KTH Computer Science
and Communication

Algorithmic Verification Techniques for Mobile Code

IREM AKTUG

Doctoral Thesis
Stockholm, Sweden 2008

TRITA-CSC-A2008:13
ISSN-1653-5723
ISRN-KTH/CSC/A–08/13-SE
ISBN 978-91-7415-123-7

KTH CSC TCS
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi 8 Oktober 2008 10:00 i F3, Kungl Tekniska Högskolan, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Modern computing platforms strive to support mobile code without putting system security at stake. These platforms can be viewed as open systems, as the mobile code adds new components to the running system. Establishing that such platforms function correctly can be divided into two steps. First, it is shown that the system functions correctly regardless of the mobile components that join it, provided that they satisfy certain assumptions. These assumptions can, for instance, restrict the behavior of the component to ensure that the security policy of the platform is not violated. Second, the mobile component is checked to satisfy its assumptions, before it is allowed to join the system. This thesis presents algorithmic verification techniques to support this methodology. In the first two parts, we present techniques for the verification of open systems relative to the given component assumptions. In the third part, a technique for the quick certification of mobile code is presented for the case where a particular type of program rewriting is used as a means of enforcing the component assumptions.

In the first part of this study, we present a framework for the verification of open systems based on explicit state space representation. We propose Extended Modal Transition Systems (EMTS) as a suitable structure for representing the state space of open systems when assumptions on components are written in the modal μ-calculus. EMTSs are based on the Modal Transition Systems (MTS) of Larsen and provide a formalism for graphical specification and facilitate a thorough understanding of the system by visualization. In interactive verification, this state space representation enables proof reuse and aids the user guiding the verification process. We present a construction of state space representations from process algebraic open system descriptions based on a maximal model construction for the modal μ-calculus. The construction is sound and complete for systems with a single unknown component and sound for those without dynamic process creation. We also suggest a tableau-based proof system for establishing temporal properties of open systems represented as EMTS. The proof system is sound in general and complete for prime formulae.

The problem of open system correctness also arises in compositional verification, where the problem of showing a global property of a system is reduced to showing local properties of components. In the second part, we extend an existing compositional verification framework for Java bytecode programs. The framework employs control flow graphs with procedures to model component implementations and open systems for the purpose of checking control-flow properties. We generalize these models to capture exceptional and multi-threaded behavior. The resulting control flow graphs are specifically tailored to support the compositional verification principle; however, they are sufficiently intuitive and standard to be useful on their own. We describe how the models can be extracted from program code and give preliminary experimental results for our implementation of the extraction of control flow graphs with exceptions. We also discuss further tool support and practical applications of the method.

In the third part of the thesis, we develop a technique for the certification of safe mobile code, by adapting the proof-carrying code scheme of Necula to the case of security policies expressed as security automata. In particular, we describe how proofs of policy compliance can be automatically generated for programs that include a *monitor* for the desired policy. A monitor is an entity that observes the execution of a program and terminates the program if a violation to the property is about to occur. One way to implement such a monitor is by rewriting the program to make it *self-monitoring*. Given a property, we characterize self-monitoring of Java bytecode programs for this property by an annotation scheme with annotations in the style of Floyd-Hoare logics. The annotations generated by this scheme can be extended in a straightforward way to form a correctness proof in the sense of axiomatic semantics of programs. The proof generated in this manner essentially establishes that the program satisfies the property because it contains a monitor for it. The annotations that comprise the proofs are simple and efficiently checkable, thus facilitate certification of mobile code on devices with restricted computing power such as mobile phones.

Acknowledgements

I owe my deepest gratitude to my supervisor, Dilian Gurov. He always managed to find the time and patience for guiding me in the past five years. Through our long discussions, and his ingenious comments I have learned how to be a scientist. Without his continuous encouragement and friendly support combined with invaluable expert advice, this thesis would have never been finished.

I thank my co-advisor Mads Dam. I have learned so much from him, especially when working together. I also thank all my co-workers: Marieke Huisman, Andreas Lundblad, Katsiaryna Naliuka and all who have worked in the S3MS project, but especially Fabio Massacci and Frank Piessens. The colleagues in the TCS group have been all very helpful and kind.

I also want to thank two great scientists from my former university METU, Halit Oğuztüzün and Cem Bozşahin, who have supported me beginning from my undergraduate days and gave me the initial excitement of science. I think of you every step of the way as I try to live up to the title "diligent".

Mika Cohen has been my longest lasting office mate, my first friend in Sweden, my first logician acquaintance and many more things. He was followed by Andreas Lundblad, the ultimate work mate: java guru, sailor, good course partner, ideal co-worker.

I thank everyone who has put up with me as I produced the work in this thesis and complained incessantly in the meanwhile: Volkan Bilyar, all my friends at IMIT Adam Strak, Steffen Albrecht, Sezi Yamaç, my dear home mate Bagşen Aktaş, Anders Johansson and my friends from Turkey Utku Erdoğdu, Bariş Sertkaya, Sinan Kalkan, Ruken Çakici, Ayse Müge Sevinç, Ayşe Abbasoğlu, all the friends I have made at summer schools but especially the one and last but not least my eternal student and co-blogger Bariş Tanrıkulu.

I am most grateful to my fairies, Şerife Tekin and Idil Aktuğ, as always. They give me inspiration, warmth; help me swim deep and fly high and write fluent.

I am indebted to my beloved friends Gokçen Baş, Elcil Kaya, Zeren Ergönül, Elif Bato and Gonca Barıt from Izmir Science High School. Though we may not see each other every year, I rejoice your being at all times. You are a part of me.

Finally, I am indebted to Mr. Karl Jonas Henrik Lundell for my uppehållstillstånd, for so many great(!) jokes, for reaching all the high points (e.g. shelves) and for my perfect boyfriend and his amazingly kind parents. One does not need to be a good scientist to prove that Karl is the best, for it is obvious.

I dedicate this thesis to my parents and to my two grandmothers... Siz olmasaniz ben ne olurdum, ne ben olurdum.

Contents

Chapter 1

Introduction

The effectiveness of the Internet in delivering software has resulted in the widespread development and use of mobile code, despite the obvious security risks involved. *Mobile code* is the name given to applications downloaded from a (possibly untrusted) source to be executed locally. One of the most well known examples of mobile code is Java Web applets, small Java applications embedded into web pages. Other examples include code embedded in documents such as e-mails, and Microsoft Office documents. Mobile code emerged to distribute computations in order to enhance user experience without sacrificing network bandwidth and server computing power [108] and it spread to promote ubiquitous computing. Modern smart card platforms such as Java Card support post-issuance downloading of applications [93] and most mobile phones currently have access to new applications through Java midlets [92], both subject to the dynamic installation procedure typical to mobile code.

A system that executes mobile code is designed as an *open system*. This is a general term used to refer to a system where certain components are not yet instantiated and are rather represented in the system by their specification. Hence in an open system, every component is either given by an implementation or by a specification in the form of, for instance, a property expressed with a suitable formalism. Open systems may arise in many situations. Components may not have been implemented yet or particular implementations may not be of importance as they are foreseen to change with later updates. An open system has earlier been defined as a system that executes in an environment which is not fully known in advance. This definition is subsumed by our definition, if the partially specified environment is viewed as a component of the open system. The missing components of an open system may be mobile components that come from another source and do not even join the system before it starts executing. A web browser can be viewed this way, with slots for Java applets that may be downloaded and run as the browser is executing. The execution of open systems is hard to predict and visualize, due to interactions between the new components and the system as well as the interactions

between the new components themselves. In the first two chapters of this thesis, we offer algorithmic techniques to analyze open systems for the purposes of simulation and verification.

In chapter 2, we introduce a formal framework for reasoning about open systems. As a first step we address the problem of modeling open system execution for both visualization and verification purposes. In this chapter, we assume that each component implementation is given by a process algebra term and that each component without implementation is specified by a temporal property. For instance, such a property can be that the component will terminate or that it will always interact with the system using a predetermined set of actions. We replace this specification by a model that is suitable for various analysis with connection to the rest of the system. The ideal model to represent a property is its characteristic model, i.e. a model that includes any behavior that can be performed by a component with the property. This way the characteristic model embodies all implementations that can be used as the specified component. We offer a characteristic model construction for properties in the modal μ-calculus, following earlier work for less expressive logics [77, 51, 71]. We use these models to create open system models, however, this construction is an interesting result in itself as it is the first characteristic model construction for modal μ-calculus that we know of. Given an open system, we describe how to construct a model for the entire system by composing the models for the components according to the system structure; the implementation of the component is used in the process if available, the characteristic model constructed for the given property, otherwise. This model is constructed automatically, can easily be visualized and helps the system developer to understand the execution of the open system.

The correctness of an open systems can be formulated as a verification problem, i.e. given an open system and a property desired of the system, whether the open system satisfies the property. A formal scheme that decomposes the desired system property into properties on components can be practically used to ensure correctness of an open system in the following way: the open system is verified for the system property by substituting characteristic models of corresponding properties in the place of missing components prior to execution, and each new component is checked to satisfy the corresponding property before it joins the system at runtime. Property decomposition has been studied in the context of compositional verification, first introduced by the *assume-guarantee paradigm* of Pnueli [95]. The goal in compositional verification has been to divide a system verification problem into smaller parts by verifying properties of individual components, and inferring a property of the system which is formed by a composition of these components.

When the correct functioning of an open system is specified as a temporal property, open system models constructed as described above can be used to establish correctness. Given an open system and a desired property in modal μ-calculus, we further present in this chapter a proof system that can be used to show that the open system has the property (provided the components that later join the system obey their specifications). A proof tableau in this system mentions states

of a model of the open system, one constructed as described above. Proof tableaux can be constructed automatically since rule application is deterministic. Open system verification is also addressed in chapter 3. In this chapter, we extend a compositional verification framework developed for the purpose of Java smart card application verification by Gurov *et al.* [53]. In this framework, the component implementations are given as Java bytecode classes and the specifications are control flow properties written in a fragment of modal μ-calculus. The models that we construct for this setting are essentially control flow graphs that are tailored to support compositional verification. We propose two extensions to the basic program model of the framework described in [53], for the purpose of increasing the precision of the verification. The first extended model is sensitive to exceptional control flow, while the second to multi-threaded control flow. We describe how these models can be extracted from Java bytecode programs of which the first extraction procedure has been implemented. The models can then be verified using pushdown automata based model checkers.

Correct functioning of an open system, as is the case for any system, depends on the correct functioning of its components. In chapter 2 and 3, we present methods to show correctness of open systems assuming the components that later join the system have certain properties. When these components are mobile ones that are possibly obtained from untrusted sources, ensuring that they actually respect the assumed properties becomes a major issue. Though verification techniques for these components and the desired properties are usually available, it is most often not feasible to apply these in a mobile setting where application "loading" is expected to take no more than a few seconds and where the computational resources may be very limited. Many companies and end users have been affected by rogue code, such as viruses distributed in the form of executable e-mail attachments that cause system crashes. Such attacks by mobile code have been successful since traditional security models have fallen short to prevent them.

Practical techniques that have been applied for ensuring mobile code security include *sandboxing* and *code-signing*. The first limits the privileges granted to mobile code to a subset of available operations and amounts to hiding platform functionality from untrusted applications. In the second approach, the mobile code producer (e.g. the program developer, the Internet distributer) attaches to its application a private key, which is used by the platform to identify this producer when the code is obtained from an untrusted source. This technique depends on a trust relationship that the producer is aware of and respectful to the requirements of the platform that executes the code which takes a long time and a high cost to establish. Recent years have seen a wide interest in the *proof-carrying code* (PCC) approach to mobile code security, introduced by Necula [90]. This technique offers a way to establish trust in the mobile program (rather than its producer) using formal methods. The code producer ships its code with a proof that the code respects the property required by the *code consumer* (e.g. the host system that executes the code). Before the code is run on the consumer's side, the proof is checked to validate that the code respects the property and hence is used to certify that the code is

safe to execute. Since the proof checking procedure is sound, malicious code with a fake proof would always be rejected.

In the last chapter of the thesis, we focus on mobile code safety in the context of mobile devices, such as phones and PDAs. Increase in functionality available in these devices puts even more on stake for the execution of mobile code. For instance, unauthorized access of code to GSM services on a mobile phone may cause direct financial loss to the device owner. We adapt the PCC scheme to the particular case of security policies expressed as security automata. In particular, we describe how proofs of safe execution can be automatically generated when the program includes a *monitor* for the desired property. A monitor is an entity that observes the execution of a program and terminates the program if a violation to the property is about to occur. One way to implement such a monitor is by rewriting the program to make it *self-monitoring*. Given a property, we characterize self-monitoring programs for this property by an annotation scheme with annotations in the style of Floyd-Hoare logics. The annotations generated by this scheme can be extended in a straightforward way to form a proof in the sense of axiomatic semantics of programs. The proof generated in this manner essentially establishes that the program satisfies a property because it contains a monitor for the property. The annotations that comprise the proofs are simple and are expected to be efficiently checkable on a mobile device.

Organization The thesis consists of three independent parts, which can be read separately. By part I, II and III, we refer to chapter 2, 3 and 4, respectively. Part I and II are concerned with compositional verification problems. Part III focuses on generating proofs of correct self-monitoring. The problems undertaken are introduced in more detail in the respective part, along with the approach taken to handle them. Similarly, results achieved in each part are summarized in the respective concluding sections.

1.1 Contributions

The work presented in chapter 2 resulted in the following papers:

1. I. Aktug and D. Gurov, "Towards State Space Exploration Based Verification of Open Systems" to appear in Proceedings of the 4^{th} International Workshop on Automated Verification of Infinite-State Systems (AVIS'05), April 2005, Edinburgh, Scotland

2. I. Aktug and D. Gurov, "State Space Representation for Verification of Open Systems", in Proceedings of the 11^{th} International Conference on Algebraic Methodology and Software Technology (AMAST '06), volume 4019 of Lecture Notes in Computer Science, pages 5-20, July 2006, Kuressaare, Estonia

The work presented in chapter 3 resulted in the following paper:

1. M. Huisman, I. Aktug and D. Gurov, "Program Models for Compositional Verification", in Proceedings of the 10^{th} International Conference on Formal Engineering Methods (ICFEM'08), volume 5256 of Lecture Notes in Computer Science, pages 147-166, October 2008, Kitakyushu-City, Japan

The work presented in chapter 4 resulted in the following papers:

1. I. Aktug and K. Naliuka, "ConSpec: A Formal Language for Policy Specification", in Proceedings of The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07), volume 197-1 of Electronic Notes in Theoretical Computer Science, pages 45-58, September 2007, Dresden, Germany
   Full version submitted to *Science of Computer Programming*, March 2008

2. I. Aktug, M. Dam and D. Gurov, "Provably Correct Runtime Monitoring", in the Proceedings of the 15^{th} International Symposium on Formal Methods (FM '08), volume 5014 of Lecture Notes in Computer Science, pages 262-277, May 2008, Turku, Finland,
   Full version accepted for publication in *Journal of Logic and Algebraic Programming*

My contribution In the first part of the thesis, the initial idea of state space representation of open systems as a means of open system verification, analogical to the verification of closed systems, is due to my advisor Dilian Gurov. The novel notions presented here for achieving this purpose are joint work and have been developed in our common discussions. Finally, the workout of the construction of section 2.6 and the workout of the proofs are due to me. The papers published as a result of the work have also been written jointly.

In the second part, I have worked with Marieke Huisman to create the extended models. The multi-threaded model is our work while the exceptional model is joint work with both Marieke Huisman and Dilian Gurov. Marieke Huisman has layed down most of the formalizations in writing. In the resulting paper, I am responsible in large for extensions of the applet analyzer to handle the new models. In particular, I have updated the original analyzer implementation to work with newer versions of Soot and extended it for the exceptional model.

Finally, in the last part the creation of ConSpec is largely due to me, although I have to give credit to Fabio Massacci for his insights on the matching problem. The concrete and symbolic security automata notions introduced as intuitive formalisms for capturing semantics of ConSpec policies are also largely due to me; they have been shaped in discussions with Dilian Gurov. Papers on the policy language are joint work with Katsiaryna Naliuka. She wrote the motivation and the related work, the rest of the text is mine. The annotation scheme is a result of long discussions between me, Dilian Gurov and Mads Dam. (Later, Andreas Lundblad also joined this group.) The formulation of the result on level II characterization is due to Mads Dam. I am the one that first layed down the ideas in writing, when preparing the

S3MS deliverables. I have also worked out the ideas for handling language features like exceptions and inheritance in the annotations. I have done all proofs in this part, except the proof of theorem 4.22, which is Dilian Gurov's work. I have only detailed and extended this proof.

Chapter 2

State Space Representation Based Verification of Open Systems

2.1 Introduction

Modern software is designed as a collection of components. Modularity brings flexibility to both the development and use of software. For instance, components are developed by different partners and put together at later stages or some component of the system is replaced, after an initial phase of use, by a new component which performs the same task in a more efficient manner. Mobile components can even join the system after it has been put in operation.

In such scenarios, each intermediate system which "misses" components can be thought of as an *open* system. An open system is a system with "holes" in it standing for the missing components. Each hole is accompanied by some property which is a condition that the component to fill the hole has to satisfy. In contrast, a closed system has all its components fixed. An open system captures an infinite set of closed systems, where each hole is filled with some component that satisfies the corresponding property.

A common way of specifying desired behavior is through expressing it as a collection of properties in some temporal logic. Verification of an open system amounts to showing that all the closed systems captured by the open system display these properties. This can only be achieved through a symbolic representation of the open system behavior. In this chapter, we propose a framework for the verification of open systems based on an explicit state space representation. In our approach, we represent the behavior of the open system as a finite structure which is comprised of states, transitions and an acceptance condition which excludes certain non-terminating behavior. The variety in behavior induced by the assumptions on the not-yet-available components is captured through *necessary* and *admissible* transitions, which respectively correspond to common and possible behavior of the closed systems captured by the open system.

Such an explicit state space representation supports various phases of the development of open systems:

- In *the modeling phase*, this formalism can be used as an alternative means of graphical specification. Certain kinds of properties are easier to express graphically than in temporal logics.

- In *automatic verification*, it provides a visualization of the system behavior. This is mostly beneficial if the automatic proof construction fails and an understanding of the open system behavior becomes necessary for debugging. Furthermore, computing the whole state space enables proof reuse when the same system is to be checked for several properties.

- In *interactive verification*, such a state space representation is all the more vital. While it is possible to use conventional methods like encoding system behavior with alternating automata [74] for cases that allow for automatic verification, the human factor in interactive verification requires a more intuitive representation.

In a process algebraic setting, the behavior of an open system can be specified by an *open process term with assumptions* (OTA). An OTA has the shape $\Gamma \rhd E$ and consists of a process term E equipped with a list of behavioral assumptions Γ of the shape $X : \Phi$, where X is a process variable free in E and Φ is a temporal property. Such an open term denotes a set of closed systems, namely those that can be obtained by substituting each free process variable in E with a closed component satisfying the respective assumptions specified in Γ. A property of an OTA is then a property shared by all the closed systems in its denotation.

Modal Transition System (MTS) is a notion that was designed to capture system behavior graphically for the purposes of specification [77]. Each MTS specifies a set of processes through an interval determined by necessary and admissible transitions. MTSs characterize Hennessy-Milner logic (HML), a modal process logic with box and diamond modalities. This makes MTSs a natural representation of open systems when assumptions on the behavior of the abstract components are specified in HML. However, an element of recursion needs to be present in a logic in order to express temporal properties. Therefore, HML is in general not sufficient to specify the missing components of an open system. Similarly, MTSs are not expressive enough for representing the state space of open systems when assumptions are temporal properties. We extend MTSs so that we can represent the state space of open systems when the component assumptions are written in modal μ-calculus, a process logic that adds the expressive power of least and greatest fixed point recursion to HML. Besides the must and may transitions of MTS, our notion, *Extended Modal Transition System* (EMTS) has sets of states (instead of single states) as targets to transitions - an extension which is needed for dealing with disjunctive assumptions. In addition, we add well-foundedness constraints to the structure to handle least fixed point assumptions.

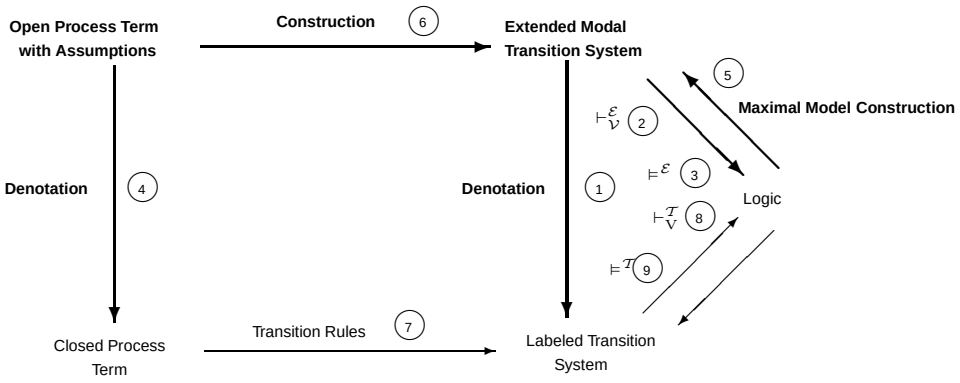


Figure 2.1: Overview of Notions

Given the open system specification as an OTA, the state space representation in the form of an EMTS can be automatically extracted. The first phase of this process corresponds to a maximal model construction, performed for each component assumption. The second phase consists of the composition of the maximal models according to the structure of the OTA term. This two-phase construction is *sound* (resp. *complete*) if the denotation of the OTA is a subset (resp. superset) of the denotation of the resulting EMTS. We show soundness of the construction for systems without dynamic process creation, and soundness and completeness for systems without parallel composition. Furthermore, we present a proof system to prove properties of EMTSs, expressed in modal μ-calculus. Proof search in this proof system can be implemented efficiently, thus complementing the modeling phase in open system development with automatic verification.

The proof system based method of Dam and Gurov [33] is an example of interactive verification of open systems. Reasoning about open systems in such a proof-theoretic manner can essentially be viewed as a symbolic execution of OTA where the state space is implicit. EMTSs can be used to make (the explored part of) the state space explicit in such a proof, thus enabling the visualization of the behavior of the system which aids the current interaction as well as future verification efforts. It also serves proof reuse as mentioned above. We leave possible interactive approaches based on EMTSs to future work.

2.1.1 Overview of Notions and Results

Figure 2.1 shows an overview of the central notions used in this chapter and the relations between them. New concepts and constructions proposed by the chapter are indicated in bold. *OTA* and *EMTS* are introduced in sections 2.4 and 2.5, respectively, built on the already well-developed concepts of *closed process term* and *labeled transition system*.

We propose open process terms with assumptions on the free variables (OTA) for modeling open systems. Such an open term *denotes* an infinite set of closed terms, namely all those which can be obtained from the open term by substituting the free variables with closed terms satisfying the respective assumptions. The relationship between labeled transition systems (LTS) and extended modal transition systems (EMTS) corresponds to that between closed and open terms (or that between closed and open systems). The *denotation* of a state of an EMTS is the set of LTS states that this state relates to with a suitably defined *simulation* relation.

The logic we use in this study is the modal μ-calculus (see section 2.4 for a short introduction). The assumptions in an OTA and the properties to be checked for the open system are both expressed in this temporal logic. We say an EMTS state satisfies a temporal logic formula (Item 3 in figure 2.1), when the all the LTS in its *denotation* satisfy the property (Item 1).

A proof system by Bradfield and Stirling [19, 104] can be used for showing that states of an LTS satisfy a temporal property expressed in modal μ-calculus (Item 8). We adapt this proof system to show (modal μ-calculus) properties of EMTS states (Item 2). A summary of both proof systems along with an account of their more significant differences can be found in section 2.7. The soundness and completeness properties of the logic, which we show for prime formulae make our proof system adequate for proving satisfaction for these type of properties.

The LTS corresponding to a closed process term can be constructed using transition rules. For the construction of the state space of an OTA in the form of an EMTS, we present here an automatic construction (Item 6) through maximal models. Given a temporal logic formula, an EMTS that characterizes it can be constructed using the maximal model construction presented in section 2.6. The maximal models for assumptions of an OTA are combined according to the structure of the process term to produce the EMTS of the open system.

If the various transformations are correctly defined, the diagram in figure 2.1 should commute. In particular, in the context of a given labeled transition system, the construction of an EMTS from an OTA should preserve the denotation (Items 6 and 1 vs. Items 4 and 7). For the automatic construction in this chapter, this is the case if the open system contains a single unknown component (see section 2.6.3). Similarly, it is possible to prove the satisfaction of a property by a state of the EMTS in a sound and complete proof system (Item 3) if and only if, for each LTS, the set of all states that are denoted (Item 1) by this state satisfy the property (Item 8).

