`

(3) `q -> q' [tr\]`

(4) `q -> q' [tr]`

(5) `q -> q' [tr\c\]`

(6) `q -> q' [tr\c ]`

(7) `q -> q' [tr\\a]`

Note that the expressions (2), (3), and (4) are equivalent. Similarly, expressions (5) and (6) are equivalent as well. In addition, when using a trigger in a transition it is not necessary to write their arguments in the trigger component of the transition. This does not affect the possibility of using such arguments in both the conditions, and the actions. For instance, given the trigger `t(int x) = {Goo g.goo(x)entry}`, one may define the transition `q0 -> q1 [t\x == 8]`, where `x` is the argument of the trigger `t`.

Now, let us illustrate some of the previous expressions with the following example:

```
TRANSITIONS {
  q0 -> q1 [tn \ c == 8 \ v = 0; \gen(e);]
  q0 -> q2 [f \ c == 2 \ ]
  q1 -> q3 [g \ \ IF (b) THEN inc(v); foo();]
  q2 -> q2 [ae?]
  q3 -> q2 [g \ true \ \log("info");]
}
```

3.4.3 TEMPLATES

In addition to *ppDATE*s which exist up-front, and 'run' from the beginning of a program's execution, new *ppDATE*s can be created by existing ones. For instance, one

may want to create a separate 'observer' for each new user logging into a system. For that, one needs to be able to define parameterised *ppDATE*s, which we call *templates*, and allow *ppDATE*s to create new instantiations of them. Fig. 3.3 illustrates an example of a *ppDATE* template called *login-logout* which, given a user `u`, describes the property "the user has to log in to perform a deposit".

Section `TEMPLATES` lists tagged *ppDATE* templates. Below, we show the syntax of this section.

```
TEMPLATES {
 TEMPLATE id_template (params) {
    VARIABLES { -- definition of the variables -- }
    TRIGGERS  { -- definition of the triggers -- }
    PROPERTY name { -- definition of the property -- }
 }
}
```

Each template is described within a subsection `TEMPLATE`, whose header is followed by a (unique) name `id_template` assigned to the template, and a list of parameters `params` used to generalise the definition of the templates. Each element in `params` has the form `Type var`, where `Type` is either a reference type (i.e. Java class), or one of the following special types: `Trigger`, `Condition`, `Action`, `HTriple`, and `MethodName`. These special types can be used to abstract triggers, conditions, actions, Hoare triples, and method names, respectively, in a template. Regarding `var`, it represents the abstraction of a value (of the corresponding type).

Note that as a template describes a *ppDATE*, the subsections `VARIABLES`, `TRIGGERS`, and `PROPERTY` are defined just like it is described in Sec. 3.4.2. In addition, the triggers defined in a template may have the same name as the triggers defined in a (non-template) *ppDATE*. Whenever this happens, the template will always refer to its own definition of the trigger. Below, we illustrate how the *ppDATE* template in Fig. 3.3 could be written using this syntax.

```
TEMPLATES {
 TEMPLATE login-logout (User u,Trigger tr) {
  TRIGGERS  {
    login_ex(String username, int pwd) = {u.login(username, pwd)exit()}
    logout_ex() = {u.logout()exit()}
  }
  PROPERTY deposit {
    STATES {
       STARTING { logout ; }
       ACCEPTING { login ; }
       BAD { bad ; }
    }
    TRANSITIONS {
       logout -> login [login_ex]
       logout -> bad [tr]
       login  -> logout [logout_ex]
       login  -> login [tr]
    }
  }
 }
}
```

Regarding the instantiation of a template, it is accomplished by using the action `create` on the transition of a *ppDATE*. This action receives as arguments the name

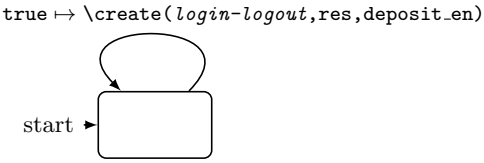


Figure 3.4: *ppDATE* in charge of creating instances of the template *login-logout*.

of the *ppDATE* template to be instantiated and a list of values to instantiate the parameterised arguments of the template, and it generates the instance of the template. For example, Fig.3.4 illustrates a *ppDATE* which creates an instance of the template *login-logout* (Fig. 3.3) upon declaration of an object of class `User`. Here, `res` represents the (concrete) object of class `User` which was created. In addition, the trigger `User.new`$^{\uparrow}$ is activated when such a creation occurs.

3.4.4 CINVARIANTS

Section `CINVARIANTS` lists the definitions of class invariants which may need to be considered during the verification of the properties. Its syntax is described as follows:

```
CINVARIANTS {
 class { invariant }
}
```

Here, `class` represents a Java class in the program under scrutiny whose implementation has to preserve the invariant definition described by `invariant`. Such invariants follow JML-like syntax and pragmatics. Below we illustrate an example of this section.

```
CINVARIANTS {
 Foo { v <= 10 }
 Foo { count >= 0 }
}
```

Note that if no class invariants are needed on a specification, then this section may be omitted. In addition, the actual version of the tool only uses the class invariants during the static verification of the Hoare triples. However, we are currently working to include the verification of class invariants at runtime as well.

3.4.5 HTRIPLES

Section *HTRIPLES* lists tagged Hoare triples. Its syntax is described as follows:

```
HTRIPLES {
  HT hoare_triple_name {
    PRE { -- precondition -- }
    METHOD { -- method to verify -- }
    POST { -- postcondition -- }
    ASSIGNABLE { -- variables modified -- }
  }
}
```

Each Hoare triple is described within a subsection `HT`, whose header is followed by the name assigned to the Hoare triple. This name is unique for each Hoare triple, and it is used to associate the Hoare triples with the states of a *ppDATE*. Subsection `HT` is composed by four parts: `PRE`, which describes the pre-condition of the Hoare triple; `POST`, which describes the post-condition of the Hoare triple; `METHOD`, which describes which is the method that has to fulfil the Hoare triple; and `ASSIGNABLE`, which lists the variables that might be modified when the method under scrutiny on the Hoare triple is executed. Here, `PRE`, `POST`, and `ASSIGNABLE` follow JML-like syntax and pragmatics.

Regarding `METHOD`, it is an expression of the form `file.method(types)`, where `method` is the name of the method related to the Hoare triple, `file` is the Java class where the previous method is implemented, and `types` is a list of arguments that can be used to differentiate methods with the same name. Note that the use of `types` is optional. Regarding `PRE`, `POST`, and `ASSIGNABLE`, these parts may be omitted. If so, they will have as default values `true`, `true`, and `\everything`, respectively.

Below, we provide an example illustrating the use of this section.

```
HTRIPLES {
  HT inc_ok {
    PRE { v }
    METHOD { Foo.inc }
    POST { count == \old(count)+1 }
    ASSIGNABLE { count }
  }
  HT inc_err {
    PRE { !v }
    METHOD { Foo.inc }
    POST { count == \old(count) }
    ASSIGNABLE { \nothing }
  }
  HT add1_ok {
    PRE { true }
    METHOD { Goo.add(int) }
    POST { x == \old(x) + n }
    ASSIGNABLE { \everything }
  }
  HT add2_ok {
    METHOD { Goo.add(int,int) }
    POST { x == n + m }
  }
}
```

Note that this section may contain several subsections `HT`, one per each Hoare triple. In addition, if no Hoare triples are included as part of a *ppDATE*, then this section may be omitted.

3.4.6 METHODS

Section *METHODS* is an optional section which allows to include method declarations as part of a specification. These methods will be included as part of the implementation of the monitor generated by the tool. Its syntax is described as follows:

```
METHODS {
  type method(arguments) { -- method implementation -- }
}
```

Methods are declared following standard Java notation. However, access modifiers (i.e., *public*, *protected*, *private*) are not necessary when declaring these methods. If a method is declared as *static* method, then monitor variables will not be accessible within that particular method. Below, we illustrate an example of this section.

```
METHODS {
  boolean compare (int x, int y) { return (x == y); }
  int four() { return 4 ; }
}
```

3.4.7 Remarks

3.4.7.1 Comment Lines

One may include comments in a *ppDATE* specification by writing %% followed by the comment.

3.4.7.2 Key Words

The key words are words appearing in the grammar of *ppDATE*. We strongly recommend not to use these words as part of your implementation. Otherwise, you may run into parsing issues. The key words used in *ppDATE* are the following:

| ACCEPTING | ACTEVENTS | ASSIGNABLE | BAD
| CINVARIANTS | FOREACH | GLOBAL | HT
| HTRIPLES | IMPORTS | METHOD | METHODS
| NORMAL | PINIT | POST | PRE
| PROPERTY | STARTING | STATES | TEMPLATE
| TEMPLATES | TRANSITIONS | TRIGGERS | VARIABLES
| call | entry | execution | exit | final | import
| uponHandling | uponThrowing | where

3.4.7.3 Private Variables in the Hoare triples

When writing a Hoare triple it is possible to include private variables both in its precondition, and its postcondition. StaRVOOrS will automatically annotate the source code with appropriate JML clauses (i.e. `spec_public`) preceding the (private) variables definition, at the time of statically verifying the Hoare triples. However, note that if later one of these Hoare triples has to be verified at runtime, then the monitor will require access to the private variables values. In such situations, the user will have to decide how to deal with these variables and modify the generated files accordingly. For instance, one can introduce getter methods in order to give access to the monitor to the values of the private variables, and then update the generated file HoareTriplesPPD.java (under ppArtifacts) to use those methods. See Sec. 3.5.4 for more details about the files generated by the tool.

3.4.8 Extra Features

This section describes some extra features added to the tool and its input language, which are not covered by the semantics described in [11]. Note that in Sec. 3.4.8.6 we provide arguments regarding how the soundness of the *ppDATE* semantics is preserved when considering these features.

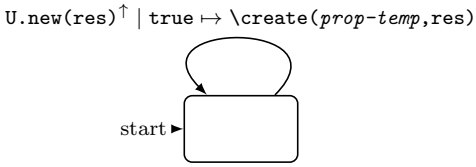


Figure 3.5: *ppDATE* in charge of creating instances of the template *prop-temp*.

3.4.8.1 PINIT Definition in Section PROPERTY

When describing a *ppDATE* specification, it is quite common to have some *ppDATE*s only focus on creating instances of a template upon declaration of an object. Such *ppDATE*s would look like the *ppDATE* illustrated in Fig. 3.5, which creates an instance of the template `prop-temp` every time an object of the class `U` is created. Here, `\result` represents the (concrete) object of class `U` which was created. In addition, the trigger `U.new`$^{\uparrow}$ is activated when such a creation occurs.

Therefore, we decided to include a special subsection `PINIT` as part of the section `PROPERTY`, which can be used to specify these kinds of *ppDATE*s in a simple manner. The syntax of this subsection is as follows:

```
PROPERTY property_name {
   PINIT { (template_name,Class) }
}
```

where `property_name` is the name of the property (as described in Sec.3.4.2.4), `template_name` is the name of a *ppDATE* template defined in section TEMPLATE, and `Class` is the name of the class associated to the declared object. Below, we illustrate how the *ppDATE* from Fig.3.5 is described using this special subsection.

```
PROPERTY example {
   PINIT { (prop-temp,U) }
}
```

Alike Fig. 3.5, property `example` describes a *ppDATE* which has a single state with only a loop transition which is fired every time an object of the class `U` is created, leading to an instantiation of the template `prop-temp` using the created object as argument.

3.4.8.2 Where clause

When declaring a trigger, any of its arguments can be bound to a variable which is not directly related to the method arguments. For instance, let us assume that we have to perform some processing on a particular value, and that we want that, depending on its result, a *ppDATE* fires a transition (or not). Then, by using a where clause right after a trigger definition one can use one argument of the trigger as a bound for that particular value. Consider the next example:

```
TRIGGERS {
 goo(int x,boolean y) = {Goo g.foo(x)entry} where {y = g.IsValid();}
}
```

Here, we do not have any interest in the whole object `g`, but we simply need to know if its a valid object or not, fact which can be computed using the method `IsValid()`,

in order to send the *ppDATE* to either the state `q2`, or `bad`, respectively. Then, one can use the boolean argument `y` of the trigger for binding the result of that method. This would allow us to write transitions like the following ones:

```
TRANSITIONS {
  q1 -> q2  [goo\ y]
  q1 -> bad [goo\ !y]
}
```

Here, remember that it is not necessary to write the arguments of a trigger in the trigger component of a transition, but one can refer to them in both the conditions and the actions.

Furthermore, any variable which is not directly bound to the method arguments is initialized in the where clause. This is done by checking that there is at least one assignment statement with the unbound variable on the left-hand side.

Note that the statements in the where clause can be any valid JAVA statements and these can call any relevant method from imported packages, and that the use of curly brackets in this clause is compulsory.

3.4.8.3 Foreach construct

The `FOREACH` construct can be used as a simplistic alternative to the use of *ppDATE* templates. Consider the following *ppDATE*:

```
GLOBAL {
  TRIGGERS {
    log(User user) = {Interface f.login(User user)entry}
    out(User user) = {Interface f.logout(User user)entry}
  }
  PROPERTY example {
    STATES {
      ACCEPTING { logout ; }
      BAD { bad ; }
      STARTING { login ; }
    }
    TRANSITIONS {
       logout -> login [log\\ \create(deposit-temp,user)]
       logout -> bad [out]
       login  -> logout [out]
       login  -> bad [log]
    }
  }
}

TEMPLATES {
 TEMPLATE deposit-temp (User u) {
  TRIGGERS {
    dep(int amount) = {u.deposit(amount)entry}
   }
   PROPERTY deposit { --- }
 }
}
```

On this *ppDATE*, every time a user logs in the interface, an instance of the template `deposit-temp` is created in order to runtime verify the property `deposit` for that user.

Now, let us introduce a similar *ppDATE* to the one described above, but written using the foreach construct:

```
GLOBAL {
  TRIGGERS {
    log(User user) = {Interface f.login(User user)entry}
    out(User user) = {Interface f.logout(User user)entry}
  }
  PROPERTY example {
    STATES {
      ACCEPTING { logout ; }
      BAD { bad ; }
      STARTING { login ; }
    }
    TRANSITIONS {
       logout -> login [log]
       logout -> bad [out]
       login  -> logout [out]
       login  -> bad [log]
    }
  }

  FOREACH (User u) {
   TRIGGERS {
    dep(int amount) = {User u1.deposit(amount)entry } where {u = u1;}
   }
   PROPERTY deposit { --- }
  }
}
```

In this version of the *ppDATE*, as soon as an object of the class `User` is created, a *ppDATE* verifying the property `deposit` is generated. Here, remember that this is not what happen the template version of the *ppDATE*, where the *ppDATE* verifying property `deposit` is only created when a user logs in.

Using a foreach construct may seem simpler than using a *ppDATE* template. However, one have to consider the following points when using it:

(i) This construct introduces a context to the *ppDATE*, i.e., the triggers, variables and transitions will now be in a particular context. Hence, each trigger should specify its context so that the *ppDATE* which will be affected will only be the one belonging to that particular context. This is done by using the where clause associated to each trigger. In addition, variables may be affected for the introduction of a context. This happens when different contexts have variables with the same name. By default, variables are match to the innermost context. However, it is possible to indicate which one is the context of the variable by using the following special notation: `::amount`, `::u::amount`. The former notation refers to the variable `amount` in GLOBAL (i.e. top level), whereas the latter refers to the variable `amount` in the context of the foreach (i.e. `Foreach (User u)`).

(ii) *ppDATE*s for verifying the properties within a foreach are always going to be generated upon creation of an object, even if the execution of the program does not require to verify them.

(iii) This construct can only refer to reference types.

(iv) *ppDATE* templates are much more expressive than this construct.

Anyhow, note that we mainly decided to include this construct in our language because:

(I) in the case where one just wants to generalise a specification regarding objects of a particular reference type, this construct may be simpler to use compared to the use of a template;

(II) *DATE* users can start writing *ppDATE* specifications right away, without being limited to learn to use templates first;

(III) it allows to migrate *DATE* specifications to *ppDATE* specifications in a simple manner.

3.4.8.4 Channel Communication

ppDATE offers a simplistic manner for automata communication by using action *!*. However, in certain situations it would be desirable to send information (i.e. an object) from one automaton to the other. Thus, we introduce the use of channels for accomplishing such communications.

A channel will broadcast its messages to all the *ppDATE*s listening to it at the moment of broadcasting. In order to use them, one have to include in the `VARIABLES` section of `GLOBAL` (at top level in the case a foreach construct is used) a declaration of the following form:

```
GLOBAL {
  VARIABLES {
    Channel channelName ;
  }
  ....
}
```

One should use in a transition the action `channelName.send(o)` to send a message (i.e. object `o`) through the channel. In order to receive the massage, the receiver *ppDATE*s should include a trigger similar to the following one:

```
TRIGGERS {
   rec(type obj) = {channelName.receive(obj)entry}
}
```

These triggers are activated by the events generated when an object is send through the channel, i.e. by the action `channelName.send`. Then, in the transition enabled by the occurrence of `rec` one can access to the sent object by referring to `obj` (target object instance of the class `type`).

3.4.8.5 Clocks

Clocks can be used in StaRVOOrS as timers which produce (internal) events once a certain time interval has elapsed. In particular, clocks introduce the possibility of defining real-time properties using *ppDATE*.

In order to use them, one has to, first, declare them in the `VARIABLES` section (see Sec. 3.4.2.1) of the *ppDATE* specification. For instance, one would declare a clock `c` in the following manner: `Clock c = new Clock();`. In addition, one has to define a special trigger to capture the timeout (event) produced by the clock. Below, we illustrate the syntax for these kind of triggers.

```
name() = { clock@time }
```

Here, `name` is a label which works as an identifier for the trigger; `clock` is the name of the declared clock, e.g. in our previous declaration `c` is the name of the clock; `time` is the amount of seconds after which the clock will produce the timeout captured by this trigger, each time the clock is reseted. For simplicity, one can also define the clock as `clock@%time`, meaning that the clock will automatically reset after producing the timeout.

Note that clocks are contextual, i.e. if one declares a clock within either a `FOREACH` (see Sec. 3.4.8.3), or a `TEMPLATE` (see Sec. 3.4.3), a spare clock will be created for each monitored object (or template instance). Moreover, if a clock is declared in the `GLOBAL` section (see Sec. 3.4.2), then only one clock is created, and it will be available in all the inner contexts of the specification.

In addition, clocks are automatically started as soon as their contexts start existing. For instance, whenever the monitor starts working, all of the clocks declared in the `VARIABLES` section, within the context of the `GLOBAL` section, are started.

Clocks implementation

Clocks are implemented as Java objects to simplify their usage. Below, we provide the signature of the different methods which are part of the *Clock* class, and we briefly describe what they do. All these methods can be used in the transitions of the *ppDATE*s, e.g. `c.reset()` can be used to reset clock `c`.

- `void reset()`: as its name indicates, resets the clock, i.e. the clock is restarted;
- `double current()`: returns the number of seconds which have elapsed since the clock was started (or reseted);
- `int compareTo(double seconds)`: compares `seconds` to the current value of the clock, i.e. the value returned by method `current`. If they are the same value, then this method returns zero. If the latter is bigger than the former, then this returns a positive integer. Otherwise, this method returns a negative integer;
- `void off()`: Switches off the clock;
- `void on()`: Turns (back) on the clock;
- `void pause()`: Pauses the clock;
- `void resume()`: Resumes the clock.

Comparison to Timed Automata

Properties described using timed automata can also be described by using a *ppDATE* with clocks. For instance, let us assume we have two clocks, c_1 and c_2, such that c_1 triggers after 42 seconds, and c_2 triggers after 10 seconds. In addition, the trigger of one clock causes the reset of the other one. Fig. 3.6 illustrates this example as a timed automaton. This automaton can also be written as the following *ppDATE* (in the StaRVOOrS input language):

```
GLOBAL {
  VARIABLES {
    Clock c1 = new Clock();
    Clock c2 = new Clock();
  }
  TRIGGERS {
    c1Trigger() = { c1@42 }
    c2Trigger() = { c2@10 }
  }
```

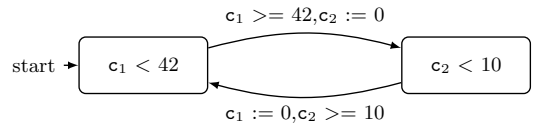


Figure 3.6: A timed automaton example.

```
  PROPERTY example {
    STATES {
       STARTING { start }
       NORMAL { normal }
    }
    TRANSITIONS {
       start -> normal [ c1Trigger \ c1.compareTo(42) >= 0 \ c2.reset();]
       normal -> start [ c2Trigger \ c2.compareTo(10) >= 0 \ c1.reset();]
    }
  }
}
```

Note that this example is an adaptation from an example provided in [43].

3.4.8.6 Towards the Semantics of the Extra Features

As mentioned above, all of the extra features depicted in this section are not covered by the *ppDATE* semantics described in [11]. Here, we do not provide any formal treatment for the semantics of these features. Still, we foresee that their use is safe, i.e., the soundness of the specifications is preserved. Below, we provide some arguments regarding why the soundness of *ppDATE* semantics would not be alter when these extra features are used.

Regarding the `PINIT` definition, as it is simply syntactic sugar to simplify the writing of some particular *ppDATE*s, its semantics would be the same as the one for the *ppDATE*s written in the ordinary manner, i.e., it preserves soundness.

Regarding the `where clause`, it could only break the soundness of a specification if a variable is bound to a particular value returned by a method, and this method throws an exception. However, in [11] programs are assumed to be side-effect free, i.e., they are restricted to not have any effect on the system which could in turn be observed by the monitor. Thus, the use of this clause preserves soundness.