Organization In the next two sections, we give a brief background on compositional reasoning and on structures that have been used for capturing properties. In sections 2.4 and 2.5, we formally introduce the OTA and EMTS notions, respectively. We present an OTA-to-EMTS construction in section 2.6, which includes our maximal model construction for the modal μ-calculus. We introduce the proof system for EMTSs, along with correctness results in section 2.7. In section 2.8, we summarize other approaches related to ours. We conclude with a summary of the chapter and suggestions for future work.

2.2 Compositional Reasoning

Compositional reasoning aims to avoid state space explosion by taking advantage of the natural decomposition of the system in components. The goal is to verify properties of individual components, and infer a property of the system which is formed by a composition of these components and thus avoid to compute the state space of the whole system.The problem of open system verification naturally arises in compositional verification. We consider here the approaches that have influenced our work.

In the rest of the text, we assume the reader to be familiar with HML and the temporal logics LTL, CTL, ∀CTL,∀CTL* (a comprehensive survey is by Emerson [37]).

The earliest formalization of this idea is Pnueli's *assume-guarantee paradigm* [95]. A formula in Pnueli's logic is a triple $\langle\psi\rangle\, P\, \langle\phi\rangle$ where ψ and ϕ are temporal formulae and P is a program. The formula is true if, whenever program P is part of a system satisfying the formula ψ, the system also satisfies ϕ. The following inference rule captures the assume-guarantee style proof strategy:

$$\frac{\begin{array}{c}\langle\psi\rangle\, P\, \langle\phi\rangle \\ \langle true\rangle\, P'\, \langle\psi\rangle\end{array}}{\langle true\rangle\, P' \parallel P\, \langle\phi\rangle} \tag{2.1}$$

This rule uses knowledge about components P and P' to infer a property of the system consisting of their composition $P' \parallel P$. Provided the environment it runs in satisfies ψ, component P guarantees the satisfaction of ϕ. The system $P' \parallel P$ then has this property if the rest of the system components P' create an environment in which ψ is satisfied.

The decomposition of the required property ϕ into an adequate assumption (or *local property*) ψ for component P requires knowledge of the system and in most cases is achievable only through human input. In order to automate the rest of the tasks in compositional reasoning, a checker must have several properties. It must be able to check that a property is true of all systems which can be built using a given component. More generally, it must be able to restrict to a given class of environments when doing this check. It must also provide facilities for performing temporal reasoning. In order to obtain such a checker, Grumberg and Long suggested the use of the simulation notion, a preorder on finite state models that captures the notion of "more behaviors" [51]. Let $\preceq$ denote this simulation relation so that the statement P' simulates P is denoted as $P \preceq P'$, and can be informally understood as P' can perform all behaviors of P.

The compositional reasoning principle of Grumberg and Long requires that for every local property, there exists a *maximal model*, which can be thought of as the most generic model that satisfies the property ψ in that it simulates all models that satisfies the property. Let P and P' be (finite state) structures defined in detail below, and $\parallel$ be the composition operator. Let $\mathcal{M}_\psi$ denote the maximal model for property ψ. This framework uses the following rule for compositional verification:

$$\frac{P \models \psi \qquad \mathcal{M}_\psi \parallel P' \models \phi}{P \parallel P' \models \phi} \tag{2.2}$$

The rule reduces checking $P \parallel P' \models \psi$ to the following steps: (1) decomposition of the global property ϕ to the local property ψ on component P, (2) construction of a maximal model $\mathcal{M}_\psi$ for ψ, (3) the check that P satisfies ψ and (4) the check that $\mathcal{M}_\psi \parallel P'$ satisfies ϕ. While the first step requires human intervention in general, note that steps 3 and 4 can be performed using standard model checking algorithms. We summarize work on step 2 below.

The correctness of Rule 2.2 depends on the relationships between the various components of the framework:

1. The preorder should preserve satisfaction of formulae of the logic, i.e. if a formula is true for a model, it should also be true for any model which is smaller in the preorder. For programs P and P',

$$P \preceq P' \Rightarrow \forall(\phi.\, P' \vDash \phi \Rightarrow P \vDash \phi)$$

2. Composition should preserve simulation. For programs P, P' and P'',

$$P \preceq P' \Rightarrow (P \parallel P'' \preceq P' \parallel P'')$$

3. For the local property ψ, there exists a maximal model. For program P and property ψ,

$$P \models \psi \iff P \preceq \mathcal{M}_\psi$$

The soundness of the rule follows then from these properties in the following way. Let us call $P \models \psi$ premise (1) and $\mathcal{M}_\psi \parallel P' \models \phi$ premise (2). From premise (1) and property (3), we can infer that $P \preceq \mathcal{M}_\psi$ and using property (2), we further infer that $P \parallel P' \preceq \mathcal{M}_\psi \parallel P'$. Then using property (1) and premise (2), we can infer that $P \parallel P' \models \phi$.

The finite models used in this study are synchronous parallel compositions of Kripke structures under fairness assumptions. Kripke structures are essentially transition systems where each state is labeled by a set of atomic propositions, and are often used to model finite-state systems for model checking purposes.

Definition 2.1 (Kripke Structures with Fairness Constraints [51])**.** A *structure* $P = (S, S_0, AP, L, \rightarrow, F)$ is a tuple of the following form.

1. S is a finite set of *states*.
2. $S_0 \subseteq S$ is set of *initial states*.
3. AP is a finite set of *atomic propositions*.
4. L is a (labeling) function that maps each state to the set of atomic propositions true in that state.

5. $\rightarrow \subseteq S \times S$ is a transition relation.

6. F is a Streett acceptance condition, represented by pairs of sets of states.

The fairness constraint encoded by F selects a subset of paths (i.e. infinite state sequences) of the Kripke structure as fair. When CTL formulae are interpreted, path quantifiers are restricted to fair paths. Let $\rho = s_0 s_1 \ldots$ be a path of the structure P. We define $\inf(\rho) = \{s \mid s = s_i \text{ for infinitely many } i\}$. A path ρ is accepted by the Streett acceptance condition F if for every $(P, Q) \in F$ (where $P, Q \subseteq S$), $\inf(\rho) \cap P \neq \varnothing$ implies $\inf(\rho) \cap Q \neq \varnothing$. Finally, a path of the structure is *fair* if it is accepted by the structure's acceptance condition. Other acceptance conditions can also be used to select fair paths. For instance, fairness constraints are also a part of the structure we use for state space representation of open systems, where *a parity condition* is used for this purpose (see section 2.5).

We can now define a simulation relation on two Kripke structures with fairness constraints:

Definition 2.2 (Structure Simulation [51])**.** Let $P = (S, S_0, AP, L, \rightarrow_P, F)$ and $P' = (S', S'_0, AP', L', \rightarrow_{P'}, F')$ be two structures with $AP \subseteq AP'$. A relation $H \subseteq S_P \times S_{P'}$ is a simulation relation if for all $s \in S$ and $s' \in S'$, the following holds:

1. $L(s) \cap AP' = L'(s')$

2. for every fair path $\rho = s_0 s_1 s_2 \ldots$ from $s = s_0$ in P, there exists a fair path $\rho' = s'_0 s'_1 s'_2 \ldots$ from $s' = s'_0$ in P' such that for every $i \geq 0$, $H(s_i, s'_i)$.

H is a simulation from P to P' if and only if for every initial state $s_0 \in S_0$ there is an initial state $s'_0 \in S'_0$ such that $H(s_0, s'_0)$. If there is such a simulation relation from P to P', then we say P' simulates P.

It can easily be checked that this simulation relation is a preorder for Kripke structures with fairness constraints, and that it preserves satisfaction of formulae of CTL. Thus condition (1) for the correctness of the compositional rule is established. The composition operator that Grumberg and Long use corresponds to Moore machine composition. Each transition of the composition is a joint transition of the components, and states of the composition are pairs of component states that agree on their common atomic propositions. This ensures that conditions (2) and (3) are satisfied.

In section 2.5, we present a simulation relation for EMTSs, which also includes fairness constraints.

Computing Maximal Models

The automatization of maximal model construction is one of the keys to the applicability of compositional verification as captured by Rule 2.2. Grumberg and Long describe a tableau construction for ∀CTL formulae in [51].

A construction for ∀CTL* was later offered by Kupferman and Vardi [71]. Maximal model construction is explored by Gurov *et al.* for reasoning about sequential programs with procedures which have potentially infinite-state behavior [53]. Programs are modeled by control flow graphs and the logic used in this study is the fragment of modal μ-calculus without diamond modalities and least fixed points. We take a closer look at this framework in the next chapter. Maximal model construction is performed by stepwise transformation of formulae a normal form, for which the mapping to maximal models is defined directly.

Compositional Reasoning for Open Systems

Verification of open systems can be performed by a compositional proof system due to Dam *et al.* presented for CCS processes in [31] and for Erlang programs in [32].

The proof system is a Gentzen-style, compositional proof system. The sequents are of the form $\Gamma \vdash \Delta$ where the set Γ consists of "assumed" assertions, while the set Δ consists of "guaranteed" ones. Such a sequent is valid if, whenever all assertions of Γ hold, at least one assertion of Δ holds. The assertions are of three forms: a process may be asserted to satisfy a temporal formula (e.g. $E : \phi$), a process may be asserted to be able to perform a certain transition and evolve to another process (e.g. $E \xrightarrow{\alpha} F$) or a relation between ordinal variables (e.g. $\kappa < \kappa'$) may be stated. The ordinal variables are used to relate the rates of progress for fixed point formulae appearing in different parts of a sequent.

In this system, compositional reasoning is accomplished through a general rule of subterm cut:

$$\frac{\Gamma \vdash Q : \psi, \Delta \qquad \Gamma, x : \psi \vdash P : \phi, \Delta}{\Gamma \vdash P[Q/x] : \phi, \Delta}$$

The rule expresses that if the assumptions in Γ guarantees that the process term Q satisfies property ψ, and if the same set of assumptions Γ and the assumption that the process variable x satisfies the temporal property ψ guarantee that the term P (which potentially includes occurrences of x) satisfies ϕ, then the same set of assumptions Γ guarantee that the system obtained by replacing each occurrence of x in P by Q satisfies the property ϕ.

A proof in this proof system is guided by the temporal logic formula to be verified and a global discharge condition is employed which recognizes proofs by well-founded induction.

The open system verification problem can be formulated in this framework by placing the assumptions on components in the set Γ, while the assertion that the structure of the system along with the desired property is asserted as a process algebra term in Δ. The notion of OTA was developed with this intuition.

2.3 Structures for Capturing Properties

Attempts to characterize formulae with finite structures resulted from different concerns, such as specification and verification. Modal Transition Systems (MTS) can be seen as a notion arising from the first concern. MTS is a graphical specification language in the process algebra framework that was designed as an intuitive alternative to Hennessy-Milner logic. Whereas, for verification purposes, various types of automata have been employed. For instance, maximal models used in many compositional reasoning frameworks have been constructed in the form of automata. We aimed for a structure to represent the state space of open systems that is visualizable in order to facilitate a thorough understanding of the system and at the same time can be directly used for verification. Therefore, we have been inspired by both MTSs and automata when developing our notion. Our structure, EMTS, is based on modal transition systems with an acceptance condition borrowed from automata in order to encode prohibited infinite runs of the system.

Modal Transition Systems

MTSs were designed as a graphical specification language in the process algebra framework by Larsen [77]. Each MTS specifies a set of processes through an interval determined by necessary and admissible transitions.

Definition 2.3 (MTS)**.** A *modal transition system* is a structure $\mathcal{S} = (S, A, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$ where S is a set of states, A is a set of actions and $\longrightarrow_{\Box}, \longrightarrow_{\Diamond} \subseteq S \times A \times S$ are transitions, satisfying the consistency condition $\longrightarrow_{\Box} \subseteq \longrightarrow_{\Diamond}$.

In the rest of this section, we let s range over MTS states. Larsen refers to states of an MTS as "specifications", stressing the fact that each state of an MTS specifies a set of processes. A process can be viewed as an MTS where the must and may transitions coincide, $\longrightarrow_{\Diamond} = \longrightarrow_{\Box}$.

An MTS can be refined stepwise to an *implementation* that performs all the *must* transitions ($\longrightarrow_{\Box}$) of the MTS but performs only a subset of the *may* transitions ($\longrightarrow_{\Diamond}$). The stepwise refinement indicates a preorder between MTSs so that as the specification gets finer, the set of processes specified by the states of the MTS gets smaller.

Definition 2.4 (Refinement)**.** A *refinement* $\mathcal{R}$ is a binary relation on the states S of the modal transition system $\mathcal{S} = (S, A, \longrightarrow_{\Box}, \longrightarrow_{\Diamond})$ such that whenever $s_1 R s_2$ and $a \in A$, then the following holds:

1. Whenever $s_1 \xrightarrow{a}_{\Diamond} s_1'$, then $s_2 \xrightarrow{a}_{\Diamond} s_2'$ for some s_2' with $s_1' \mathcal{R} s_2'$
2. Whenever $s_2 \xrightarrow{a}_{\Box} s_2'$, then $s_1 \xrightarrow{a}_{\Box} s_1'$ for some s_1' with $s_1' \mathcal{R} s_2'$

The state s_1 is said to be a refinement of state s_2, denoted $s_1 \lhd s_2$, if $s_1 \mathcal{R} s_2$ for some refinement relation $\mathcal{R}$. Refinement can be generalized to the states of two different MTSs in the natural way.

Given a labeled transition system $\mathcal{T} = (S_\mathcal{T}, A, \longrightarrow_\mathcal{T})$, a process $t \in S_\mathcal{T}$ of $\mathcal{T}$ *implements* a structure $\mathcal{S}$ if there is a refinement relation which contains $(t, \mathcal{S})$, that is if $t \lhd \mathcal{S}$.

The combinators of the employed process algebra can be lifted to MTSs, which makes them suitable for modeling open systems when component assumptions are given in HML. Furthermore, this enables a component to be replaced by its refinement.

In [18], Boudol and Larsen introduce a concept similar to maximal models. The class of formulae of HML for which a maximal model in the form of an MTS can be constructed is termed *graphically representable*. The authors show that a formula is graphically representable (i.e. by a single MTS state) if and only if it is consistent and *prime*. A formula is prime if, whenever it implies a disjunction, then it implies one of the disjuncts, i.e.

$$\phi \text{ is prime} \iff \forall \phi_0, \phi_1.(\phi \Rightarrow (\phi_0 \vee \phi_1)) \Rightarrow (\phi \Rightarrow \phi_0) \vee (\phi \Rightarrow \phi_1)$$

Non-prime formulae are also representable with MTSs but by multiple (though finitely many) MTS states. It is also shown that for each MTS (state), a characteristic formula exists in Hennessy-Milner Logic so that s_1 is a refinement of s_2 if and only if it satisfies s_2's characteristic formula, and when viewed as specifications both the state s and its characteristic formula are implemented by the same set of processes. Finally, these results establish a Galois connection between the logical consequence preorder on consistent prime formulae and the refinement preorder on MTS states.

Automata Theoretic Approaches

The establishment of a clean connection between Büchi automata and linear temporal logic (LTL) enabled verification-related problems such as satisfiability and model-checking to be reduced to standard automata-theoretic problems [114, 111]. The idea is to associate with each linear temporal logic formula a finite automaton over infinite words that accepts exactly the computations that satisfy the formula. As a result of this correspondence, optimal algorithms from automata theory could be imported to verification. In this manner, a linear time, automata-based linear temporal logic verification algorithm has been constructed [111].

Similar efforts for branching time logics require tree automata to be used, because in branching time logics, each moment in time may separate into several possible futures, while in linear time logics, each moment in time has a unique possible future [76]. Tree automata run on infinite trees instead of infinite words, and have been used with a number of different acceptance conditions in the literature, of which the most frequently used are Müller, Rabin, Streett, and parity conditions. (For a comprehensive survey of automata on infinite trees see Thomas [107].) Tree automata yield, when used in this context, exponential algorithms for verification. In order to reduce complexity, generalized forms of nondeterministic tree automata

arose, e.g. alternating tree automata [89] and amorphous automata [14]. For instance, Kupferman *et al.* present an automata-based model checking algorithm in [74], reducing the problem to a special non-emptiness problem for alternating automata, thus obtaining a linear time model-checking bound for CTL.

Automata-based algorithms for modal μ-calculus are relatively less explored as the construction of . Emerson and Jutla have shown that modal μ-calculus formulae and nondeterministic automata on trees are equiexpressive [38]. This result is reached by establishing that the parse tree of a modal μ-calculus formula can be seen as an alternating tree automaton with, for instance, Streett acceptance condition, and by converting this alternating tree automaton to an equivalent nondeterministic tree automaton. This second step is in general not possible since alternating tree automata are a generalization of nondeterministic tree automata, but alternating tree automata obtained from modal μ-calculus formulae have a special property of being "history-free" that enables the conversion. In order to characterize modal μ-calculus, Janin and Walukiewicz created the μ-automata, automata that are alternating automata with parity condition when restricted to binary trees, but more general otherwise. This automata type has equivalent expressive power to the that of μ-calculus.

The reason we introduce a new formalism to capture modal μ-calculus formulae is our interest in representing the state space of processes satisfying a property by a common structure. Although highly expressive, we believe that the aforementioned structures do not provide an intuitive representation of the state space in terms of states and transitions. The combination of complicated transition relations with acceptance conditions (consider for instance alternating automata with Streett acceptance [73]), make automata an unattractive choice for graphical specification. Therefore, our structure for capturing properties, and eventually state space of open systems, brings together the may and must transitions of MTSs with a parity acceptance condition. Our maximal model construction for modal μ-calculus was inspired rather by the construction of Kaivola which converts formulae from the alternation-depth class Π_2 fragment of the modal μ-calculus to Büchi Automata [65].

2.4 Specifying Open Systems

A system, the behavior of which is parameterized on the behavior of certain components, is conveniently represented as a pair $\Gamma \rhd E$, where E is an open process-algebraic term, and Γ is a list of assertions of the shape $X : \Phi$ where X is a process variable free in E and Φ is a closed formula in a process logic.

In the present study, we work with the class of Basic Parallel Processes (BPP)[23]. The terms of BPP are generated by:

$$E ::= \mathbf{0} \mid X \mid a.E \mid E + E \mid E \parallel E \mid \mathit{fix}\ X.E$$

where X ranges over a set of process variables *ProcVar* and a over a finite set of

actions A. We assume that *ProcVar* is partitioned into assumption process variables *AssProcVar* used in assertions, and recursion process variables *RecProcVar* bound by the *fix* operator. A term E is called *linear* if every assumption process variable occurs in E at most once. The operational semantics of closed process terms (called processes and ranged over by t) is standard. In the rest of this text, the symbol "$\parallel$" signifies *merge composition*, while the symbol "|" is used as a symbol for parallel composition in general.

$$\frac{\cdot}{a.E \xrightarrow{a} E} \qquad \frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 \parallel E_2 \xrightarrow{a} E_1' \parallel E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 \parallel E_2 \xrightarrow{a} E_1 \parallel E_2'} \qquad \frac{E_1[\mathit{fix}\ X.E_1/X] \xrightarrow{a} E_1'}{\mathit{fix}\ X.E_1 \xrightarrow{a} E_1'}$$

As a process logic for specifying both behavioral assumptions on components and open system properties, we consider the modal μ-calculus [70]. We have selected this logic as it subsumes most other well-known logics like CTL and LTL. The formulae of modal μ-calculus are generated by:

$$\Phi ::= \mathrm{tt} \mid \mathrm{ff} \mid Z \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [a]\,\Phi \mid \langle a \rangle\,\Phi \mid \nu Z.\Phi \mid \mu Z.\Phi$$

where Z ranges over a set of propositional variables *PropVar*.

Variable X in $\sigma X.\Phi$, where $\sigma \in \{\nu, \mu\}$, is called *guarded* if every occurrence of X in Φ is in the scope of some modality operator $\langle a \rangle$ or $[a]$. We say that a formula is guarded if every bound variable in the formula is guarded. A formula Φ is a *normal* formula if $\sigma_1 Z_1$ and $\sigma_2 Z_2$ are two different occurrences of binders in Φ then $Z_1 \neq Z_2$ and no occurrence of a free variable Z is also used in a binder σZ in Φ. Let Φ be a normal formula and $\sigma_1 X.\Psi_1$ and $\sigma_2 Z.\Psi_2$ be subformulae of Φ, then X *subsumes* Z if $\sigma_2 Z.\Psi_2$ is a subformula of $\sigma_1 X.\Psi_1$.

Definition 2.5 (Semantics of Modal μ-calculus)**.** The semantics of the modal μ-calculus is given in terms of the denotation $||\Phi||^{\mathcal{T}}_{\mathrm{V}} \subseteq S_{\mathcal{T}}$ relative to a labeled transition system $\mathcal{T} = (S_{\mathcal{T}}, A, \longrightarrow_{\mathcal{T}})$ and a valuation $\mathrm{V} : PropVar \to S_{\mathcal{T}}$ as follows:

$$\begin{aligned}
&||\mathrm{tt}||^{\mathcal{T}}_{\mathrm{V}} = S_{\mathcal{T}} \\
&||\mathrm{ff}||^{\mathcal{T}}_{\mathrm{V}} = \varnothing \\
&||Z||^{\mathcal{T}}_{\mathrm{V}} = \mathrm{V}(Z) \\
&||\Phi_1 \vee \Phi_2||^{\mathcal{T}}_{\mathrm{V}} = ||\Phi_1||^{\mathcal{T}}_{\mathrm{V}} \cup ||\Phi_2||^{\mathcal{T}}_{\mathrm{V}} \\
&||\Phi_1 \wedge \Phi_2||^{\mathcal{T}}_{\mathrm{V}} = ||\Phi_1||^{\mathcal{T}}_{\mathrm{V}} \cap ||\Phi_2||^{\mathcal{T}}_{\mathrm{V}} \\
&||\,\langle a \rangle\,\Phi||^{\mathcal{T}}_{\mathrm{V}} = \{t \mid \exists t'.\, t \xrightarrow{a}_{\mathcal{T}} t' \wedge t' \in ||\Phi||^{\mathcal{T}}_{\mathrm{V}}\} \\
&||\,[a]\,\Phi||^{\mathcal{T}}_{\mathrm{V}} = \{t \mid \forall t'.\, t \xrightarrow{a}_{\mathcal{T}} t' \wedge t' \in ||\Phi||^{\mathcal{T}}_{\mathrm{V}}\} \\
&||\mu Z.\Phi||^{\mathcal{T}}_{\mathrm{V}} = \bigcap\{T \subseteq S_{\mathcal{T}} \mid T \supseteq ||\Phi||^{\mathcal{T}}_{\mathrm{V}[T/Z]}\} \\
&||\nu Z.\Phi||^{\mathcal{T}}_{\mathrm{V}} = \bigcup\{T \subseteq S_{\mathcal{T}} \mid T \subseteq ||\Phi||^{\mathcal{T}}_{\mathrm{V}[T/Z]}\}
\end{aligned}$$

An alternative (equivalent) interpretation of extremal fixed points is through approximants. We provide here a characterization for a set of ordinals Ord, elements of which are ranged over by α, κ. The symbol λ ranges over limit ordinals. Let $(\sigma Z.\Phi)^\alpha$ be the *α-approximant* (alternatively α-unfolding) of $\sigma Z.\Phi$ with the following interpretation:

$$\begin{array}{ll} ||(\nu Z.\Phi)^0||^{\mathcal{T}}_{\mathrm{V}} = S_{\mathcal{T}} & ||(\mu Z.\Phi)^0||^{\mathcal{T}}_{\mathrm{V}} = \varnothing \\ ||(\nu Z.\Phi)^{\alpha+1}||^{\mathcal{T}}_{\mathrm{V}} = ||\Phi||^{\mathcal{T}}_{\mathrm{V}[||(\nu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}}/Z]} & ||(\mu Z.\Phi)^{\alpha+1}||^{\mathcal{T}}_{\mathrm{V}} = ||\Phi||^{\mathcal{T}}_{\mathrm{V}[||(\mu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}}/Z]} \\ (\nu Z.\Phi)^\lambda = \bigcap\{||(\nu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}} \mid \alpha < \lambda\} & (\mu Z.\Phi)^\lambda = \bigcup\{||(\mu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}} \mid \alpha < \lambda\} \end{array}$$

Approximants are used in connection to theorem 2.6, and in the correctness proof for the maximal model construction, found in Appendix A.1.1. The following theorems present results on modal μ-calculus and approximants.

Theorem 2.6. *The following hold for modal μ-calculus formulae:*

(i) $||\sigma Z.\Phi||^{\mathcal{T}}_{\mathrm{V}} = ||\Phi[\sigma Z.\Phi/Z]||^{\mathcal{T}}_{\mathrm{V}}$, $\sigma \in \{\mu, \nu\}$. (Unfolding theorem)

(ii) $||(\mu Z.\Phi)||^{\mathcal{T}}_{\mathrm{V}} = \bigcup_\alpha ||(\mu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}}$ (Knaster-Tarski theorem)

(iii) $||(\mu Z.\Phi)^\kappa||^{\mathcal{T}}_{\mathrm{V}} = \bigcup_{\alpha<\kappa} ||\Phi||^{\mathcal{T}}_{\mathrm{V}[||(\mu Z.\Phi)^\alpha||^{\mathcal{T}}_{\mathrm{V}}/Z]}$

We define satisfaction $t \models^{\mathcal{T}}_{\mathrm{V}} \Phi$ by $t \in ||\Phi||^{\mathcal{T}}_{\mathrm{V}}$. In what follows, we omit the subscript V from $||\Phi||^{\mathcal{T}}_{\mathrm{V}}$ when Φ is a closed formula. Satisfaction is lifted to sets of states in the natural way, so that a set of states $S \subseteq S_{\mathcal{T}}$ satisfies a property Φ, $S \models_{\mathrm{V}} \Phi$, only if for all $s \in S, s \models_{\mathrm{V}} \Phi$.

We say that an OTA $\Gamma \rhd E$ is *guarded* when the term E and all modal μ-calculus formula Φ in Γ are guarded. Similarly, we say an OTA is linear when the term it contains is linear.

The behaviors specified by an open term with assumptions are given with respect to a labeled transition system $\mathcal{T}$ that is closed under the transition rules and is closed under substitution of processes for assumption process variables in subterms of the OTA. The states of LTS correspond to processes in our process algebra. The denotation of an OTA is then the set of all processes obtained by substituting each assumption process variable in the term by a process from $\mathcal{T}$ satisfying the respective assumptions.

Definition 2.7 (OTA Denotation)**.** Let $\Gamma \rhd E$ be an OTA, $\mathcal{T}$ be an LTS, and ρ_R : RecProcVar $\to S_{\mathcal{T}}$ be a recursion environment. The *denotation* of $\Gamma \rhd E$ relative to $\mathcal{T}$ and ρ_R is defined as:

$$[\![\Gamma \rhd E]\!]^{\mathcal{T}}_{\rho_R} \triangleq \{E\rho_R\rho_A \mid \forall (X : \Phi) \in \Gamma.\ \rho_A(X) \models^{\mathcal{T}} \Phi\}$$

where ρ_A : AssProcVar $\to S_{\mathcal{T}}$ ranges over assumption environments.

Example. Consider an operating system in the form of a concurrent server that spawns off handler processes each time it receives a request. These processes run system calls for handling the given requests to produce a result (modeled by the action $\overline{out}$). The handler is defined as $Handler \triangleq In \parallel \overline{out}.\mathbf{0}$ where $In \triangleq in.In$. Although it is possible to communicate with request handlers through the attached channel (modeled by the action *in*), they do not react to further input. A property one would like to prove of such a server is that it stabilizes whenever it stops receiving new requests. Eventual stabilization can be formalized in the modal μ-calculus as $\mathbf{stab} \triangleq \nu X.\mu Y.\, [in]\, X \wedge \left[\overline{out}\right] Y$. We can reduce this verification task to proving that the open system modeled by the OTA

$$X : \mathbf{stab} \rhd X \parallel Handler$$

which consists of *Handler* and any stabilizing process X, eventually stabilizes.

2.5 Extended Modal Transition Systems

We propose Extended Modal Transition Systems (EMTS) as a structure for explicit state space representation of open systems with temporal assumptions. In this chapter, we summarize the main definitions around this notion.

The notion of EMTS is based on Larsen's Modal Transition Systems presented in 2.3. In addition to may and must transitions for dealing with modalities, EMTSs include sets of states (instead of single states) as targets to transitions to capture disjunctive assumptions, and a set of prohibited infinite runs defined through a coloring function to represent termination assumptions.

Definition 2.8 (EMTS)**.** An *extended modal transition system* is a structure

$$\mathcal{E} = (S_\mathcal{E}, A, \longrightarrow_\mathcal{E}^\Diamond, \longrightarrow_\mathcal{E}^\Box, c)$$

where

(i) $S_\mathcal{E}$ is a set of *abstract states*,

(ii) A is a set of *actions*,

(iii) $\longrightarrow_\mathcal{E}^\Diamond,\ \longrightarrow_\mathcal{E}^\Box \subseteq\ S_\mathcal{E} \times A \times 2^{S_\mathcal{E}}$ are *may* and *must transition relations*, and

(iv) $c :\ S_\mathcal{E} \to \mathbb{N}^k$ is a *coloring function* for some $k \in \mathbb{N}$.

A must transition from a state is a shared transition of all those closed systems captured by the state, while a may transition is a transition that may or may not be exhibited by a closed system state captured by the EMTS state.

A *run* (or may–run) of $\mathcal{E}$ is a possibly infinite sequence of transitions $\rho_\mathcal{E} = s_0 \xrightarrow{a_0}_\mathcal{E} s_1 \xrightarrow{a_1}_\mathcal{E} s_2 \xrightarrow{a_2}_\mathcal{E} \ldots$ where for every $i \geq 0$, $s_i \xrightarrow{a_i}_\mathcal{E}^\Diamond S$ for some S such that

$s_{i+1} \in S$. Must–runs are defined similarly. We distinguish between two kinds of *a-derivatives* of a state s: $\partial_a^{\Diamond}(s) \triangleq \{S \mid s \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} S\}$ and $\partial_a^{\Box}(s) \triangleq \{S \mid s \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Box} S\}$.

The coloring function c induces a set $W_{\mathcal{E}}$ of prohibited infinite runs by means of a *parity acceptance condition* (cf. [88, 38]) as follows. The function c is extended to infinite runs $\rho_{\mathcal{E}} = s_0 \overset{a_0}{\longrightarrow}_{\mathcal{E}} s_1 \overset{a_1}{\longrightarrow} \ldots$ so that $c(\rho_{\mathcal{E}}) = (c(s_0)(1) \cdot c(s_1)(1) \ldots, \ldots, c(s_0)(k) \cdot c(s_1)(k) \ldots)$ is a k-tuple of infinite words where $c(s)(j)$ denotes the j^{th} component of $c(s)$. Let $inf(c(\rho_{\mathcal{E}})(i))$ denote the set of infinitely occurring colors in the i^{th} word of this tuple. Then the run $\rho_{\mathcal{E}}$ is said to be prohibited, $\rho_{\mathcal{E}} \in W_{\mathcal{E}}$, if and only if $max\,(inf\,(c(\rho_{\mathcal{E}})(i)))$ is odd for some $1 \leq i \leq k$, i.e. in one of these k infinite words, the greatest number that occurs infinitely often is odd.

The choice of parity acceptance for capturing alternation of fixed points in modal μ-calculus formulae is natural as was noted by Emerson and Jutla [38]. The typical coloring function used in earlier parity conditions (for instance the one used by Emerson and Jutla in [38]) color states with a single natural number. It is nonetheless possible to obtain a set of state-set pairs, to encode the set of infinite runs represented by our parity condition by means of a Streett acceptance condition. Streett acceptance, introduced for Kripke structures in section 2.2, can also be used for EMTSs to select a subset of infinite runs. Given the EMTS $\mathcal{E}$, let max_j be the largest number that occurs in the j^{th} entry of the states of $S_{\mathcal{E}}$ and let the colors used in the EMTS be m-tuples. The coloring function c of the EMTS can be used to specify a set of pairs Ω such that for $L_{ij}, U_{ij} \subseteq S_{\mathcal{E}}$ where $1 \leq j \leq m$ and $1 \leq 2 * i + 1 \leq max_j$, $(L_{ij}, U_{ij}) \in \Omega$ are constructed as follows:

- $L_{ij} = \{s \in S_{\mathcal{E}} \mid c(s)(j) = 2 * i + 1\}$ and
- $U_{ij} = \{s \in S_{\mathcal{E}} \mid \exists i'.c(s)(j) = 2 * i' \wedge i' \geq i\}$

In this way, a run is *not* prohibited only if the odd color in the j^{th} entry of an infinitely often visited state is canceled out by infinitely often visiting a state which has a larger, even color in the same entry.

Next, we define a simulation relation between the states of an EMTS as a form of mixed fair simulation (cf. e.g. [51, 22]).

Definition 2.9 (Simulation)**.** $R \subseteq S_{\mathcal{E}} \times S_{\mathcal{E}}$ is a *simulation relation* on the states of $\mathcal{E}$ if whenever $s_1 R s_2$ and $a \in A$:

1. if $s_1 \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} S_1$, then there is a set of states S_2 such that $s_2 \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} S_2$ and for each $s_1' \in S_1$, there exists a $s_2' \in S_2$ such that $s_1' R s_2'$;

2. if $s_2 \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Box} S_2$, then there is a set of states S_1 such that $s_1 \overset{a}{\longrightarrow}_{\mathcal{E}}^{\Box} S_1$ and for each $s_1' \in S_1$, there exists a $s_2' \in S_2$ such that $s_1' R s_2'$;

3. if the run $\rho_{s_2} = s_2 \overset{a_1}{\longrightarrow}_{\mathcal{E}} s_2^1 \overset{a_2}{\longrightarrow}_{\mathcal{E}} s_2^2 \overset{a_3}{\longrightarrow}_{\mathcal{E}} \ldots$ is in $W_{\mathcal{E}}$ then every infinite run $\rho_{s_1} = s_1 \overset{a_1}{\longrightarrow}_{\mathcal{E}} s_1^1 \overset{a_2}{\longrightarrow}_{\mathcal{E}} s_1^2 \overset{a_3}{\longrightarrow}_{\mathcal{E}} \ldots$ such that $s_1^i R\, s_2^i$ for all $i \geq 1$ is also in $W_{\mathcal{E}}$.

We say that abstract state s_2 *simulates* abstract state s_1, denoted $s_1 \preceq s_2$, if there is a simulation relation R such that $s_1 R s_2$. Simulation can be generalized to two different EMTSs $\mathcal{E}_1$ and $\mathcal{E}_2$ in the natural way.

Labeled transition systems can be viewed as a special kind of EMTS, where: $\longrightarrow_{\mathcal{E}}^{\Box} = \longrightarrow_{\mathcal{E}}^{\Diamond}$, the target sets of the transition relation are singleton sets of states, and the set of prohibited runs W is empty.

We give the meaning of an abstract state relative to a given LTS, as the set of concrete LTS states simulated by the abstract state.

Definition 2.10 (Denotation)**.** Let $\mathcal{E}$ be an EMTS, and let $\mathcal{T}$ be an LTS. The *denotation* of abstract state $s \in S_{\mathcal{E}}$ is the set $[\![s]\!]_{\mathcal{T}} \triangleq \{t \in S_{\mathcal{T}} \mid t \preceq s\}$. This notion is lifted to sets of abstract states $S' \subseteq S_{\mathcal{E}}$ in the natural way:

$$[\![S']\!]_{\mathcal{T}} \triangleq \bigcup \{[\![s]\!]_{\mathcal{T}} \mid s \in S'\}$$

In the rest of this chapter, we assume that EMTSs obey the following *consistency* restrictions: $\longrightarrow_{\mathcal{E}}^{\Box} \subseteq \longrightarrow_{\mathcal{E}}^{\Diamond}$, $s \xrightarrow{a}_{\mathcal{E}}^{\Box} S$ implies S is non-empty, and W does not contain runs corresponding to infinite must–runs of the EMTS.

In section 2.7, we present a proof system for proving properties of abstract states. For this purpose, we define when an abstract state s satisfies a modal μ-calculus formula Φ. The global nature of the set W in EMTSs makes it cumbersome to define the denotation of a fixed point formula compositionally as a set of abstract states. We therefore give an indirect definition of satisfaction, by means of the denotation $[\![s]\!]_{\mathcal{T}}$ of a state s.

Definition 2.11 (Satisfaction)**.** Let $\mathcal{E}$ be an EMTS, $s \in S_{\mathcal{E}}$ be an abstract state of $\mathcal{E}$ and Φ be a modal μ-calculus property. Then s satisfies Φ under valuation $\mathcal{V} : \mathrm{PropVar} \to 2^{S_{\mathcal{E}}}$, denoted $s \models_{\mathcal{V}}^{\mathcal{E}} \Phi$, if and only if for any LTS $\mathcal{T}$ $[\![s]\!]_{\mathcal{T}} \models_{\mathrm{V}}^{\mathcal{T}} \Phi$ where valuation $\mathrm{V} : \mathrm{PropVar} \to 2^{S_{\mathcal{T}}}$ is induced by $\mathcal{V}$ as $\mathrm{V}(Z) \triangleq \bigcup \{[\![s]\!]_{\mathcal{T}} \mid s \in \mathcal{V}(Z)\}$.

Example. The state space of the open system introduced in the previous section is captured by the EMTS in figure 2.2. In figures 2.2 and 2.3 start states of the EMTSs are marked by a green arrow and blue, red, green circles correspond to the state colors 0, 1 and 2, respectively. For any labeled transition system $\mathcal{T}$, the processes simulated by the state s_1 are those denoted by the open term $X : \mathbf{stab} \rhd X \parallel Handler$. The EMTS consists of six abstract states, each state denoting the set of processes which it simulates. For instance, states s_5 and s_6 in the example denote all processes which can engage in arbitrary interleavings of *in* and $\overline{out}$ actions, but so that *in* has to be enabled throughout while $\overline{out}$ has not. Infinite runs stabilizing on $\overline{out}$ actions are prohibited by the coloring of s_3 and s_6.

Consider the processes a) *fix* $A.in.A$, b) *fix* $A.(in.A + \overline{out}.(fix\ B.in.B))$ and c) *fix* $A.(in.A + \overline{out}.A)$, for which corresponding EMTSs are shown in figure 2.3. In order to show that processes are denoted by the open system $X : \mathbf{stab} \rhd X \parallel Handler$, simulation relations between the start states of the EMTSs of processes and the

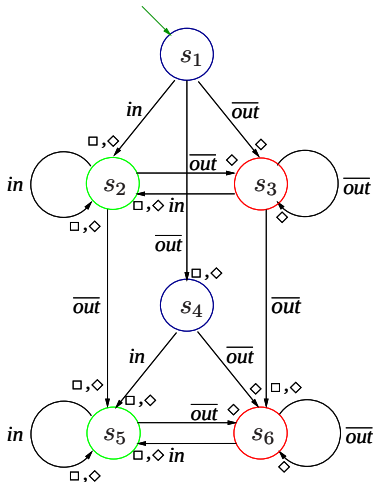


Figure 2.2: EMTS for $X : \boldsymbol{stab} \rhd X \parallel Handler$

start state of the EMTS of the open system should be established. With the help of these two figures, it is possible to see that the second process is in the denotation of this open system while the first and third processes are not:

1. The relation $R_1 = \{(t_1, s_1), (t_1, s_2)\}$ is not a simulation relation because of the second item of definition 2.9. It is not possible to build a simulation relation that contains the pair (t_1, s_1), since t_1 does not have any successors to be paired with s_4. Since the transition from s_1 to s_4 is a must transition for the action $\overline{out}$, in order to be simulated by s_1, t_1 should have an $\overline{out}$ successor.

2. The relation $R_2 = \{(t_2, s_1), (t_2, s_2), (t'_2, s_4), (t'_2, s_5)\}$ is a simulation relation for the second process and the open system. Furthermore, this is the only possible simulation relation that contains the pair (t_2, s_1).

3. The relation $R_3 = \{(t_3, s_1), (t_3, s_2), (t_3, s_4), (t_3, s_5)\}$ is the obvious candidate for a simulation relation for the third process. R_3 is not a simulation relation since the pair (t_3, s_5) does not satisfy the third condition of definition 2.9. Because the color of state s_3 and the colors of both its $\overline{out}$-successors, s_3 and s_6, are odd, processes simulated by this state are not permitted to stabilize on $\overline{out}$. But t_3 can perform such a stabilizing run, hence t_3 is not simulated by s_3.

2.6 From Specification to State Space Representation

We propose a two-phase construction ε that translates an open term $\Gamma \rhd E$ to an EMTS, denoted $\varepsilon(\Gamma \rhd E)$. In the first phase, an EMTS is constructed for each

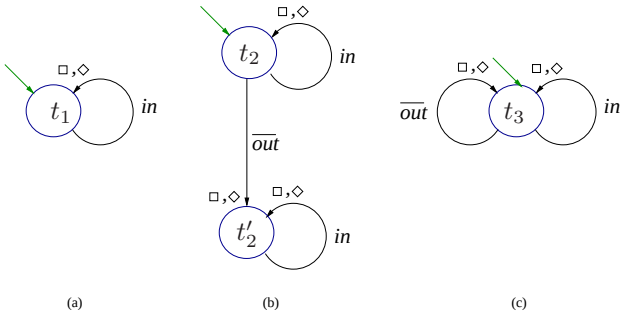


Figure 2.3: EMTSs for processes a) *fix* $A.in.A$, b)*fix* $A.(in.A + \overline{out}.(fix\ B.in.B))$ and c) *fix* $A.(in.A + \overline{out}.A)$

underspecified component. This part is essentially a maximal model construction for the modal μ-calculus. The second phase consists of combining the EMTSs produced in the first step according to the structure of the term E.

We will illustrate the construction with the use of examples. In the examples below, the set of actions is $A = \{a, b\}$. Blue, red and green circles around state names correspond to integers 0, 1 and 2 respectively and are used to indicate the color of the state. The number of circles around a state indicates the length of the color tuple for this state. The outermost circle around a state corresponds to the leftmost entry of its color, while the innermost circle corresponds to the rightmost. For example, a green outer circle in combination with a red inner circle means that the state has color (2,1). Color tuples are contracted into equivalent but shorter tuples when possible.

2.6.1 Maximal Model Construction

We define the function ε which maps modal μ-calculus formulae to triples of the shape $(\mathcal{E},\ S,\ \lambda)$, where $\mathcal{E} = (S_\mathcal{E}, A, \longrightarrow^\Diamond_\mathcal{E}, \longrightarrow^\Box_\mathcal{E}, c)$ is an *EMTS*, $S \subseteq S_\mathcal{E}$ is a set of *start states* of $\mathcal{E}$, and $\lambda : S_\mathcal{E} \to 2^{PropVar}$ is a *labeling function*.

The function is defined inductively on the structure of Φ as shown in figure 2.4. The meaning of open formulae that arises in intermediate steps are given by the by the valuation which assigns the whole set of processes $S_\mathcal{T}$ to each propositional variable. Essentially, the particular valuation used does not play a role in the final EMTS, since the properties used as assumptions of an OTA are closed.

In the definition, let $\varepsilon(\Phi_1)$ be $((S_{\mathcal{E}_1},\ A,\ \longrightarrow^\Diamond_{\mathcal{E}_1},\ \longrightarrow^\Box_{\mathcal{E}_1},\ c_1),\ S_1,\ \lambda_1)$ and $\varepsilon(\Phi_2)$ be $((S_{\mathcal{E}_2},\ A,\ \longrightarrow^\Diamond_{\mathcal{E}_2},\ \longrightarrow^\Box_{\mathcal{E}_2},\ c_2),\ S_2,\ \lambda_2)$ where $S_{\mathcal{E}_1}$ and $S_{\mathcal{E}_2}$ are disjoint sets. The new state s_{new} is not in $S_{\mathcal{E}_1}$ and a and a' are actions in A. The coloring functions $c_1 : S_{\mathcal{E}_1} \to \mathbb{N}^{k_1}$ and $c_2 : S_{\mathcal{E}_2} \to \mathbb{N}^{k_2}$ color the states of $\mathcal{E}_1$ and $\mathcal{E}_2$ with integer tuples of length k_1 and k_2 respectively.

For a set S, $S\ |_\Box$ denotes the largest transition-closed set contained in S such

that there is no element $s \in S\ |_{\Box}$ with the empty set as the target to a must transition, that is, there is no s such that for some $a \in A$, $s \xrightarrow{a}_{\mathcal{E}}^{\Box} \varnothing$ and each state s is reachable from some start state.

The EMTS for formula tt consists of the single state s_{tt} with may transitions to itself for every action (See figure 2.5(a)), while the EMTS for ff is the empty EMTS. The EMTS for a propositional variable consists of a single state with may transitions to s_{tt} for each action. Essentially, the particular valuation used for propositional variables does not play a role in the final EMTS, since the properties used as assumptions of an OTA are closed. Nevertheless, the meaning of open formulae that arises in intermediate steps are given by the by the valuation which assigns the whole set of processes $S_{\mathcal{T}}$ to each propositional variable. This is achieved by constructing the EMTS for the propositional variable Z as a single start state, which has may transitions to s_{tt} for each action (See figure 2.5(b)).

For the modal cases, a new state s_{new} is set as the start state. The EMTS for $\varepsilon([a]\,\Phi)$ has a single may transition for a, which is to the set of initial states of $\varepsilon(\Phi)$ (See figure 2.6(a)). This is to ensure all simulated processes satisfy Φ after engaging in an a. Additionally, there is a may transition to s_{tt} for all other actions. The EMTS for $\varepsilon(\langle a\rangle\,\Phi)$ includes a must transition for a from this start state to the start states of $\varepsilon(\Phi)$, along with may transitions for all actions to s_{tt} forcing the simulated processes to have an a transition to some process satisfying Φ and allowing any other transitions besides (See figure 2.6(b)).

The states of the EMTS for the conjunction of two formulae is the cross product of the states of the EMTSs constructed for each conjunct, excluding pairs with incompatible capabilities (See figure 2.7(a)). The color of a state of $\varepsilon(\Phi_1 \wedge \Phi_2)$ is the concatenation of the colors of the paired states. In the case of disjunction, the set of start states of $\varepsilon(\Phi_1 \vee \Phi_2)$ is the union of the start states of $\varepsilon(\Phi_1)$ and $\varepsilon(\Phi_2)$ which reflects the union of their denotation (See figure 2.7(b)). The color of a state is given by padding with 0's from either the left or right.

The construction for fixed point formulae is a powerset construction which is similar to the constructions given in [30] and [65] for the purpose of constructing Büchi Automata for linear time and the alternation-depth class Π_2 fragments of the μ-calculus, respectively. The states of $\varepsilon(\sigma Z.\Phi)$ consist of sets of states of $\varepsilon(\Phi)$ and its start states are singletons containing some start state of $\varepsilon(\Phi)$. An invariant of the maximal model construction is that start states do not have incoming transitions. (The case for $\varepsilon(\mathrm{tt})$ is the only exception and can be easily adapted to satisfy the invariant.) For a transition of state $q = \{s_1, \ldots, s_n\}$ of $\varepsilon(\sigma Z.\Phi)$, each state s_i has a transition in $\varepsilon(\Phi)$. A member state of the target of this transition, then, contains a derivative for each s_i. A member of the target state additionally contains an initial state of $\varepsilon(\Phi)$ if one of the derivatives included is labeled by Z. The definition of figure 2.4 makes use of the target set function $\partial_{\mathcal{P}}$ defined below.

Definition 2.12 (Target Set Function $\partial_{\mathcal{P}}$)**.** Let Φ be a modal μ-calculus formula, σ be either μ or ν, $\varepsilon(\Phi)$ be ($\mathcal{E}_1$,S,λ) where $\mathcal{E}_1 = (S_{\mathcal{E}_1}, A, \longrightarrow_{\mathcal{E}}^{\Diamond}, \longrightarrow_{\mathcal{E}}^{\Box}, c)$ is an EMTS, $S \subseteq S_{\mathcal{E}_1}$ is a set of start states, $\lambda : S_{\mathcal{E}_1} \to 2^{PropVar}$ is a function that maps states

- $\varepsilon(\mathrm{tt}) \triangleq ((\{s_{\mathrm{tt}}\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \emptyset,\ \{s_{\mathrm{tt}} \mapsto 0\}),\ \{s_{\mathrm{tt}}\},\ \{s_{\mathrm{tt}} \mapsto \emptyset\})$
  where $s_{\mathrm{tt}} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\}$ for all $a \in A$.

- $\varepsilon(\mathrm{ff}) \triangleq ((\emptyset,\ A,\ \emptyset,\ \emptyset,\ \emptyset),\ \emptyset,\ \emptyset)$

- $\varepsilon(Z) \triangleq ((\{s_{new}, s_{\mathrm{tt}}\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \emptyset,\ \{s_{\mathrm{tt}} \mapsto 0, s_{new} \mapsto 0\}),\ \{s_{new}\},\ \{s_{new} \mapsto \{Z\}, s_{\mathrm{tt}} \mapsto \emptyset\})$
  where $s_{new} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\}$ and $s_{\mathrm{tt}} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\}$ for all $a \in A$.