Regarding the `FOREACH` construct, as it is intended to be a simplistic alternative to the use of *ppDATE* templates, their semantics could be described in a similar manner to the one described in [11] for the templates. Thus, we can argue that the use of this construct preserves soundness.

Regarding the use of `channels` and `clocks`, in order to guarantee that their use preserves the soundness of *ppDATE* semantics, the semantics described in [11] should be extended to handle these features properly. In order to do so, we can use as a base the semantics described in [45] for *DATE*. *DATE* provides the use of `channels` and `clocks` as well. In fact, we have implemented these features in the StaRVOOrS input language by following their implementation in *DATE*. Thus, as these features are considered to be sound in *DATE*, by considering similar semantics for them as depicted in [45] for *ppDATE*, we can argue that the use of these features preserve soundness.

3.5 Using StaRVOOrS

In this section we depict how StaRVOOrS works by running the tool on the coffee machine example introduced in Sec. 3.2. Both the *ppDATE* specification written in the input language of the tool, and a simplistic implementation of the coffee machine system, together with two big case studies based on Mondex [2] and SoftSlate (a real Java cart application) [3], can be found in [4], under the section *Downloads*.

3.5.1 Coffee Machine Specification and Implementation

This section illustrates both the *ppDATE* specification written in the input language of StaRVOOrS, and a (simplistic) implementation for the coffee machine system, which were informally introduced in Sec. 3.2.

3.5.1.1 *ppDATE* Specification for the Coffee Machine

```
IMPORTS { import main.CMachine; }

GLOBAL {
 EVENTS {
   brew_entry() = {CMachine cm.brew()}
   brew_exit() = {CMachine cm.brew()uponReturning()}
   cleanF_entry() = {CMachine cm.cleanF()}
 }
 PROPERTY prop {
  STATES {
    BAD { bad ; }
    NORMAL { q2 (brew_error,clean_filter_error) ;}
    STARTING { q (brew_ok,clean_filter_ok) ; }
  }
  TRANSITIONS {
    q -> q2 [brew_entry \ cm.cups < cm.limit ]
    q2 -> q [brew_exit ]
    q2 -> bad [ brew_entry ]
    q2 -> bad [ cleanF_entry ] }
 }
}

HTRIPLES {
 HT brew_ok {
   PRE {cups < limit}
   METHOD {CMachine.brew}
   POST {cups == \old(cups)+1}
   ASSIGNABLE {cups} }
 HT brew_error {
   PRE {cups < limit}
   METHOD {CMachine.brew}
   POST {cups == \old(cups)}
   ASSIGNABLE {cups} }
 HT clean_filter_ok {
   PRE {true}
   METHOD {CMachine.cleanF}
   POST {cups == 0}
   ASSIGNABLE {cups} }
```

```
 HT clean_filter_error {
   PRE {true}
   METHOD {CMachine.cleanF}
   POST {cups == \old(cups)}
   ASSIGNABLE {cups} }
}
```

3.5.1.2 Coffee Machine Implementation

```
public class CMachine {
   public int cups;
   public int limit;
   public boolean active;

   CMachine(int limit) {
        this.limit = limit;
        cups = 0;
        active = false;
   }

   public void cleanF() {
       if (!active)
          cups = 0;
   }

   public void brew() {
       if (!active && cups < limit)
          cups++;
   }
}
```

3.5.2 Running StaRVOOrS

In order to run StaRVOOrS, as it is illustrated in Fig. 3.7, the following input should be provided:

(i) the path to the main directory of the Java files to be verified, for instance, `Example/CoffeeBrew`.

(ii) a description of the *ppDATE* specification for the provided program written in the input language of the tool, for instance, `Example/prop_brew.ppd`.

(iii) the path of the output directory where the files generated by the tool are going to be placed, for instance, `Example`.

3.5.2.1 Flags

When running StaRVOOrS, one may include flags to indicate different options. Most of these flags are use as follows:

```
StaRVOOrS [-OPTIONS] <java_source_files> <ppDATE_file> <output_add>
```

Below, we list them.

- -n (or --none_verbose)
  None verbose monitor generation.
- -x (or --xml)
  The .xml file generated by KeY is not removed.

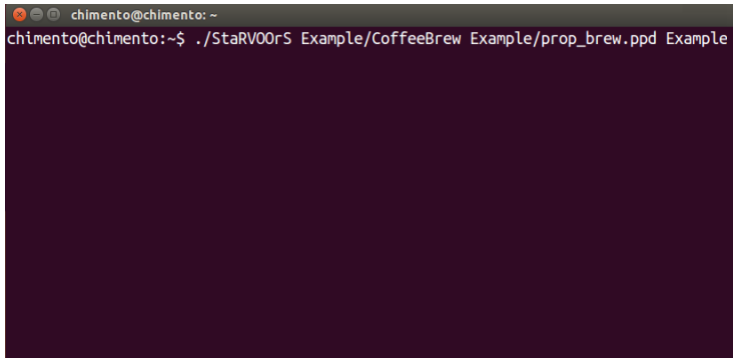


Figure 3.7: Running StaRVOOrS

- -r (or --only_rv)
  Monitor generation without performing deductive verification with KeY.
- -k (or --killbad)
  All the ppDATEs which have reached a bad state are killed, i.e. terminated.
- -d (or --distributed)
  Improves the output provided by the monitor in the context of active objects.[1]
- -v (or --version)
  Shows StaRVOOrS version number. Usage: `StaRVOOrS -v`
- -p (or --only_parse)
  Only parse the ppDATE file. Usage: `StaRVOOrS -p <ppDATE_file>`
- -h (or --help)
  Describes the different flags available. Usage: `StaRVOOrS -h`

Note that the flags -p, -h, and -v, have to be used in isolation. All of the other flags can be combined.

3.5.3 StaRVOOrS ouput

Fig.3.8 illustrates all the files generated by StaRVOOrS when it is used to analise the running example. This output consists of: the monitor files generated by Larva (folder `aspects` and folder `larva`), the files generated by StaRVOOrS to runtime verify partially proven Hoare triples (folder `ppArtifacts`), an instrumented version of the source code (folder `CoffeMachine`), a report summarising the results obtained during the static verification of the Hoare triples (*report.txt*), the optimised version (if any) of the provided *ppDATE* specification (*prop_brew_optimised.ppd*), and the *DATE* specification obtained as a result of translating the (optimised) *ppDATE* (*prop_brew.lrv*). Note that StaRVOOrS does not modify the provided source code, it creates an instrumented version of it. Thus, at the time of monitoring the code, the instrumented version of the source code is the one which should be used.

3.5.4 StaRVOOrS execution insights

StaRVOOrS is a fully automated tool. However, in order to have a better understanding of its execution, below we will explain it in three stages. Note that during

[1] This feature was only tested in ProActive.

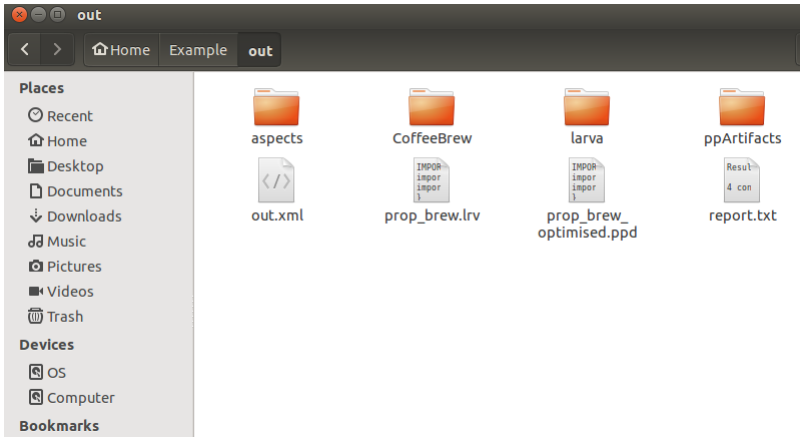


Figure 3.8: StaRVOOrS output

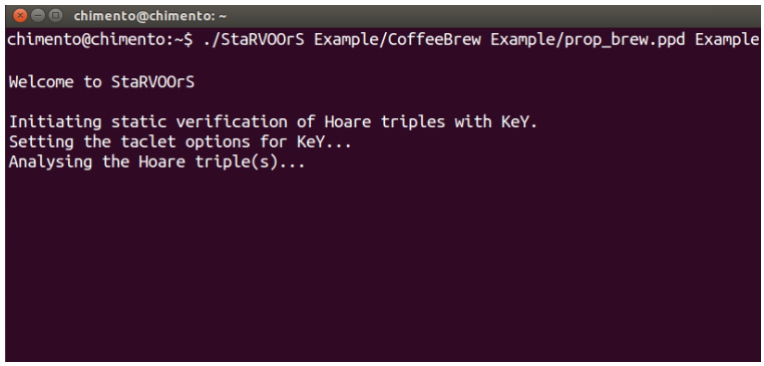


Figure 3.9: Initiating Static Verification

each one of this stages, StaRVOOrS will produced some output on the terminal. We will illustrate such an output through figures.

The first stage corresponds to the static verification of the Hoare triples using KeY. Fig. 3.9 shows the output produced by the tool on the terminal during this stage. At first, KeY (taclet) options are set. These options are parameters which, for instance, indicate to KeY which rules of its sequent calculus it is able to use during the verification of a property. For the time being, we are just using the standard options. Then, KeY is ran.

While KeY analyses all the Hoare triples, every time a proof attempt is saturated, some information related to this analysis is given as output in the terminal. Fig. 3.10 illustrates this. Once KeY is done verifying all the Hoare triples, it generates a temporary file *out.xml*, which is removed once StaRVOOrS is done[2], describing its results. This file is used by StaRVOOrS to optimise the *ppDATE* specification for runtime checking. However, in order to give to the user some understandable feedback about what happened during the static verification of the contracts, StaRVOOrS generates a file *report.txt* which briefly explains the content of the .xml file.

[2]Note that one may use the flag -x to include this file as part of the output.

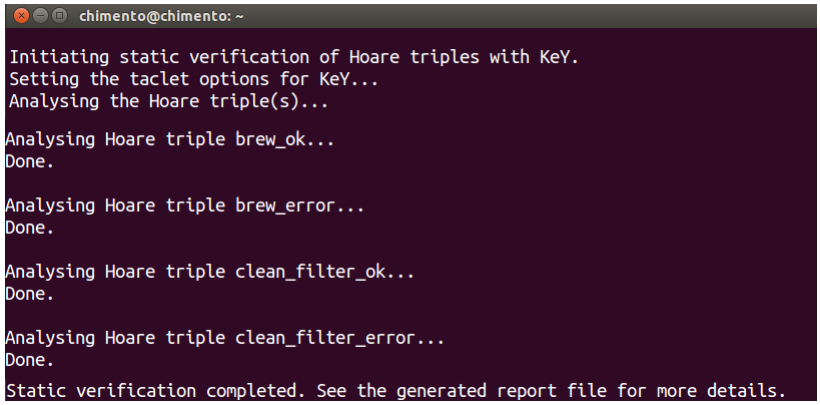


Figure 3.10: Output shown on the terminal during static verification

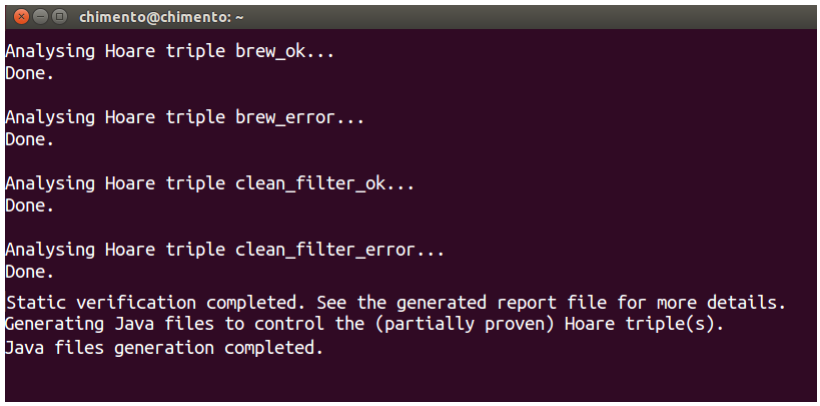


Figure 3.11: Optimization and files generation after static verification

The second stage corresponds to the refinement of the specification. In this stage, all the Hoare triples which were fully proven are removed from the *ppDATE*, and those which were only partially proven are modified by strengthening their pre-conditions including the conditions which lead to an unclosed path on a proof. For instance, in our running example the report would include the information that the pre-condition of the Hoare triple `brew_ok` is strengthen with the addition of the condition `active == TRUE`.

Whenever it is necessary at runtime to verify partially proven Hoare triples, StaRVOOrS instruments the provided Java files by adding a new parameter to the method(s) associated to the Hoare triple(s). This new parameter is used to distinguish different calls to the same method. This change is introduced in the refined *ppDATE* specification as well. Besides, StaRVOOrS generates several files within folder the `ppArtifacts` which are used to runtime verify the Hoare triples. For instance, the file *HoareTriplesPPD.java* contains the implementation of the methods which are used to verify the pre- and post-conditions of the Hoare triples. In addition, file *IdPPD.java* will be used to generate the value of the new parameter added to the methods. Once this stage is over, the terminal will look like Fig. 3.11.

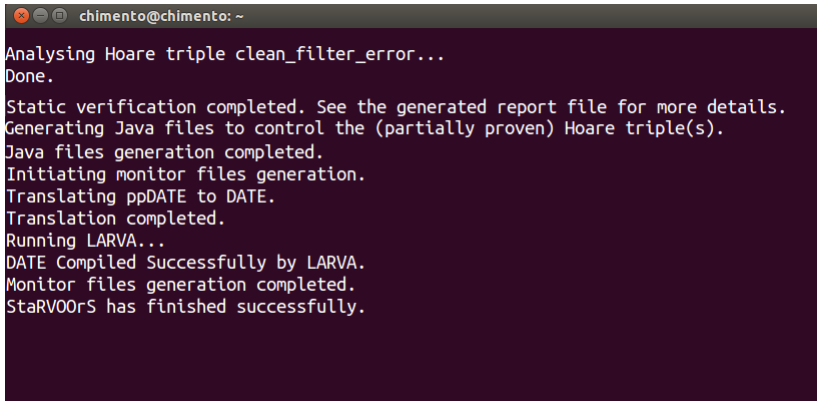


Figure 3.12: Monitor Generation

The third stage corresponds to the generation of the runtime monitor. In order to do so, the refined *ppDATE* specification is translated by StaRVOOrS to a *DATE* specification (file *prop_brew.lrv* in our running example). Then, Larva is used to generate the monitor files from the *DATE*. After the execution of Larva is completed, leading to the generation of the files in the folders `aspects` and `larva`, StaRVOOrS execution is completed as well. The terminal will reflect this, as it is illustrated in Fig. 3.12.

3.5.5 Running the application with the generated monitor

To run the (generated) instrumented version of the program, let us call it P, together with the monitor, one can generate an executable jar file, and then run it on a Java virtual machine. We will use Java 1.7 to compile the Java files. However, due to compatibility issues with Larva, when compiling the aspects one has to use the version 1.5. Note that the aspects have to be compiled using an AspectJ compiler. We recommend the *ajc* compiler. In addition, to run the jar file one has to use *aj5* (like command *java*, but with support for AspectJ), or similar. Below, we provide a short script explaining how to create such a jar file.

First, go inside the output directory.

cd Example/out

Second, copy the folders `larva` and `ppArtifacts` into the main folder of *P*.

cp -r larva CoffeeBrew
cp -r ppArtifacts CoffeeBrew

Third, create a directory named `Build`, and compile *P* using the option *-target 1.7* in such a way that the compiled files are placed within `Build`.

mkdir Build
*javac -target 1.7 $(find CoffeeBrew -name *.java) -d Build*

Next, create an executable jar file from the files in `Build`.

jar cfe coffeeM.jar main.CMachine -C Build .

Now, one has to weave the aspects into the jar file. In order to do so, the files in folder aspects have to be compiled using an AspectJ compiler. Here, we use *ajc*. Note that it is usually recommend to generate a new jar file when the aspects are weaved.

ajc -1.5 -sourceroots aspects/ -inpath coffeeM.jar -outjar coffeeM_asp.jar

Finally, this weaved executable jar file corresponds to the compilation of P together with the monitor. Therefore, running it would mean the one is running a monitored version of P. To execute the weaved jar file one can use *aj5* as follows:

aj5 -jar coffeeM_asp.jar

CHAPTER
FOUR

TESTING MEETS STATIC AND RUNTIME VERIFICATION

J. M. Chimento, W. Ahrendt, G. Schneider

Abstract

Test driven development (TDD) is a technique where test cases are used to guide the development of a system. This technique introduces several advantages at the time of developing a system, e.g. writing clean code, good coverage for the features of the system, and evolutionary development. In this paper we show how the capabilities of a testing focused development methodology based on TDD and model-based testing, can be enhanced by integrating static and runtime verification into its workflow. Considering that the desired system properties capture data- as well as control-oriented aspects, we integrate TDD with (static) deductive verification as an aid in the development of the data-oriented aspects, and we integrate model-based testing with runtime verification as an aid in the development of the control-oriented aspects. As a result of this integration, the proposed development methodology features the benefits of TDD and model-based testing, enhanced with, for instance, early detection of bugs which may be missed by TDD, regarding data aspects, and the validation of the overall system with respect to the model, regarding the control aspects.

4.1 Introduction

Minimising bugs is a major objective in software development, but accomplishing this objective to a satisfactory degree is often difficult. In fact, few experts are overly surprised when bugs are found even in well-known programs or algorithms, e.g. [49]. The need of software development techniques which help programmers to spot bugs early on is apparent.

Programmers can use several techniques which help to develop implementations with fewer bugs. The most used technique to increase confidence in the correctness of the developed software is undoubtedly *testing*. To a lesser extent *formal methods* are used. They offer stronger guarantees, but they are not applied nearly as widely as their potential suggests.

Besides the more traditional way of performing testing, *test driven development* (TDD) [20] is a technique where test cases are used to drive the development of the program. Therein, test cases form a light-weight 'specification' of program units, guiding the programmer who aims at satisfying the given test-cases. Using this technique, programmers tend to write cleaner code with good coverage for the desired system features, as every feature is accounted with test cases. This helps limiting the introduction of bugs.

Another testing technique is *model-based testing* (MBT) [91], which in turn is part of *model based development*. In MBT, tests are automatically generated (also) from model artifacts, and frequently executed to check whether the test passes or not (after providing a checker for expected outputs, the *oracle*). In order to perform MBT one must write a *model* from which the test cases are obtained.

In this paper we show how the capabilities of a testing focused development methodology based on TDD and MBT, can be enhanced by integrating static and runtime verification into its workflow. Considering that the desired system properties can be separated into data-oriented aspects (e.g. how a method modifies the fields of a class) and control-oriented aspects (e.g. proper flow of execution of the methods), we integrate TDD with (static) deductive verification [9, 56], and we integrate MBT with runtime verification [58, 63, 74]. The former integration comes as an aid in the development, and debugging, of the data aspects, whereas the latter helps the development, and debugging, of the control-oriented part.

Regarding the data aspects, we first define (empty) methods needed in the classes, and we write *contracts* (i.e. *Hoare triples*) for them. Then, we write test cases covering all contracts, and we proceed by applying TDD. (As we so far only have empty methods, tests will in principle fail for the lack of even an initial implementation.) After some iterations in TDD, where method implementations are developed, and some early bugs may be discovered and fixed, we use deductive verification to formally verify the methods. If some of the contracts cannot be (fully) verified, we generate (potentially failing) test cases covering the parts of the implementation that could not be proven correct, and continue by applying TDD focused on these new tests. Then, we iterate on these steps, until the verification of the methods associated to these contracts is saturated, i.e. we got to fully verified the methods, or the deductive verifier has not enough information to finish the proof.

Regarding the control aspects, we start by writing a model for these aspects. Next, we use MBT to generate test cases, and continue with the development of the program by attempting to get a desired coverage over the model, e.g. transition coverage. After this, we produce a monitor specification from the model, in order to then runtime verify the overall system implementation with respect to the model. This monitor specification can be further extended to cover aspects not covered by the model. (In particular, forbidden behaviour is often not made explicit in models, but very much so in monitor specifications.)

As a result of this integration, our proposed methodology features the benefits of

using TDD and MBT, but enhanced with:

- early detection of bugs which may be missed when applying traditional TDD;
- high code coverage for the unit tests by (proof) construction;
- the validation of the overall system behaviour with respect to the model (understood as a specification)
- the inclusion of aspects often neglected in models (and in MBT), like nested methods calls and forbidden behaviour.

The authors have earlier made technical contributions which are used in this work, in deductive verification [9], proof based test generation [13], and combined static and runtime verification [11]. The corresponding tools are used, together with other tools, in the examples we discuss (see Sec. 4.4). The proposed development process does, however, not depend on the exact tools used in the different steps.

Structure of the paper. Sec. 4.2 provides a brief introduction to TDD, MBT, and static and runtime verification. Sec. 4.3 presents an overview of our proposed methodology. Sec. 4.4 illustrates in more detail our methodology through its application in the development of a small Java program. Sec. 4.5 elaborates on the benefits of using our proposed methodology. Sec. 4.6 discusses related work and Sec. 4.7 concludes the paper.

4.2 Background

In this section we briefly introduce the concepts we build upon in this work.

4.2.1 Test Driven Development

Test driven development (*TDD*) is a software development technique [20]. In this technique, the test cases serve as a guide for developing the different parts (units) of the system. Pragmatically, the test cases can be seen as (unit) specifications, however in a limited sense, as the wanted behaviour is only given for exactly these tests, and the programmer has to extrapolate from that herself.