- $\varepsilon(\Phi_1 \wedge \Phi_2) \triangleq (((S_{\mathcal{E}_1} \times S_{\mathcal{E}_2})|_{\Box},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ W),\ (S_{\mathcal{E}_1} \times S_{\mathcal{E}_2})|_{\Box} \cap (S_1 \times S_2),\ \lambda)$ where

$$
\begin{array}{rcl}
\longrightarrow^{\Diamond}_{\mathcal{E}} & \triangleq & \{(s,r) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S' \times \cup\partial^{\Diamond}_{a}(r) \mid s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S'\} \\
 & \cup & \{(s,r) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \cup\partial^{\Diamond}_{a}(s) \times R' \mid r \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_2} R'\} \\
 & \cup & \{(s,r) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S' \times R' \mid s \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S' \wedge r \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_2} R' \wedge S' \notin \partial^{\Box}_{a}(s) \wedge R' \notin \partial^{\Box}_{a}(r)\} \\
\longrightarrow^{\Box}_{\mathcal{E}} & \triangleq & \{(s,r) \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} (S' \times \cup\partial^{\Diamond}_{a}(r)) \mid s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S'\} \\
 & \cup & \{(s,r) \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} (\cup\partial^{\Diamond}_{a}(s) \times R') \mid r \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_2} R'\} \\
c & \triangleq & \{(s,r) \mapsto c_1(s) \cdot c_2(r) \mid s \in S_{\mathcal{E}_1} \wedge r \in S_{\mathcal{E}_2}\} \\
\lambda & \triangleq & \{(s,r) \mapsto \lambda_1(s) \cup \lambda_2(r) \mid s \in S_{\mathcal{E}_1} \wedge r \in S_{\mathcal{E}_2}\}
\end{array}
$$

- $\varepsilon(\Phi_1 \vee \Phi_2) \triangleq ((S_{\mathcal{E}_1} \cup S_{\mathcal{E}_2},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c),\ S_1 \cup S_2,\ \lambda_1 \cup \lambda_2)$ with:

$$
\begin{array}{rcl}
\longrightarrow^{\Diamond}_{\mathcal{E}} & \triangleq & \longrightarrow^{\Diamond}_{\mathcal{E}_1} \cup \longrightarrow^{\Diamond}_{\mathcal{E}_2} \\
\longrightarrow^{\Box}_{\mathcal{E}} & \triangleq & \longrightarrow^{\Box}_{\mathcal{E}_1} \cup \longrightarrow^{\Box}_{\mathcal{E}_2} \\
c & \triangleq & \{s \mapsto c_1(s) \cdot 0^{k_2} \mid s \in S_{\mathcal{E}_1}\} \cup \{s \mapsto 0^{k_1} \cdot c_2(s) \mid s \in S_{\mathcal{E}_2}\}
\end{array}
$$

- $\varepsilon([a]\,\Phi_1) \triangleq ((S_{\mathcal{E}_1} \cup \{s_{new}, s_{\mathrm{tt}}\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}_1},\ c),\ \{s_{new}\},\ \lambda)$ with:

$$
\begin{array}{rcl}
\longrightarrow^{\Diamond}_{\mathcal{E}} & \triangleq & \longrightarrow^{\Diamond}_{\mathcal{E}_1} \cup \{s_{\mathrm{tt}} \overset{a'}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\} \mid a' \in A\} \cup \{s_{new} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S_1\} \\
 & \cup & \{s_{new} \overset{a'}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\} \mid a' \neq a \wedge a' \in A\} \\
c & \triangleq & c_1 \cup \{s_{new} \mapsto 0^{k_1}\} \cup \{s_{\mathrm{tt}} \mapsto 0^{k_1}\} \\
\lambda & \triangleq & \lambda_1 \cup \{s_{new} \mapsto \emptyset\} \cup \{s_{\mathrm{tt}} \mapsto \emptyset\}
\end{array}
$$

- $\varepsilon(\langle a\rangle\,\Phi_1) \triangleq \varepsilon(\mathrm{ff})$ if $S_1 = \emptyset$
  $\varepsilon(\langle a\rangle\,\Phi_1) \triangleq ((S_{\mathcal{E}_1} \cup \{s_{new}, s_{\mathrm{tt}}\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c),\ \{s_{new}\},\ \lambda)$ otherwise, with:

$$
\begin{array}{rcl}
\longrightarrow^{\Diamond}_{\mathcal{E}} & \triangleq & \longrightarrow^{\Diamond}_{\mathcal{E}_1} \cup \{s_{new} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S_1\} \cup \{s_{new} \overset{a'}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\} \mid a' \in A\} \cup \{s_{\mathrm{tt}} \overset{a'}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s_{\mathrm{tt}}\} \mid a' \in A\} \\
\longrightarrow^{\Box}_{\mathcal{E}} & \triangleq & \longrightarrow^{\Box}_{\mathcal{E}_1} \cup \{s_{new} \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S_1\} \\
c & \triangleq & c_1 \cup \{s_{new} \mapsto 0^{k_1}\} \cup \{s_{\mathrm{tt}} \mapsto 0^{k_1}\} \\
\lambda & \triangleq & \lambda_1 \cup \{s_{new} \mapsto \emptyset\} \cup \{s_{\mathrm{tt}} \mapsto \emptyset\}
\end{array}
$$

- $\varepsilon(\sigma Z.\Phi_1)\ ((2^{S_{\mathcal{E}_1}}|_{\Box}, A, \longrightarrow^{\Diamond}_{\mathcal{E}}, \longrightarrow^{\Box}_{\mathcal{E}}, c_\sigma), 2^{S_{\mathcal{E}_1}}|_{\Box} \cap \{\{s\} \mid s \in S_1\}, \lambda)$ where $\sigma \in \{\nu, \mu\}$ with:

$$
\begin{array}{rcl}
\longrightarrow^{\Diamond}_{\mathcal{E}} & \triangleq & \{\{s_1, \ldots, s_n\} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S \mid \exists i.\exists S'_i.s_i \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S'_i \wedge \\
 & & S = \partial_{\mathcal{P}}((\cup\partial^{\Diamond}_{a}(s_1), \ldots, S'_i, \ldots, \cup\partial^{\Diamond}_{a}(s_n)), S_1, \lambda_1, Z)\} \\
 & \cup & \{\{s_1, \ldots, s_n\} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S \mid \forall j.\exists S'_j.s_j \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S'_j \wedge S'_j \notin \partial^{\Box}_{a}(s_j) \wedge \\
 & & S = \partial_{\mathcal{P}}((S'_1, \ldots, S'_n), S_1, \lambda_1, Z)\} \\
\longrightarrow^{\Box}_{\mathcal{E}} & \triangleq & \{\ \{s_1, \ldots, s_n\} \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S \mid \exists i.\exists S'_i.s_i \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S'_i \wedge \\
 & & S = \partial_{\mathcal{P}}((\cup\partial^{\Diamond}_{a}(s_1), \ldots, S'_i, \ldots, \cup\partial^{\Diamond}_{a}(s_n)), S_1, \lambda_1, Z)\}
\end{array}
$$

$$
c_\nu(\{s_1, \ldots, s_n\})(j) \triangleq \begin{cases} \underset{1 \le i \le n}{maxodd}(c_1(s_i)(j)) & \text{if } \forall i.Z \notin \lambda_1(s_i) \\ \prod\limits^{even}_{s \in S_{\mathcal{E}_1}} c_1(s)(j) & \text{if } \exists i.Z \in \lambda_1(s_i) \end{cases}
$$

$$
c_\mu(\{s_1, \ldots, s_n\})(j) \triangleq \begin{cases} \underset{1 \le i \le n}{maxodd}(c_1(s_i)(j)) & \text{if } \forall i.Z \notin \lambda_1(s_i) \\ \prod\limits^{odd}_{s \in S_{\mathcal{E}_1}} c_1(s)(j) & \text{if } \exists i.Z \in \lambda_1(s_i) \end{cases}
$$

$$
\lambda \triangleq \{\{s_1, \ldots, s_n\} \mapsto \bigcup_{1 \le i \le n} \lambda_1(s_i) - \{Z\} \mid \{s_1, \ldots, s_n\} \in 2^{S_{\mathcal{E}_1}}\}
$$

Figure 2.4: Maximal Model Construction

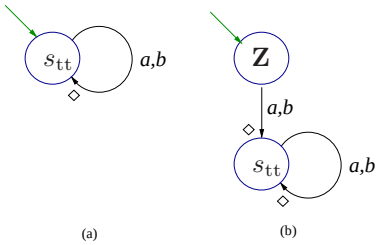


Figure 2.5: (a) $\varepsilon(\mathrm{tt})$, (b) $\varepsilon(Z)$

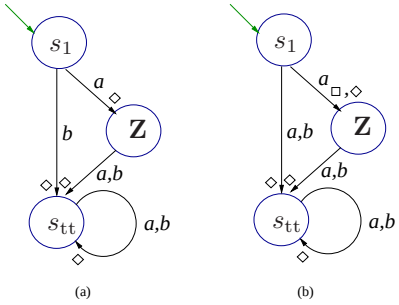


Figure 2.6: (a) $\varepsilon([a]\,Z)$, (b) $\varepsilon(\langle a\rangle\,Z)$

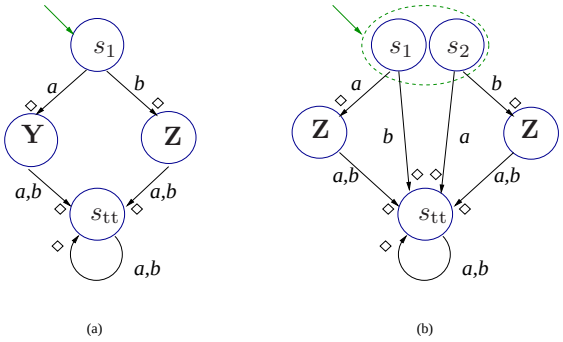


Figure 2.7: (a) $\varepsilon([a]\,Y \wedge [b]\,Z)$, (b) $\varepsilon([a]\,Z \vee [b]\,Z)$

of $\mathcal{E}$ to propositional variables, $c : S_{\mathcal{E}} \rightarrow N^k$ is a coloring function that maps states of $\mathcal{E}$ to k-tuples, and let $Z \in PropVar$ be a propositional variable. Given a tuple consisting of a target set for each element of a state of $\varepsilon(\sigma Z.\Phi)$, the function $\partial_{\mathcal{P}} : (2^{S_{\varepsilon_1}} \times \ldots \times 2^{S_{\varepsilon_1}}) \times 2^{S_{\varepsilon_1}} \times (S_{\mathcal{E}_1} \rightarrow 2^{PropVar}) \times PropVar \rightarrow 2^{2^{S_{\varepsilon_1}}}$ defines the target set of a transition of $\varepsilon(\sigma Z.\Phi)$ for this state as follows:

$$\begin{aligned}\partial_{\mathcal{P}}((S_1,\ldots,S_n),S,\lambda,Z) \triangleq\ & \{\{s_1,\ldots,s_n\} \mid \forall i.s_i \in S_i \wedge \nexists j.Z \in \lambda(s_j)\} \cup \\ & \{\{s_1,\ldots,s_n,s_0\} \mid \forall i.s_i \in S_i \wedge \\ & \qquad \exists j.Z \in \lambda(s_j) \wedge s_0 \in S\}\end{aligned}$$

Each component of the color of state q is determined by comparing the corresponding entries of the member states s_i. When for at least one of these states s_i, this entry is odd, the greatest of the corresponding odd entries is selected as the entry of q, otherwise the maximum entry is selected for the same purpose. The color of q is further updated if it contains a state s_i labeled by Z. When Z identifies a greatest fixed point formula, each entry of the constructed tuple is defined to be the least even upper bound of the integers used in this entry of $\varepsilon(\Phi)$. Whereas, when Z identifies a least fixed point formula, the least odd upper bound of the integers is the entry for the color of q. Figures 2.7(a) and 2.9(a,b) illustrate how the alternation of fixed points is handled. In this example, the innermost fixed point is a greatest fixed point which means that the color of the state labeled by the variable identifying this fixed point (**Z**) is not changed going from figure 2.7(a) to figure 2.9(a). On the other hand, the outer fixed point is a least fixed point therefore the least odd upper bound of the colors of figure 2.9(a) is computed and the result (1) is used to color the state labeled with the variable that identifies this fixed point (**Y**) in figure 2.9(b).

In figure 2.10, an example which requires colors of states to be tuples with multiple entries is given.

This part of the construction potentially causes an exponential blow-up in the number of states. Ideally, an algorithm of this step would start with the set of start state singletons and grow the EMTS by computing the target of one transition at each step. Then, the average number of states would be much less since most of the subset-states are not reachable from the start states. In figure 2.11, we can see how the state space grows from the state state singletons and figure 2.12 shows the EMTS constructed.

2.6.2 Construction for Terms

We extend the function ε to the domain of OTAs so that $\varepsilon(\Gamma \rhd E) = (\mathcal{E},S,\lambda)$, where $\mathcal{E} = (S_{\mathcal{E}}, A, \longrightarrow^{\Diamond}_{\mathcal{E}}, \longrightarrow^{\Box}_{\mathcal{E}}, c)$ is an EMTS, $S \subseteq S_{\mathcal{E}}$ is the set of *start states* of $\mathcal{E}$, and $\lambda : S_{\mathcal{E}} \rightarrow 2^{RecProcVar}$ is a *labeling function.*

The function ε is defined inductively on the structure of E as shown in figure 2.8. The EMTS corresponding to the nil process **0** consists of an abstract state without outgoing transitions, indicating that no transition is allowed for processes simulated

- $\varepsilon(\Gamma \rhd \mathbf{0}) \triangleq (\{s_{new}\},\ A,\ \emptyset,\ \emptyset,\ \{s_{new} \mapsto 0\}),\ \{s_{new}\},\ \{s_{new} \mapsto \emptyset\})$

- $\varepsilon(\Gamma \rhd X) \triangleq \varepsilon(\Phi)$ if $X \in AssProcVar$
  where $\Phi = \bigwedge_{X:\Psi \in \Gamma} \Psi$ (defaults to tt when Γ contains no assumption on X).

- $\varepsilon(\Gamma \rhd X) \triangleq (\{s_{new}\},\ A,\ \emptyset,\ \emptyset,\ \{s_{new} \mapsto 0\}),\ \{s_{new}\},\ \{s_{new} \mapsto \{X\}\})$ if $X \in RecProcVar$

- $\varepsilon(\Gamma \rhd a.E_1) \triangleq ((S_{\mathcal{E}_1} \cup \{s_{new}\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c),\ \{s_{new}\},\ \lambda_1 \cup \{s_{new} \mapsto \emptyset\})$ with:
  $\longrightarrow^{\Diamond}_{\mathcal{E}} \triangleq \longrightarrow^{\Diamond}_{\mathcal{E}_1} \cup \{s_{new} \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S_1\}$
  $\longrightarrow^{\Box}_{\mathcal{E}} \triangleq \longrightarrow^{\Box}_{\mathcal{E}_1} \cup \{s_{new} \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S_1\}$
  $c \triangleq c_1 \cup \{s_{new} \mapsto 0^{k_1}\}$

- $\varepsilon(\Gamma \rhd E_1 + E_2) \triangleq ((S_{\mathcal{E}_1} \cup S_{\mathcal{E}_2} \cup (S_1 \times S_2),\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c),\ S_1 \times S_2,\ \lambda)$
  $\longrightarrow^{\Diamond}_{\mathcal{E}} \triangleq \longrightarrow^{\Diamond}_{\mathcal{E}_1} \cup \longrightarrow^{\Diamond}_{\mathcal{E}_2} \cup \{(s,r) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S' \mid s \in S_1 \wedge r \in S_2 \wedge (s \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S' \vee r \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_2} S')\}$
  $\longrightarrow^{\Box}_{\mathcal{E}} \triangleq \longrightarrow^{\Box}_{\mathcal{E}_1} \cup \longrightarrow^{\Box}_{\mathcal{E}_2} \cup \{(s,r) \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S' \mid s \in S_1 \wedge r \in S_2 \wedge (s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S' \vee r \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_2} S')\}$
  $c \triangleq \{s \mapsto c_1(s) \cdot 0^{k_2} \mid s \in S_{\mathcal{E}_1}\} \cup \{r \mapsto 0^{k_1} \cdot c_2(r) \mid r \in S_{\mathcal{E}_2}\}$
  $\cup \{(s,r) \mapsto c_1(s) \cdot c_2(r) \mid (s,r) \in S_1 \times S_2\}$
  $\lambda \triangleq \lambda_1 \cup \lambda_2 \cup \{(s,r) \mapsto \lambda_1(s) \cup \lambda_2(r) \mid s \in S_1 \wedge r \in S_2\}$

- $\varepsilon(\Gamma \rhd \mathit{fix}\ X.E_1) \triangleq ((S_{\mathcal{E}_1},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c_1),\ S_1,\ \lambda)$ with:
  $\longrightarrow^{\Diamond}_{\mathcal{E}} \triangleq \{s \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S \mid (s \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S) \vee$
  $(\exists s_1 \in S_1.s_1 \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S \wedge\ X \in \lambda_1(s) \wedge s \text{ is reachable from } s_1)\}$
  $\longrightarrow^{\Box}_{\mathcal{E}} \triangleq \{s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S \mid (s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S) \vee$
  $(\exists s_1 \in S_1.s_1 \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S \wedge\ X \in \lambda_1(s) \wedge s \text{ is reachable from } s_1)\}$
  $\lambda \triangleq \{s \mapsto (\lambda_1(s) - \{X\}) \mid s \in S_{\mathcal{E}_1}\}$

- $\varepsilon(\Gamma \rhd E_1 \parallel E_2) \triangleq ((S_{\mathcal{E}_1} \times S_{\mathcal{E}_2} \times \{1,2\},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}},\ \longrightarrow^{\Box}_{\mathcal{E}},\ c),\ S_1 \times S_2 \times \{1,2\},\ \lambda)$
  $\longrightarrow^{\Diamond}_{\mathcal{E}} \triangleq \{(s,r,x) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S' \times \{r\} \times \{1\} \mid s \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_1} S'\}$
  $\cup \{(s,r,x) \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} \{s\} \times R' \times \{2\} \mid r \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}_2} R'\}$
  $\longrightarrow^{\Box}_{\mathcal{E}} \triangleq \{(s,r,x) \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S' \times \{r\} \times \{1\} \mid s \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_1} S'\}$
  $\cup \{(s,r,x) \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} \{s\} \times R' \times \{2\} \mid r \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}_2} R'\}$
  $c \triangleq \{(s,r,1) \mapsto c_1(s) \cdot 0^{k_2} \mid s \in S_{\mathcal{E}_1} \wedge r \in S_{\mathcal{E}_2}\}$
  $\cup \{(s,r,2) \mapsto 0^{k_1} \cdot c_2(r) \mid s \in S_{\mathcal{E}_1} \wedge r \in S_{\mathcal{E}_2}\}$
  $\lambda \triangleq \{(s,r,x) \mapsto \emptyset \mid s \in S_{\mathcal{E}_1} \wedge r \in S_{\mathcal{E}_2} \wedge x \in \{1,2\}\}$

Figure 2.8: EMTS Construction for Process Algebra Terms

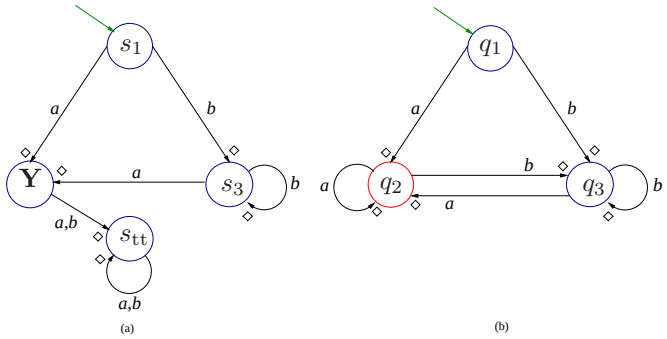


Figure 2.9: (a) $\varepsilon(\nu Z.\,[a]\,Y \wedge [b]\,Z)$, (b) $\varepsilon(\mu Y.\nu Z.\,[a]\,Y \wedge [b]\,Z)$

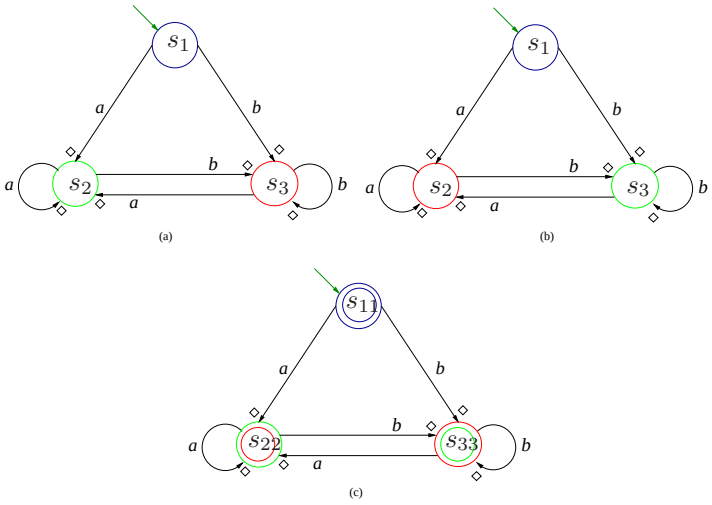


Figure 2.10: (a) $\varepsilon(\nu Y.\mu Z.\,[a]\,Y \wedge [b]\,Z)$, (b) $\varepsilon(\nu Z.\mu Y.\,[a]\,Y \wedge [b]\,Z)$, (c) $\varepsilon((\nu Y.\mu Z.\,[a]\,Y \wedge [b]\,Z) \wedge (\nu Z.\mu Y.\,[a]\,Y \wedge [b]\,Z))$

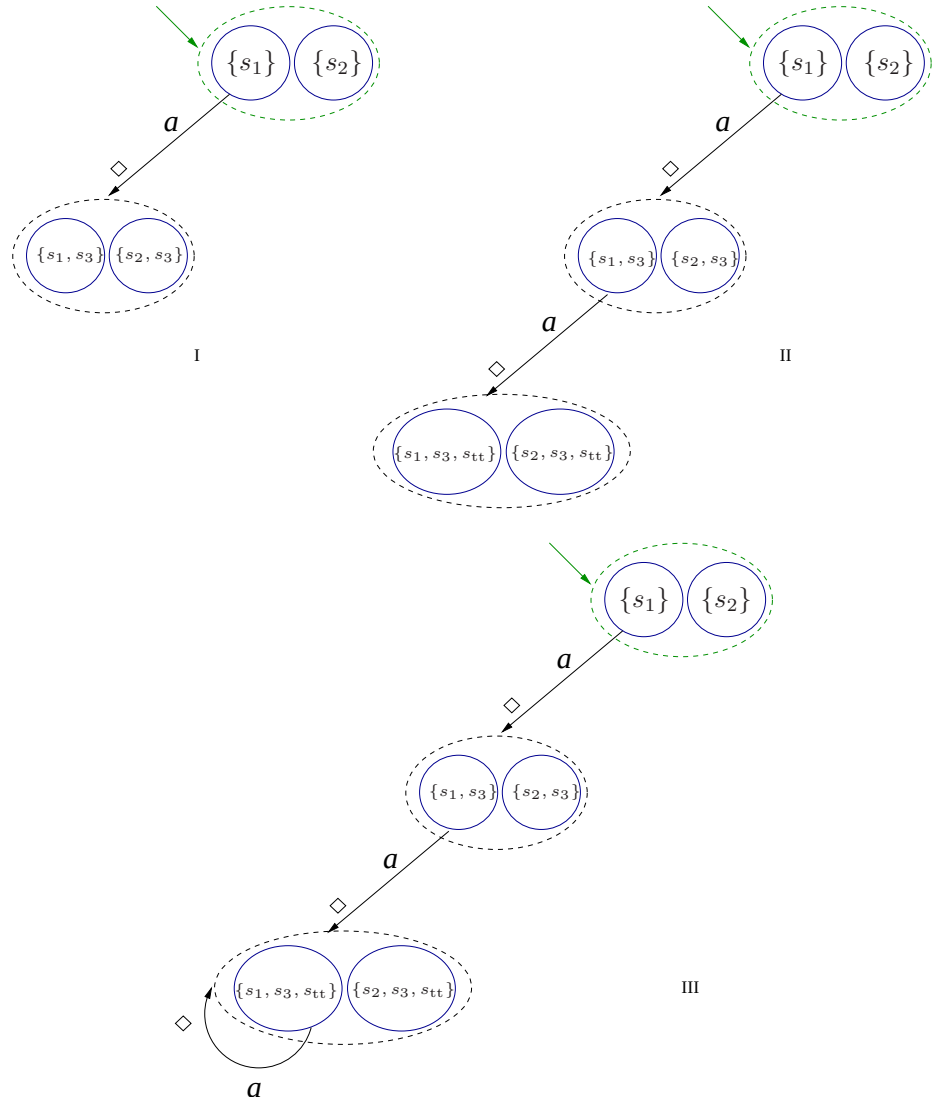


Figure 2.11: Three Steps of Constructing EMTS for $\mu Z.\, [a]\, Z \vee [b]\, Z$ from $\varepsilon([a]\, Z \vee [b]\, Z)$

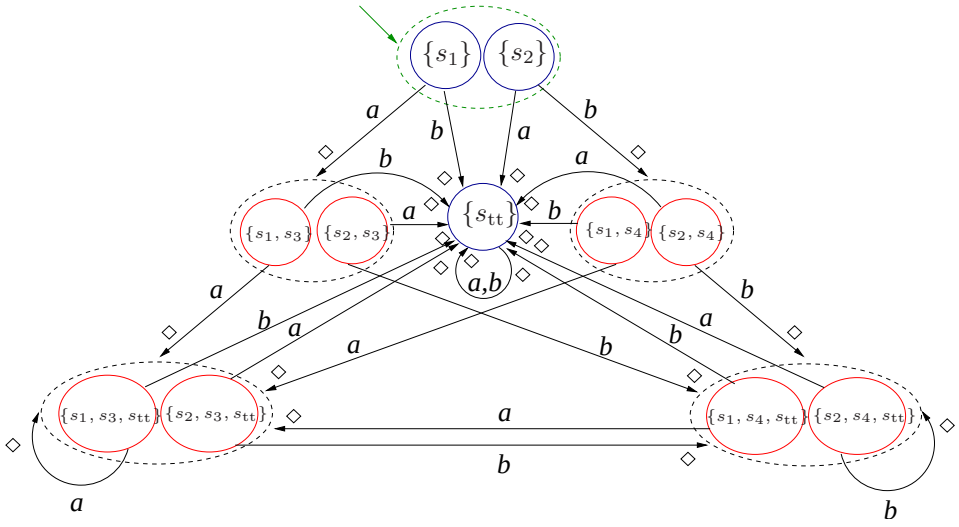


Figure 2.12: $\varepsilon(\mu Z.\,[a]\,Z \vee [b]\,Z)$

by this state. If a process variable X in the term E stands for an underspecified component of the system, that is if X is an assumption process variable, then the EMTS for X is a maximal model for the conjunction of the properties specified for this component in the assumption list Γ.

The EMTS for a recursion process variable X is a single state without outgoing transitions, since the capabilities of the processes simulated are determined by the binding *fix*-expression. The function λ labels the state X. Given the EMTS for the term of the *fix*-expression where X is free, the transitions of the start states are transferred to the states labeled by X.

The EMTS for a subterm prefixed by an action a is given by a start state with a must a-transition to the set of start states of the EMTS for the subterm. The EMTS for the sum operator consists of an EMTS where the start states are the cross product of the start states of the EMTSs for the subterms. It is assumed for this case that there are no incoming transitions to the start states of the EMTSs being combined. This is an invariant of the construction, except the case for tt which can be trivially converted to an equivalent EMTS to satisfy the property.

Finally, the states of the EMTS for a parallel composition of two components consists of a state from each component. Each state has transitions such that one of the components make a transition while the other stays in the same state. Each state is further marked by 1 or 2 to keep track of which component has performed the last transition; this is necessary to enable a run of the composition if the interleaved runs are enabled.

2.6.3 Correctness Results

The aim of the above construction is to capture by means of an EMTS exactly those behaviors denoted by the given OTA. The construction is *sound* (resp. *complete*) if the denotation of the OTA is a subset (resp. superset) of the denotation of the resulting EMTS. Our first theorem establishes the soundness and completeness of the maximal model construction.

Theorem 2.13. *Let $\mathcal{T}$ be a transition-closed LTS, Φ be a closed and guarded modal μ-calculus formula and $\varepsilon(\Phi) = (\mathcal{E}, S, \lambda)$. Then $[\![S]\!]_{\mathcal{T}} = ||\Phi||^{\mathcal{T}}$.*

Proof. The proof is done by induction on the structure of the logical formula and can be found in Appendix A.1.1. □

Our next result shows that the construction is sound and complete when assumptions exist on only one of the components that are running in parallel and the rest of the system is fully determined.

Theorem 2.14. *Let $\mathcal{T}$ be a transition-closed LTS, $\Gamma \rhd E \| t$ be a guarded linear OTA where E does not contain parallel composition, and t is closed, and let $\varepsilon(\Gamma \rhd E \,\|\, t) = (\mathcal{E}, S, \lambda)$. Then $[\![S]\!]_{\mathcal{T}}$ is equal to the set $[\![\Gamma \rhd E \,\|\, t]\!]_{\rho_0}$ up to bisimulation, where ρ_0 maps each recursion process variable X to $\mathbf{0}$.*

Theorems 2.13 and 2.14 are proved by induction on the structure of the logical formula and the process term, respectively. The proof of the latter theorem can be found in Appendix A.1.2.

In the general case, when multiple underspecified components run in parallel, we only have soundness: our construction is sound for systems without dynamic process creation. For systems with dynamic process creation, the construction does not terminate.

Theorem 2.15. *Let $\mathcal{T}$ be a transition-closed LTS, $\Gamma \rhd E$ be a guarded linear OTA where every recursion process variable in the scope of parallel composition is bound by a* fix *operator in the same scope, and let $\varepsilon(\Gamma \rhd E) = (\mathcal{E}, S, \lambda)$. Then the set $[\![S]\!]_{\mathcal{T}}$ includes $[\![\Gamma \rhd E]\!]_{\rho_0}$ up to bisimulation.*

The proof of the theorem is as the proof of theorem 2.14, but includes a more general case for parallel composition and can be found in Appendix A.1.2.

Example. Take a system made up of two components that run in parallel with the only available actions being a and b. The assumption on the first component is called **NeverDoes** a and means that this component can never perform action a and dually the assumption on the second component, **NeverDoes** b, is that action b is always disabled. The state space of the system constructed through ε is given in figure 2.13 after some simplifications. Unfortunately, the start state s_1 of this EMTS simulates any process and is clearly a proper superset of the intended set

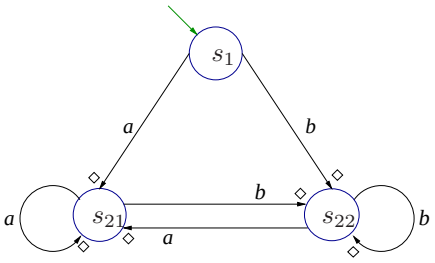


Figure 2.13: $\varepsilon(X : \mathbf{NeverDoes}\, a, Y : \mathbf{NeverDoes}\, b \rhd X \| Y)$

of processes. This state space also captures systems where an a transition becomes available although it was initially disabled while trying to capture the fact that the first component may start at an arbitrary instant. This over-approximation makes it impossible to prove some simple properties of the open system through the constructed state space. One such property is $\langle a \rangle \,\mathrm{ff} \vee \langle b \rangle\, [a]\, \mathrm{tt}$ which states that either it is impossible to perform an a initially or for each initial b-transition there exists a follower a-transition. Proving such a property of an EMTS requires the presence of a must transition.

Our last result reflects the fact that verification of open systems in the presence of parallel composition is undecidable for the modal μ-calculus in general. Completeness results can, however, be obtained for various fragments of the μ-calculus, such as ACTL, ACTL* and the logic of [53]. In our approach, the tasks of constructing a finite representation of the state space in the form of an EMTS and the task of verifying properties of this representation are separated. This allows different logics to be employed for expressing assumptions on components and for specifying system properties, giving rise to more refined completeness results.

2.7 Proof System

In this section we present the proof system we use for showing that a state of an EMTS satisfies a modal μ-calculus property. Our proof system $\Sigma_{\mathcal{E}}$ is a specialization of a proof system $\Sigma_{\mathcal{T}}$ by Bradfield and Stirling for showing properties of sets of LTS states. It is sound and complete for prime formulae.

In both systems, a proof tree is constructed using the corresponding proof rules. The construction starts with the goal and progresses in a goal-directed fashion, checking at each step if a terminal node was reached. A successful tableau (or proof) is a finite proof tree having successful terminals as leaves. Below, we contrast the major components of the two proof systems for a better understanding: sequents, proof rules and conditions for being a successful/unsuccessful terminal, in particular discharge conditions for repeat nodes.

Sequents Sequents of $\Sigma_{\mathcal{T}}$ (left) include a set of LTS states S while sequents of $\Sigma_{\mathcal{E}}$ (right) include a single state s of the EMTS. Φ and Ψ are modal μ-calculus properties. The similarity of the sequents is natural since the abstract state s corresponds to a set of concrete states $[\![s]\!]_{\mathcal{T}}$, its denotation with respect to the labeled transition system, $\mathcal{T}$.

$$S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \qquad\qquad s \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$$

Rules The rules of the two proof systems are shown in figure 2.14. Common rules of the two proof systems reduce the goal in a similar manner. The rule of disjunction in our proof system is not as powerful as the one in Stirling's. When we are to show an abstract state s satisfies property $\Phi_1 \vee \Phi_2$, we have to choose one of Φ_1 and Φ_2 for s to satisfy since s can not be split. In our proof system, $\Sigma_{\mathcal{E}}$, we can show that the state s satisfies $\Phi_1 \vee \Phi_2$, only if $[\![s]\!]_{\mathcal{T}}$ satisfies one of these properties Ψ_1 and Ψ_2 in every $\mathcal{T}$. This results in our proof system to be *prime-complete* instead of complete.

Proof trees (possibly) branch in $\Sigma_{\mathcal{E}}$ for $\Box$ and $\Diamond$-rules since each goal contains a single abstract state and not a set of states. The choice in the $\Diamond$-rule of $\Sigma_{\mathcal{T}}$ result in a goal with a single state, while the choice between $\Box$-successors in $\Sigma_{\mathcal{E}}$ results in a set of states. The Cut rule does not exist in the original proof system of Stirling. We extended $\Sigma_{\mathcal{T}}$ with the Cut rule in order to be able to reflect branchings of a proof tree of $\Sigma_{\mathcal{E}}$ in a proof tree of $\Sigma_{\mathcal{T}}$, when we translate proof trees for showing soundness and completeness of our proof system. Finally, we do not have a Thin rule. In order to have such a rule in $\Sigma_{\mathcal{E}}$, we would have to define when a state "includes" another, but for now we can only test two states for identity.

Terminals The conditions for being a terminal is also similar for $\Sigma_{\mathcal{T}}$ and $\Sigma_{\mathcal{E}}$. For both proof systems a successful tableau (or proof) is a finite proof tree having successful terminals as leaves. We now describe the conditions for a node to be a (successful/unsuccessful) terminal for both proof systems.

Terminals for $\Sigma_{\mathcal{T}}$ A node n in a proof tree labelled by a sequent $S \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$ is denoted $n : S \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$. If $n : S \vdash^{\mathcal{T}}_{\mathrm{V}} Z$ is a node where Z identifies a fixed point formula $\sigma Z.\Phi$, and there is a ancestor node $n' : S' \vdash^{\mathcal{T}}_{\mathrm{V}} Z$ above n with at least one application of a rule other than Thin and Cut in between, $S' \supseteq S$ and for any other fixed point variable Y on this path, Z subsumes Y, then node n is called a σ-terminal. So no further rules are applied to it[1]. The most recent node making n a σ-terminal is called n's companion. The conditions for a leaf sequent $R \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$ to be a successful (resp. unsuccessful) terminal are as follows.

[1]In order to show completeness for our proof system, we relax this condition without harming the soundness and completeness results.

Name	$\Sigma_{\mathcal{T}}$ Rule	$\Sigma_{\mathcal{E}}$ Rule
$\wedge$	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \wedge \Phi_2}{S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \quad S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_2}$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \wedge \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \quad s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_2}$
$\vee$	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \vee \Phi_2}{S_1 \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \quad S_2 \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_2}\ S = S_1 \cup S_2$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \vee \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1} \quad \dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \vee \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_2}$
$\Box$	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} [a]\Phi}{\partial_a(S) \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} [a]\Phi}{s_1 \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi \ \ldots\ s_n \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi}\{s_1, ..., s_n\} = \cup\, \partial^{\Diamond}_a(s)$
$\Diamond$	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} \langle a\rangle \Phi}{f_a(S) \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\ f_a : s \mapsto s' \in \partial_a(s)$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \langle a\rangle \Phi}{s_1 \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi \ \ldots\ s_n \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi}\ \{s_1, \ldots, s_n\} \in \partial^{\Box}_a(s)$
σZ	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} \sigma Z.\Phi}{S \vdash^{\mathcal{T}}_{\mathrm{V}} Z}$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \sigma Z.\Phi}{s \vdash^{\mathcal{E}}_{\mathcal{V}} Z}$
Z	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} Z}{S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\ Z$ identifies $\sigma Z.\Phi$	$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} Z}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi} Z$ identifies $\sigma Z.\Phi$
Thin	$\dfrac{S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}{R \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\ S \subset R$	
Cut	$\dfrac{\mathrm{S} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}{\mathrm{S}_1 \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \quad \mathrm{S}_2 \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\ \mathrm{S} = \mathrm{S}_1 \cup \mathrm{S}_2$	

In the rules above, σ ranges over μ and ν.

Figure 2.14: Proof Rules for $\Sigma_{\mathcal{T}}$ and $\Sigma_{\mathcal{E}}$

Successful Terminals for $\Sigma_{\mathcal{T}}$

1. $\Psi = \mathrm{tt}$, or else $\Psi = Z$, Z is free in the initial formula, and $R \subseteq \mathbf{V}(Z)$
2. $R = \varnothing$
3. $\Psi = Z$ where Z identifies a fixed point formula $\sigma Z.\Phi$, and the leaf sequent is a σ-terminal with companion node $n : S \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$, then
   a) If $\sigma = \nu$, then the terminal is successful.
   b) If $\sigma = \mu$, then the terminal is successful if there is no infinite chain of composable trails $T_0 \circ T_1 \circ T_2 \ldots$ of n (see definition 2.17 below).

Unsuccessful Terminals for $\Sigma_{\mathcal{T}}$

1. $\Psi = \mathrm{ff}$, or $\Psi = Z$ and Z is free in the initial formula with $\mathrm{R} \not\subseteq \mathbf{V}(Z)$

2. $\Psi = \langle a \rangle \Phi$ and for some $r \in R$, $\partial_a(\{r\}) = \varnothing$

3. $\Psi = Z$ where Z identifies a minimal fixed point formula $\mu Z.\Phi$, there is an ancestor node $S \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$ with at least one application of a rule other than Thin and Cut in between, $S \subset R$ and for any other fixed point variable Y on this path, Z subsumes Y.

We now explain the notion of trail used in the above definition.

Definition 2.16 (Dependent)**.** If node $n' : S' \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi'$ is an immediate successor of node $n : S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$, then state $s' \in S'$ at n' is a *dependent* of state $s \in S$ at n if:

- s $= s'$ and the rule applied to n is $\wedge$, $\vee$, σZ, Z, or Thin, or
- $s \xrightarrow{a}_{\mathcal{T}} s'$ and the rule is $\Box_a$, or
- $s' = f_a(s)$ and the rule is $\Diamond_a$ applied with choice function f_a.

Definition 2.17 ($\mathcal{T}$-Trail)**.** Assume that node n_k:$S_k \vdash^{\mathcal{T}}_{\mathrm{V}} Z$ is a μ-terminal and node n_0:$S_0 \vdash^{\mathcal{T}}_{\mathrm{V}} Z$ is its companion. A trail T of the companion node n_0 is a sequence of state–node pairs $(s_0, n_0), \ldots, (s_k, n_k)$ from state $s_0 \in S_0$ at n_0 to $s_k \in S_k$ at n_k, such that for all $0 \leq i < k$, one of the following holds:

1. $s_{i+1} \in S_{i+1}$ at n_{i+1} is a dependent of $s_i \in S_i$ at n_i, or
2. n_i is the immediate predecessor of a σ-terminal node $n' \neq n_k$ whose companion is n_j for some $j : 0 \leq j \leq i$, and $n_{i+1} = n_j$ and $s_{i+1} \in S_{i+1}$ at n' is a dependent of $s_i \in S_i$ at n_i.

Two trails T_1 and T_2 of the same companion node are *composable*, if the last pair of T_1 and the first pair of T_2 mention the same state; in this case their composition is denoted by $T_1 \circ T_2$.

Trails mention state–node pairs. Later in this chapter, we work with actual sequences of transitions (runs). We therefore define a mapping α to extract the corresponding run from a given trail.

Definition 2.18 (α: Trail to Run Conversion)**.** Let $T = (s_0, n_0), \ldots, (s_k, n_k)$ be a $\mathcal{T}$-trail of proof tree Σ. The corresponding run $\alpha(T)$ is inductively defined as follows:

$$\alpha(s, n) \triangleq \varepsilon$$

$$\alpha((s_1, n_1) \cdot (s_2, n_2) \cdot T) \triangleq \begin{cases} (s_1 \xrightarrow{a}_{\mathcal{T}} s_2) \cdot \alpha((s_2, n_2) \cdot T) & \Box_a \text{ or } \Diamond_a\text{-rule is applied to } n_1 \\ \alpha((s_2, n_2) \cdot T) & \text{otherwise.} \end{cases}$$

Terminals for $\Sigma_{\mathcal{E}}$ The conditions for being a σ-terminal and definition of companion node is similar in our proof system, where σ-terminals and their companions mention the same state. If $n : r \vdash^{\mathcal{E}}_{\mathcal{V}} Z$ is a node where Z identifies a fixed point formula, node n is called *a σ-terminal* if the following holds: there is an identical

ancestor node of n, $n' : r \vdash^{\mathcal{E}}_{\mathcal{V}} Z$ and for any other fixed point variable Y on this path, Z subsumes Y. The most recent node making n a σ-terminal is named n's *companion*. Similar to $\Sigma_{\mathcal{T}}$, no rules are applied to σ-terminals in our proof system. The conditions for a leaf node $r \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$ of a proof tree to be a successful terminal are listed below.

Successful Terminals for $\Sigma_{\mathcal{E}}$

1. $\Psi = \mathrm{tt}$, or else $\Psi = Z$, Z is free in the initial formula, and $r \in \mathcal{V}(Z)$
2. $\Psi = [a]\,\Phi$ and $\cup\partial^{\Diamond}_{a}(r) = \varnothing$
3. $\Psi = Z$ where Z identifies a fixed point formula $\sigma Z.\Phi$, and the sequent is a σ-terminal with companion node $n : r \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$, then
   a) If $\sigma = \nu$, then the terminal is successful.
   b) If $\sigma = \mu$, then the terminal is successful if every infinite run of the EMTS that corresponds to an infinite sequence of trails of the companion node n_0 is in $W_{\mathcal{E}}$. (The notion of trail is explained below.) When the set $W_{\mathcal{E}}$ is encoded using a coloring function c, the condition is that for any set S_T of trails of n_0, there should exist $1 \leq j \leq k$, so that $max(\bigcup_{T \in S_T} c(\alpha(T))(j))$ is odd. This ensures, for an infinite run $w_{n_0} = \alpha(T_1) \circ \alpha(T_2) \circ \alpha(T_3) \ldots$ where for all $i \geq 1$, T_i is a trail of n_0, that there exists some $1 \leq j' \leq k$ such that $max\,(inf(c(w_{n_0})(j')))$ is odd.

Unsuccessful Terminals for $\Sigma_{\mathcal{E}}$

1. $\Psi = \mathrm{ff}$, or else $\Psi = Z$, Z is free in the initial formula, and $r \notin \mathcal{V}(Z)$
2. $\Psi = \langle a \rangle\,\Phi$ and $\cup\partial^{\Box}_{a}(r) = \varnothing$
3. $\Psi = Z$ where Z identifies the least fixed point formula $\mu Z.\Phi$, and the sequent is a σ-terminal with companion node n_0, then the terminal is unsuccessful if there exists a set S_T of trails of n_0 such that for every $1 \leq j \leq k$, $max\,(\bigcup_{T \in S_T} c(\alpha(T))(j))$ is even. This means that some infinite run w_{n_0} of the EMTS, which corresponds to an infinite sequence of trails of the companion node n_0, is not in $W_{\mathcal{E}}$.

The notion of $\mathcal{E}$-trails of an EMTS is defined analogously to $\mathcal{T}$-trails of an LTS. Since the nodes of the proof trees mention single states, the notion of dependent becomes superfluous. What is more, $\mathcal{E}$-trails of a node always begin and end with the same state, and are thus always composable. The mapping α from definition 2.18 which converts an $\mathcal{T}$-trail to an $\mathcal{T}$-run can be applied in exactly the same way to $\mathcal{E}$-trails and proofs trees in $\Sigma_{\mathcal{E}}$. We present here the definitions of $\mathcal{E}$-trail and the corresponding mapping α for completeness.

Definition 2.19 ($\mathcal{E}$-Trail)**.** Assume that node $n_k{:}r \vdash^{\mathcal{E}}_{\mathcal{V}} Z$ is a μ-terminal and node $n_0{:}r \vdash^{\mathcal{E}}_{\mathcal{V}} Z$ is its companion. A trail T of the companion node n_0 is a sequence of state–node pairs $(r, n_0), \ldots, (r, n_k)$ such that for all $0 \leq i < k$, one of the following holds:

1. $n_{i+1} : r_{i+1} \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi_{i+1}$ is an immediate successor of $n_i : r_i \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi_i$, or
2. n_i is the immediate predecessor of a σ-terminal node $n' : r' \vdash^{\mathcal{E}}_{\mathcal{V}} Z'$ where $n' \neq n_k$ whose companion is $n_j : r' \vdash^{\mathcal{E}}_{\mathcal{V}} Z'$ for some $j : 0 \leq j \leq i$, $n_{i+1} = n_j$, and $r_{i+1} = r'$.

Definition 2.20 (α: Trail to Run Conversion)**.** The function α and is defined for $\mathcal{E}$-trails as follows:

$$\alpha(r, n) \triangleq \varepsilon$$

$$\alpha((r_1, n_1) \cdot (r_2, n_2) \cdot T) \triangleq \begin{cases} (r_1 \xrightarrow{a}_{\mathcal{E}} r_2) \cdot \alpha((r_2, n_2) \cdot T) & \Box_a \text{ or } \Diamond_a\text{-rule is applied to } n_1 \\ \alpha((r_2, n_2) \cdot T) & \text{otherwise.} \end{cases}$$

Example For the open system $X : \mathbf{stab} \rhd X \parallel Handler$, a corresponding EMTS was given in figure 2.2. Eventual stabilization of all processes denoted by the abstract state s_1 in this EMTS can be shown using $\Sigma_{\mathcal{E}}$. We include in figure 2.15 a part of this proof tree that is representative:

Node n_{15} is discharged with companion node n_9 without appealing to colorings since X identifies a greatest fixed point formula. To discharge node n_{19} with n_{16}, however, we need to make sure that all infinite runs of $\mathcal{E}$ corresponding to infinite sequences of trails of n_{16} are in W. Node n_{16} has only one unique trail $T_u = ((s_3, n_{16}), (s_3, n_{17}), (s_3, n_{18}), (s_3, n_{19}))$. The maximum color occurring in the corresponding run $c(s_3 \xrightarrow{\overline{out}} s_3)$ is 1. So we have that $max\,(c(\alpha(T_u))(1))$ is odd. Therefore we can conclude that the infinite run $s_3 \xrightarrow{\overline{out}} s_3 \xrightarrow{\overline{out}} \ldots$, which this trail gives rise to, is prohibited. Hence the terminal is successful.