Performing TDD consists of the following steps:

(i) Write test cases that initially fail;
(ii) Write code making the tests pass;
(iii) Refactor the code.

These steps are usually known as Red, Green, and Refactor, respectively. The idea is that before implementing the methods of the system one should, first, write test cases for all of them. Such test cases will immediately fail, as the methods are not (properly) implemented yet. Then, one proceeds to implement the methods. The implementation of a method is considered to be ready once its test cases succeed. Finally, one should remove from the implementation all the duplication of code (if any) introduced in order to make the test pass.

In general, by using TDD, programmers limit the introduction of bugs to a certain extent. In addition, this technique presents other benefits like writing clean code, good coverage for the features of the system, and evolutionary development.

On the negative side, developers usually complain that they do not think in terms of tests and that it takes more time to develop the code, so it is imperative to break such resistance to change the way they develop software. After adopting TDD though, many programmers agree with the benefits of using it [16].

4.2.2 Model Based Testing

Unit testing focuses on writing tests which analyse the computation performed by the unit on the *data*. In contrast to that, model-based testing (MBT) [91] provides

better support for testing *control*-oriented aspects, e.g. the flow of execution of the methods in the program under test. Most models that are used to generate tests for control-oriented aspects are based on variants of *finite-state machines*.

In general, MBT tools can automatically generate test cases from the model which might also contain the expected output in order to automate the decision on whether the test passes or not [6, 90]. In addition, they may generate failing traces which simplifies the detection of pitfalls in the program under test.

More concretely, MBT involves doing the following:

(i) Writing an abstract model (sometimes the model is annotated to capture the relationship between tests and requirements);
(ii) Generating *abstract* tests from the model, which implies defining a test selection and coverage criteria;
(iii) Generating *concrete* test cases, which implies the creation of an *adaptor* to convert abstract tests into concrete test cases;
(iv) Executing the tests on the system under test (SUT) and assigning verdicts;
(v) Analysing the test results and taking corrective action.

Note that a fault in the test case might not necessarily mean that there is a problem with the implementation: the verdict might be due to a fault in the adaptor code or in the model.

Among the benefits of using MBT, it is usually mentioned [91] that it increases the possibility of finding errors, it reduces testing cost and time (programmers spend less time and effort on writing tests and analysing results as it generates shorter test sequences), it improves the test quality (by considering coverage of the model and of the SUT), it might detect requirements defects, it gives traceability between requirements and the model, and between informal requirements and generated test cases, and that it helps the updating of test suites when the requirements evolve.

On the negative side, among other things MBT cannot guarantee to find all differences between the model and the implementation, it needs skilled model designers, and it is mostly used for functional testing. Moreover, unless you keep an updated table relating requirements with the model, you might get the wrong model from outdated requirements. Finally it is indeed an overhead to write the model (which might be wrong) and to develop the adaptor (which might also introduce errors).

4.2.3 Deductive Verification

In deductive verification, correctness properties of a program (unit) are captured in logical formulae, e.g., in first-order logic, high-order logic, program logic, etc. These formulas are then proved by deduction in a (logic) calculus [9, 56].

There are three main approaches that one may adopt to perform deductive verification. Let us call these three approaches *Proof Assistants*, *Program Logic*, and *Verification Condition Generation*.

Proof Assistants are interactive theorem provers which, in general, target some high-order logic [28, 92]. These provers are not language-oriented. Instead, they provide a language in which both the syntax and the semantics of the program under scrutiny have to be described. In addition, the correctness properties have to be modelled within the logic handled by the proof assistant. Then, one may use the proof assistant to develop the proof of the properties.

Concerning *Program Logic*, *Hoare Logic* [66] may be the most well-known program logic to analyse programs. Hoare logic offers both a clear notation to describe programs and their properties, and a set of axioms and inference rules which may be used to verify the properties [81]. In this logic, properties are described by using *Hoare triples*.

In the *Verification Condition Generation* approach, programs are annotated with assertions representing the desired correctness properties [71]. Then, this assertions

may be used to generate first-order logic verification conditions which later may be discharged by using some automatic prover [50].

A benefit of deductive program verification is that once a property (contract) for a given unit is proven, there is a very high confidence that the method is correct (provided the property is correct). Another advantage is that one does not need to run the program, reducing the need to find test cases and to set or simulate runtime environments.

One disadvantage of this technique is that it is not possible, in general, to be applied automatically. Also, the method requires contracts of called (library) code and loop invariants. So one can argue that it requires a highly specialised person to do such verification, as the critics go for many other formal methods techniques. Besides, many properties of the program cannot be proved statically and are required to be analysed during program execution.

4.2.4 Runtime Verification

Runtime verification (RV) [58, 63, 74] is a technique focused on monitoring software executions. It detects violations of properties which occur while the program under scrutiny 'runs'. Moreover, RV provides the additional possibility of reacting to the incorrect behaviour of the program whenever an error is detected.

Properties verified with RV are specified using any of the following approaches: (i) annotating the source code of the program under scrutiny with *assertions* [72]; (ii) using a high level specification language [78]; or (iii) using an automaton-based specification language [10, 45].

In order to perform the verification of the properties, RV introduces the use of *monitors*. A monitor is a piece of software that runs in parallel to the program under scrutiny, controlling that the execution of the latter does not violate any of the properties. In addition, monitors usually create a log file where they add entries reflecting the verdict obtained when a property is verified.

In general, monitors are automatically generated from the annotated/specified properties [42, 46, 80], which is of course a big advantage. Another advantage is that one can check also properties which are not provable statically, thus complementing static verification. Finally, the fact of monitoring the real execution makes the technique appealing since this particular execution, and deployment, may not have been covered at testing time.[1]

The main disadvantages of this technique is that one can only capture errors that are witnessed by current executions and cannot say much, in general, about other runs. Depending on the context, adding a monitor adds time and space overheads which might be prohibitive in some cases (e.g., in small devices, or when the response time of the system is critical).

4.3 Combining Testing with Static and Runtime Verification

In this section we provide an overview of the proposed development methodology. As a starting point, for presentation purposes, we describe a methodology using two styles of testing, TDD and MBT, not yet using deductive or runtime verification. Thereafter, we enhance the methodology by integrating (static) deductive verification and runtime verification in the workflow. A detailed example demonstrating the usage of the methodology will be provided in the next section (Sec. 4.4).

[1] The monitor can also log the execution of the program in order to perform a 'post mortem' analysis which could give more insights into why the error occurred.

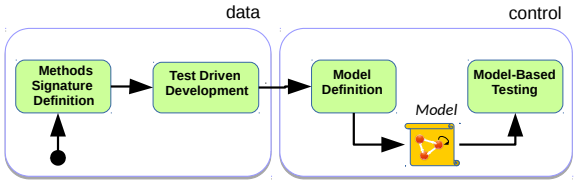


Figure 4.1: A testing focused development workflow.

4.3.1 A Testing Focused Development

Fig. 4.1 illustrates an abstract view of a purely testing focused workflow. Based on the insight that the desired properties of a system can be largely divided into data- and control-oriented aspects, we can view the methodology as consisting of two stages focusing on data and control, respectively.

Regarding the data stage, first we define the signatures of the methods, and provide stub implementations to enable compilation. Then, we use TDD as explained in Sec. 4.2.1. Here, the various aspects of the desired computation on the data have to be accounted with (unit) test cases.

Regarding the control stage, we start by writing a model focusing on the control aspects of the system. Then, we continue developing our program by using model-based testing, in a similar manner to how *Behaviour Driven Development* (BDD) [34] is performed. BDD is an extension of TDD where one focuses on the behaviour of the system instead of units of code. In general, every feature of the system is divided into scenarios of the form *GIVEN-WHEN-THEN*, e.g. GIVEN certain condition, WHEN some operation is performed, THEN something should happen. In [44], Colombo *et al.* show how the BDD features can be written as models for model-based testing. For instance, the scenario,

GIVEN we are in state *unlogged*
WHEN method `log` is ran successfully
THEN we are in state *logged*

would be represented in a model as a transition from the initial state *unlogged* to the state *logged*, which is triggered whenever the method `log` is ran successfully.

In the spirit of this pattern, we continue by generating test cases which trigger the transitions of the model, aiming at triggering each transition at least once. In terms of BDD, this would be similar to considering a whole scenario every time we iterate in the development cycle.

Thus, one would continue iterating on this stage until transition coverage over the model is accomplished. Note that failing to accomplish this would probably mean that the implementation is erroneous (assuming that the model is correct of course).

Finally, we proceed to complete the overall implementation of the system, by implementing the system level layer(s). In the simplest case, in a stand alone, command line application in, say, Java, this may correspond to implementing the class containing the method `main`.

Once the development of the overall implementation of system is completed, we will have an implementation that is likely to feature:

- clean code;
- good coverage over the data aspects;
- high coverage over the control aspects;

However,

- the unit test cases only specify the wanted behaviour for some specific inputs, not for all inputs;
- we have no information regarding the unit test coverage;
- all unit test cases need to be written by hand;
- we have no evidence that the overall system implementation fulfills the control aspects of the desired properties.

4.3.2 A methodology integrating testing and verification

The aforementioned shortcomings of the purely testing focused methodology indicate the potential for an improved methodology, which we present in the following.

In [10, 11], Ahrendt *et al.* show how runtime monitors can be optimised by combining the use of runtime verification with deductive verification. In these works, the authors consider the integration of data- and control-aspects in the specification, but their separation in the verification. From that work, we inherit the overall idea to use static and runtime verification in combination, however in a different way. In the development process we propose here, static and runtime verification techniques are not integrated with each other directly, but are integrated with TDD and MBT, respectively. As a result, we obtain the workflow illustrated in Fig. 4.2.

Regarding the data stage, we start by defining the signatures of all the methods associated to data aspects, providing a stub implementation to allow their compilation. Next, we define *contracts*, i.e., properties written as pre/post-conditions (Hoare triples), for the different methods. These contracts focus on the data aspects. We then proceed to apply TDD, adding one test at a time, and make it pass by further developing the implementation. Each one of this test should cover a scenario where a contract holds, and each contract should be associated to (at least) one test.

Once we have implemented the methods, we proceed to use deductive verification in an attempt to statically verify the implementation with respect to its contract. For each method, this either results in (1) a closed proof, i.e. the contract is fully verified, or (2) an unclosed (partial) proof, i.e., the contract is not (fully) verified.

In case of (1), this means that the method fulfils the contract. In case of (2), either (i) there is a bug in the program, or (ii) the deductive verifier has not enough information to finish the proof. Here, we can use tools like KeY [9] or StaDy [77] to reason about whether (i) or (ii) is most likely. These tools use *symbolic execution* [64, 70], KeY through the use of its feature KeyTestGen [13], and StaDy through the use of PathCrawler [93], to generate test cases covering exactly those executions through the method that correspond to the open proof branches. In general, if the test case succeeds right away, this can be an indicator that the verifier has not enough information to finish the proof. If however the test does not pass, we modify the implementation to make the test succeed, i.e. we apply TDD. Thus, we have a retrofitting loop between deductive verification and TDD, where deductive verification provides new, *automatically generated* tests for TDD. This loop will continue until either the verification of the method is saturated, i.e. we got to fully verified the contract associated to the method, or the deductive verifier has not enough information to finish the proof. Either way, we will have a test suite providing guarantees towards a high code coverage for the unit tests. In Sec. 4.4 we will show an example on how this retrofitting loop can detect bugs which could not be detected right away by using traditional TDD. Also, note that we make use of deductive verification in a much more lightweight manner than what is done traditionally. If proofs fail due to limitations of the effort level we can put in the development ecosystem at hand, the failed proofs are still of good use, as they are the source of new tests.

Turning to the control stage, we start by working in the exact same manner as described in Sec. 4.3.1. However, once we have implemented the system level

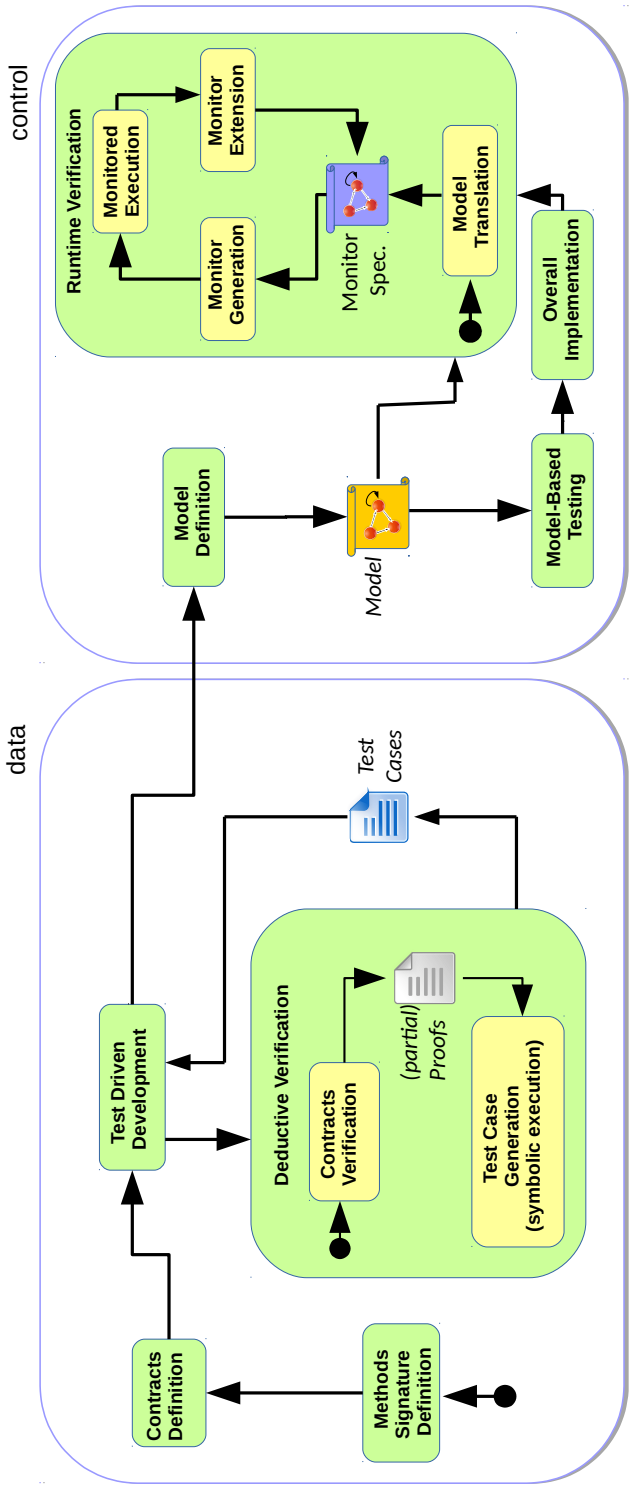


Figure 4.2: Integrating deductive and runtime verification in the workflow.

layer, we move on to the use of runtime verification. For that, we first need to produce a monitor specification from the model. By considering the results given by Falzon *et al.* [55], we can convert the model into a monitor specification in a quite straightforward manner. (This step could in principle be partly automated, but is not at the moment.) We then use this specification to automatically generate a monitor (e.g. using the LARVA tool [46]), in order to runtime verify the overall system implementation with respect to the model. Here, we use the test cases generated with model-based testing in the previous step as traces to guide the monitored execution.

After certain rounds of runtime verifying the overall system implementation, we can proceed to further extend the monitor in order to cover aspects not covered by the model. For instance, we can add new transitions to violating states to capture forbidden behaviour. In addition, by using runtime verification we can include the analysis of control aspects of nested method calls, as model-based testing is mainly focused on the (almost) top level of the call stack.

Once the development of the overall implementation of system is completed, we will have an implementation that is likely to feature:

- clean code;
- high unit test coverage (in terms of specific metrics) guaranteed by the use of deductive verification [13];
- high test coverage over the control aspects (transition coverage on the model);
- good evidence that the overall implementation of the system fulfills the control aspects (achieved by the use of runtime verification).

4.4 The methodology in action

In this section we describe the use the development methodology described in Sec. 4.3.2 in a running example, consisting on the (Java) development of a small bank system where users log in to perform transactions. Below, we provide a brief description of the system. Throughout the section, even without explicit mention, Fig. 4.2 is a good reference for the current position in the workflow.

A repository with the whole documentation of the system, and the developed sources, is available from [7]. The repository contains several branches covering the steps above, e.g., branch *step1* covers the first step, branch *step2* covers the second step, and so on. These versioned sources will allow the interested reader to have a proper understanding on the work performed at each step, and to have a clear view on how the development evolves from one step to the other.

Running Example: Bank System Our running example consists on the development in Java of a small bank system where users log in to perform transactions. This system has the following classes:

- `Account`, representing the accounts of the bank,
- `Category`, representing different user categories,
- `DataBase`, emulating a database,
- `HashTable`, an open addressing Hashtable with linear probing as collision resolution,
- `SystemCentral`, used to keep track of centralised data,
- `User`: representing the users of the bank,
- `UserInterface`: representing the interface offered to the users in order to interact with their accounts,
- `Main`.

Classes `Account`, `Category`, `HashTable`, and `User` are developed in the data stage; whereas classes `UserInterface`, and `Main` are developed in the control stage. Note that the classes `DataBase` and `SystemCentral` are used to emulate, only, the interaction

```
/*@ public normal_behaviour
  @ requires size < capacity;
  @ ensures
  @ (\exists int i; i >= 0 && i < capacity; h[i] == u);
  @ ensures size == \old(size) + 1;
  @ assignable size, h[*];
  @ also
  @ public normal_behaviour
  @ requires size >= capacity;
  @ assignable \nothing;
  @ */
public void add(Object u, int key) { }
```

Figure 4.3: Contracts for method add.

with the database, and the centralised data for the bank. Thus, their development is not a proper part of the running example.

On this section, we discuss the development of the classes `HashTable` (steps 1,2, and 3), `UserInterface` (step 4), and `Main` (steps 5 and 6). In addition, in the presentation we mainly deal with the following data and control aspects, respectively: (i) the set of logged user will be implemented using an open addressing hashtable with linear probing as collision resolution (data aspect); (ii) a user has to be logged to perform a transaction (control aspect).

4.4.1 Method Signature Definition

We start applying our methodology by defining the signatures of all the methods associated to data aspects, providing a stub implementation to allow their compilation.[2]

4.4.2 Contract Definition

We then continue by defining contracts accounting for the data aspects of the system. Here, we use the *Java Modelling Language* (JML) [68] to write the contracts. By using JML one can specify both pre- and postconditions of methods calls, and class invariants. JML contracts are annotations in the source code, preceding the corresponding method signature.

Regarding class `HashTable`, Fig. 4.3 shows the contracts defined for method `add`, which is used to add an object into a hashtable. The first contract corresponds to the case where there is still room for adding a new object to the hashtable. It says that, after adding the object, there exists an index in the hashtable where the new object is stored. The second contract corresponds to the case where the hashtable is full. Fig. 4.4 shows one of the contracts defined for method `delete`, which is used to remove objects from the hashtable. It corresponds to the case where there is an object in the position of the computed hash code for `key`. Then, the object is replaced by null in the hashtable, the size of the hashtable decreases by one, and the removed object is returned by the method. The objects in the other positions should remain the same.

[2]See branch *initial-code* in [7].

```
/*@ public normal_behaviour
  @ requires key >= 0 && size > 0;
  @ requires h[hash_function(key)] != null;
  @ ensures \result == \old(h[hash_function(key)]);
  @ ensures h[hash_function(key)] == null
  @         && size == \old(size) - 1;
  @ ensures (\forall int j; j >= 0 && j < capacity
  @          && j != hash_function(key); h[j] == \old(h[j]));
  @ assignable size, h[*];
  @ */
public Object delete(int key) { }
```

Figure 4.4: One of the contracts for method delete.

```
    @Test
    public void test_add_1(){
        int idx = hash.hash_function(3);
        hash.add(new Integer(42),idx);

        assertEquals(hash.h[idx],new Integer(42));
    }
    @Test
    public void test_add_2(){
        hash.add(new Integer(3),0);
        hash.add(new Integer(38),2);
        hash.add(new Integer(42),0);

        HashTable aux = new HashTable(3) ;
        aux.add(new Integer(3),0);
        aux.add(new Integer(42),1);
        aux.add(new Integer(38),2);
        assertArrayEquals(hash.h,aux.h);
    }
```

Figure 4.5: Test cases for method `add`.

4.4.3 Test Driven Development

Once the contracts are in place, we proceed to define test cases and use TDD to implement the methods. Here, we use *JUnit* [26] to write and check the unit test cases.[3] For instance for method `add`, we may have two test cases (see Fig. 4.5). The first covers the case where the position of the computed hash code for the object is free, the other test covers the case where the corresponding position is occupied, i.e., the method should look for the nearest following index which is free.

Fig. 4.6 and Fig. 4.7 show (parts of) the developed implementations for method `add` and `delete`. Both implementations contain bugs which are not detected by the test cases that were used in this step. In method `add`, the statement `i++`, should actually be inside an `else` branch of the if statement; and in method `delete`, we are not computing the hash code of `key` before checking the hashtable.