Example As a further example, we show how an attempt to show the same system satisfies the property $stab2 = \mu Y.\, \nu X.\, [in]\, X \wedge [\overline{out}]Y$ fails. This property requires that the overall number of $\overline{out}$ actions is finite, though these may be interleaved with arbitrarily many *in* actions. This is not necessarily the case in this open system, for example if the process we plug in the process $fix\ A.(in.\overline{out}.A)$ for the underspecified component. This process satisfies the *stab* property and so is eligible to join the system. But the resulting system clearly violates the *stab*2 property defined above. Each *in* action is matched with at least one $\overline{out}$ action, thus infinitely many of the former action will result in infinitely many of the latter action. We expect to find no successful proof tree for the goal $s_1 \vdash^{\mathcal{E}}_{\mathcal{V}} \mu Y.\, \nu X.\, [in]\, X \wedge [\overline{out}]Y$. Since there are no diamond modalities or disjunctions in the formula *stab*2, there

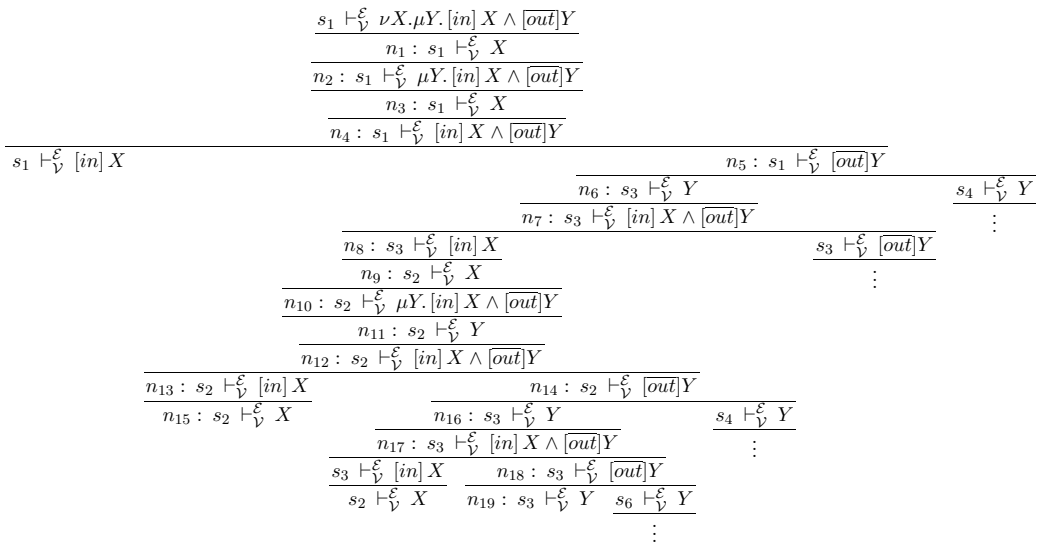


Figure 2.15: Successful Proof Tree Example

is only one possible proof tree. We present a part of this tree in figure 2.16, which includes the unsuccessful terminal node n_{20}.

In order for the terminal n_{20} to be successful for every *unique* trail T_u of the companion node n_8, there should exist $1 \leq j \leq k$ such that $max\,(c(\alpha(T_u))(j))$ is odd, since Y identifies a least fixed point formula. The unique trails of n_{20} and the corresponding partial runs are given below:

1. $T_{u1} = ((s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{13}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{18}), (s_3, n_{20}))$
   $\alpha(T_{u1}) = s_3 \xrightarrow{in} s_2 \xrightarrow{\overline{out}} s_3$

2. $T_{u2} = ((s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{13}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{17}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{18}), (s_3, n_{20}))$
   $\alpha(T_{u2}) = s_3 \xrightarrow{in} s_2 \xrightarrow{in} s_2 \xrightarrow{\overline{out}} s_3$

3. $T_{u3} = ((s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{14}), (s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{13}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{18}), (s_3, n_{20}))$
   $\alpha(T_{u3}) = s_3 \xrightarrow{\overline{out}} s_3 \xrightarrow{in} s_2 \xrightarrow{\overline{out}} s_3$

4. $T_{u4} = ((s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{14}), (s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{13}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{17}), (s_2, n_{15}), (s_2, n_{16}), (s_2, n_{18}), (s_3, n_{20}))$
   $\alpha(T_{u4}) = s_3 \xrightarrow{\overline{out}} s_3 \xrightarrow{in} s_2 \xrightarrow{in} s_2 \xrightarrow{\overline{out}} s_3$

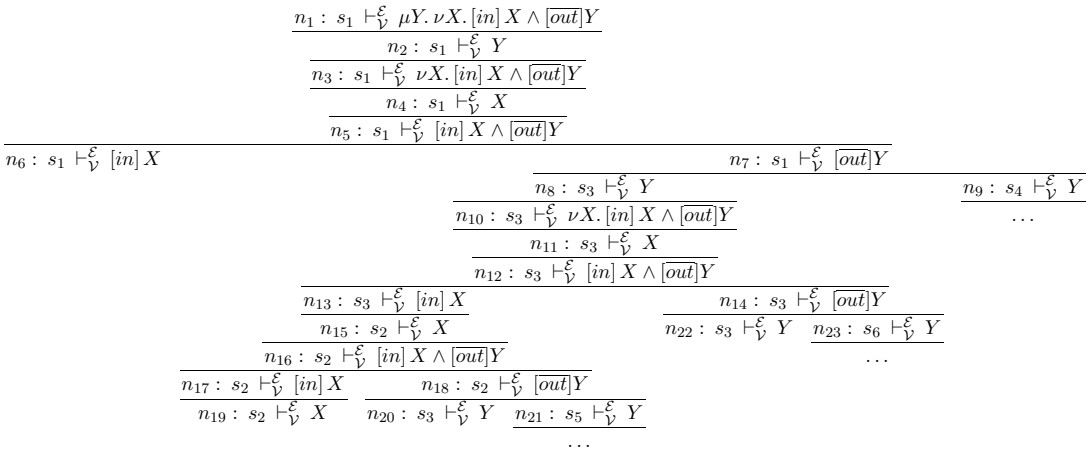


Figure 2.16: Unsuccessful Proof Tree Example

These trails show that the state s_2 occurs infinitely often in some infinite runs, which means that the color 2 will be occurring infinitely often. For all unique trails T_{ui}, where $1 \geq i \geq 4$, of the companion node n_8 with the terminal n_{20}, $max\,(c(\alpha(T_{ui}))(1))$ is 2, which is even. So the condition is not met and n_{20} is not a successful terminal.

Terminal n_{22} has the same companion node n_8. Similarly, all but one of the unique trails of n_8 with the terminal n_{22} mention s_2. (The unique trail $T_{u'} = ((s_3, n_8), (s_3, n_{10}), (s_3, n_{11}), (s_3, n_{12}), (s_3, n_{14}), (s_3, n_{22}))$ and the corresponding run $\alpha(T_{u'})$=$s_3 \xrightarrow{\overline{out}} s_3$ does not mention s_2 so the condition is met for this trail with j=1.) Since unique trails that do not meet the condition exist, this terminal is also unsuccessful.

Algorithms Based on the Proof System It is possible to give an algorithm for showing that an abstract state satisfies a formula. The application of the rules in our proof system are deterministic, except for the rules of disjunction and diamond, and in these cases the number of possible applications are finite. For the discharge condition, color sequences of all unique trails of a companion node should be checked to have a dominating odd entry. It is possible to construct all unique trails of a companion node. For a unique trail it is possible to "come back" to a state-node pair only once hence there are only finitely many unique trails for a terminal.

$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \wedge \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \quad s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_2}$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \wedge \Phi_2}{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \quad [\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_2}\wedge$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \vee \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1}$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \vee \Phi_2}{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1}\vee$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_1 \vee \Phi_2}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_2}$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_1 \vee \Phi_2}{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_2}\vee$
$s \vdash^{\mathcal{E}}_{\mathcal{V}} [a]\Phi$ and $\cup\partial^{\Diamond}_a(s) = \emptyset$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} [a]\Phi}{\emptyset \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\Box_a$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} [a]\Phi}{s_1 \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi \quad \ldots \quad s_n \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi}\ \{s_1,\ldots,s_n\} = \cup\partial^{\Diamond}_a(s)$	$\dfrac{\dfrac{\dfrac{\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} [a]\Phi}{\partial_a([\![s]\!]_{\mathcal{T}}) \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\Box_a}{[\![\cup\partial^{\Diamond}_a(s)]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Thin}^*}}{[\![s_1]\!]_{\mathcal{T}} \cup \ldots \cup [\![s_{n-1}]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \quad [\![s_n]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Cut}}{\vdots}\text{Cut}}{[\![s_1]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \quad [\![s_2]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Cut}$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \langle a\rangle\Phi}{s_1 \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi \quad \ldots \quad s_n \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi}\ \{s_1,\ldots,s_n\} \in \partial^{\Box}_a(s)$	$\dfrac{\dfrac{\dfrac{\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \langle a\rangle\Phi}{f_a([\![s]\!]_{\mathcal{T}}) \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\Diamond_a}{[\![\{s_1,\ldots,s_n\}]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Thin}^*}}{[\![s_1]\!]_{\mathcal{T}} \cup \ldots \cup [\![s_{n-1}]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \quad [\![s_n]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Cut}}{\vdots}\text{Cut}}{[\![s_1]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi \quad [\![s_2]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\text{Cut}$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} Z}{s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi}Z$ identifies $\sigma Z.\Phi$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} Z}{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi}\sigma Z$
$\dfrac{s \vdash^{\mathcal{E}}_{\mathcal{V}} \sigma Z.\Phi}{s \vdash^{\mathcal{E}}_{\mathcal{V}} Z}$	$\dfrac{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \sigma Z.\Phi}{[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} Z}Z$

Figure 2.17: Translation $\pi_{\mathcal{T}}$

2.7.1 Soundness and Completeness

To prove soundness of the proof system $\Sigma_{\mathcal{E}}$ presented above, we show that every successful tableau for sequent $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ in $\Sigma_{\mathcal{E}}$ can be transformed into a successful tableau for sequent $[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$ in $\Sigma_{\mathcal{T}}$ for any given $\mathcal{T}$. Soundness of $\Sigma_{\mathcal{E}}$ follows then from the soundness of $\Sigma_{\mathcal{T}}$ by definition 2.11.

We achieve this transformation by defining a translation from rule instances in $\Sigma_{\mathcal{E}}$ to proof trees in $\Sigma_{\mathcal{T}}$ and we show that its extension to proof trees in $\Sigma_{\mathcal{E}}$ translates successful tableaux in $\Sigma_{\mathcal{E}}$ to successful tableaux in $\Sigma_{\mathcal{T}}$.

Definition 2.21 (Translation)**.** Translation $\pi_{\mathcal{T}}$, mapping rule instances in $\Sigma_{\mathcal{E}}$ to proof trees in $\Sigma_{\mathcal{T}}$ for a given $\mathcal{T}$, is defined through figure 2.17. In the definition, f_a is a choice function.

Soundness

Lemma 2.22 (Correctness)**.** *For each rule instance in $\Sigma_{\mathcal{E}}$, translation $\pi_{\mathcal{T}}$ assigns a correct proof tree in $\Sigma_{\mathcal{T}}$, so that each premise (resp. conclusion), $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$, of the rule is matched by a leaf (resp. root), $[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$, and all unmatched leaves of the constructed proof tree are successful terminals.*

Translation $\pi_{\mathcal{T}}$ extends to proof trees in $\Sigma_{\mathcal{E}}$ in the obvious way as defined by the matching sequents.

Corollary 2.23. *For proof tree $A_{\mathcal{E}}$ with root sequent $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ in $\Sigma_{\mathcal{E}}$, $A_{\mathcal{T}} = \pi_{\mathcal{T}}(A_{\mathcal{E}})$ is a proof tree with root sequent $[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$ in $\Sigma_{\mathcal{T}}$, such that each leaf $s_i \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi_i$ of $A_{\mathcal{E}}$ is matched by a leaf $[\![s_i]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi_i$ in $A_{\mathcal{T}}$, and all unmatched leaves in $A_{\mathcal{T}}$ are successful terminals.*

Definition 2.24 (β:Trail Translation)**.** Given a proof tree $A_{\mathcal{E}}$ with nodes labeled $m_0 \dots m_l$, the function β converts a trail, $T = (t_0, n_0), \dots, (t_k, n_k)$, of the corresponding LTS tree $\pi_{\mathcal{T}}(A_{\mathcal{E}})$ to the $\mathcal{E}$-trail, $E = (s_0, n_0), \dots, (s_{k'}, n_{k'})$ by replacing each node with the matching one and deleting the remaining ones:

$$\begin{array}{rcl} \beta(\varepsilon) & \triangleq & \varepsilon \\ \beta((t_1, n_1) \cdot T) & \triangleq & \begin{cases} ((s_{1'}, n_{1'}) \cdot \beta(T)) & \text{if } n_{1'} \text{ matches } n_1 \text{ and } s_{1'} \text{ is in } n_{1'} \\ \beta(T) & \text{if no node matches } n_1 \text{ in } A_{\mathcal{E}} \end{cases} \end{array}$$

Notice that whenever a pair $(t_i,\ n_i)$ is replaced by $(s_{i'}, n_{i'})$, $t_i \in [\![s_{i'}]\!]_{\mathcal{T}}$, since node n_i contains the sequent $[\![s_{i'}]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi'$.

After a corresponding proof tree in Stirling's proof system is created, it remains to show that if the terminals of the former tree are successful, the terminals of the latter tree will also be successful.

Lemma 2.25. *$s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ implies $[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$ for any LTS $\mathcal{T}$. That is, if there is a successful tableaux for $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ in $\Sigma_{\mathcal{E}}$, then for any $\mathcal{T}$ there is a successful tableaux for $[\![s]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$ in $\Sigma_{\mathcal{T}}$.*

Proof. Assume $A_{\mathcal{E}}$ is a successful tableau for sequent $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$, and assume $\mathcal{T}$ is an LTS. To establish the result it suffices, due to Corollary 2.23, to show that each leaf of tableaux $A_{\mathcal{T}} = \pi_{\mathcal{T}}(A_{\mathcal{E}})$ matching a (successful) terminal of $A_{\mathcal{E}}$ is a successful terminal. Consider the successful terminal $m : r \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$ in $A_{\mathcal{E}}$ and the matching node of $A_{\mathcal{T}}$, $n : [\![r]\!]_{\mathcal{T}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$. The proof that the latter is also a successful terminal is trivial when $\Psi = \mathrm{tt}$, when $\Psi = Z$ and Z is free in the initial formula, when Z identifies the formula $\nu Z.\Psi'$, or when $\Psi = [a]\,\Psi'$ and $\cup\partial^{\Diamond}_{a}(r) = \varnothing$.

The only interesting case occurs when $\Psi = Z$ and Z identifies $\mu Z.\Psi'$. In such a case, the condition of subset inclusion is trivially satisfied, so it remains to show that no infinite sequences of composable trails of the companion node n' exist in $A_{\mathcal{T}}$, i.e. there is no $\kappa_{n'} = T_1 \circ T_2 \circ \dots$, such that for all $i \geq 1$, T_i begins with a pair (t_i, n'), where $t_i \in [\![r]\!]_{\mathcal{T}}$.

Assume such an infinite sequence, $\kappa_{n'} = T_1 \circ T_2 \ldots$, exists. Since the trails are composable, we also know that for all $i \geq 1$, T_i ends with the pair $(t_{i'}, n)$ for some $t_{i'} \in [\![r]\!]_{\mathcal{T}}$. Then the corresponding sequence of the EMTS for terminal node m and companion node m', $w_{m'} = \beta(T_1) \circ \beta(T_2) \ldots$ is an infinite sequence of $\mathcal{E}$-trails, and for all $i \geq 1$, $\beta(T_i)$ starts and ends with pairs (r,m') and (r, m) respectively. Because $m : r \vdash^{\mathcal{E}}_{\mathcal{V}} Z$ is a successful terminal, the corresponding infinite run $\alpha(w_{m'})$ is guaranteed to be in W. Then, by the definitions of simulation and denotation, the run $\alpha(\kappa_{n'})$ of $\mathcal{T}$ cannot be infinite, and therefore neither can $\kappa_{n'}$ be infinite. We thus reached a contradiction. □

Theorem 2.26 (Soundness)**.** *$s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ implies $s \models^{\mathcal{E}}_{\mathrm{V}} \Phi$.*

Proof. Follows directly from lemma 2.25, the soundness of Bradfield and Stirling's proof system (cf. [19, 104]), and definition 2.11. □

Completeness

We base our completeness argument on the existence of a *universal* LTS $\mathcal{U}$, having the property that every LTS $\mathcal{T}$ is isomorphic to a transition–closed sub–structure of $\mathcal{U}$. Because of the shape of our disjunction proof rule, completeness can only be shown for formulae Φ, all subformulas of which are *prime* (cf. [18]). A formula Ψ is prime if whenever it logically implies a disjunction $\Psi_1 \vee \Psi_2$ then it also implies one of the disjuncts.

By the completeness proof of Bradfield and Stirling's proof system [19], [104], whenever $[\![s]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} \Phi$ we know that there is a space of canonical proofs of $[\![s]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi$, which differ in the choice functions that are made use of in the $\Diamond$-rule applications. For the case when $\mathcal{E}$ is finite–state and all subformulas of Φ are prime, we show how to construct a proof $A_{\mathcal{U}}$ of $[\![s]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi$ that captures several of these canonical proofs using the branching provided by Cut-rule. This proof is constructed mutually with another proof $A^*_{\mathcal{U}}$ for the same goal that is built using "macros" from $\pi_{\mathcal{U}}$ instead of single rules. Both proofs are built in the weaker version of Bradfield and Stirling's proof system, where repeat nodes are not necessarily terminals. In the final step of the Completeness proof, $A^*_{\mathcal{U}}$ is translated backward into a proof of $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ using the reverse function $\pi^{-1}_{\mathcal{U}}$.

The constructions of $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$ guide each other. At each step, the rule application in $A_{\mathcal{U}}$ determines the corresponding subtree (or "macro") application taken from the range of translation $\pi_{\mathcal{U}}$ in $A^*_{\mathcal{U}}$ except when the rule is Thin. In turn, $A^*_{\mathcal{U}}$ determines the number of Cut-rules that are to be applied after each $\Box$ and $\Diamond$ rule in $A_{\mathcal{U}}$. As a result, each rule application in $A_{\mathcal{U}}$ is matched by the same rule application in $A^*_{\mathcal{U}}$ except for Thin. In order to describe this, we define in the construction process a γ function which matches nodes of $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$. If the rule/macro to be used is $\Diamond$, $A^*_{\mathcal{U}}$ additionally constraints the choice function used in $A_{\mathcal{U}}$ so that the extension of the former to the set mentioned in the matching node of $A_{\mathcal{U}}$ gives the latter. Finally, when a repeat node of $A_{\mathcal{U}}$ is to be taken as a terminal is also determined by whether the matching node is a terminal in $A^*_{\mathcal{U}}$.

$A_{\mathcal{U}}$ is similar to the canonical proofs from the completeness proof of $\Sigma_{\mathcal{T}}$ [19, 104] in the judicious application of the Thin rule, and in that the validity of the sequent is preserved at each application. Besides being applied to the goal as the first rule, Thin is applied in the rest of the tree only if the goal is of the form $S \vdash^{\mathcal{U}}_{\mathbf{V}} \sigma Z.\Psi$. The application of Thin should reduce the goal $S \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$ to the subgoal $||\Phi||^{\mathcal{T}}_{\mathbf{V}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Phi$, where $||\Phi||^{\mathcal{T}}_{\mathbf{V}}$ is defined as the set of all states in $S_{\mathcal{T}}$ that satisfy Φ under valuation $\mathbf{V}$. $A_{\mathcal{U}}$ may include applications of a special version of the Cut-rule, while no Cut-rule applications occur in canonical proofs. Cut may be applied after $\Box$ and $\Diamond$ rule in $A_{\mathcal{U}}$, in such a way that the set mentioned in each of the subgoals produced will be identical to the set mentioned in the original node. (Note that this application merely duplicates the current subgoal hence giving the possibility to combine several proof trees for the same subgoal in a single proof tree.)

Theorem 2.27 (Completeness)**.** *Let $\mathcal{E}$ be a finite–state EMTS, $s \in S_{\mathcal{E}}$, and let Φ have prime subformulae only. Then $s \models^{\mathcal{E}}_{\mathrm{V}} \Phi$ implies $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$.*

Proof. The proof is carried by translating a proof tree in $\Sigma_{\mathcal{T}}$ to one in $\Sigma_{\mathcal{E}}$ when the former tree obeys certain conditions. The transformation is carried through the inverse translation π^{-1}. The proof can be found in Appendix A.1.3 $\Box$

It is important to note that the proof system due to Dam *et al.* presented in section 2.2, can handle more complicated verification problems than is currently possible in our framework. For instance, it is possible to perform verification on systems with dynamically changing configuration due to dynamic process spawning. Nevertheless, our approach has also advantages. In the above proof system, (the explored part of) the state space is only implicitly present in a proof. Building an explicit representation of the state space allows proof reuse utilizing the (part of) the behavior already explored during proof search. When the verification task is undecidable (as in the present case, unless the temporal logic is appropriately restricted), one has to rely on interactive methods, and then visualizing the state space can be a significant aid in guiding the proof. Finally, separating the task of building the state space from the task of checking its properties (even if in a synchronized fashion as in local model checking) allows user interaction to focus on the first, potentially undecidable task, and thus be freed from the second task which is decidable for any finite representation of the state space.

2.8 Related Work

In this section, we present related work. We mention some extensions to MTSs that are in certain ways similar to EMTSs but were created for use in abstraction frameworks. Other methods that can be employed in open system verification like robust satisfaction and partial model checking are also summarized.

2.8.1 MTS Extensions for Abstraction

Abstraction is used to deal with the state space explosion problem and infinite state spaces in model checking [25]. In this approach, knowledge about system and specification-to-be-met is used to extract a simplified model of the system. An abstraction framework therefore consists of a set of concrete objects, a set of abstract objects and an abstraction relation that maps concrete objects to abstract ones. We say that an abstraction framework is *sound* if, taken a concrete system and one of its abstract representations in the framework, any property that is satisfied by the abstract model is also satisfied by the concrete model. Thus in a sound abstraction framework, desired properties of a system can be checked on the simplified model. Abstraction aims to hide away details of the model and concentrate on the aspects necessary to verify a property in order to reduce the state space while still retaining the necessary information to perform the verification.

The crucial problem in this context is to come up with an appropriate abstract model of the original system. This process is not straightforward considering that the abstract model should be both simple enough to promote efficiency and adequate for verification purposes. In *abstraction refinement*, this problem is solved by beginning the verification process with a relatively simple initial abstraction of the system and refining this model iteratively based on the feedback from the verification process. For instance, in *counter-example guided abstraction refinement* [24], the abstract model is checked against the desired property at each iteration. If the model satisfies the property, the iterative process terminates with a positive answer since the abstract model is conservative. If the verification ends unsuccessfully, a counter-example on this model is produced. If this counter-example is also a behavior of the original model, one can deduce that the formula does not hold for the original system, hence the iterative process terminates with a negative answer. But since the model is abstract and may have behaviors that do not belong to the actual system behavior, the counter-example may be spurious. In this case, the abstract model is refined using this counter-example.

Another issue to consider when developing an abstraction framework is the choice of abstract model that is used to represent concrete systems. MTSs have been used by Godefroid et al [49] as such a (abstract) representation for the purposes of checking properties written in modal μ-calculus. Since the refinement preorder on MTSs preserves such properties, if an MTS satisfies a property, any implementation of the MTS also satisfies the property. Most of the conventional state-transition models (see [25] for examples) used in abstraction frameworks are over-approximations and thus can only be used for verifying safety properties. The advantage of employing MTSs in abstraction frameworks is that MTSs define an interval thanks to the lower bound set on behavior by must transitions. This enables both safety and liveness properties to be deduced.

Two structures inspired by the notion of MTS were designed for representing state space abstractions in abstraction refinement frameworks. *Kripke modal transition systems* (KMTS) were introduced by Huth et al. [63]. KMTSs include two

labeling functions L_{must}, L_{may} that label states with atomic propositions. These specify an interval of propositions to be satisfied by each state, besides the interval of transitions specified by may and must transitions. *Generalized Kripke modal transition systems* (GKMTS), introduced for counter-example guided abstraction refinement by Grumberg and Shoham [52], are KMTSs with *hyper* must-transitions that have sets of states as targets to transitions. The simulation relation between two GKMTS states, referred to as *generalized mixed simulation* by the authors, is defined with an intention to preserve CTL properties. This relation is similar to our simulation relation as defined in definition 2.9, but without the well-foundedness requirement stated by the third part of the definition.

We have noted earlier that soundness is a key property of abstraction frameworks. If the abstract model satisfies a property, the property should also be satisfied by the concrete model. Otherwise, the abstraction is clearly not useful for verification. On the other hand, though sound, an abstraction framework may be too weak. For a concrete model and a property, it may not be possible to find an abstract model such that this model has the property. An abstraction framework is *complete* if for each concrete model and a property that the concrete model satisfies, there is an abstract model for the concrete model that satisfies the property. Completeness is an important property for an abstraction framework regardless of the way abstractions are obtained from concrete models. Of particular interest is abstraction frameworks that are complete when only finite models are considered as abstractions.