Not only do the test we gave for `add` not reveal the bug, more generally, additional hand-written tests are also likely to miss this bug. The only way to trigger it is to `add`

[3] All the test cases are available from [7], under the path *src/test/java/bank*.

```
public void add (Object u, int key) {
     /* code ommited for presentation */
     while (h[i] != null && j < capacity) {
           if (i == capacity-1) i = 0;
           i++;
           j++;
     }
    /* code ommited for presentation */  }
```

Figure 4.6: Part of the implementation of method `add`.

```
public Object delete (int key) {
    if (key >= 0) {
        if (h[key] == null) return null;
        else { Object ret = h[key] ;
               h[key] = null ;
               size = size - 1;
               return ret;
        }
    } else { return null; } }
```

Figure 4.7: Implementation of method `delete`.

an element (with non-zero hash) to a hashtable with *exactly one* free position which is *precisely* at index zero. Only in this scenario, we would add the element in a wrong position, as the index zero will be skipped during the iterations of the while (due to the missing else), leading to have an erroneous value for the variable `i` whenever the loop breaks, i.e. `j` reaches the value of `capacity`. Also the test case analysing method `delete` (not given here) succeeds because the value of `key` coincides with its hash code, and it is between the bounds of the hashtable. However, these bugs are detected in later steps.

4.4.4 Deductive Verification

Contract Verification

After all the methods associated to the data aspects are implemented, we use KeY [9] to verify them. KeY is a deductive verification tool for data-centric *functional correctness* properties of Java programs. Given a Java program with JML annotations on its methods, KeY generates formulae in Java Dynamic Logic, and attempts to prove them. In addition, KeY comes with a user interface where users can interact with the prover, and inspect proof trees.

Regarding the class `HashTable`, all of its methods are automatically verified, with exception of the methods `add` and `delete`. In relation to method `add`, as it contains a loop to look for the next available index, then KeY needs more information to deal with its first contract, i.e. it needs a loop invariant. Thus, we introduce a loop invariant, and run KeY one more time. Still, but now due to the bug, KeY is not able to fully prove the contract. This time the issue is that KeY cannot prove that the body of the loop fulfils the loop invariant. In particular, it cannot prove the invariant `i < capacity`. By taking a look at the information in the proof tree, we can realise that whenever `i` is set to zero in the inner `if`, it is immediately increased by one afterwards. Thus, the index zero will be always skipped. After fixing this

```
public Object delete (int key) {
    if (key >= 0) {
        int i = hash_function(key);
        if (h[i] == null) return null;
        else { Object ret = h[i] ;
               h[i] = null ;
               size = size - 1;
               return ret;
        }
    } else { return null; }    }
```

Figure 4.8: Fixing the implementation of method `delete`.

issue by wrapping the statement `i++` with an `else` branch in the `if` statement, KeY fully verifies the contract.

In relation to method `delete`, due to the bug KeY cannot fully verify the contract depicted in Fig. 4.4. In order to analyse the issue, this time instead of looking at the information in the proof tree, we proceed to generate new test cases for it covering the issue. Note that we take this decision to show that if a user does not know how to read a proof tree, she still can use our methodology for developing her system.

Test Case Generation

State-of-the-art deductive verification technology can be employed for more purposes than full-fledged formal verification, like test case generation [13] and runtime monitor optimisation [11]. In the current context, as a next step, we use *KeyTestGen* [13] to automatically generate the test cases, complementing the tests which we used for guiding the implementation in the TDD phase. Specifically, the tool generates tests covering the cases that could *not* be verified by KeY. Note that, if the verification for a certain class of initial values and inputs did not succeed, this means that either the implementation is correct but KeY was not able to show that (with the given effort level), or that implementation is indeed incorrect. The generated test cases help us to distinguish these cases, and locate additional bugs.

Therefore, we run KeyTestGen including the postcondition of the contract in the oracle. This generates the file *TestGeneric0_delete.java*, which contains a test case, in this case a failing one, i.e., a counter-example for the contract.[4] Its execution throws an exception where an index is out of bounds when accessing the hashtable. We then apply TDD again, with focus on making this new test succeed. The exception indicates that the hash code of the `key` is not computed before checking the hashtable. We fix the implementation of `delete` as it is illustrated in Fig. 4.8. Afterwards, KeY fully verifies the contract.

4.4.5 Model Definition

We now move on to the control focused stage, starting by defining the model describing the control aspects of the system.

Regarding the class `UserInterface`, Fig. 4.9 depicts the model representing the following control aspect: *A user should be logged to perform a transaction.* We use a modelling language where transitions have the form $pre \xrightarrow{\texttt{foo}|post \mapsto action}_{q_1} q_2$. A transition from state q_1 to state q_2 can only be taken when method `foo` is called in

[4] This file is available from [7], under the path *src/test/java/bank*.

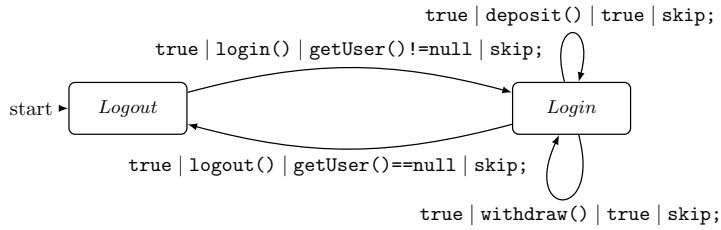


Figure 4.9: Model for control aspects of the system.

```
public boolean fooGuard(){
    return state = State.Q1 && pre ;
}
@Action
public void foo() {
    state = State.Q2;
    adapter.foo();
    assertTrue(post);
    action;
}
```

Figure 4.10: Model transition in modelJUnit terms.

a configuration where *pre* is satisfied. When a transition is taken, *post* has to be checked. If it holds, *action* has to be executed and q_2 is entered. If however *post* does not hold, this is considered a failure in the execution, revealing a bug in the implementation. In terms of modelJUnit (see Sec. 4.4.6), these transitions can be implemented as illustrated in Fig. 4.10.

4.4.6 Model-based Testing

In order to perform MBT, here we use *modelJUnit* [90], an extension of JUnit which supports model-based testing. In this extension, the models are written as Java classes, and the test cases are automatically generated from the model. Thereby, we continue our development by writing the model from Fig. 4.9 in modelJUnit syntax, and use the tool to automatically generate test cases, with focus on triggering each transition of the model (at least once)[5].

Once the class is fully implemented, modelJUnit is able to generate a test case which accomplishes transition coverage over the model. Fig. 4.11 illustrates the trace followed by this test, where the tuple (q_1, foo, q_2) means, "given that we are in state q_1, after executing foo we move to state q_2". Note that this trace is produced by modelJUnit.

4.4.7 Overall Implementation

Next, we implement the method `main` in class `Main`. For simplicity, we implement this method as a loop where the user is requested to enter the desired action, to be executed by the corresponding method in class `UserInterface`. Fig. 4.12 illustrates

[5]The files BankAdapter.java, BankModel.java, and BankTest.java, which implement the model are available from [7], under the path *src/test/java/bank*.

```
done (Logout, login, Login)
done (Login, logout, Logout)
done (Logout, login, Login)
done (Login, deposit, Login)
done Random reset(true)
done (Logout, login, Login)
done (Login, withdraw, Login)
done (Login, logout, Logout)
done (Logout, login, Login)
done (Login, deposit, Login)
```

Figure 4.11: Trace followed by the test case accomplishing transition coverage over the model.

```
switch (inputLine) {
    case "deposit":
         System.out.print("Enter␣the␣amount:");
         amount = in.next();
         aux = Integer.parseInt(amount);
         f.deposit(aux);
         break;
    case "withdraw":
         System.out.print("Enter␣the␣amount:");
         amount = in.next();
         aux = Integer.parseInt(amount);
         f.deposit(aux);
         break;
```

Figure 4.12: Part of the implementation for method `main`.

part of the (buggy) implementation for this method. As the code for calling both `deposit` and `withdraw` is practically identical, the programmer may just copy and paste it. The programmer may have forgotten to replace, after pasting, the call to method `deposit` by a call to method `delete`.

4.4.8 Runtime Verification

Model Translation

Once the method `main` is ready, we proceed by *runtime verifying* that the entire implementation fulfils the control aspects w.r.t. to (at first) the model given in Fig. 4.9.

First, by following the ideas in [55], we translate this model (Fig. 4.9) into a *DATE* specification [45]. Fig. 4.13 depicts part of the obtained *DATE*[6]. For space reasons, we have omitted the transitions and new states related to the methods `deposit` and `withdraw`. Regarding *DATE*, transitions are of the form $q_1 \xrightarrow{e|cond \mapsto act} q_2$. A transition from state q_1 to state q_2 is taken when event e occurs while the condition *cond* holds. If the transition is taken, action *act* is executed. The most important events are calls to, and returns from, methods, e.g., $\texttt{foo}^{\downarrow}$ and $\texttt{foo}^{\uparrow}$ (for a method `foo`).

[6]The file prop_deposit.ppd containing this translation is available in the root of [7].

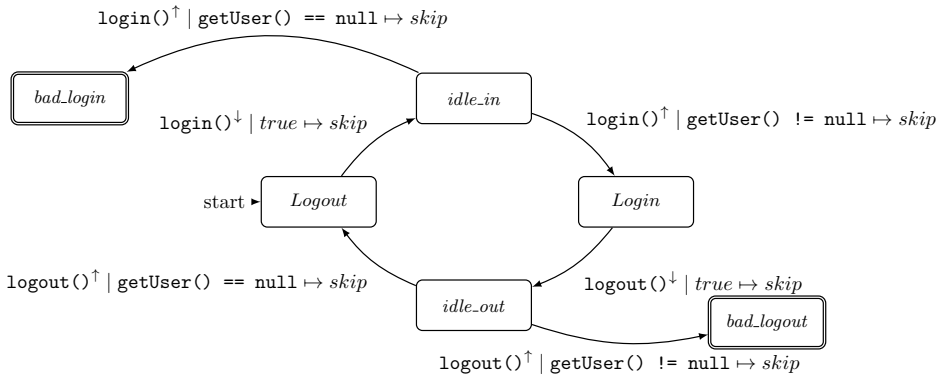


Figure 4.13: Part of the *ppDATE* specification generated from the model.

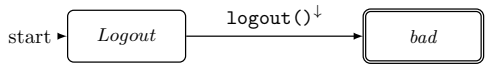


Figure 4.14: Extending the monitor for safety checks.

Monitor Generation

Second, we use the runtime verifier LARVA [46] to automatically generate monitor code. This code includes the Java classes implementing the monitor, and AspectJ code to link the application to the monitor code.[7]

Monitored Execution

Finally, we use the trace previously produced by modelJUnit as a guide to (runtime) check that the method `main` features the expected behaviour. In this particular example, as we are dealing with a small program, we follow this trace manually, i.e. we run the system and interact with its interface, to trigger the methods in the same order as they are called in Fig. 4.11. Clearly, there is good potential for automating such a step, connecting MBT tools and RV tools.

During the according execution of our program, by looking at the log file generated by the monitor we can notice that when we attempted to execute the method `withdraw`, the system called method `deposit` instead. Thus, we inspect the code and realise that the case for withdraw is actually making a method call to `deposit`. Then, we fix this issue by writing the appropriate call for the case of withdraw, and re-run the trace. This time, the execution of the trace fulfils the model from Fig. 4.9.

Extending the Monitor

Once we have analysed the overall implementation of the system w.r.t. to the model, we can extend the monitor to also cover safety properties. For instance, we can add the transition depicted in Fig. 4.14 to express that `logout` is never called while the user is not logged in. Also, we can check the integrity of the data flow through nested method calls. Fig. 4.15 shows the implementation of `deposit` in `UserInterface`.

[7] The files generated by LARVA, see [7], in *src/main/larva* and *src/main/aspects*.

```
public void deposit(int money){
    if (u != null && money > 0)
        u.getAccount().deposit(money);
}
```

Figure 4.15: Implementation of `deposit` in `UserInterface`.

This method has an inner call to the method `deposit` of the class `Account`. Then, we can have a monitor checking that both methods `deposit` are called with the exact same argument[8].

4.5 Discussion

In this section we elaborate on the benefits of integrating deductive and runtime verification into the workflow of the methodology introduced in Sec. 4.3.2.

By integrating TDD with deductive verification, we enhance the methodology with the following features: (i) early detection of certain bugs which are likely to be missed by using TDD alone, e.g., the bug in the implementations `delete` and `add` (Sec. 4.4); (ii) high (unit) test coverage—in terms of specific metrics—guaranteed by construction.

Concerning (i), as the (unit) test cases which are used for TDD only specify the expected behaviour for specific inputs, so it may be the case that the method has a bug but the test cases do not cover it. Note that, in TDD, it is the tests which guide the development of the implementation. Corner cases not covered by the tests can easily be buggy in code developed for those tests. For instance, the bug in the implementation of method `add`, described in Sec. 4.4, only occurs in a very particular case. In these cases, the use of deductive verification helps to detect the bug, as this verification technique analyses every possible run of the method. Even partial verification results help to direct the developers attention towards areas of further investigation.

Concerning (ii), test cases using symbolic execution can guarantee several kinds of coverages. For instance, depending on how it is set up, *KeYTestGen* can automatically generate test cases which guarantee either *full feasible path coverage*, *full feasible branch coverage*, or *Modified Condition / Decision coverage* [13]. Note that all the previous coverage metrics subsume *statement coverage*.

Regarding the integration of MBT with runtime verification, this enhances the methodology by adding good evidence that the overall implementation of the system fulfils the model used for MBT. As mentioned in Sec. 4.2.2, one of the disadvantages of MBT is that it cannot guarantee to find all differences between the model and the (overall) implementation of the system. However, by using the test cases (i.e. traces) generated from the model as a guide, we can use runtime verification to analyse whether the system behaves as it is described in the model.

There is room for improvement, and plenty of future work, in the proposed integration. For instance, implementing a tool which automatically generates a *DATE* specification from a modelJUnit model would improve the use of the proposed methodology, as the users would only have to worry about writing the (initial) monitor specification. In addition one could integrate (probably domain specific) tools automating the runtime verification of those traces generated by the MBT tool. For instance, one could use *Selenium* [8] to perform such a task on web applications.

[8]The file args_integrity.ppd available in [7], describes a monitor verifying this property.

4.6 Related Work

Test driven development [20] is a widely used development methodology. In the literature, one can find many extensions for this methodology, e.g. behaviour driven development [34]. In this paper, we have elaborated on the use of these methodologies, and we have discussed the benefits offered by our methodology in comparison to using this technique in isolation.

The combination of testing with either static analysis or static verification is an active area of research, e.g. [33, 47, 87, 89]. In general, these works aim at test case generation for either debugging, or verifying source code. Those objectives come in the play in our work, but integrated in a development method.

In [77], Petiot *et al.* present the tool *StaDy*. This tool applies test case generation in combination with deductive verification in order to increase the confidence regarding the correctness of C source code. In [13] Ahrendt *et al.* introduce *KeyTestGen*. This tool can be used to automatically generate test cases from partial proofs developed by the deductive verifier KeY [9] (for Java). Both *StaDy* and *KeyTestGen* can be used in a similar manner. First, one uses deductive verification to attempt to prove a property. If this attempt does not succeed, test case are generated for classes of executions which were not verified. The oracles of the generated test cases check whether the test is a counter example of the property. In our work, we make use if this principle, by integrating it in a development method.

Another related area of research is the combination of model-based testing with runtime verification. In [17], traces are automatically generated from nondeterministic models by using MBT. Then, these traces are used as a guide to verify at runtime the overall implementation of a system in order to analyse whether the system presents a good evidence for fulfilling the behaviour described by the model. This work accomplishes a similar integration to the one proposed in our methodology. However, it does not explore the possibility of extending the monitor obtained from the model to cover more properties at runtime.

In [55], Falzon *et al.* study the combination of test case generator QuickCheck [6] and the runtime verifier Larva [46], by presenting a technique which extracts runtime monitors from QuickCheck models, keeping the same semantics. This work focuses on the use of runtime verification to check the behaviour of a system w.r.t. the model used to generate the monitor. In our methodology, we use the ideas from this work to chain model-based testing and runtime verification, i.e. we are using both techniques instead of just runtime verification.

4.7 Conclusions

In this paper we presented a development methodology based on the combination of test driven development (TDD) and model-based testing (MBT), enhanced by the integration of (static) deductive verification and runtime verification on its workflow. We have also elaborated on the benefits obtained from the integration of these techniques (Sec. 4.5). The authors see the work as a contribution to integrated methodologies which take advantage of a variety of established practices and tools, making state-of-the-art **Formal** Methods profitable **i**n **S**oftware **E**ngineering processes.

CHAPTER
FIVE

INFERRING GLOBAL TRACE CONDITIONS FROM PARTIAL LOCAL PROOFS

J. M. Chimento

Abstract

System level modelling of properties focuses on describing legal sequences of method executions in the system from a control perspective. On the other hand, unit properties deal with data aspects of the system, local to the execution of the methods. Although there are works relating data and control aspects of a system, a yet quite unexplored area is the relation between global conditions regarding the execution of system traces, and unit properties local to the states of the model. The objective of this paper is to contribute to filling in this gap by presenting a methodology that uses the power of low effort, typically unsuccessful, proof attempts to verify individual transitions in the model, in order to infer global trace conditions. Given an initial property associated to a state of the model, this methodology starts by doing a reachability analysis, i.e. analysing which transitions can be taken to reach this state. Next, the methods associated to the incoming transitions are used to statically verify that these transitions have the desired property as a postcondition. This results in (possibly) local partial proofs for them. Then, closed-path conditions are extracted from the partial proofs, and are backwards propagated through the states of the model. Such conditions guarantee that, if any of these transitions is taken, a system trace fulfilling them will lead the system towards the desired state in the model, and the initial property will hold when that state is reached. Therefore, closed-path conditions come as a generalisation of the idea of weakest precondition for the methods. These steps are repeated until the initial state is reached. Finally, the property backwards propagated to the initial state, together with a system trace going from the initial state of the model to the desired state, which is created using the results of the reachability analysis previously performed, represent a trace condition for the system. Applications for the use of trace conditions include (global) test case generation, state invariants verification, and runtime verification.

5.1 Introduction

Having confidence in the correctness of a piece of software is essential for its deployment. Without a doubt, *testing* is the most widely used technique to increase such a confidence. In addition, one can also turn to *formal methods* (FM) to accomplish the same objective. However, in spite of the fact that FM offer strong guarantees to analyse software correctness, they are not used as broadly as their potential suggests.

An ordinary criticism against the use of FM is that it may not be easy to come up with a formal specification to describe the behaviour of the system, and once you do have your specification, fully verifying it may not be a simple task. In fact, failing to prove a property could mean that either the system has an error, or the chosen prover does not have enough information (or power) to finish the proof.

In general, at the time of analysing a property, a failed proof is discarded right away. However, having a *partial proof*, i.e. a proof which does not fully verify the property, can actually contribute a lot towards the analysis of software correctness, even if this proof was obtained by performing a low effort verification attempt, e.g. failing at automatically proving the property, and then not performing any interactive step in an attempt to finish the proof.

For instance, there are tools like *KeyTestGen* [13], which can be used to automatically generate test cases from a partial proof, in order to test the parts of the proof which cannot be verified. These test cases can be used to deal with the issue mentioned above. In general, if one of these test cases fails, then there is an error in the system. Otherwise, the prover does not have enough information to finish the proof. In addition, to check on other possible uses for partial proof one can also refer to [10, 11] regarding runtime monitors optimisation, and [38] regarding the enhancement of the test driven development methodology.

In this work we present a methodology that uses the power of partial proofs obtained from low effort (static) verification attempts, to infer global trace conditions for a system. Given a model describing (part of) the behaviour of the system, and a property π associated to a state of this model, our proposed methodology starts by doing a reachability analysis, i.e. an analysis to get all of the incoming transitions to the state. Next, the methods associated to the incoming transitions are used to statically verify that these transitions have the desired property as a postcondition, resulting in (possibly) local partial proofs for them. Then, closed-path conditions are extracted from the partial proofs, and are backwards propagated through the states of the model. Such conditions guarantee that, if one of these transitions is taken, a system trace fulfilling them will lead the system towards the desired state in the model, and π will hold when that state is reached. Therefore, closed-path conditions play the role of sufficient preconditions for the methods associated to the transitions, i.e. preconditions which come as a generalisation of the idea of weakest precondition for the methods. These steps are repeated until the initial state is reached. Finally, the property backwards propagated to the initial state, together with a system trace going from the initial state of the model to the desired state, which is created using the results of the reachability analysis previously performed, represent a trace condition for the system.

The main contributions of this work are:

- A methodology to infer global trace conditions from partial proof of local properties;
- The identification of applications for the inferred global trace conditions, including (global) test case generation, runtime verification, and state invariants verification.

Global test cases can be used for fault detection, and to guarantee global coverage for the methods of the system from the knowledge about certain testing coverage at a local (unit) level. Regarding runtime verification, global trace conditions can be

used to instrument the monitor with special artefacts for guiding the execution of the system. Finally, one use for state invariants is the analysis of explicit typestates for objects. Sec. 5.6 elaborates on this.

Structure of the paper. Sec. 5.2 introduces the notion of partial proof, and the power behind them. Sec. 5.3 describes the model used in this work to model system behaviour. Sec. 5.4 introduces the notion of global trace condition. Sec. 5.5 presents a methodology to infer global trace conditions from partial proofs. Sec. 5.6 elaborates on how one can use our proposed methodology in the context of testing, and (runtime) verification. Sec. 5.7 presents some case studies illustrating concrete uses of our methodology. Sec. 5.8 discusses related work, and Sec. 5.9 concludes the paper.

5.2 The Power of Partial Proofs

We start this section by describing what a partial proof is. Then, we proceed to talk about the power behind them.

5.2.1 Partial Proof

To talk about the power behind a partial proof, first, we have to elaborate on what a *partial proof* is. Here, we will use *dynamic logic* (DL). DL is a modal logic for reasoning about programs. This logic extends first-order logic with two modalities, $\langle p\rangle\phi$ and $[p]\phi$, where p is a program and ϕ is another DL formula. The formula $\langle p\rangle\phi$ is true in a state s if there *exists* a terminating execution of p, starting in s, which results in a state where ϕ holds. The formula $[p]\phi$ holds in a state s if *all* the terminating executions of p, starting in s, result in a state where ϕ holds. For deterministic programs p, the only difference between the two modalities is that termination is *stated* in $\langle p\rangle\phi$, and *assumed* in $[p]\phi$.

DL formulae can be proved using *deductive verification* [9, 56]. On this technique formulae are proved by deduction over a logic calculus. For instance, the deductive verifier KeY [9] is a tool for data-centric functional correctness properties of Java programs which, given a Java program with annotated properties written in the *Java Modelling Language* (*JML*) [72], generates proof obligations in (Java) DL from these annotations, and then uses a *sequent calculus* to attempt to verify them. Such a calculus focuses in the use of the *symbolic execution* paradigm [70].

Here, we consider the use of the sequent calculus mentioned above to elaborate on the definition of *partial proof*. However, we will not introduce this calculus. Instead, we will illustrate a rather simple example on how a sequent look like, and how such a sequent could be proved by using this calculus. Given a set of formulae Γ, the sequent $\Gamma \vdash \langle p\rangle\phi$ holds if p, when starting in a state fulfilling all formulae in Γ, terminates in a state fulfilling ϕ. In addition, due to the use of symbolic execution DL has to be extended by *explicit substitutions*. While symbolically executing p, its effects are gradually, starting from the front, turned into explicit substitutions. Thereby, after some proof steps, a certain prefix of p has turned into a substitution σ, representing the effects so far, while a remaining program p' is yet to be run. During the verification of p, an intermediate proof node may look like $\Gamma \vdash \sigma\langle p'\rangle\phi$. Such a node tells us that, if Γ was true before the original program p, and σ is the accumulated effect up to now, then ϕ will be true after the execution of the remaining program p'.

As an example, consider a proof of the following DL sequent:

$$\texttt{x} > 0, \texttt{y} > 0 \vdash \langle \texttt{x=y;y=x+1;x=y-x;if(y\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \qquad (5.1)$$

(where p_1, p_2, and q are (sequences of) program statements and ϕ is some postcondition). Sequent (5.1) says that in each state where `x` and `y` are positive, the program provided in the modality will terminate, and result in a state where ϕ holds.

The proof for the sequent above would start by, in a certain number of steps, turning the three leading assignments into explicit substitutions, apply the first to second one, apply the result of this substitution to the third, and perform some simplifications, arriving at

$$\texttt{x} > 0, \texttt{y} > 0 \vdash (\texttt{x} \leftarrow \texttt{y} \parallel \texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow \texttt{y-x})\langle \texttt{if(y\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (5.2)$$

where $(\texttt{x} \leftarrow \texttt{y} \parallel \texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow \texttt{y-x})$ represents the explicit (parallel) substitution obtained as a result from the symbolic execution of the first three statements. When clashes occurs in parallel substitutions, as it is the case for `x` in (5.2), a 'right-win' semantics is adopted in order to resolve them. Thereby, (5.2) is reduced to:

$$\texttt{x} > 0, \texttt{y} > 0 \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle \texttt{if(y\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (5.3)$$

In general, most proofs branch over case distinctions, often triggered by Boolean decisions in the source code. Such a branching occurs by applying rules like the following, simplified,[1] if rule:

$$\textsf{if}\ \frac{\Gamma, \sigma(b) \vdash \sigma\langle s_1 \ \ \omega\rangle\phi \qquad \Gamma, \sigma(\neg b) \vdash \sigma\langle s_2 \ \ \omega\rangle\phi}{\Gamma \vdash \sigma\langle \texttt{if}\ b\ s_1\ \texttt{else}\ s_2 \ \ \omega\rangle\phi}$$

In our example, applying the if rule to sequent (5.3) results in splitting the proof into two branches, with the following sequents, respectively:

$$\texttt{x} > 0, \texttt{y} > 0, (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)(\texttt{y\%2} = 0) \ \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle p_1\,; q\rangle\phi$$
$$\texttt{x} > 0, \texttt{y} > 0, (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)(\neg(\texttt{y\%2} = 0)) \ \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle p_2\,; q\rangle\phi$$

Applying the substitution on the left side of either sequent results in:

$$\texttt{x} > 0, \texttt{y} > 0, \texttt{(y+1)\%2 == 0} \ \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle p_1\,; q\rangle\phi \quad (5.4)$$
$$\texttt{x} > 0, \texttt{y} > 0, \neg(\texttt{(y+1)\%2 == 0}) \ \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle p_2\,; q\rangle\phi \quad (5.5)$$

Once all proof branches are closed, i.e. they are verified, we have a *complete proof* of the root sequent. However, a proof attempt may result into a proof object where not all the branches are closed. We refer to this kind of proofs as *partial proofs*. In addition, we refer to the set of formulae in a sequent, e.g. '$\texttt{x} > 0, \texttt{y} > 0, \texttt{(y+1)\%2 == 0}$' in sequent (5.4), as *branch condition*. In other words, a branch condition is a set of formulae leading the proof to one of its branches. Moreover, if the branch condition leads the symbolic execution of the program in the sequent towards a certain statement in its body, then we say that such a branch condition is a *path condition*.

5.2.2 Partial Proofs Capabilities

In the above example, consider a partial proof where sequent (5.4) is fully proved, but sequent (5.5) is only partially proved. From this partial proof, we can conclude that the following modification of the root sequent (5.1) is valid:

$$\texttt{x} > 0, \texttt{y} > 0, \texttt{(y+1)\%2==0} \vdash \langle \texttt{x=y;y=x+1;x=y-x;if(y\%2==0)\{}p_1\texttt{\}else\{}p_2\texttt{\};}q\rangle\phi \quad (5.6)$$

(We added `(y+1)%2==0` to the left side of (5.1), as additional assumption). This sequent can be proven by replaying the original proof, where now both branches

[1] The simplified rule ignores possibles side effects or exceptions which might be caused by b.

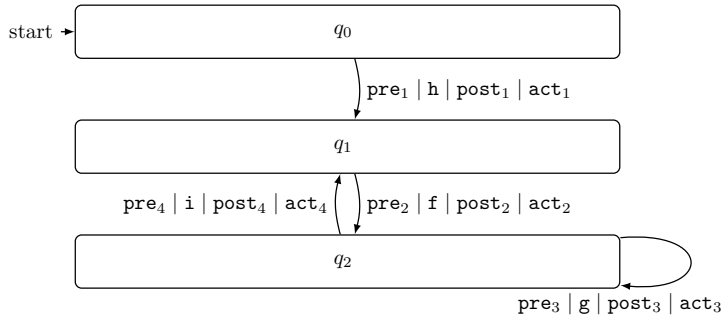


Figure 5.1: Model example

would close. The path leading to the closed proof of sequent (5.4) will replay identically. In addition, sequent (2.3) now can be proved because the following variant of sequent (6) can be closed immediately, due to contradicting assumptions:

$$\texttt{x} > 0, \texttt{y} > 0, \texttt{(y+1)\%2==0}, \neg(\texttt{(y+1)\%2 == 0}) \vdash (\texttt{y} \leftarrow \texttt{y+1} \parallel \texttt{x} \leftarrow 1)\langle p_2 ; q\rangle\phi$$

An interesting remark is that if we were using a *perfect* prover, i.e. a prover that only returns *false* whenever the program under scrutiny does not fulfil the property being analysed, then in sequent (2.4) the set of formulae '`x>0`, `y>0`,`(y+1)%2==0`', would correspond to the *weakest precondition* of the program in the modality. However, in an ordinary set up the prover may fail to close a proof branch even if the sub-goal on that branch is valid. This may happen because of either a lack of 'proving power', the strategies being used, a lack of code annotations (e.g. loop invariants), or alike. Still, even if the proof attempt fails, one can get close branches out of it. Thus, the branch condition of a closed branch forms a *sufficient precondition*, which is not necessarily the weakest precondition. Thereby, the true power behind partial proofs resides in the possibility of finding sufficient preconditions by extracting branch conditions from closed branches.

5.3 System Level Modelling

Before focusing our attention on trace conditions, we introduce a formalism to specify legal sequences of methods or states, i.e. a manner to describe valid traces of the system. Here, to model the behaviour of the system we will use a labelled transitions system based on the 'QuickCheck' models introduced by Falzon *et al.* in [55]. In these models, transitions are of the form $q \xrightarrow{\texttt{pre}|\texttt{foo}|\texttt{post}|\texttt{act}} q'$. A transition from state q to state q' is available to be taken when method `foo` is called in a configuration where the precondition *pre* is satisfied. When a transition is available, the postcondition *post* has to be checked once `foo` terminates its execution. If it holds, action *act* has to be executed, and q' is entered, i.e. the transition is taken. However, if *post* does not hold, then this is considered a failure in the execution. In addition, pre and postconditions may depend on system and model variables.

We assume that the arguments of the methods are implicitly included in the transitions right next to the method name, e.g. we write `foo` instead of `foo(x)`. Besides, we assume that actions in the transitions can modify only model variables, i.e. the model is restricted to not having any effect on the system. Formally,

Definition 27. Given a set of system variables *Sys* and a set of model variables V, a model m is a tuple (Σ, Q, t, q_0) such that:

- Σ is the set of method names.
- Q is the finite set of states.
- t is the transition relation among states in Q, where each transition is tagged with (i) a precondition; (ii) a method; (iii) a postcondition; (iv) an action which may change the valuation of the model variables: $t \subseteq Q \times cond_{Sys \cup V} \times \Sigma \times cond_{Sys \cup V} \times action \times Q$.
- $q_0 \in Q$ is the initial state.

We will write $q \xrightarrow{\texttt{pre}|\texttt{f}|\texttt{post}|\texttt{a}}_m q'$ to mean that, given a model m whose transition relation is t, $(q,\texttt{pre},\texttt{f},\texttt{post},\texttt{a},q') \in t$. We omit the subscript m whenever it is clear from the context. In addition, note that the notation $cond_{Sys \cup V}$ represents the syntactic category of *boolean expressions* over system and model variables.

As an example, Fig. 5.1 depicts a model consisting of three states q_0 (initial state), q_1, and q_2, and four transitions $q_0 \xrightarrow{\texttt{pre}_1|\texttt{h}|\texttt{post}_1|\texttt{act}_1} q_1$, $q_1 \xrightarrow{\texttt{pre}_2|\texttt{f}|\texttt{post}_2|\texttt{act}_2} q_2$, $q_2 \xrightarrow{\texttt{pre}_3|\texttt{g}|\texttt{post}_3|\texttt{act}_3} q_2$, and $q_2 \xrightarrow{\texttt{pre}_4|\texttt{i}|\texttt{post}_4|\texttt{act}_4} q_1$, among these states.

5.4 Traces and Trace Conditions

This section introduces the notion of *trace condition*. In Sec. 5.4.1 we define what a trace is, and we use traces to provide semantics for the models presented in Sec.5.3. In Sec. 5.4.2 we introduce the notion of a *trace condition*. Finally, in Sec. 5.4.3 we elaborate on the idea of backwards propagation of conditions.

5.4.1 Trace

In this work we consider that a *system trace* w, or simply *trace*, is a sequence of tuples consisting of a valuation of (visible) system variables, i.e. mappings from variables to values (of adequate types), a method, and a list of concrete values for its arguments. Formally,

Definition 28. Given the set of system variables Sys, a trace w is a sequence of tuples in $\Theta_{Sys} \times \Sigma \times \overline{cArgs}$, i.e. $w \in (\Theta_{Sys} \times \Sigma \times \overline{cArgs})^*$.
Note that we characterize the set of system variables valuations Θ_{Sys}, with typical element θ; and that $\overline{cArgs}$ represents a sequence of concrete values of adequate type for the arguments of the method in Σ.

For instance, assuming that each method has exactly one argument, the following expression depicts a trace associated to the model illustrated in Fig. 5.1: $\ll(\theta_0, h, c_0), (\theta_1, f, c_1), (\theta_2, g, c_2) \gg$, where c_0, c_1, and c_2 are actual parameters of adequate type.

In addition, we assume that values of model variables are stored in *valuations* ν, which are characterized by the set $\mathcal{V}$. As mentioned in Sec. 5.3, model variables are changed only in actions a of transitions, and that both actions and conditions in transitions can depend on system variables as well as on model variables. Given a condition *cond*, we write $(\theta, \nu) \models cond$ to denote that *cond* is satisfied by valuations θ and ν.

We can now define the semantics of a model by identifying how a trace generated by the system changes its states.

Definition 29. Given a model $m = (\Sigma, Q, t, q_0)$, a *trace* $w \in (\Theta_{Sys} \times \Sigma \times \overline{cArgs})^*$ *shifts a monitor from configuration* $(q, \nu) \in Q \times \mathcal{V}$ *to configuration* $(q', \nu') \in Q \times \mathcal{V}$,

written $(q,\nu) \overset{w}{\Rightarrow} (q',\nu')$, by induction over w:

$$
\begin{array}{l}
(q,\nu) \overset{\varepsilon}{\Rightarrow} (q',\nu') \overset{df}{=} q = q' \wedge \nu = \nu';\\
(q,\nu) \xRightarrow{(\theta,f,\overline{args}):w} (q',\nu') \overset{df}{=} \exists\, q'',\nu'', pre, post, a \cdot\\
\quad q \xrightarrow{\texttt{pre|f|post|a}}_m q'' \wedge \theta \uplus \nu \models pre \wedge \nu'' = a(\nu) \wedge eval(f,\theta) \uplus \nu'' \models post\\
\quad \wedge\ (q'',\nu'') \overset{w}{\Rightarrow} (q',\nu');\\
(q,\nu) \xRightarrow{(\theta,f,\overline{args}):w} (q',\nu') \overset{df}{=} (q,\nu) \overset{w}{\Rightarrow} (q',\nu') \wedge \nexists\, q'', v'', pre, post, a \cdot\\
\quad q \xrightarrow{\texttt{pre|f|post|a}}_m q'' \wedge \theta \uplus \nu \models pre.
\end{array}
$$

where the operator $\uplus$ denotes the union of valuations, such that $\theta \uplus \theta' = \{(x,val) \mid (x,val) \in \theta \text{ or } (x,val) \in \theta'\}$. In addition, a trace $w \in (\Theta_{Sys} \times \Sigma \times \overline{cArgs})^*$ is said to be valid if, for some $q \in Q$ and valuation ν, $(q_0,\nu_0) \overset{w}{\Rightarrow} (q,\nu)$ holds, where ν_0 represents the initial valuation of the model variables.

We will write $a(\nu)$ to represent the model variables valuation obtained from the execution of action a, in terms of valuation ν. In addition, the operator $eval : \Sigma \times \Theta_{Sys} \to \Theta_{Sys}$, given a method f and a system variables valuation θ, returns the system variables valuation obtained after the execution of f, in terms of θ. Besides, the previously introduced notation $\ll(\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n)\gg$, is simply syntactic sugar to represent the trace $(\theta_0, f_0, \overline{c}_0) : \cdots : (\theta_n, f_n, \overline{c}_n) : \varepsilon$.

Note that a model only describes legal behaviour of a system. Therefore, the case where there exist a transition whose precondition holds, but the postconditions does not hold once the corresponding method terminates its execution, i.e. non legal behaviour, it is not covered by the semantics provided in the definition above.

Finally, we say that a trace w leads a model to a state q, if w is a valid and q is the last state reached by it. Formally,

Definition 30. Given a model $m = (\Sigma, Q, t, q_0)$, and a state $q \in Q$, a trace w leads m to state q, written $\mathsf{leads}_m(w,q)$, if:

$$\mathsf{leads}_m(w,q) \overset{df}{=} \exists\, \nu_0, \nu \cdot (q_0,\nu_0) \overset{w}{\Rightarrow} (q,\nu).$$

5.4.2 Trace Condition

Before introducing the concept of *trace condition*, first, we have to introduce some notation. The expression $\varphi(x, y, \cdots, z)$ represents a *first order logic* (FOL) formula with free variables $x, y, \cdots, z$. Similarly, the expression $\varphi(\overline{x})$ represents a FOL formula whose free variables are, at most, the variables in $\overline{x}$, i.e. $FreeVars(\varphi) \subseteq \overline{x}$.

Now we can proceed to define trace conditions. Trace conditions are properties which impose restrictions over a sequence of tuples consisting of methods and the list of their arguments; and the (initial) values of the variables of the system. Formally,

Definition 31. Given a sequence $\ll(f_0,\overline{x}_0), \cdots, (f_n,\overline{x}_n)\gg \in (\Sigma \times \overline{args})^*$, a FOL formula φ, the set of system variables Sys, and assuming that $\overline{x}_i$ are the arguments of method f_i, i.e. $f_i(\overline{x}_i)$, with no overlapping names w.r.t. to the arguments of the others methods, a trace condition is an expression of the form:

$$\varphi(\overline{Sys}, \overline{x}_0, \cdots, \overline{x}_n) \bullet \ll(f_0,\overline{x}_0), \cdots, (f_n,\overline{x}_n)\gg$$

We refer to $\varphi(\overline{Sys}, \overline{x}_0, \cdots, \overline{x}_n)$ as '*trace property*', and to $\ll(f_0,\overline{x}_0), \cdots, (f_n,\overline{x}_n)\gg$ as '*trace pattern*'.

For instance, assuming that each method has only one argument of type integer, and that the system has a variable v, then the following expression depicts a trace condition associated to the model illustrated in Fig. 5.1:

$\varphi(v, x_0, x_1, x_2)\bullet \ll (h, x_0), (f, x_1), (g, x_2) \gg$, where $\varphi(v, x_0, x_1, x_2) = v > 10 \wedge x_0 > 0 \wedge x_1 < x_2$.

It is not hard to notice in this example that the trace pattern consists of a sequence of the methods of the system paired up with their arguments, and that the trace property involves the use of all of such arguments. Thus, trace conditions impose restrictions on traces in a non local manner, i.e. trace conditions are *global* to the system as their restrictions are imposed along the whole trace.

Regarding their satisfaction, we say that a trace condition holds if there exists a trace that models it. Formally,

Definition 32. Given a sequence $\ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg \in (\Sigma \times \overline{args})^*$, and a trace condition $\varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)\bullet \ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg$, this trace condition is satisfied by a trace $\ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg$ if,

$$\theta_0 \uplus [x_0 \mapsto \overline{c}_0, \cdots, x_n \mapsto \overline{c}_n] \models \varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)$$

We will use the following notation to represent that a trace condition is satisfied by a given trace,

$\ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg \models_{tc} \varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)\bullet \ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg$.

In addition, a trace condition $\varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)\bullet \ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg$ is *sufficient* for a state q of a model m, and a FOL formula ϕ local to q, if it is satisfied by any trace $\ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg$, such that this trace leads m to q, and ϕ is modelled by θ_0. We will refer to these kind of trace conditions as *sufficient trace conditions*. Formally,

Definition 33. Given a model m, a state q of m, and a FOL formula ϕ local to q, a trace condition $\varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)\bullet \ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg$ is sufficient for q and ϕ if:

$\forall \ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg \cdot$
$\ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg \models_{tc} \varphi(Sys, \overline{x}_0, \cdots, \overline{x}_n)\bullet \ll (f_0, \overline{x}_0), \cdots, (f_n, \overline{x}_n) \gg$
$\Rightarrow \mathsf{leads}_m(\ll (\theta_0, f_0, \overline{c}_0), \cdots, (\theta_n, f_n, \overline{c}_n) \gg, q) \wedge eval(f_n, \theta_n) \models \phi$

Thereby, a sufficient trace condition guarantees that state q will be reached in m, and that ϕ will be fulfilled when that occurs.

Regarding applications, trace conditions can be used in the context of testing, runtime verification, and static verification. In this work, we only focus on applications that are benefited from the use of our trace conditions inference methodology. In particular, inferred trace conditions are used for generating global test cases, guiding the execution of the system while monitoring properties, and analysing the use of explicit typestates for objects. Sec. 5.6 elaborates on each one of these applications in detail.

5.4.3 Backwards Computation of Trace Conditions

The main contribution of this work is the inference of global trace conditions from local partial proofs. Given a system, a model m describing its behaviour, the set Sys of system variables, and a FOL formula $\varphi_0(\overline{Sys})$, local to a state q of m, we want to compute conditions from the (partial) proof of φ_0, and *backward propagate* them through m, in order to generate trace conditions for the system. Such trace conditions have to lead the execution of the system towards reaching q, in a manner that φ_0 will be fulfilled when q is reached, i.e. they will be sufficient trace conditions for q and φ_0.

From a trace condition perspective, φ_0 can be seen as $\varphi_0(\overline{Sys})\bullet \ll \gg$. Remember that we will perform a backwards computation of the trace condition. Thus, we start

the computation in state q, i.e. the state to be reached. Thereby, no methods have been considered as part of the sequence in the trace condition yet.

Now let us assume the existence of the transition $q' \xrightarrow{pre_0|\mathtt{f}_0|post_0|a_0}_m q$. Then, we can use a verifier to generate a sufficient precondition (see Sec 1.3.2) for method $\mathtt{f}_0$ from a local (partial) proof of φ_0.

Such a precondition will be (possibly) described in terms of both the system variables and the arguments of $\mathtt{f}_0$. Thus, we represent it as the FOL formula $\varphi_1(\overline{Sys}, \overline{x}_0)$. Thereby, the previous trace condition can be changed to $\varphi_1(\overline{Sys}, \overline{x}_0)\bullet <$ $< (\mathtt{f}_0, \overline{x}_0) \gg$. Such a trace condition means that if we are in state q', φ_1 holds, and method $\mathtt{f}_0$ is executed, then we will reach state q in a set up where φ_0 is fulfilled.