There is a strong relationship between the existence of maximal models in a framework and its completeness. For a concrete model M and a formula ϕ that it satisfies, a maximal model of ϕ can always be used as the abstract model for M. (Completeness should not be confused with the finite-model property; there should be a clear abstraction relationship between the concrete models and their abstractions, for instance like our simulation relation or a Galois connection as in abstract interpretation [29].) Hence, if a maximal model exists for each property in the framework, the framework is complete, as noted by Dams and Namjoshi in [35]. The authors further note that any abstraction framework that uses one of the aforementioned models (MTS, KMTS, GKMTS) as abstractions is incomplete with respect to branching-time logics such as CTL and the modal μ-calculus. They state that modal transition systems must be extended with hyper must-transitions and fairness conditions to achieve completeness for existential and liveness properties, respectively.

In order to create a complete framework using finite models for branching-time logics, Dams *et al.* introduce *focused transition systems* (FTS) as abstractions [34]. FTSs are essentially KMTSs extended with the focus capability which enables splitting abstract states into sub-states and with a fairness constraint given by a set of infinite sequences of states. The focus feature achieves what we achieve by hyper transitions and the fairness constraint is given as a set of prohibited runs like the earlier version of the EMTS notion, presented in [1]. Both an abstract state and less abstract states that are obtained by splitting an abstract state can be included

in a FTS, and the relationship of between the abstract state and its sub-states is explicitly given. Both the satisfaction relation between FTSs and propositional modal μ-calculus formulae and the abstraction relation between FTSs is defined in terms of (infinite) games. An FTS may not satisfy a property (i.e. the "model checking game" for an FTS and a property given by an alternating tree automaton can fail) even if all concrete systems abstracted by this FTS have the property. What is important for the authors is that, given a property and a concrete system that satisfies the property, there exists an FTS that satisfies the property and that abstracts the concrete system. The authors present a maximal model construction that converts property automata to FTS. Possibly because an algorithmic maximal model construction is not their goal, the authors do not concern themselves with presenting the fairness constraints in a finite manner as we do in the EMTS notion.

The same authors propose in [35] to use tree automata as abstractions in abstraction frameworks. The abstraction relation between automata is naturally determined by the language accepted by automata, i.e. automaton A abstracts automaton B if A accepts all trees accepted by B. Since checking tree-language inclusion is expensive (EXPTIME-hard), the authors define a simulation pre-order on automata that is sound. Deciding the existence of such a simulation relation between two (finite) automata is in NP and can be done by a deterministic algorithm. Such an algorithm can be used to check whether an automaton satisfies a property by checking the existence of a simulation relation between the automaton and the property automaton. The use of automata as abstract models simplifies the proofs of soundness and completeness for the framework.

2.8.2 Other Methods for the Verification of Open Systems

The term open system has been used in the literature for referring to systems whose behavior depends on its interaction with the environment, and is not fully determined by its internal state [56, 72, 7]. In this sense of the term, a property of an open system M is a property that is satisfied by the composition of the system with any environment M'. For such an open system M to satisfy property Φ, then, the composed system $M \mid M'$ should satisfy the property. In [72], this notion of satisfaction is called *robust satisfaction* and is considered for systems where M is given as a finite state (possibly nondeterministic) Moore machine that communicates with the environment via input and output variables. The problem is considered for the logics CTL, CTL* and μ-calculus and is solved by reduction to the emptiness problem of alternating tree automata. Given a system as the process algebra term E_M, the problem of robust satisfaction can be stated in our framework as showing that the open system modeled by the OTA

$$X : \mathrm{tt} \rhd E_M \mid X$$

satisfies Φ where no unbound process variable occurs in E_M. It is important to note that, in our current framework transitions of the system M can not be disabled by any other component X that runs in parallel since we use CSP-like parallel

composition without communication. In order to model such a dependency of X on the environment, a synchronous notion of parallel composition should be employed instead.

Another method that takes advantage of the compositional nature of the system to deal with state space explosion is *partial model checking* introduced by Anderson [8]. In this method, computing the state space of a concurrent system is avoided by removing some component while transforming the specification accordingly. Given a process t and a property Φ, Andersen defines transformation compute the *quotient property* Φ/t, making use of compositional reasoning. For any process t' that t is composed with, the quotient property Φ/t satisfies the following:

$$t'|t \models \Phi \qquad \Longleftrightarrow \qquad t' \models \Phi/t$$

The "only if" direction of this equivalence allows us to model check t' against the quotient property Φ/t instead of model checking $t'|t$ against the original property Φ. The task of model checking the system is then simplified, assuming the size of the quotient property is not much larger than the size of the original property. In order to keep the size of the quotient property reasonably small, it is simplified at each step with the help of heuristics.

Martinelli observes in [83] that "security protocols can be conveniently described by open systems". He gives two examples. In the first example, an attacker tries to listen to the conversation between the two agents A and B. This can be modelled by the open system consisting of the process algebra terms E_A and E_B which represent the agents and X, a placeholder for the attacker with unpredictable behavior:

$$X : \mathrm{tt} \rhd E_A \mid E_B \mid X$$

The second example again mentions two parties willing to communicate, but this time one of them can not be trusted. Let A and B be these two agents and suppose B has unknown behavior and may try to exploit the protocol to gain advantage. If the agent A is specified by the term E_A, in this open system X is a placeholder for the possibly malicious agent B:

$$X : \mathrm{tt} \rhd E_A \mid X$$

In these two examples, it is possible to show that the communication is secure in any scenario through showing properties of the open systems above provided the agents are specified in a suitable process algebra and a suitable logic is used for expressing properties. However, in [83] the verification is performed in a slightly different manner. Only systems with one unknown component are considered. Suppose the known participants of the system are given as the term E_S and the desirable property of the system is Φ, then partial model checking techniques mentioned above are used to find a property Ψ on the unknown component X such that

$$X : \mathrm{tt} \rhd E_S \mid X \text{ has property } \Phi \qquad \Longleftrightarrow \qquad X \text{ has property } \Psi$$

Then, the remaining task is to find if Ψ is satisfiable, that is to find if there exists a malicious partner which may have the property Ψ. In the first system above, for instance, this would mean that a potential attacker exists that can obtain the secret message. Although his verification method is different, Martinelli's work is important for us since it provides us with a possible application area.

2.9 Conclusion

2.9.1 Summary and Contributions

Summary In this chapter, a finite structure is introduced that captures the state space of open systems modeled as basic parallel processes, provided the component assumptions are given in the modal μ-calculus. We provide a method that extracts such a structure from open system descriptions in the form of process algebra terms with assumptions, and show it sound for terms without dynamic process creation and complete for systems with a single underspecified component. We also adapt an existing proof system for the purposes of showing behavioral properties of open systems. The proof system, based on the state space representation of the open system, is sound for the logic and complete for prime formulae.

We offer a framework for verification of open systems based on state space representation. The open system is input to the framework as a process algebraic term with assumptions. A state space representation is extracted from this specification. Open system properties can then be checked on the state space representation by means of a proof system. The main feature of the approach is that the focus of the verification process is on understanding the behavior of the system rather than proving properties. System behavior is captured by a finite structure which can easily be visualized; the more tedious part of showing properties through this representation can be performed automatically. This is an advantage over other methods for open system verification, especially when a visualization of the behavior is important, for instance for finding counter-examples in debugging.

Contributions Below are the contributions of this first part of the thesis:

- a finite structure (EMTS) suitable for the representation of the state space of open systems in that it supports:
  - visualization of the state space, i.e. the system behavior,
  - graphical specification,
  - state space exploration for interactive techniques,
  - verification,
  - proof reuse,
- a sound and prime-complete proof system that can be used to prove properties using the state space representation,

- a maximal model construction for the modal μ-calculus that builds an EMTS with the same denotation as the formula,
- and an automatic construction that extracts the state space of an open system described as an open process algebraic term; the construction is exact provided that the open system has a single unknown component and over-approximating when the open system does not include dynamic process creation.

2.9.2 Future Work

Characterization The correctness results on the automatic construction of EMTSs from open system specifications can be made more precise in a number of ways. First of all, theorem 2.15 states that the construction over-approximates if the open system includes parallel composition but no further information is given on precision. We believe that any additions of must-transitions or the subtraction of may-transitions in an effort to make the constructed structure more precise would result in an under-approximation. Such a result would indicate that our construction comes as close to capturing the open term with an EMTS as possible. A more practically interesting problem is to determine which temporal properties can still be shown using the constructed EMTS, in the presence of the over-approximation. The question can be put more formally as follows: for which class of temporal properties Φ, it is the case that the open system $\Gamma \rhd E$ satisfies Φ if the start states of the constructed EMTS satisfy Φ?

$$\forall \mathcal{T}.\ (\varepsilon(\Gamma \rhd E) \text{ satisfies } \Phi \Rightarrow [\![\Gamma \rhd E]\!]_{\mathcal{T}} \text{ satisfies } \Phi)$$

Yet another set of correctness results could be established by using different logics to express assumptions on components and to express open system properties.

Parallel composition In order to extend the scope of the open systems handled in our framework, it is essential to consider different types of parallel composition in system specification and eventually in state space construction. Currently we consider only systems where components act in parallel but independently of each other, which leaves out many interesting open systems.

Interactive exploration The problem of verifying open systems when assumptions are expressed in modal μ-calculus is undecidable in general due to parallel composition (consider for instance dynamic process spawning [33]). Therefore, user interaction is necessary to perform the verification task in general. Hence, the integration of interactive methods to the framework would enable it to handle a larger class of open systems. A most natural way to accomplish this is to introduce interactive state space exploration, which corresponds to the symbolic execution of OTAs in a stepwise manner. Some extensions to the notion of EMTS like sub- and super-states, and to the notion of OTA like transition assertions that keep historical information about component behavior are foreseen as necessary for this purpose. Another major problem in this context would be to fix a scheme for setting the

colors of states in order to model terminating behavior, especially when fixed point alternation is present in some component assumption.

Tool development Automatic construction of section 2.6 can be formulated as an algorithm along with the proof system of section 2.7. These algorithms complete the automatization of the framework and lay the basis for tool development. A tool for the verification of open systems based on the work presented in this chapter can potentially contain the following features:

- *Specification*: The specification of the open system is to be given either in the form of an OTA or can be directly provided as an EMTS. The OTA may include assumptions expressed in different fragments of modal μ-calculus.
- *Automatic State Space Extraction* The construction presented in section 2.6 can be used for automatic extraction of the state space when the specification of the system is given as an OTA.
- *Interactive State Space Exploration* As an alternative, interactive exploration may be employed for forming the (partial) state space when automatic extraction is not possible. This feature is to appear similar to the "simulation" command of Concurrency Workbench [27].
- *Visualize State Space* EMTSs can be visualized by the aid of a graph visualizer. (Minimization may be necessary though for a comprehensible visualization.)
- *Verification* Verification of modal μ-calculus properties of the state space will be possible by the implementation of an algorithm combining the encoding of the proof system of section 2.7 and a proof search strategy.

Case Studies A tool as outlined above can be used for the evaluation of the approach through case studies. Though the possible applications are limited by the lack of modeling of data in the framework, it is nevertheless possible to find examples that illustrate how helpful the tool is for working with open systems. Applications in security, similar to those suggested by Martinelli in [83], can pose as realistic case studies. These case studies may require, however, different logics and process algebra be employed in the framework.

Chapter 3

Program Models for Compositional Verification

3.1 Introduction

In this chapter, we focus on the verification of open systems in a compositional verification setting. Here we model programs as flow graphs, in contrast to process terms of the previous chapter. Furthermore, we describe the extraction of flow graphs from actual code (Java bytecode). A maximal model construction is also employed here to create models for partially specified components of open systems, though this construction is somewhat simpler as the logic used to specify "missing" components is only a fragment of the modal μ-calculus used in the previous part.

Compositional verification, as was formerly introduced, addresses the problem of proving the correctness of a system based on the properties of its components. This requires to reason about the correctness of the system in a modular fashion: reasoning about components separately and then deducing a property about the composed system. Such modular reasoning allows verification techniques to scale, since the verification of large systems can be made feasible by dividing the problem into the verification of system components. But what is equally important is that compositional verification enables a certain flexibility in realizing a correct system. It enables to work with specifications in the design phase of software development, and still to have assurance in system correctness. This is necessary, for example, for the development of safe open platforms with the ability to accommodate mobile components. Designers of such a platform would not have access to all the mobile components that may be installed on the platform after it has been put to use, but can still guarantee correctness properties thanks to compositional reasoning. The guarantee for the correct functioning of the platform can be formulated as a set of properties which hold of the platform, provided certain assumptions hold on mobile components. These assumptions can then be checked at runtime (whenever a new component is to join the system) and can furthermore be used as guidelines by

mobile component developers. Compositional verification techniques thus provide support for secure dynamic loading of applications by specifying local requirements on applications to be installed, and verifying that these are sufficient to guarantee the global security requirements.

In this chapter, we focus on the compositional verification method introduced by Gurov, Huisman and Sprenger (see [53] for an overview of the approach and previous results). The method supports the following abstract compositional verification principle, where $\mathcal{G}_1$ and $\mathcal{G}_2$ are programs with procedures (i.e. components), modeled as *control flow graphs*, and $\uplus$ denotes control flow graph composition:

$$\frac{\mathcal{G}_1 \models \sigma \qquad \theta_I(\sigma) \uplus \mathcal{G}_2 \models \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models \phi}\ \mathcal{G}_1 : I \tag{3.1}$$

Informally, this rule lays out two tasks to prove that the composition $\mathcal{G}_1 \uplus \mathcal{G}_2$ satisfies property ϕ, it is sufficient to find a "local" property σ of flow graph $\mathcal{G}_1$ (typically an unavailable component) for which one can verify that: *(i)* σ indeed holds for $\mathcal{G}_1$, and *(ii)* the local property σ ensures the global property ϕ. Task *(i)* is deferred until component $\mathcal{G}_1$ becomes available. Task *(ii)* assumes knowledge of the names of the provided and required methods of $\mathcal{G}_1$ (i.e. its interface I), and is achieved by constructing a maximal model for the local property in the form of a control flow graph which respects the provided interface (denoted with $\theta_I(\sigma)$ above) and by showing that its composition with $\mathcal{G}_2$ satisfies ϕ. In both tasks, the verifications can be performed algorithmically, using finite-state and pushdown automata-based model checking, respectively.

We are interested in both *structural* and *behavioral* control flow properties of programs. A structural property is a property of the (finite) flow graph itself, such as "every path from the entry of method m_1 to a call instruction to method m_2 passes a call instruction to method m_3". A behavioral property is a property of the (infinite state) behavior induced by the flow graph, such as "in any execution of the program, method m_1 calls method m_2 at most once". All formulae are expressed in the fragment of the modal μ-calculus [70] with boxes and greatest fixed-points only. The technique of Gurov *et al.* supports the principle above for the case where the local requirement σ is a structural property. In general, behavioral properties do not give rise to a unique maximal flow graph. To handle local behavioral requirements, behavioral requirements should be precisely characterized by a set of structural requirements (see [61] for such a translation).

A maximal flow graph with respect to a property σ is a flow graph that simulates all other flow graphs satisfying property σ. This notion is based on the notion of maximal model for compositional reasoning [51] as was introduced in the previous chapter, but in addition takes the provided interface into account: a maximal flow graph only simulates flow graphs with the same interface. We assume all local properties to be structural, therefore use a maximal model construction (detailed in [53]) for structural properties. Furthermore, the way the components are composed in the framework is rather simple in contrast to the systems of the previous chapter: composition $\uplus$ of the flow graphs corresponds to set union.

The program model of this technique captures only sequential control flow, and does not distinguish exceptional from normal control flow. In this part of the thesis, we extend the model with exceptions and multi-threading, closely following the semantics described in the Java language and virtual machine specifications [50, 81]. The extensions not only allow to increase the precision of the models, but also to explicitly reference exceptional and concurrent behavior in property specifications. Moreover, we prove that for both extensions the compositional verification principle still applies.

A tool set has been developed to support the compositional verification method and its utility has been demonstrated on an industrial case study provided by smart card producer Gemplus [62]. The tool set which will be described in greater detail in this chapter, contains, among other components, the *program analyzer* that extracts the program model from Java bytecode classes. The program analyzer is a static analysis tool, built on top of the SOOT Java Optimization Framework [109]. In this chapter, we also describe how models that capture exceptional and multi-threaded control flow can be extracted from bytecode programs and give experimental results for the new implementation of the program analyzer that produces models with the former extension.

Overview. In section 3.5, we present related work. Section 3.2 gives a brief overview of the approach of Gurov *et al.* to compositional verification with definitions for the basic program model and described the tool set. In section 3.3, we present an extension of the program model for precise representation of exceptional behavior, while section 3.4 is concerned with the extension with multi-threading. Finally, section 3.6 draws conclusions and discusses further extensions.

3.2 Compositional Verification of Sequential Programs with Procedures

First we give a summary of the approach of Gurov *et al.* and their earlier results on compositional verification of control-flow properties for sequential flow graphs without exceptions. We also briefly describe the tool set supporting the method. Details of both the framework and the tool set can be found in [53].

3.2.1 Program Model

Both control flow graph structure and behavior are defined in terms of specifications which we formalize below.

Definition 3.1 (Model, Specification)**.** A *model* over a set of labels L and a set of atomic propositions A is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a labeled transition relation, and $\lambda\colon S \rightarrow \mathcal{P}(A)$ a valuation assigning to each state a set of atomic propositions. A *specification* $\mathcal{S}$ is a pair $(\mathcal{M}, \mathbb{E})$, with $\mathcal{M}$ a model and $\mathbb{E} \subseteq S$ a set of *entry* states.

The reachable part of a specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ is defined by $\mathcal{R}(\mathcal{S}) = (\mathcal{M}', \mathbb{E})$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from $\mathbb{E}$. Simulation on two models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The disjoin union $\mathcal{M}_1 \uplus \mathcal{M}_2$ is defined as $(S_1 \uplus S_2, L_1 \cup L_2, \{in_i(s) \xrightarrow{a} in_i(s') | s \xrightarrow{a} s' \in \mathcal{M}_i\}, A_1 \cup A_2, \lambda)$, where $\lambda(in_i(s)) = \lambda_i(s)$ and in_i (for $i \in \{1, 2\}$) injects S_i into $S_1 \uplus S_2$. The disjoint union is lifted to specifications in the straightforward way and serves as the composition operation in 3.1. The simulation relation between specifications is standard.

Definition 3.2 (Specification Simulation)**.** Specification $(\mathcal{M}_1, \mathbb{E}_1)$ *is simulated by* specification $(\mathcal{M}_2, \mathbb{E}_2)$, denoted $(\mathcal{M}_1, \mathbb{E}_1) \leq (\mathcal{M}_2, \mathbb{E}_2)$, if there is a simulation R on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in \mathbb{E}_1$ there is some $t \in \mathbb{E}_2$ with $(in_1(s), in_2(t)) \in R$.

Notice that simulation is preserved by disjoint union.

$$\mathcal{S}_1 \leq \mathcal{T}_1 \wedge \mathcal{S}_2 \leq \mathcal{T}_2 \Rightarrow \mathcal{S}_1 \uplus \mathcal{S}_2 \leq \mathcal{T}_1 \uplus \mathcal{T}_2 \tag{3.2}$$

Let $\mathcal{M}eth$ be an infinite set of method names, and let $\mathcal{C}ont\mathcal{V}al$ be a possibly infinite set of control values (disjoint from $\mathcal{M}eth$) specific for each instantiation of the model (in the program model with exceptions, for instance, it is a set of exception names). Both sets are disjoint from any reserved symbols.

Every control flow graph comes equipped with an interface, specifying the provided and required methods, and the set of legal control values.

Definition 3.3 (Flow Graph Interface)**.** A *flow graph interface* is a triple $I = (I^+, I^-, C)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of *provided* and *required* methods, and $C \subseteq \mathcal{C}ont\mathcal{V}al$ is a finite set of control values, respectively. We say I is *closed* if $I^- \subseteq I^+$. The *composition* of two interfaces $I_1 = (I_1^+, I_1^-, C_1)$ and $I_2 = (I_2^+, I_2^-, C_2)$ is defined as $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-, C_1 \cup C_2)$.

Method specifications are the basic building blocks of flow graphs. For the basic program model for sequential programs with procedures, method specifications are defined as below.

Definition 3.4 (Basic Method Specification)**.** A *method graph* for $m \in \mathcal{M}eth$ over a set $M \subseteq \mathcal{M}eth$ of method names is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$, where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e. each node is tagged with the method name). A *method specification* for $m \in \mathcal{M}eth$ over M is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ such that $\mathcal{M}_m$ is a method graph for m over M and $\mathbb{E}_m \subseteq V_m$ a non-empty set of *entry points* of m.

The atomic proposition r is used to mark the return nodes of the method, i.e. a node $v \in V_m$ is a *return point* of m if it is tagged with the atomic proposition r, i.e. if $r \in \lambda_m(v)$.

A flow graph is a collection of method specifications, joined with the disjoint union operator $\uplus$.

```
class Number {

   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }
}
```

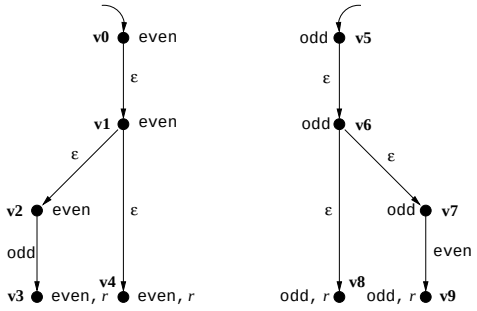


Figure 3.1: A simple Java class and its flow graph

Definition 3.5 (Flow Graph Structure)**.** *Flow graphs $\mathcal{G}$ with interface I*, written $\mathcal{G} : I$, are inductively defined by

- $(\mathcal{M}_m, \mathbb{E}_m) : (\{m\}, M, C)$ if $(\mathcal{M}_m, \mathbb{E}_m)$ is a method specification for m over M and C,
- $\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ if $\mathcal{G}_1 : I_1$ and $\mathcal{G}_2 : I_2$.

A flow graph $\mathcal{G} : I$ is *closed* if its interface I is closed. We use $\leq_s$ to denote *structural simulation* between flow graphs.

In the basic program model, a flow graph $\mathcal{G} : I$ is a model over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$ as given by the method specification for this model.

Example 3.6. Figure 3.1 shows a simple Java class and the (simplified) flow graph it induces in the basic program model. The flow graph consists of two method specifications - one for method `even` and one for method `odd`. Entry nodes are depicted as usual through edges without source.

3.2.2 Flow Graph Extraction

The tool set developed to support this compositional verification technique contains the *Program Analyzer (PA),* that extracts flow graphs from Java (bytecode) classes. These graphs are over-approximations of actual program behaviors, as specified by the Java semantics. PA is built on top of the Soot Java Optimization Framework [109].

Soot produces control flow graphs similar to ours, which are then further processed by PA to produce control flow graphs as described above. Bytecode programs are first converted by Soot into code in an intermediate language called Jimple. Then, a safe over-approximation of the application's call graph is produced, utilizing a class hierarchy analysis. For example, if the analysis cannot resolve which

method will be called by a virtual method call, a call edge is generated for every method that can potentially be invoked by this call. Further, Soot produces a control flow graph for each method, abstracting away all values. PA incorporates the information coming from the call graph to control flow graphs produced by Soot to form flow graphs in the format of the program model. Extending PA for a different program model amounts to using additional information produced by Soot's different analysis when translating Soot's control flow graphs into flow graphs of the program model.

Flow Graph Behavior

The behavior of a flow graph $\mathcal{G}$, denoted $b(\mathcal{G})$, is also defined as an instance of the general notion of specification. Since the local guarantees must be properties over the flow graph structure, we only have to define the behavior of closed flow graphs. Simulation on the behavioral level is denoted by $\leq_b$: $\mathcal{G}_1 \leq_b \mathcal{G}_2 \Leftrightarrow b(\mathcal{G}_1) \leq b(\mathcal{G}_2)$. As will be discussed later, for the compositional verification principle to apply for a concrete program model, structural simulation has to imply behavioral simulation:

$$\mathcal{G}_1 \leq_s \mathcal{G}_2 \quad \Rightarrow \quad \mathcal{G}_1 \leq_b \mathcal{G}_2 \tag{3.3}$$

Basic Program Model The behavior of the basic flow graphs (where $\mathcal{C}ontVal = \varnothing$) is defined as follows.

Definition 3.7 (Basic Flow Graph Behavior)**.** Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : (I^+, I^-)$ be a closed flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The *behavior* of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bs}, A_b, \lambda_b)$, s.t. $S_b = V \times V^*$, that is, states are *configurations* of control points and stacks, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_{bs}$ is defined as follows:

[transfer]	$(v, \sigma) \xrightarrow{\tau}_{bs} (v', \sigma)$	if	$v \xrightarrow{\varepsilon}_m v',\ v \models \neg r$
[call]	$(v_1, \sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{bs} (v_2, v_1' \cdot \sigma)$	if	$m_1, m_2 \in I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1',\ v_1 \models \neg r,$ $v_2 \models m_2,\ v_2 \in E$
[return]	$(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{bs} (v_1, \sigma)$	if	$m_1, m_2 \in I^+,\ v_2 \models m_2 \wedge r,\ v_1 \models m_1$

The set of entry states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$, where ϵ denotes the empty sequence.