By applying a reasoning in state q' alike the one above, assuming the existence of a transition $q'' \xrightarrow{pre_1|\mathtt{f}_1|post_1|a_1}_m q'$, we can get the trace condition $\varphi_2(\overline{Sys}, \overline{x}_1, \overline{x}_0)\bullet <$ $< (\mathtt{f}_1, \overline{x}_1), (\mathtt{f}_0, \overline{x}_0) \gg$. This trace condition means that whenever we are in state q'', φ_2 holds, and methods $\mathtt{f}_1$ and $\mathtt{f}_0$ are ran successively, then we will reach state q in a set up where the property φ_0 is fulfilled.

Hence, we can continue applying a similar reasoning multiple times until reaching the initial state of the model. This will result in the generation of a trace condition $\varphi_n(\overline{Sys}, \overline{x}_{n-1}, \cdots, \overline{x}_1, \overline{x}_0)\bullet \ll (\mathtt{f}_{n-1}, \overline{x}_{n-1}), \cdots, (\mathtt{f}_1, \overline{x}_1), (\mathtt{f}_0, \overline{x}_0) \gg$. Similar to what we have described above, this trace condition means that if we are in the initial state of m, φ_n holds, and all methods $\mathtt{f}_i$ are ran successively following the order established by the sequence in the trace condition, then we will go through all the intermediate states of the model until reaching state q. Such state will be reached in a set up where the property φ_0 is fulfilled. Thus, we inferred a trace condition which, under the assumption that φ_n holds for some trace, is a sufficient trace condition.

Note that in the previous computation we have not considered the possibility of having loops in the model. In sec. 5.5, we elaborate on how one can deal with models containing loops when backwards propagating conditions trough their states.

5.5 Inferring Trace Conditions

In this section we propose a backwards computation methodology to infer (sufficient) global trace conditions from partial verification of a FOL formula of local method calls. This methodology consists of the following three stages: *Reaching Transitions Analysis*, *Backwards Reachability Tree Computation*, and *Trace Condition Inference*. In short, in the first stage we extract information from the model about the reachability of its states. This information is used in the following stage, in addition to sufficient preconditions extracted from (partial) deductive proofs, to compute a backwards reachability tree. Finally, in the last stage the backwards reachability tree is analysed in order to infer trace conditions for the system. Below, first, we provide an intuitive example for the use of our methodology. Then, we describe each one of the stages mentioned above in detail.

5.5.1 Example

In this example we will consider the model illustrated in Fig. 5.1, assuming that each method has exactly one argument x_i, where i is the name of the corresponding method, that the system has only one variable var, that we want a FOL formula $\varphi_0(var)$ to be fulfilled when reaching state q_2, and that each loop in the model can be traversed only once.

To begin with, we extract the reachability information from the model and generate the following mapping `it`, representing the incoming transitions of the state provided as argument:

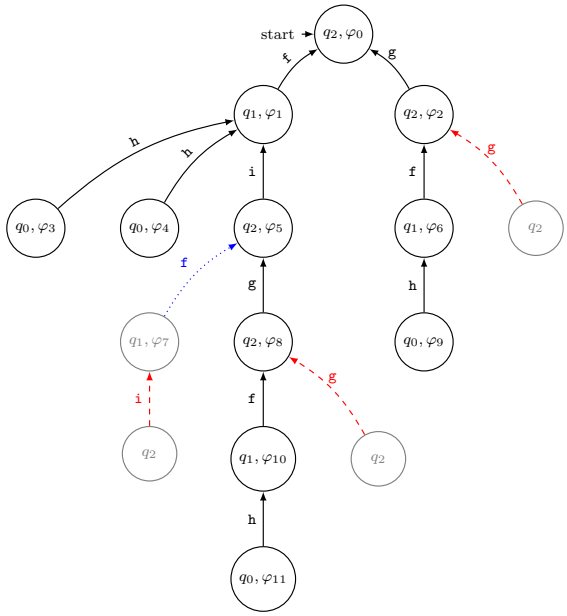


Figure 5.2: Backwards reachability tree computation example

$$\texttt{it}(q_2) = \{\ q_2 \xrightarrow{\texttt{pre}_3|\texttt{g}|\texttt{post}_3|\texttt{act}_3} q_2\ ,\ q_1 \xrightarrow{\texttt{pre}_2|\texttt{f}|\texttt{post}_2|\texttt{act}_2} q_2\ \}$$
$$\texttt{it}(q_1) = \{\ q_2 \xrightarrow{\texttt{pre}_4|\texttt{i}|\texttt{post}_4|\texttt{act}_4} q_1\ ,\ q_0 \xrightarrow{\texttt{pre}_1|\texttt{h}|\texttt{post}_1|\texttt{act}_1} q_1\ \}$$
$$\texttt{it}(q_0) = \{\ \}$$

Basically, mapping `it` says that both state q_1 and q_2 have two incoming transitions, and that the (initial) state q_0 has none.

Next, we compute the *backwards reachability tree* (BRT). Let us assume that Fig. 5.2 depicts the result obtained from this computation, by allowing only one iteration for the loops in the model. In this figure, nodes contain their corresponding state (in the model), and the FOL formula to be fulfilled when reaching them, i.e. either the initial trace property, or a condition backwards propagated to the state. Moreover, transitions are of the form $node_2 \xrightarrow{\texttt{foo}} node_1$, meaning that the current state in $node_1$ can be reached from the current state of $node_2$ when method `foo` is executed. In addition, we assume that, as explained below, during the computation of this BRT, dashed (red) transitions were filtered, and the dotted (blue) transition was pruned. This, led to removed the nodes in gray from the tree.

The construction of this tree starts by setting up the root for the BRT. The root consists of a node with current state q_2 (target state), and FOL formula φ_0. We will label this node by the tuple (q_2, φ_0). A similar representation will be used for the other nodes. Note that, for simplicity, we have omitted the arguments of the property in the description of the node.

Then, from the mapping `it` we know that q_2 could be reached from q_1 running `f`, and from itself by running `g` (loop transition). Thus, we proceed by trying to verify the Hoare triples $\{true\}\,\texttt{f}(x_f)\,\{\varphi_0(var)\}$ and $\{true\}\,\texttt{g}(x_g)\,\{\varphi_0(var)\}$. Let us assume that the failed (low effort) verification attempt of each one of the previous Hoare triples results in the generation of the sufficient preconditions $\varphi_1(var, x_f)$ and $\varphi_2(var, x_g)$ for methods `f` and `g`, respectively. Thus, the nodes (q_1, φ_1) and (q_2, φ_2) are created, and added to the BRT with transitions $(q_1, \varphi_1) \xrightarrow{\texttt{f}} (q_2, \varphi_0)$ and $(q_2, \varphi_2) \xrightarrow{\texttt{g}} (q_2, \varphi_0)$.

Note that one node will be created for each extracted sufficient precondition. For instance, while being in node (q_1, φ_1), if the proof of $\{true\}\,\mathtt{h}(x_h)\,\{\varphi_1(var, x_f)\}$ has two closed branch conditions, then, as illustrated in Fig. 5.1, we create a node for each extracted sufficient precondition, e.g. φ_3 and φ_4.

We iterate over similar steps for computing the rest of the BRT. Still, it is worth mentioning two more scenarios: transition filtering, and both transition and state pruning.

Regarding transition filtering, we have to do this whenever adding the transition to the BRT would generate a trace condition whose trace pattern would perform an extra iteration through a loop in the model. For instance, if in node (q_2, φ_2) we consider the transition for method `g`, then the loop transition associated to this method would be traversed twice. However, we have assumed that loops can only be traversed once. Thus, that transition has to be filtered.

Regarding pruning, whenever the computation of a branch in the BRT finishes in a node whose current state is not the initial state of the model, that node has to be pruned from the BRT. In addition, we have to go up traversing the branch removing nodes and transitions until either reaching a node whose current state is the initial state, or the node has more than one incoming branch. For instance, let us consider the node (q_1, φ_7), which can be reached in the BRT by traversing the following sequence of nodes: $\ll(q_0, \varphi_0), (q_1, \varphi_1), (q_2, \varphi_5), (q_1, \varphi_7)\gg$. On this node, we assume that the only method with a closed branch condition is `i`. However, as executing this method would trigger an extra traversal of a loop in the model, the transition associated to this method has to be filtered. This means that no transitions are available to reach the current state of the node. Thus, as the current state of node (q_1, φ_7) is not the initial state, this state is pruned. This leads to pruning the transition $(q_1, \varphi_7) \xrightarrow{\mathtt{f}} (q_2, \varphi_5)$. Finally, as the node (q_2, φ_5) has a bifurcation, the pruning is over.

Finally, we analyse the generated BRT in order to infer trace conditions from it. By doing depth-first search in the BRT, we can obtain the following sequence of nodes:
$\ll(q_0, \varphi_3), (q_1, \varphi_1), (q_2, \varphi_0)\gg$
$\ll(q_0, \varphi_4), (q_1, \varphi_1), (q_2, \varphi_0)\gg$
$\ll(q_0, \varphi_{11}), (q_1, \varphi_{10}), (q_2, \varphi_8), (q_2, \varphi_5), (q_1, \varphi_1), (q_2, \varphi_0)\gg$
$\ll(q_0, \varphi_9), (q_1, \varphi_6), (q_2, \varphi_2), (q_2, \varphi_0)\gg$

Then, as explained in Sec. 5.5.4, by considering these sequences of nodes, the following trace conditions can be inferred, respectively:
$\varphi_3(var, x_h, x_f)\bullet\ll(h, x_h), (f, x_f)\gg$
$\varphi_4(var, x_h, x_f)\bullet\ll(h, x_h), (f, x_f)\gg$
$\varphi_{11}(var, x_h, x_g, x_i, x_f)\bullet\ll(h, x_h), (f, x_f), (g, x_g), (i, x_i), (f, x_f)\gg$
$\varphi_9(var, x_h, x_f, x_g)\bullet\ll(h, x_h), (f, x_f), (g, x_g)\gg$

In conclusion, we went from having a FOL formula $\varphi_0(var)$ local to the state q_2, to have four trace conditions imposing global restrictions, i.e. the trace property over trace patterns.

5.5.2 Reaching Transitions Analysis

We start applying our methodology by extracting information from the model regarding how to reach each one of its states. Let us assume that we have a model $m = (\Sigma, Q, t, q_0)$. Then, basically, for each state $q \in Q$, we want to know which are the transitions leading to it. The information extracted from the model will be represented with a mapping of signature $\mathtt{rt} : Q \to 2^{Transition}$, such that,

$$\mathtt{rt}(q) = \{q' \xrightarrow{\mathtt{pre|foo|post|act}}_m q \;\mid\; q' \in Q,\; q' \xrightarrow{\mathtt{pre|foo|post|act}}_m q \in t\}$$

```
Node = {
  parent   : Node,
  children : Set<Node>,
  initial  : State
  current  : State,
  property : Property,
  method   : Method * List<Args>
  iter     : Int
}
```

Figure 5.3: Record `Node` used to represent backwards reachability trees

5.5.3 Backwards Reachability Tree Computation

In this stage, we proceed to compute the BRT. Given a model m, and two of its states q and q', a BRT to get from q to q' corresponds to a tree structure where each path from the root of the tree to its leaves represents a reversed path through the states and transitions of m to get to state q', starting in state q. In other words q is in the leaves of the tree, and q' is in its root. We start computing the tree from a (root) node representing q', i.e. the state which we want to reach, and each branch goes on until arriving at a (leaf) node representing q, i.e. the state where the path starts. Hence, the BRT denomination for this tree.

To perform such computation we will use record `Node`, which is (abstractly) depicted in Fig. 5.3. This record can be used as a tree structure, where `parent` points to the parent of the node, and `children` is the set of all the children of the node. In addition, field `initial` is used to keep track of the initial state of the model, field `current` represents the state of the model associated to the node, field `property` represents either the FOL formula to analyse in the root, or the backtracked (branch) condition to the node, field `method` represents the method (and its arguments) that has to be executed to reach its parent, and field `iter` represents the allowed amount of iterations over loops in the model.

Having said this, we proceed to elaborate on the computation of the BRT. Algorithm 1 represents its whole computation. Given a state *initial*, i.e. the state where the paths start, a state *toreach*, i.e. the state to be reached at the end of the paths, a FOL formula φ associated to *toreach*, and the integer *iterations*, i.e. the amount of iterations that each loop can be traversed on a path, this algorithm starts computing the BRT by creating its root. Here, *nil* represents absence of data, i.e. the root of the tree has neither a parent, nor method associated to it. Once the root is defined, function `brt` is applied to it, and finally the computed BRT is returned. Note that the application of function `brt` is technically the actual computation of the BRT.

Separating the creation of the root from the computation of the body simplifies the whole computation of the BRT. Below, we provide two pseudo algorithms illustrating different versions of function `brt`: one where the models do not have any loops, i.e. Algorithm 2, and one where the models may contain loops, i.e. Algorithm 3.

Algorithm 2 starts by using the mapping `rt` to get the reachability information for the current state of *node*, i.e. from which states in the model the *node.current* can be reached. These results are stored in the set *reachable*.

Then, for each transition *tr* on this set we proceed by attempting to verify whether the method in *tr* establishes the FOL formula *node.property*. This can be simply done by trying to prove the Hoare triple $\{true\}\, tr.method\, \{node.property\}$. A Hoare

Algorithm 1: Backwards Reachability Tree

Input: State *initial*, State *toreach*, Property φ, Int *iterations*
Output: Backwards reachability tree
begin
  root ⟵ `Node` { *nil*, $\emptyset$, *initial*, *toreach*, φ, *nil*, *iterations* }
  return `brt`(*root*)

Algorithm 2: `brt` (No Loops Version)

Input: Node *node*
Result: Backwards reachability tree computation
begin
  reachable ⟵ `rt`(*node*.`current`)
  for *tr* ∈ *reachable* **do**
    conditions ⟵ *verify*(*node*.`property`, *tr*)
    if *conditions* = $\emptyset$ **then**
      continue
    newNodes ⟵ *makeNodes*(*conditions*)
    addChildren(*node*, *newNodes*)
  map `brt` *node*.`children`
  return

triple $\{\pi\}\, m\, \{\pi'\}$ means that, if m is invoked and π holds at the time of its invocation, then π' should be satisfied upon termination of the execution of m. Here, we assume that method *verify* represents the call to a deductive verifier. This method receives as arguments the property to be verified, i.e., *node.property*, and the transition to be analysed, i.e., *tr*, and returns a set of sufficient preconditions for the method *tr.method*. Note that one could also use the pre and postconditions in *tr*, to *refine* the triples. This is the reason why *verify* receives the whole transition as an argument, instead of just receiving the method associated to it.

The previous verification attempt will result into either a complete, or a partial proof, from which one should extract *closed branch conditions*, i.e. branch conditions associated to closed branches of the proof. These closed branch conditions, in conjunction to the precondition of the Hoare triple, correspond to a sufficient precondition for *tr.method*. These results are stored in the set *conditions*.

Next, we analyse whether the set *conditions* is empty or not. In the affirmative case, there are no sufficient preconditions that we can use to compute the BRT. In that case, we proceed to analyse the following element in *reachable*. Otherwise, we have sufficient preconditions which can be used in the BRT computation. For each one of these conditions a new node has to be created, and added to the set of children of *node*. Each one of these nodes will belong to a different branch of the BRT. We assume that the operator *makeNodes* takes care of creating a set with all the new nodes from the set of conditions provided as an argument to it; and the operator *addChildren* receives *node* and a set of nodes *newNodes* as arguments, and adds *newNodes* into the set of children of *node*.

Finally, once all the elements in set *reachable* are analysed, function `brt`, is applied to all the children of *node* to proceed with the computation of the BRT. The computation of (each branch of) the BRT will continue until the initial state

Algorithm 3: `brt` (Loops Version)

```
Input: Node node
Result: Backwards reachability tree computation
begin
    candidates ⟵ rt(node.current)
    reachable ⟵ filter(candidates, node.iter)
    if reachable = ∅ then
        prune(node)
        return
    for tr ∈ reachable do
        conditions ⟵ verify(node.property, tr)
        if conditions = ∅ then
            continue
        newNodes ⟵ makeNodes(conditions)
        addChildren(node, newNodes)
    map brt node.children
    return
```

of the model is reached. In this case, as there are no loops in the model, the initial state cannot be reached from any other state. Thus, it is not possible to continue backwards propagating properties. Therefore, the computation terminates.

Regarding Algorithm 3, it corresponds to an extension of Algorithm 2 which takes into account the possibility that a model may contain loops.

To begin with, this algorithm uses the mapping `rt` to get the reachability information for the current state of *node*. However, as it may be the case that by considering certain transitions, the model would exceed the allowed amount of iterations when traversing the loops, the reachability information has to be filtered to prevent this issue from ever happening. Thereby, the information stored in *reachable* now corresponds to the filtered reachability information for the current state of *node*. We assume that operator *filter* receives as an argument the set of transitions to be analysed, i.e., *candidates*, and discards the appropriate transitions.

Next, we proceed to analyse if the set *reachable* is empty, or not. In the affirmative case, all of the possible transitions to reach the current state of *node* were filtered. Therefore, we stop computing the branch of the BRT associated to *node*. However, as the current state of all the leaves of a BRT has to be the initial state, we may have to start removing nodes from the branch until reaching either (I) a node whose current state is the initial one, or (II) a node which has a bifurcation, i.e. it is part of another path. We assume that operator *prune* takes care of this task. This operator receives as an argument the node to be removed from the BRT, i.e., *node*, and it removes this node and all is predecessors until reching either scenario (I) or (II).

Then, the remainder of this algorithm continues alike Algorithm 2. Note that the computation of (each branch of) the BRT will continue until any of the following four scenarios occurs: (i) the initial state of the model is reached and there are not loops in the model; (ii) the initial state of the model is reached and there are loops in the model, but the initial state cannot be reached from any other state; (iii) a node has to be pruned and the initial state is reached while pruning ; or (iv) a node has to be pruned and its father has more than one child. In (i) and (ii) the computation terminates because it is not possible to continue backwards propagating properties

through the model. In (iii) the computation terminates because the initial state is reached, and a trace condition can now be generated for that branch of the BRT. In (iv) the computation terminates because the pruning of nodes reaches a point of the BRT where the father of the node has more than one child, i.e., the father should not be removed from the BRT as it is part of another path. Thus, the pruning stops.

5.5.4 Trace Condition Inference

Finally, once we have computed the BRT, one can use a *depth-first search* (DFS) algorithm to get sequences of nodes representing paths from the root of the tree to its leaves.

Let us assume that the sequence of nodes $\ll n_k, \cdots, n_0 \gg$ was obtained by using DFS on the BRT, and n_0 is the root of the tree. Then, this sequence represents a path from the state *initial* to state *toreach*, in a set up where φ holds when *toreach* is reached. In particular, if we think in terms of trace conditions, the sequence $\ll n_k.\texttt{method}, \cdots, n_1.\texttt{method} \gg$ corresponds to the trace pattern, and $n_k.\texttt{phi}$ represents the trace property, which has to be initially fulfilled in order to, later, reach the chosen state while satisfying φ. Thus, from the previously obtained sequence of nodes, we get the following trace condition:

$$n_k.\texttt{phi}\bullet \ll n_k.\texttt{method}, \cdots, n_1.\texttt{method} \gg$$

In addition, as all the backwards propagated conditions through the model were sufficient preconditions, if this trace condition is satisfiable, i.e. $n_k.\texttt{phi}$, then we can conclude that it is actually a sufficient trace condition.

5.6 Applications

In this section we elaborate on possible applications for our proposed methodology (see Sec. 5.5) in the context of testing, and (runtime) verification.

5.6.1 Global Test Case Generation

Testing is an area where proofs can be used to (automatically) generate unit test cases from a formula. In general, some prover which applies the symbolic execution paradigm [70], e.g. deductive verifier KeY [9], is used in an attempt of verifying the formula. Then, test cases can be (automatically) generated for each branch of the proof.

By following a similar proof-based approach, if we consider a FOL formula as the initial formula in our methodology, then we can infer trace conditions and use them to (automatically) generate global test cases for a particular run of the system. Thus, we can go from generating test cases focused on a system unit, e.g. testing a method, to generating test cases focused on a system component, e.g. testing the interaction between different methods of the same object.

Let us come back to the example introduced in Sec. 5.5.1. To generate global test cases from the trace conditions $\varphi_9(var, x_h, x_f, x_g)\bullet \ll (h, x_h), (f, x_f), (g, x_g) \gg$, we could generalise the manner in which *KeyTestGen* [13] automatically generates unit test cases from proofs.

Considering the trace property φ_9 as a *test data constraint*, i.e. a constraint which must be satisfied by the initial data to be generated for the test case, we start by generating concrete data satisfying it. This concrete data corresponds to a first order interpretation of φ_9. *KeyTestGen* finds such interpretations with the help of Z3 [50].

The next step corresponds to the generation of the oracle from φ_0 (i.e. the initial formula). As φ_9 was obtained through backwards propagation, we already know

that if the initial data satisfies it, then φ_0 will be fulfilled when state q_2 is reached. Therefore, one could conclude that it is enough to create a trivial oracle for the test case, i.e. an oracle that returns always true. However, creating the actual oracle from φ_0 works as a sanity check for the test cases. In particular, whenever the provided model does not describe the whole behaviour of the system, including an actual check for φ_0 as oracle for the test case could help us to detect unexpected issues (see Sec. 5.7.2).

Finally, the test case is created in two stages: *preamble*, and *execution of the methods*. The objective of the preamble is to reproduce the initial state of the first order interpretation. Here, all the necessary Java objects are created, and the different program variables and fields are initialise with concrete data. Regarding the execution of the methods, an executable environment is created to run all of the methods in the trace pattern, e.g. $\ll(h, x_h), (f, x_f), (g, x_g)\gg$.

Therefore, the successful generation and check of such a test case would guarantee that it is possible to get an initial configuration for the system, whose execution would lead to state q_2 in a set up where φ_0 is fulfilled. Thus, the generated test case offers guarantees for a whole execution of the system, i.e. it is global.

Regarding the use of global test cases, one may consider using the knowledge about certain testing coverage for the methods of the system at a local (unit) level to guarantee a global coverage. For instance, let us assume that the execution paths p_1, p_2, and p_3, guarantee full path coverage for a certain method `foo`. In addition, if the initial values used to test `foo` fulfil a condition π_1, then the execution path p_1 is traversed. Similarly, condition π_2 leads to the traversal of p_2, and π_3 leads to the traversal of p_3. Then, by considering any of the previous conditions as the initial formula in our methodology, if we succeed to generate trace conditions from every condition, then the set of all the inferred trace conditions guarantees that there exists system traces such that every execution path for `foo` will be traversed when reaching the appropriate state in the model. Thus, our methodology guarantees a full global path coverage, provided a full unit path coverage was accomplished.

Another possible use for global test cases is fault detection. Basically, we can use as initial formula in our methodology a comparison of a field against an invalid concrete value. Then, succeeding to infer a trace condition from that formula would mean that there is an error in the system, given that there exists a trace (pattern) leading the execution of the system towards the desired state, but when it is reached the field under scrutiny has an invalid value assigned to it. Later, by analysing the global test case generated from the infer trace condition one can analyse the system globally, looking for the fault.

5.6.2 Runtime Verification

Runtime verification (RV) [63,74] is a technique concerned with the runtime monitoring of software executions. It detects violations of properties that occur while the program under scrutiny is executed. One downside of this technique is that it introduces some overhead to the execution of the system which, in some cases, may cause trouble, e.g. when the response time of a system is critical and the added overhead delays it. In addition, when RV is used before system deployment in order to analyse certain system behaviour, one has to come up with appropriate (system) traces to monitor.

In [55] Falzon *et al.* show how to translate a model like the one we have introduced in Sec. 5.3 to a runtime monitor specification. By following these ideas, we can generate a runtime monitor which checks the desired initial FOL formula when reaching the appropriate state in the monitor by following a trace pattern.

In addition, when analysing an interactive application, by considering the trace conditions inferred with our methodology, the monitor can be instrumented with artefacts for guiding the execution of the system. For instance, when reaching a

state in the automaton, the monitor could add an entry in a log suggesting which methods would be recommended to run next. Basically, it would suggest to run any of the methods for which a sufficient precondition was backwards propagated to that state. Thereby, following the monitor suggestions would be the same as traversing a trace pattern. Thus, the execution of the system is led by the monitor towards reaching a desired state. Hence, guided RV would simplify the task of coming up with appropriate (system) traces for monitoring the system when using RV before system deployment.

5.6.3 State Invariants Verification

The previously described applications allow us to check whether there exists an execution of the system such that a FOL formula φ is fulfilled whenever certain state q of the model is reached. However, it would also be interesting to know whether φ is fulfilled for every single execution of the system reaching q, and as long as the model resides in q, i.e. whether φ is a local *state invariant*.

By extending Algorithm 3 in our methodology (see Sec. 5.5) with an analysis of *invariability* for φ, we can statically verify whether this FOL formula is a state invariant or not. Here, by invariability we mean that the execution of any of the methods in the system is not going to interfere with the truth value of the FOL formula while residing in a certain state of the model.

Basically, we start by inferring trace conditions from a BRT computed considering the invariability analysis. Next, if the trace patterns of all the these trace conditions cover every path in the model, i.e. we can get from the initial state to state q in every possible manner, and every trace pattern holds initially for every (system) trace, then φ is a state invariant, as we can generate test cases from the trace conditions which guarantee that φ is fulfilled when q is reached, the invariability analysis guarantees that the methods of the system are not going to interfere with φ truth value while being on q, and q can be reached from the initial state through every possible path.

One possible use for state invariants could be the analysis of explicit *typestates* for objects [52, 86]. Typestates are properties of objects which describe valid sequences of methods calls. Basically, typestates can be interpreted as automata which describe the legal behaviour of the instances of an object regarding the appropriate manner in which the methods of the object can be called.

Some applications add explicit additional fields to their specifications in order to describe and analyse typestates, e.g. Mondex [88]. Thereby, if the behaviour described by the provided model coincides with the behaviour described by the typestates, then the value of the field used to control the typestate should be fixed in each one of the states of the model. Thus, an equality comparison between the field controlling the typestate and its appropriate (concrete) value should be a state invariant of the corresponding state in the model, as their value should always coincide.

Note that checking for invariability may not be an easy task. One would have to analyse all of the methods of the system, to know which methods may interfere with the system variables. Then, if the property to be analysed in certain node depends on system variables, one would have to check whether it is possible to run one of the interfering methods in the current state of the node. If so, then the property cannot be considered a state invariant, as its system variables could be modified while residing in that state.

5.7 Evaluation

This section depicts concrete examples on the use of our methodology, regarding the applications introduced in Sec. 5.6. Here, we will consider the analysis of two Java

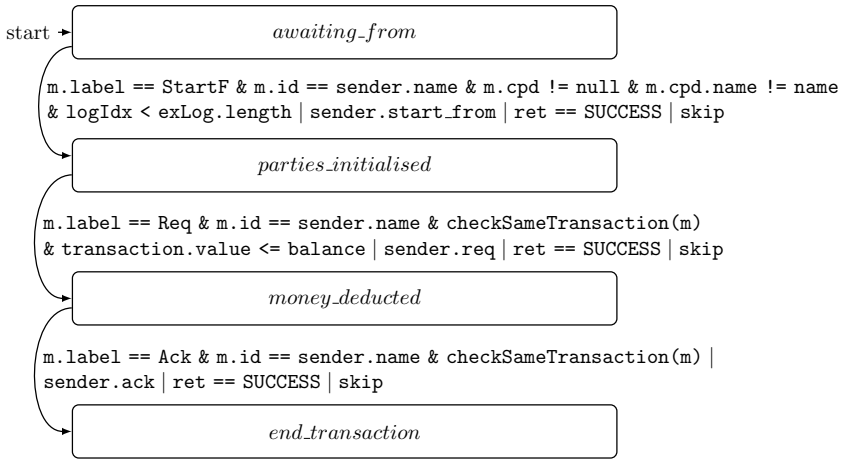


Figure 5.4: Model for sender purse in Mondex

applications: Mondex, and FiTS. All the files and sources used in the examples below, with the exception of FiTS sources that are available in [76], can be found in [4].

5.7.1 Mondex Case Study: Global Test Case Generation

Mondex is an electronic purse application for smart cards products [2], which has been studied as a verification benchmark problem since 2006 [95]. It essentially consists of a financial transaction system which supports transferring funds between electronic *purses*, i.e. accounts. Whenever a person makes a transaction, electronic money is taken from their purse and transferred to the target purse. Regarding the transactions, they are performed following a multi-step message exchange protocol: (1) the source and destination purses should (independently) register with the central fund transferring manager; (2) await a request to deduct funds from the source purse; (3) await a request to add the funds to the destination purse; and finally (4) an acknowledgement is sent to indicate that the transfer took place before the transaction ends.

In this work we will use the implementation of Mondex introduced in [10]. It consists of a Java implementation which runs on a desktop computer, instead of a Java Card implementation for smart cards. A minor difference between these implementations is that in the Java version some methods return values in order to indicate whether their output is normal or erroneous, instead of raising exceptions alike the Java Card implementation.

In particular, we will focus on analysing the behaviour of a sender purse. `ConPurse` objects represent the state of an electronic purse. These objects offer several methods which are used to perform the different transactions. For a sender purse, we should consider only methods (i) `start_from_operation`, (ii) `req_operation`, and (iii) `ack_operation`. Method (i) is used to start taking part on a transaction; method (ii) is used to reply to a money request, i.e. send the money to the destination purse; and method (iii) is used to finish the participation in the transaction once the destination purse acknowledges that it has received the transferred money. In addition, these objects have a field `status`, which is used to keep track of the overall status of a purse during a transaction.

Nr	Hoare triple	Path condition
1	{m_a.label==Ack & m_a.id==sender.name & checkSameTransaction(m_a)} ack(m_a) {status == 4}	status == 3
2	{m_r.label==Req & m_r.id==sender.name & checkSameTransaction(m_r) & transaction.value <= balance} req(m_r) {status == 3}	status == 1
3_a	{m_sf.label==StartF & m_sf.id==sender.name & m_sf.cpd != null & m_sf.cpd.name != name & logIdx < exLog.length} start_from(m_sf) {status == 1}	nextSeq != 32767 & status == 0 & m_sf.cpd.name >= 1 & m_sf.cpd.nextSeqNo >= 0 & m_sf.cpd.value >= 1 & m_sf.cpd.value <= balance
3_b	{m_sf.label==StartF & m_sf.id==sender.name & m_sf.cpd != null & m_sf.cpd.name != name & logIdx < exLog.length} start_from(m_sf) {status == 1}	nextSeq >= 32767 & status == 0 & m_sf.cpd.name >= 1 & m_sf.cpd.nextSeqNo >= 0 & m_sf.cpd.value >= 1 & m_sf.cpd.value <= balance

Table 5.1: Backwards reachability tree computation

Let us consider the model depicted in Fig. 5.4. This model, which is based on the specification provided for Mondex in [10], describes the behaviour of a source purse: first the purse starts its participation in a transaction, then it deducts the requested money from its balance and transfers it to the source purse, and finally it receives an acknowledgement confirming that the (deducted) money was transferred successfully.

Trace Condition Inference

As mentioned before, the field `status` is used to keep track of the overall status of a purse on a transaction. Therefore, whenever the state *end_transaction* is reached in the model, `status` should store the value corresponding to a successful termination of a transaction, i.e. 4 for a sender purse. Then, we use the provided model, considering state *end_transaction* and the property '`status == 4`', to apply our methodology in order to infer global trace conditions.

The first stage corresponds to the computation of the mapping `it` consisting of the reachability information extracted from the model. For space reasons, and due to the fact that for the model in Fig. 5.4 reachability is straightforward, we omit the description of this mapping here.

Next, we move on to the computation of the BRT by following Algorithm 1 with

arguments *awaiting_from*, *end_transaction*, `status == 4`, and 0, as the initial state, the state to reach, the FOL formula, and the allowed number of loop iterations, respectively. State *end_transaction* can only be reached from state *money_deducted* with a transition associated to method `ack`. Therefore, we could proceed to analyse the Hoare triple {`true`} `ack(m_a)` {`status == 4`}. However, the transition associated to `ack` has a precondition describing desired properties for the appropriate execution of this method on state *money_deducted*, i.e. the message corresponds to method `ack`, the message is for the sender purse, and the message corresponds to current transaction. Hence, we use this information to refine the previous Hoare triple as follows:

```
{m_a.label==Ack & m_a.id==sender.name & checkSameTransaction(m_a)}
ack(m_a)
{status == 4}
```

To attempt to verify this Hoare triple, first, we write it as a JML contract. Then, we annotate it in the Java sources. Finally, we run KeY on the annotated code. Note that, for the time being, every step performed in the previous verification attempt is developed manually. However, they can be automated. We will deal with the automation of these steps in future work (see Sec. 2.12).

Such a verification results in one open branch, and one closed branch from which the path condition '`status == 3`' is extracted. Thus, we backwards propagate this condition to state *money_deducted*, and we continue with the computation of the BRT from that state.

In total, we perform three backward propagations of conditions through the model when computing the BRT. Table 5.1 describes the Hoare triples which were analysed during the whole backwards propagation process, and the path condition(s) extracted from the proofs at each step, i.e. the (sufficient) precondition to be backwards propagated. Note that the number in the first column indicates the order in which the Hoare triples are analysed, and that in the case of method `start_from`, the proof results in two close branches, i.e. the table has one entry for each close branch (3_a and 3_b).

Once the BRT is computed, we proceed to compose trace conditions from it. Assuming that φ_{3_a} and φ_{3_b} represent the path conditions in row 3_a and 3_b from table 5.1, respectively, the following trace conditions are inferred:

φ_{3_a}(`status,nextSeq,m_sf`)$\bullet \ll$(`start_from,m_sf`), (`req,m_r`), (`ack,m_a`)$\gg$
φ_{3_b}(`status,nextSeq,m_sf`)$\bullet \ll$(`start_from,m_sf`), (`req,m_r`), (`ack,m_a`)$\gg$

Then, we proceed to use these trace conditions to generate global test cases.

Global Test Case Generation

From the trace conditions above we can write one test case per trace condition using JUnit [26]. In particular, we have written these test cases using as a base a test case automatically generated by *KeyTestGen* [13]. Fig. 5.5 illustrates part of such test cases. They consist of a preamble where the necessary data for the test cases is created and initialised, a block of code where the methods in the sequence of the trace condition are executed, and a call to the oracle to check whether the test is successful or not. Note that all the written test cases can be found in [4].

Regarding the preamble, we have to create and initialise an instance of a `ConPurse` object to play the role of sender purse on the test case. Such an instance has to fulfil the initial FOL formula, i.e., the path conditions backwards propagated to the initial state of the model. In our test cases, object `ConPurse self` corresponds to this instance. In addition, for the arguments `m_sf`, `m_r`, and `m_a` of the different methods, we create and initialise (concrete) data `Data.m_sf`, `Data.m_r`, and `Data.m_a`,

```
//Preamble
short name = (short)1;
ConPurse _o1 = new ConPurse(name);
_o1.balance = (short)42;
ConPurse self = (ConPurse)_o1;
//Execution of sequence of methods in trace condition
java.lang.Throwable exc = null;
try {
  self.start_from_operation(Data.m_sf);
  self.req_operation(Data.m_r);
  self.ack_operation(Data.m_a);
} catch (java.lang.Throwable e) {
  exc=e;
}
//Calling the test oracle
assertTrue(testOracle(exc,self));
```

Figure 5.5: Part of a test case generated from a trace condition

```
protected ArrayAccounts accounts;
public int openSession() {...}
public void closeSession(int sid) {...}
public void depositTo(String acc_nr_dt, int amount_dt) {...}
public void withdrawFrom(String acc_nr_wf, int amount_wf) {...}
```

Figure 5.6: Part of the `UserInfo` class

respectively, having into account the refined preconditions for the test data constraints (see Sec. 5.6.1).

Regarding the execution of the sequence methods, it is performed within a `try-catch` block, as it could happen that an exception is thrown during the execution of the methods when the test case is run. Variable `exc` can be used to check which kind of exception has occurred.

Finally, we write the call to the oracle, which works as a sanity check. This oracle checks whether the property '`status == 4`' is fulfilled after the execution of the sequence of methods. Both test cases are successful.

It is important to remark that, even though we have manually created all the cases, it is possible to automate their generation. We consider developing a tool which integrates this automation as future work.

5.7.2 FiTS Case Study: Runtime Verification

FiTS (a *Financial Transaction System*) [76] is a minimalistic implementation of a system which emulates the standard behaviour of a financial transaction system, where users can have several accounts and use them to deposit or withdraw money.

Basically, the FiTS implementation can be divided in two parts: a front-end providing the operations which both the administrator and the users of the system can perform; and a back-end where the user and account information database is handled. In addition, it is important to remark that several bugs were intentionally introduced on FiTS, as this system is meant to be a verification benchmark. In fact, FiTS has been used as a verification benchmark in the 3rd international Competition on Runtime Verification (CRV), held on 2016 as part of the 16th international conference

∀ `u:UserInfo`,

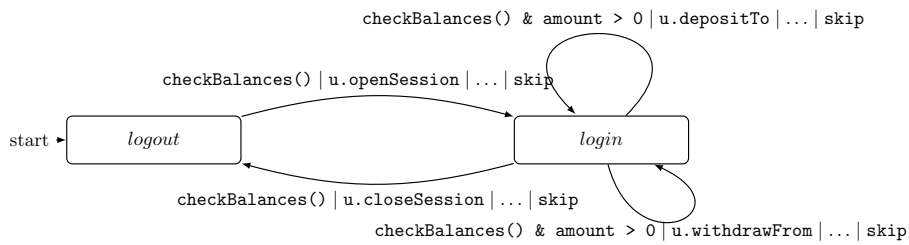


Figure 5.7: Part of the model for the back-end operations of the users

on Runtime Verification (RV'16).

In this work, we analyse the behaviour of the class `UserInfo`, which is part of the back-end of the system. This class represents the information database of the users by storing objects of the class `UserInfo`. These objects depict the state of a particular user. Fig. 5.6 illustrates the signature of the methods, and a field of interest for this case study. Field `accounts` represents the set of open accounts of the user; whenever a user logs in the system, method `openSession` is called to open a new session for the user; whenever a user logs out from the system, method `closeSession` is called to close the previously opened session for the user; whenever a user deposits money from an external source into her account, method `depositTo` is called to deposit the money in the appropriate account; and whenever the user pays a bill (i.e. an external money account), method `withdrawFrom` is run to withdraw the money from the appropriate account.

We had to introduce some minor adaptations to FiTS implementation, in order to make the source code readable by the deductive verifier KeY at the time of computing the BRT. For instance, as KeY cannot handle generics we replaced the use of the classes `ArrayList<UserSession>` and `ArrayList<UserAccount>` by our own implemented classes `ArraySessions` and `ArrayAccounts`, respectively. These two new classes, which were fully specified and verified to increase the confidence in their use, offer the same functionalities than the ones used in the original types, without losing any generality on the implementation. In addition, to avoid dealing with type casting manipulations and unboxing (i.e. unwrapping) of objects in the different proofs, the type of the fields `next_account` and `next_session_id` was changed from `Integer` to `int`.

Let us consider the model illustrated in Fig. 5.7. This model depicts a standard property of financial transaction system. A user can only pay a bill or deposit money from an external source, if she has logged in the system, i.e. methods `depositTo` and `withdrawFrom` can only be run between a use of `openSession` and `closeSession`. Note that for space reasons, as they are not of use in the computation of the BRT, we have omitted here the actual postconditions of the different transitions, and written "..." instead.

Trace Condition Inference

To infer trace conditions for the system applying our methodology we consider the following FOL formula in state *login*: "After accessed, none of the accounts of a user can have a negative balance". Formally,

Nr	Hoare triple	Path condition
1	{checkBalances()} openSession() {checkBalances()}	true
2	{checkBalances() & amount_dt > 0} depositTo(acc_nr_dt,amount_dt) {checkBalances()}	true
3	{checkBalances() & amount_wf > 0} withdrawFrom(acc_nr_wf,amount_wf) {checkBalances()}	true

Table 5.2: Backwards reachability tree computation

$\forall\ int\ i\ ;\ i >= 0\ \&\ i < accounts.size\ ;\ accounts.set[i].balance >= 0$

In the rest of this section, we represent this property as method `checkBalances()`.

The application of our methodology starts by computing the reachability information mapping `rt` from the model. Below, we describe the mapping obtained:

$\texttt{it}(login)\ \texttt{=}\ \{\ logout \xrightarrow{\texttt{checkBalances()|u.openSession|...|skip}} login\texttt{,}$
$login \xrightarrow{\texttt{checkBalances() \& amount\_dt > 0|u.depositTo|...|skip}} login\texttt{,}$
$login \xrightarrow{\texttt{checkBalances() \& amount\_wf > 0|u.withdrawFrom|...|skip}} login\ \}$
$\texttt{it}(logout)\ \texttt{=}\ \{\ login \xrightarrow{\texttt{checkBalances()|u.closeSession|...|skip}} logout\ \}$

Next, we compute the BRT following Algorithm 1 with arguments *logout*, *login*, `checkBalances()`, and 1, as the initial state, the state to reach, the FOL formula, and the allowed number of loop iterations, respectively. By considering the map entry `rt`(*login*), we analyse with KeY the refined Hoare triples depicted in the second column of Table 5.2. All these Hoare triples are fully verified by KeY. Thereby, the (path) condition `true` is backwards propagated to state *logout*, and we continue with the computation of the BRT. From this moment on, every backwards propagated condition will correspond to the condition `true`, and every analysed Hoare triple will be fully verified.

In total there are 32 backwards propagations of conditions, leading to the computation of a BRT consisting of eleven branches, where 33 transitions were filtered for exceeding the limit in the amount of loop traversals. From this BRT, 11 trace conditions can be inferred. Here, we will only focus in the following one:

$\texttt{true}\bullet \ll(\texttt{openSession}, \texttt{()}), (\texttt{closeSession}, \texttt{sid}), (\texttt{openSession}, \texttt{()}),$
$(\texttt{depositTo}, \texttt{acc\_nr\_dt}, \texttt{amount\_dt}), (\texttt{withdrawFrom}, \texttt{acc\_nr\_wf}, \texttt{amount\_wf}) \gg$

Then, we proceed to generate a runtime monitor from the provided model, in order to runtime verify property `checkBalances` when reaching the appropriate state by following the trace pattern of the inferred trace condition.

Runtime Verification

In order to use the previously inferred trace conditions for runtime verification, first, we use the translation described in [38], which is based on ideas from [55], in order to translate the model in Fig. 5.7 to a runtime monitor specification described as a *DATE* [45]. *DATE*s are finite state automata whose transitions are of the form: *event* | *condition* $\mapsto$ *action*, where *event* is a system event, i.e. entering or exiting a method, that triggers the transition, the *condition* is checked and must hold in order

to trigger the transition, and the *action* is a piece of code to be run when taking the transition (after checking the condition).

Once the model is translated to a *DATE*, we use this translation as an input for the runtime verifier LARVA [46]. This tool automatically generates a runtime monitor from the provided *DATE*. Such a monitor will be able to track the sequence of methods in the trace conditions, and check whether the desired FOL formula is fulfilled when reaching the appropriate state.

Next, we run the methods in the sequence of the trace condition, in parallel to the runtime monitor. In order to do so, we create a file with a `main` method where we set up a user and an empty account for her, and we recreate a run which traverses the appropriate sequence of methods, i.e. `openSession`, `closeSession`, `openSession`, `depositTo`, `withdrawFrom`. On this run, after opening a session for the second time, if we deposit 500 Euros, and next we pay to someone 495 Euros, then withdrawing the money from the account leaves the account in 5 Euros, i.e. property `checkBalances` holds when the state in the monitor associated to state *login* in the model is reached.

It is important to remark that, if class `UserInfo` is used in isolation, the fact that `true` is the backwards propagated trace property for every trace condition, and that all the transitions in the model are traversed by the trace pattern, would mean that there is no need to runtime verify `checkBalances`, as every possible execution of the system will fulfill it. However, whenever some of the code units of the system interact with each other, e.g. methods of a unit making (inner) calls to methods in another unit, one code of unit can introduce unexpected issues in the other one, e.g. data manipulation of an object in the front-end can push one of its fields out of the expected boundaries for it in the back-end. Thus, one should always consider generating the corresponding runtime check for the FOL formula under scrutiny, as they will detect these unexpected issues.

Let us come back to the example above, but now, instead of running the methods of class `UserInfo` in isolation, we also consider the execution of the whole system in parallel to the monitor in order to analyse the selected trace condition.

Due to the fact that the methods in the trace pattern are all part of the back-end of the system, they are not directly called in an execution. Instead, we have to focus in using the methods in the front-end which will make inner calls to the methods of interest from within their bodies, i.e. methods in the front-end are interacting with methods in the back-end.

The methods in the front-end are provided by the class `Interface`. Here, we use the methods `USER_login`, `USER_logout`, `USER_depositFromExternal`, and `USER_payToExternal`, as these methods make inner calls to the methods `openSession`, `closeSession`, `depositTo`, and `withdrawFrom`, respectively. Therefore, after setting up the user and an empty account for her, we proceed to run the following sequence of (front-end) methods, corresponding to the trace pattern of the selected trace condition, whose methods are all in back-end: `USER_login`, `USER_logout`, `USER_login`, `USER_depositFromExternal`, `USER_payToExternal`.

At the time of running the previous sequence, if we deposit 500 Euros, and next we pay to someone 495 Euros, then the monitor detects a violation of property `checkBalances` while being at state *login*, when method `USER_payToExternal` is run. Hence, something must be wrong in method `USER_payToExternal` before the inner call to `withdrawFrom` is performed.

By inspecting `USER_payToExternal` implementation, we notice that a fee is charged to the 495 Euros before the withdrawal of money. This causes the total amount of money to be transferred to go over 500 Euros. However, when checking if the balance of the account is enough for the transaction, this check is performed against the 495 Euros without the fee charge. Thereby, due to this erroneous check, the

method makes the inner call to method `withdrawFrom` with an amount higher than the balance of the account. Thus, property `checkBalances` is violated.

In conclusion, even though the methods of class `UserInfo` work fine in isolation, the manipulation of data in the front-end may cause a violation of the property of interest. However, by checking the FOL formula with the runtime monitor, we were able to spot the bug in FiTS which lead to the property violation when analysing the inferred trace condition. Thus, in spite of the fact that it may seem not necessary to do it, generating a runtime check for the FOL formula under scrutiny is able to detect a bug at runtime.

5.7.3 Mondex Case Study: State Invariant Verification

From the results obtained in Sec.5.7.1 we know that property '`status == 4`' is fulfilled whenever state *end_transaction* is reached by following the trace patterns of the inferred trace conditions. In addition, we know that the trace patterns traverse all of the possible paths in the model, and that each trace property holds initially, as the values initially assigned to their corresponding data satisfied them. Let us now analyse the invariability of this property.

The only methods which can modify the different fields of a sender purse are `start_from_operation`, `req_operation`, and `ack_operation`. In addition, as state *end_transaction* is reached whenever a transaction is over, none of these methods can be executed in that state. Thus, property '`status == 4`' is a state invariant in state *end_transaction*.

Furthermore, while being at state *money_deducted*, the only method which may modify a sender purse is `ack_operation`. Such a modification will only occur whenever the previous method is successfully executed. However, when this successful execution terminates, the model will be in a different state, i.e. *end_transaction*. Thereby, property '`status == 3`' is a state invariant in that state. Similarly, considering the successful execution of method `req_operation`, property '`status == 1`' is a state invariant in state *parties_initialised*.

In addition, as mentioned before in Sec.5.7.1, field `status` is used to keep track of the overall status of a purse on a transaction. Therefore, this field can be used as a typestate in the specification of the methods. In particular, in the case of a sender purse, whenever either state *parties_initialised*, *money_deducted*, or *end_transaction* is reached in the model, `status` should store the value 1, 3 or 4, respectively.

Thus, the state invariants described above can be used to prove the explicit use of field `status` as a typestate, due to the fact that the expected values for `status` in the different states of the model match the values used in the comparisons of the state invariants.

As a remark, the state invariants above can be used, for instance, to improve the results obtained in [10], regarding the verification of a specification for Mondex written in *ppDATE*. Basically, a *ppDATE* consists of a labeled transition system, whose states may include functional properties regarding the methods of the system under scrutiny, described as Hoare triples. Thereby, if a FOL formula is a state invariant, then it can be used to refine all of the preconditions of the Hoare triples residing in the appropriate state of the *ppDATE*.

Regarding the work in [10], from a total of 26 Hoare triples, only 2 were fully verified. The remaining 24 were partially verified. In particular, the partially verified Hoare triples do not talk about the value of field `status`, as the automaton in the *ppDATE* takes care of controlling that the methods are executed in the right order, according to the transaction protocol. However, by refining the preconditions of these partially verified Hoare triples with the appropriate knowledge about the state invariants from our previous analysis, all of them would be fully verified. Thus, by considering the use of the state invariants none of the Hoare triples would have to be

verified at runtime, whereas in [10] the 24 partially verified Hoare triple are runtime verified.

5.8 Related Work

We are not aware of any other work which we can be directly compared to the full extent of the work presented here. Still, we can relate the different concepts used in our work to the state of the art.

The main concept used here is *partial proof*, as they work as a conduit to backwards propagate conditions through the model.

In [10, 11], Ahrendt *et al.* use partial proofs to refine *ppDATE* specifications, in order to optimise the runtime monitors extracted from them using the StaRVOOrS tool. As a result, only the branches which were not closed will be verified at runtime. Alike StaRVOOrS, our methodology can be used for runtime verification. However, this tool does not consider the backwards propagation of condition through the states of the *ppDATE*s, and it runtime verifies open branches, whereas our methodology focus on the use of the closed ones. In [13] Ahrendt *et al.* present *KeyTestGen*. This tool automatically generates test cases from proofs of formulas developed by KeY. In particular, KeyTestGen offers a test case generation mode which automatically generate test cases from partial proofs, considering only open branches. Our methodology can also be used to generate test cases. However, we generate global test cases for a whole execution of the system, whereas KeyTestGen generates unit test cases for particular methods. In [38], Chimento *et al.* use partial proofs to enhance the application of *test driven development* (TDD). In particular, partial proofs obtained from the deductive verification of properties are used to (automatically) generate test cases for the open branches, provinding new test cases for TDD. This increases the confidence in the correctness of the developed software, as these new test cases will cover the execution of the parts of the methods which might be in conflict with the property under scrutiny. In spite of the fact that [38] focus on software development and the applications of our methodology, at least for the time being, only focus on software verification, both works consider the use of the open branches in partial proofs as part of their application.

Another concept which is fundamental for our work is the combination of verification techniques, in particular, using symbolic execution as a means to combine static and dynamic verification. In the this work, we have combined deductive verification with both testing and runtime verification. Regarding related work, we have already mentioned StaRVOOrS and KeyTestGen above, which combine (static) deductive verification with runtime verification, and testing, respectively. In addition, one can refer to any of [59, 60, 77, 87].

In [59], Ge *et al.* present DyTa. This tool uses the results obtained during a static verification phase for guiding the application of dynamic symbolic execution. As a result, test cases which can be used to detect potential defects in the system under test are generated. In [60], Godefroid *et al.* introduce SAGE. This tool focuses on the generation of test cases for x86 binaries using dynamic symbolic execution. In [77], Petiot *et al.* introduce the tool *StaDy*. This tool applies test case generation in combination with deductive verification to increase the confidence about the correctness of C programs. In [87], Tillman *et al.* present Pex, a white-box testing tool to generate test cases for .NET programs. The applications for our methodology show that our work can be used to combine deductive verification with either runtime verification, or testing, whereas all the works previously listed only offer one particular combination of techniques.

Regarding the inference of traces from models, more precesily the inference of trace patterns, one can always relate to model-based testing tools like modelJUnit [90],

or QuickCheck [6]. In spite of the fact that these kinds of tools can infer traces, neither the previously mentioned tools, nor any other model-based testing tool (to the best of our knowledge), support backwards propagation of conditions through the states of the model.

A very interesting relation to our work is that sufficient preconditions come as a generalisation of the *weakest preconditions* introduced by Dijkstra [53]. In our work we compute sufficient preconditions from partial proofs. Such preconditions would correspond to the weakest precondition of the methods if we were using a *perfect* prover, i.e. a prover that closes a proof whenever the program under scrutiny fulfils it. However, when using an ordinary prover one may not be able to prove certain branches of a valid property. This may occur due to the lack of 'proving power', the strategies being used, a lack of code annotations (e.g. loop invariants), or alike. In this cases, a sufficient precondition represents a set of initial states which, if fulfilled, will lead the program to a state where the postcondition is fulfilled. Still, this set does not cover the correct execution of all the parts of the program, e.g. it does not cover the parts of the program associated to open branches. Thus, a sufficient precondition is not necessarily the weakest precondition of the program. Anyhow, sufficient preconditions are enough for our methodology, as we focus on the execution of the parts of the methods associated to close branches in a proof, and not necessarily on every possible execution of the method (as it would be the case for a weakest precondition).

5.9 Conclusion

In this paper we present a new methodology which uses the power of partial proofs local to the transitions in a model, obtained from low effort verification attempts, in order to infer global trace conditions which impose restrictions over the execution of the system. This methodology backwards propagates sufficient preconditions through the states of the model until reaching the initial the state. Then, the sufficient precondition propagated to the initial state, and the path followed from the state containing the local property under scrutiny to the initial state, together form the inferred trace condition.

It is important to stress again that the sufficient preconditions extracted from partial proofs come as a generalisation of the weakest precondition of the methods under scrutiny, as they are associated to close branches of the proof of a property, but not necessarily to its full proof.

In addition, working with sufficient preconditions grants the possibility of inferring trace conditions even when performing low effort verifications attempts. For instance, it is enough to get one closed branch out of several branches to infer a trace condition. Thus, once a branch is closed, one can proceed with the application of our methodology, ignoring the rest of the open branches.

We also describe applications for the use of the inferred trace conditions. These applications include (global) test case generation, runtime verification, and state invariants verification. Therefore, this work advances the state-of-the-art of both static and dynamic verification.

This work also reports on the application of the proposed methodology in two case studies: Mondex, an electronic purse application; and FiTS, a financial transaction system. The former was selected because previous work by the authors of this article in this case study motivated the initial ideas on trace conditions inference. The latter was selected because it is a verification benchmark which purposely includes several bugs in its code. Therefore, it provides a relevant scenario for the application of our methodology.

As a future work, we are planning to implement a complete tool chain which will (automatically) apply our methodology in its full extent, whereas, at the moment, the different tools used to develop our case studies are connected manually. In addition, we consider the possibility of introducing some code annotations in order to modularise the inference of trace conditions. For instance, let us assume that a system is divided into several layers of application, e.g back-end and front-end. Then, given that we already know some sufficient preconditions for certain methods in a particular layer, e.g. back-end, we can annotate them in the source code to lift those results to another layer, e.g. the front-end. This can be of a great aid to infer sufficient preconditions describing necessary conditions to go from one layer of the system to another one.

Acknowledgements. We would like to thank Wolfgang Ahrendt as some of the ideas presented in this work evolved from discussions with him. In addition, he provided valuable input to improve the presentation of this work.

BIBLIOGRAPHY

[1] Apache Tomcat. `tomcat.apache.org/`. Accessed: 18-05-2018.

[2] MasterCard International Inc. Mondex. `www.mondexusa.com/`. Accessed: 18-05-2018.

[3] SoftSlate Commerce. `www.softslate.com/`. Accessed: 18-05-2018.

[4] StaRVOOrS. `cse-212294.cse.chalmers.se/starvoors`. Accessed: 2018-11-29.

[5] StaRVOOrS User Manual. `cse-212294.cse.chalmers.se/starvoors/files/userguide.pdf`.

[6] Quviq ab: Quickcheck documentation v1.26.2, June 2012.

[7] Bank system repository. `github.com/mchimento/Bank`, January 2018.

[8] SeleniumHQ. `http://www.seleniumhq.org/`, 2018. Accessed: 25-01-2018.

[9] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book*. LNCS. Springer, 2016.

[10] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109 of *LNCS*, pages 108–125. Springer, 2015.

[11] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Form Methods Syst Des*, 51(1), 2017.

[12] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2012.

[13] W. Ahrendt, C. Gladisch, and M. Herda. *Proof-based Test Case Generation*, pages 415–452. In Ahrendt et al. [9], 2016.

[14] W. Ahrendt, G. Pace, and G. Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISOLA'12*, LNCS 7609. Springer, 2012.

[15] W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS - Episode II - Strengthen and Distribute the Force. In *ISoLA'16 (1)*, volume 9952 of *LNCS*. Springer, 2016.

[16] M. Andersson. Test driven development and automated testin. Course given at Chalmers reporting on his experience teaching TDD to Volvo software developers, 2014.

[17] P. Arcaini, A. Gargantini, and E. Riccobene. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In *ICST'13 2013*, pages 178–187, 2013.

[18] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, et al. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

[19] C. Artho and A. Biere. Combined static and dynamic analysis. In *AIOOL'05*, volume 131 of *ENTCS*, pages 3–14, 2005.

[20] D. Astels. *Test Driven Development: A Practical Guide*. Prentice Hall PTR, 2003.

[21] J. Barnes. *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012.

[22] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'05*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[23] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. volume 7436, pages 68–84, 2012.

[24] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI'04*, pages 44–57, 2004.

[25] E. Bartocci and Y. Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.

[26] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein. *JUnit 5 User Guide (version 5.0.3)*. 2018.

[27] B. Beckert, M. Herda, S. Kobischke, and M. Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018*, pages 53–63, 2018.

[28] Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Coq'Art : the calculus of inductive constructions*. Springer, 2004.

[29] F. Bobot, J. christophe Fillitre, C. March, and A. Paskevich. Why3: Shepherd your herd of provers. In *In Workshop on Intermediate Veri cation Languages*, 2011.

[30] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, LNCS 4609, 2007.

[31] E. Bodden and P. Lam. Clara: Partially evaluating runtime monitors at compile time - tutorial supplement. In *RV'10*, volume 6418 of *LNCS*, pages 74–88, 2010.

[32] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[33] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP'11*, pages 78–83, 2011.

[34] D. Chelimsky. *The RSpec Book. Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Bookshelf, 2010.

[35] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[36] J. M. Chimento and W. Ahrendt. Inferring Global Trace Conditions From Partial Local Proofs. In *Technical Report*. 2018.

[37] J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java. In *Runtime Verification*, volume 9333 of *LNCS*, pages 297–305. Springer, 2015.

[38] J. M. Chimento, W. Ahrendt, and G. Schneider. Testing Meets Static and Runtime Verification. In *FormaliSE'18*. ACM, 2018.

[39] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM'12: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 132–146, 2012.

[40] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT, Cambridge, Mass;London;, 1999.

[41] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[42] D. R. Cok. *OpenJML: JML for Java 7 by Extending OpenJDK*. Springer, 2011.

[43] C. Colombo. LARVA System User Manual, March 2018.

[44] C. Colombo, M. Micallef, and M. Scerri. Verifying web applications: From business level specifications to automated model-based testing. In *MBT'14*, pages 14–28, 2014.

[45] C. Colombo, G. J. Pace, and G. Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS'08*, volume 5596 of *LNCS*, pages 135–149. Springer, 2009.

[46] C. Colombo, G. J. Pace, and G. Schneider. LARVA - A Tool for Runtime Monitoring of Java Programs. In *SEFM'09*, 2009.

[47] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *(ICSE'05)*, pages 422–431, 2005.

[48] F. S. de Boer, S. de Gouw, E. B. Johnsen, and P. Y. H. Wong. Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study. In S. Y. Shin and J. C. Maldonado, editors, *SAC*, pages 1573–1578. ACM, 2013.

[49] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *CAV'15*, pages 273–289, 2015.

[50] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[51] N. Decker, M. Leucker, and D. Thoma. jUnitRV - Adding Runtime Verification to jUnit. In *NASA Formal Methods*, volume LNCS 7871. Springer, 2013.

[52] R. DeLine and M. Fahndrich. Typestates for objects. volume 3086, pages 465–490. Springer Verlag, June 2004.

[53] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[54] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: Overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer*, 17(6):677–694, 2015.

[55] K. Falzon and G. Pace. Combining testing and runtime verification techniques. In *Model-based Methodologies for Pervasive and Embedded Software*, volume LNCS 7706, 2012.

[56] J.-C. Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[57] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In J. Knoop and L. J. Hendren, editors, *PLDI'02*, pages 234–245. ACM, 2002.

[58] A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. D. Monica, and A. Ingólfsdóttir. A foundation for runtime monitoring. In *RV'17*, pages 8–29, 2017.

[59] X. Ge, K. Taneja, T. Xie, and N. Tillmann. Dyta: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 992–994, 2011.

[60] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[61] D. Gries. *The Science of Programming*. Springer, 1st edition, 1987.

[62] D. Harel, D. C. Kozen, and J. Tiuryn. *Dynamic logic*. Foundations of computing. the MIT Press, Cambridge (Mass.), London, 2000.

[63] K. Havelund and G. Roşu. Runtime verification. In *Computer Aided Verification (CAV'01) satellite workshop*, volume 55 of *ENTCS*, 2001.

[64] M. Hentschel, R. Hähnle, and R. Bubel. *Symbolic Execution*, pages 385–389. In Ahrendt et al. [9], 2016.

[65] B. Hetzel. *The Complete Guide to Software Testing*. Wiley [Imprint], Hoboken, 2nd edition, 1993.

[66] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[67] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[68] M. Huisman, W. Ahrendt, D. Grahl, and M. Hentschel. *Formal Specification with the Java Modeling Language*, pages 193–241. In Ahrendt et al. [9], 2016.

[69] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[70] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[71] J. Koenig and K. R. M. Leino. Getting Started with Dafny: A Guide. In *Software Safety and Security*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 152–181. IOS Press, 2012.

[72] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, 2007.

[73] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *LNCS*. Springer, 2010.

[74] M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[75] F. Maraninchi and Y. Rémond. Running-modes of real-time systems: a case-study with mode-automata. In *12th Euromicro Conference on Real-Time Systems (ECRTS 2000), 19-21 June 2000, Stockholm, Sweden, Proceedings*, pages 257–264, 2000.

[76] G. Pace and C. Colombo. FiTS: A Financial Transaction System. `https://github.com/ccol002/ARVISpringSchool`, 2018. Accessed: 2018-11-29.

[77] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. Instrumentation of annotated C programs for test generation. In *SCAM'14*, pages 105–114, 2014.

[78] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[79] G. Reger. An overview of MarQ. In *Runtime Verification - 16th International Conference, RV 2016. Proceedings*, volume 10012 of *LNCS*. Springer, 2016.

[80] G. Reger, H. C. Cruz, and D. E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In *TACAS*, volume 9035 of *LNCS*, pages 596–610. Springer, 2015.

[81] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 2009.

[82] A. Sarcar and Y. Cheon. A new eclipse-based jml compiler built using ast merging. Technical Report UTEP-CS-10-08, University of Texas, 2010.

[83] H. Sözer. Integrated static code analysis and runtime verification. *Softw., Pract. Exper.*, 45(10):1359–1373, 2015.

[84] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[85] S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement and Proof. *Technical monograph PRG-126, Oxford University Computing Laboratory*, 2000.

[86] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[87] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[88] I. Tonin. Verifying the Mondex case study. The KeY approach. *Technical Report 2007-4, Universität Karlsruhe*, 2007.

[89] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques. In *SEFM*, LNCS, pages 382–398, 2011.

[90] M. Utting and B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[91] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, 2012.

[92] M. Wenzel. *The Isabelle/Isar Reference Manual*. 2016.

[93] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *EDCC'05*, pages 281–292. Springer, 2005.

[94] D. Wonisch, A. Schremmer, and H. Wehrheim. Zero Overhead Runtime Monitoring. In *SEFM'13*, volume 8137 of *LNCS*, pages 244–258. Springer Berlin Heidelberg, 2013.

[95] J. Woodcock. First Steps in the Verified Software Grand Challenge. In *SEW'06*, pages 203–206. IEEE Computer Society, 2006.

[96] K. Zee, V. Kuncak, M. Taylor, and M. C. Rinard. Runtime Checking for Program Verification. In *RV'07*, volume 4839 of *LNCS*, pages 202–213. Springer, 2007.