Example 3.8. Consider the flow graph from Example 3.6. Because of possible unbounded recursion, it induces an infinite-state behavior. One example execution of the program is represented by the following path from an initial to a final configu-

ration:

$$(v_0, \epsilon) \xrightarrow{\tau}_{bs} (v_1, \epsilon) \xrightarrow{\tau}_{bs} (v_2, \epsilon) \xrightarrow{\text{even call odd}}_{bs} (v_5, v_3) \xrightarrow{\tau}_{bs} (v_6, v_3) \xrightarrow{\tau}_{bs}$$
$$(v_7, v_3) \xrightarrow{\text{odd call even}}_{bs} (v_0, v_9 \cdot v_3) \xrightarrow{\tau}_{bs} (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_{bs}$$
$$(v_4, v_9 \cdot v_3) \xrightarrow{\text{even ret odd}}_{bs} (v_9, v_3) \xrightarrow{\text{odd ret even}}_{bs} (v_3, \epsilon)$$

Flow graph behavior for the basic model can be viewed as the behavior of a pushdown automaton (PDA) (see [53], Def.34). Thus, behavioral properties can be verified using PDA model checking (see [21] for a survey of verification techniques for infinite-state systems). Notice further that for basic flow graphs, structural simulation implies behavioral simulation (i.e. (3.3) holds), see [53], Thm.36.

3.2.3 Properties over Flow Graphs

As property specification language, we use a fragment of the modal μ-calculus [70] with boxes and greatest fixed-points only. A variety of useful safety properties of program control flow structure and behavior are expressible in this fragment, as illustrated in [53]. Let L be a set of labels, A a set of atomic propositions, and V a set of propositional variables. The formulae of the logic are inductively defined by:

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\phi$$

where $p \in A$, $a \in L$ and $X \in V$.

Satisfaction of properties by graphs is defined in terms of the general notion of specification in the standard way (see [70] or section 2.4). We use $\models_s$ and $\models_b$ to denote satisfaction at the structural and behavioral levels, respectively: $\mathcal{G} \models_s \phi \Leftrightarrow \mathcal{G} \models \phi$, and $\mathcal{G} \models_b \phi \Leftrightarrow b(\mathcal{G}) \models \phi$.

Example 3.9. For the flow graph in the basic program model from Example 3.6, the structural formula $\nu X.\,[\texttt{even}]r \wedge [\texttt{odd}]r \wedge [\varepsilon]X$ expresses the property "on every path from a program entry node, the first encountered call edge leads to a return node", in effect specifying that the program is tail-recursive. The behavioral formula $\neg\texttt{even} \vee \nu X.\,[\texttt{even call even}]\mathsf{ff} \wedge [\tau]X$ expresses the property "in every program execution that starts in method `even`, the first call is not to method `even` itself".

3.2.4 Maximal Flow Graphs

As mentioned above, the compositional verification technique we use is based on the construction of maximal models. A maximal model construction is provided in [53] for the logic described above. The process consists of a step-wise transformation of the formula into a semantically equivalent normal form, for which a direct mapping to maximal models is given. Since the logic used here does not include diamond modalities and least fixed points, the models are representable as standard transition systems. They do not require the extensions such as a partition of the transitions into *may* and *must* transitions (since there is no existential modality)

and fairness constraints (since no least fixed point is present). The models are constructed as specifications in the sense of Def. 3.1.

When a component is partially specified (by a property σ and an interface I), rather than given as an implementation, we construct a model to represent this component as a flow graph that simulates all flow graphs with interface I satisfying σ. However, a maximal model for the property need not be a flow graph since it may not be legal. Therefore, the interface needs to be also taken into account in the construction.

Definition 3.10. (Maximal Flow Graph) Let I be an interface and σ be a property, then the graph $\mathcal{S}_\sigma^{\mathcal{I}}$ is the *maximal flow graph* for property σ and interface I if it is the maximal model (over labels and atomic propositions as induced by I) of the property σ.

$$\forall \mathcal{S}.(\mathcal{S} \leq_s \mathcal{S}_\sigma^{\mathcal{I}} \Leftrightarrow \mathcal{S} \models \sigma \wedge \mathcal{R}(\mathcal{S}) : I)$$

The maximal model construction described above can be employed for maximal flow graph construction provided that I can be formulated by a *characteristic formula* that precisely defines all legal flow structures with interface I. More precisely, for interface I, the formula ψ_I is a characteristic formula if the following holds:

$$\forall \mathcal{S}.(\mathcal{S} \models \psi_I \Leftrightarrow \mathcal{R}(\mathcal{S}) : I) \tag{3.4}$$

Finally, given an interface I, a mapping θ_I from formulae to flow graphs is a maximal flow graph construction if for any property σ, $\theta_I(\sigma)$ is a maximal flow graph for property σ and interface I. Provided a maximal model construction θ (in the standard sense) for the logic, and the characteristic formula ψ_I for the interface I, $\theta_I(\sigma)$ can be implemented as $\theta(\psi_I \wedge \sigma)$.

Basic Program Model In the basic program model, flow graphs with interface I are models over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$ that can be characterized by the following formula ([53, Th. 31]), essentially specifying that every state is labeled by a unique method name that is preserved along edges:

$$\sigma_I \;=\; \bigvee_{m \in I^+} \nu X.P_m \wedge [I^-, \varepsilon]X \qquad P_m \;=\; m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m'$$

3.2.5 Compositional Verification

As was informally presented in the introduction, the compositional verification method of this framework is based on the following principle. (The rule is repeated here for readability purposes.) The aim is to deduce that the system which consists of the composition of the flow graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ has the property ϕ, when the flow graph $\mathcal{G}_1$ is not available but is specified by the local property σ and its interface I. This amounts to the problem of verifying the correctness of the open system consisting of the composition of a component partially specified by σ and

I and the component $\mathcal{G}_2$. In order to perform verification on the open system, we first construct the maximal flow graph with respect to property σ and interface I, denoted $\theta_I(\sigma)$. The model for the open system can then be constructed by composing this maximal flow graph with the flow graph $\mathcal{G}_2$ and checked to respect the global safety property ϕ. The open system can later be "closed" by any flow graph $\mathcal{G}_1$ with interface I and property σ joining the system. When $\mathcal{G}_1$ is to join the open system, it suffices to check that it respects σ (provided it is known to respect the interface I) in order to conclude that $\mathcal{G}_1 \uplus \mathcal{G}_2$ satisfies ϕ:

$$(\mathsf{compos})\ \frac{\mathcal{G}_1 \models_s \sigma \qquad \theta_I(\sigma) \uplus \mathcal{G}_2 \models_b \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \phi}\ \mathcal{G}_1 : I$$

If local properties σ are restricted to structural properties, the rule is sound and complete as was shown in [53], Thm.39. This result is established using the following properties:

(i) Logical satisfaction characterizes simulation, and vice versa, which holds for the logic of section 3.2.3, thus making it suitable for our purposes. This property can be stated as follows: there exists a mapping χ from finite specifications to formulae, and a mapping (maximal model construction) θ from formulae to finite specifications, such that for any specifications $\mathcal{S}, \mathcal{S}_1$ and finite $\mathcal{S}_2$ (see [53, Ths. 8, 15]):

$$\mathcal{S}_1 \leq \mathcal{S}_2 \Leftrightarrow \mathcal{S}_1 \models \chi(\mathcal{S}_2) \quad \text{and} \quad \mathcal{S} \models \phi \Leftrightarrow \mathcal{S} \leq \theta(\phi) \tag{3.5}$$

(ii) Flow graphs with interface I can be logically characterized in the sense of 3.4 and the mapping θ_I is a maximal flow graph construction in the sense of section 3.2.4.

(iii) Structural simulation is preserved by flow graph composition (the current setup satisfies this as stated by 3.2):

$$\mathcal{A}_1 \leq_s \mathcal{B}_1 \wedge \mathcal{A}_2 \leq_s \mathcal{B}_2 \Rightarrow \mathcal{A}_1 \uplus \mathcal{A}_2 \leq_s \mathcal{B}_1 \uplus \mathcal{B}_2$$

(iv) Structural simulation implies behavioral simulation (the current setup satisfies this as stated by 3.3):

$$\mathcal{A} \leq_s \mathcal{B} \Rightarrow \mathcal{A} \leq_b \mathcal{B}$$

The results (i) and (iii) are general results about specifications, that can be reused provided the extended program models are instantiations of the general notion of specification. The results (ii) and (iv) have to be re-established for each extension of the basic program model to show that the compositional verification result still applies.

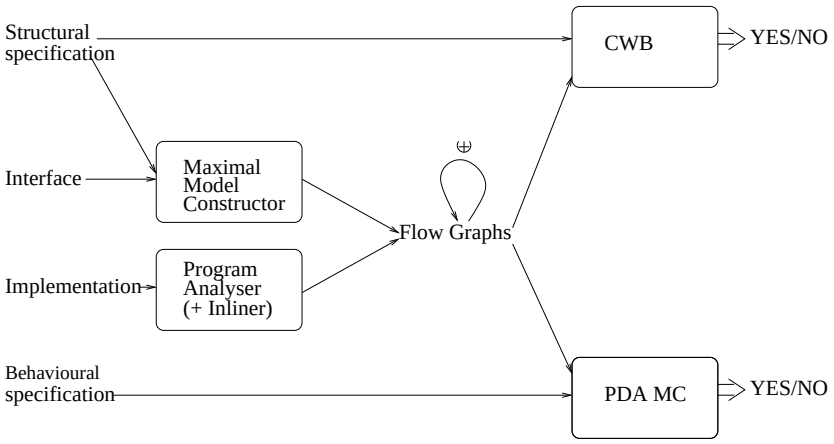


Figure 3.2: Tool Set for Compositional Verification

3.2.6 A Tool Set for Compositional Verification

A tool set was implemented to support the compositional verification method described above for the basic program model presented in section 3.2.2. figure 3.2 gives a general overview of its architecture.

For each component, an implementation (in Java bytecode), or a structural property restricting its possible implementations and an interface is required as input. If the code of the implementation is provided, the Program Analyzer is used to extract a flow graph (and if necessary, we use the Inliner to abstract the flow graph to public methods,i.e.only to methods mentioned in the interface [53]). If the specification of the component is provided instead, we construct a maximal flow graph of the given property as described in section 3.2.4 using the Maximal Model Constructor. Composition $\uplus$ of the resulting flow graphs amounts to a concatenation of the textual graph representations.

The tool set also implements translations of flow graphs into models which serve as input for different model checkers. In order to check structural properties, we exploit the fact that flow graphs can be viewed as finite Kripke structures, and convert flow graphs to CCS models. Since structural properties are μ–calculus formulae, the verification can then be done using standard model checking tools such as the Concurrency Workbench (CWB) [26]. To verify that a composed system respects a behavioral safety property, we view the behavior of a flow graph as an infinite state model generated by a Pushdown Automaton (PDA), and apply PDA model checking. We are not aware of an efficient, off-the-shelf model checker for (alternation–free) modal μ–calculus properties of PDAs. We are currently developing one ourselves.

The extensions to the Program Analyzer for handling exceptional and multi-

threaded control flow are described in the following sections. Extending the Maximal Model Constructor, Inliner and the translation into CCS and PDA models is straightforward, and not discussed further.

The tool set has been evaluated on the PACAP case study [20], an electronic purse developed for smart cards. In PACAP, a smart card may contain one purse applet and several loyalty applets, which interact to exchange information. The case study describes a potential "bad scenario" in terms of an illicit interaction involving the purse applet and the loyalty applets, one of which is malicious. The goal of the verification, presented in detail in [53], is to ensure the absence of such illicit interactions for the given implementations of the purse and loyalties.

3.3 Exceptional Control Flow

Our first extension to the program model is for the purposes of modeling the raising and catching of exceptions as detailed in the Java Virtual Machine Specification [81]. Program behavior in the presence of exceptions is sensitive to the type of the exceptional object thrown and caught. Since we do not represent data, we need to add a set of exception names to the model. For this, we take $\mathcal{C}ontVal$ (which is the empty set in the basic program model) to be $\mathcal{E}xcp$, an infinite set of exception names. We define method specifications over $M \subseteq \mathcal{M}eth$ and $E \subseteq \mathcal{E}xcp$, accordingly.

In a flow graph with exceptions, a control point may be tagged with an exception: the state is said to be exceptional if the current control point is tagged with an exception (*cf.* the state immediately after an exception is thrown, before it is caught [81]). Model extraction from actual bytecode models every instruction that might raise an exception with several transfer edges, one leading to a normal and the others leading to exceptional control points (for all possible exceptions). Explicit throw statements are modeled as internal transfer edges that always lead to an exceptional point. Catch statements are implicit: they are modeled as internal transfer from an exceptional to a normal control point.

At behavioral level, the main difference with the basic model is that the decision of which control point execution resumes after completion of a method call is postponed to the time of return, depending on whether the method call returns normally, or with an exception. Model extraction for a method that may terminate with an exception produces multiple edges labeled with this method, ending in control points tagged with exceptions, in addition to an edge that ends in a normal control point. When a method is called, the set of all possible return points (exceptional and normal) is pushed on the call stack (instead of a single one), so that the appropriate control point can be selected upon return.

Below, we instantiate the compositional verification principle for flow graphs with exceptions in such a way that conditions (ii) and (iv) from section 3.2.5 are met, by defining structure and behavior appropriately. We also discuss how model extraction is adapted, and give typical example properties that refer to the exceptional structure or behavior of a flow graph.

3.3.1 Program Model with Exceptions

As mentioned above, we take $\mathcal{C}ontVal$ as $\mathcal{E}xcp$. Interfaces of flow graphs with exceptions are thus of the form (I^+, I^-, E), where $E \subseteq \mathcal{E}xcp$. We use $I^{\mathcal{E}}$ to extract the exception component from the interface.

Method specifications are similar to method specifications in the basic program model, except that exceptions are also atomic propositions.

Definition 3.11. (Method Specification with Exceptions) A *flow graph with exceptions* for $m \in \mathcal{M}eth$ over sets $M \subseteq \mathcal{M}eth$ and $E \subseteq \mathcal{E}xcp$ is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\} \cup E$, $m \in \lambda_m(v)$ for all $v \in V_m$, and for all $e, e' \in E$, if $\{e, e'\} \subseteq \lambda_m(v)$ then $e = e'$, *i.e.,* each control point is tagged with at most one exception. A *method specification with exceptions* for $m \in \mathcal{M}eth$ over M and E is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ s.t. $\mathcal{M}_m$ is a flow graph with exceptions for m over M and E and $\mathbb{E}_m \subseteq V_M$ a non-empty set of *entry points* of m.

We use the following abbreviation: $v \models E \Leftrightarrow \exists e \in E.v \models e$. Method specifications with exceptions have to satisfy two *wellformedness* constraints: (1) entry nodes are not exceptional: $\forall v \in \mathbb{E}_m.v \not\models I^{\mathcal{E}}$; and (2) all outgoing edges from exceptional control points are internal transfer edges ending in a normal control point: $\forall v, v' \in V, e \in I^{\mathcal{E}}, l \in L_m.v \models e \wedge v \xrightarrow{l} v' \Rightarrow l = \varepsilon \wedge v' \not\models I^{\mathcal{E}}$. The second constraint is not strictly necessary, but keeps the behavior of flow graphs clean: catching an exception always results in a normal state in the same method. Throughout, we will assume all method specifications to be wellformed.

3.3.2 Extracting Flow Graphs with Exceptions from Java Classes

We extended the Program Analyzer to handle exceptions. This is accomplished in the following manner. Explicit throw statements give rise to internal transfer edges ending in an appropriately labeled exceptional control point. All other instructions that might raise an exception (such as accessing a reference, which can lead to a NullPointerException) are modeled by a choice: the current control point has multiple outgoing edges labeled ε, one ending in a normal control point and all others ending in appropriate exceptional control points. To model method invocations, edges are labeled either ε, modeling the case that the invocation instruction raises an exception, or with the method name. At most one of the edges labeled with the method name ends in a normal control point, modeling normal termination of the method, all others lead to an exceptional control point, corresponding to exceptional returns from the method. The exceptional control points are either tagged with an exception listed in the method's throw clause, or with a runtime exception that can be thrown (and not caught) in the method. The analysis of which exceptions might be returned by a method is transitive with respect to the call graph.

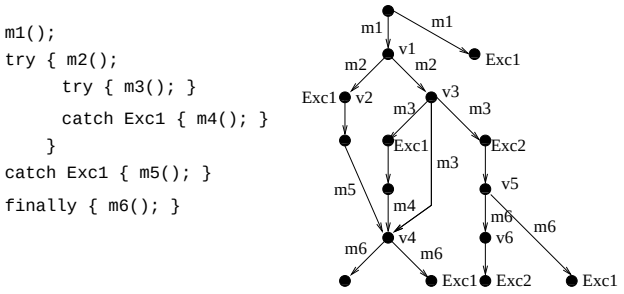


Figure 3.3: Example extraction for a try-catch-finally statement

To illustrate how PA extracts a flow graph from a `try-catch-finally` block, figure 3.3.2 shows an example code fragment[1] together with the corresponding flow graph. We assume that `Exc1` and `Exc2` are the only exceptions; `m1`, `m2` and `m3` and `m6` can throw `Exc1`, while `m3` can also throw `Exc2`. (For presentation purposes, some nodes are named.) A `try-catch` is modeled by branches in the control flow: each instruction in the try-block that could raise an exception has an outgoing edge to an exceptional control point (*e.g.*, the call to `m2` in `v1` can lead to normal point `v3`, or to exception point `v2`). If the exception is handled by one of the catch clauses, the only outgoing edge from this point leads to the control flow of the corresponding clause. For example, in `v2`, the exception is caught by the outer catch clause, leading to a call of `m5`. All edges that correspond to normal termination of the `try-catch` (*i.e.,* termination of the try-block, and termination of all catch-clauses) lead to the same control point, where the flow graph modeling the next instruction starts. If the `try-catch` block is followed by a `finally`-clause, at each possible exit of the `try-catch` block (*e.g.,* nodes `v4` and `v5` in figure 3.3.2), the graph extracted for the `finally` clause is inserted. In case the `try-catch` block ended with an exception, the exception is saved until the end nodes of the graph of the `finally` clause, thus the internal nodes of the `finally` graph are not tagged with this exception. However if an end node of the `finally` graph is normal, an edge is added to rethrow the exception. For example, if the call to `m6` in `v5` ends normally in `v6`, then `Exc2` is re-thrown. The end node of a `finally` clause can thus be either normal, tagged with an exception thrown in the `finally` block or with the exception inherited from the `try-catch` block (in case no exception is thrown by the `finally` block itself).

In order to see the results of graph extraction on a realistic piece of software, we analyzed a simulation application built on top of the JavaSim library, a tool for building discrete event process-based simulation[2]. We considered 140 types of

[1]For illustrative purposes, the extraction is described in terms of source code, however the actual implementation works on bytecode.

[2]Available via the JavaSim homepage: `http://javasim.ncl.ac.uk.`

exceptions, checked as well as unchecked, all subtypes of class `Exception`. The exceptional control flow graph includes 55 methods in 14 classes (approximately 640 lines of code), of which 7 classes belong to the JavaSim library. On a Pentium4 2.2GHz computer with 512MB memory pool, the call graph construction takes 3 minutes, and can be decreased substantially by instrumenting Soot to prevent the analysis of Java API methods. It takes 1,5 seconds to create the control flow graph, which contains 1450 nodes and 1466 edges.

3.3.3 Flow Graph Behavior with Exceptions

Modeling the behavior of flow graphs with exceptions requires a different use of the call stack than that in the basic program model. In the basic model, the return point is determined at the time of method call and is pushed on the call stack at this time. However, this is not possible when modeling exceptional behavior as it cannot be predicted at the time of the call whether termination will be normal or exceptional. Therefore, the transition that models the method call is adapted to push the set of all possible return points on the call stack. The return transition then selects the appropriate return point, i.e. with the matching exception (if any). In addition, we introduce transition labels $\mathsf{throw}\ e$ and $\mathsf{catch}\ e$; these labels make raising and recovering from exceptions observable for specification purposes.

Definition 3.12. (Behavior with Exceptions) Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The *behavior* of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{be}, A_b, \lambda_b)$ s.t. $S_b \in V \times (\mathcal{P}(V) \backslash \{\varnothing\})^*$, *i.e.,* states are pairs of control points and stacks of non-empty sets of nodes, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{\tau\} \cup \{l\ e \mid l \in \{\mathsf{throw}, \mathsf{catch}\},\ e \in I^{\mathcal{E}}\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\rightarrow_{be}$ is defined as follows:

[transfer]	$(v, \sigma) \xrightarrow{\tau}_{be} (v', \sigma)$	if	$m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,$ $v \not\models I^{\mathcal{E}},\ v' \not\models I^{\mathcal{E}}$
[call]	$(v_1, \sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{be} (v_2, V \cdot \sigma)$	if	$m_1, m_2 \in I^+,\ v_1 \models \neg r,\ v_1 \not\models I^{\mathcal{E}}$ $v_2 \models m_2,\ v_2 \in \mathbb{E}$ $V = \{v \mid v_1 \xrightarrow{m_2}_{m_1} v\},\ V \neq \varnothing$
[return]	$(v_2, V \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{be} (v_1, \sigma)$	if	$m_1, m_2 \in I^+,\ v_1 \models m_1,\ v_2 \models m_2 \wedge r,$ $v_1 \in V,\ \forall e \in I^{\mathcal{E}}.v_1 \models e \Leftrightarrow v_2 \models e$
[throw]	$(v, \sigma) \xrightarrow{\mathsf{throw}\ e}_{be} (v', \sigma)$	if	$m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,\ v' \models e$
[catch]	$(v, \sigma) \xrightarrow{\mathsf{catch}\ e}_{be} (v', \sigma)$	if	$m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \wedge e$

The set of initial states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$.

As for the basic model, the behavior of a flow graph with exceptions is the behavior of a PDA, and hence PDA model checkers can again be used for verification of behavioral properties. Since there is a close correspondence between flow graph

structure and behavior, structural simulation between flow graphs with exceptions implies their behavioral simulation (thus condition (iv) of section 3.2.5 holds).

Theorem 3.13. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be flow graphs with exceptions. If $\mathcal{G}_1 \leq_s \mathcal{G}_2$ then $\mathcal{G}_1 \leq_b \mathcal{G}_2$.*

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define relation R_b by (where $|\sigma|$ denotes the length of σ, and $\sigma(i)$ the i^{th} element in σ):

$$(v,\sigma)R_b(v',\sigma') \Leftrightarrow vRv' \wedge |\sigma| = |\sigma'| \wedge \forall i < |\sigma|.\forall w \in \sigma(i).\exists w' \in \sigma'(i).wRw'$$

It can be easily checked that R_b is a behavioral simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

3.3.4 Properties over Flow Graphs with Exceptions

Modeling exceptional control flow in flow graphs not only allows their behavior to be approximated with greater accuracy, but it also allows to express and verify properties that are related to exceptions (both at structural and at behavioral level). Typical properties of a flow graph with exceptions $\mathcal{G} : I$ expressible in our logic are:

- Exception $e \in I^{\mathcal{E}}$ is never thrown*:* $\nu X.\neg e \wedge [-]X$ (where $[K]\phi$ abbreviates $\bigwedge_{a\in K}[a]\phi$ and '$-$' stands for L). This property can be expressed both at structural and at behavioral level but with slightly different meanings. For the same flow graph, the behavioral property may be satisfied while the structural may not do to the fact that presence of recursion may render certain nodes unreachable at the behavioral level.
- Exception $e \in I^{\mathcal{E}}$ is always caught within the method where it is thrown: $\nu X.(\neg e \vee \neg r) \wedge [-]X$ (again, this property can be expressed both at structural and behavioral level).
- After exception $e \in I^{\mathcal{E}}$ is thrown, the first method that can be called is the (e.g. state-restoring) method $n \in I^{+}$: $\nu X.(\neg e \vee \nu Y.[M \setminus \{n\}]\mathsf{ff} \wedge [\varepsilon]Y) \wedge [-]X$.

It is natural to handle exceptions locally. Hence, in a compositional verification setting, global behavioral specifications would typically not mention throwing and catching of exceptions; these labels can instead be relabelled into silent τ-transitions.

The tool set has also been extended to translate control flow graphs with exceptions into CCS models. This has been used to produce the CCS model corresponding to the graph extracted for the simulation application described at the end of section 3.3.2. Then, we used the Concurrency Workbench to verify various local properties of the application. For instance, we checked whether exceptions are caught locally, i.e., within the method. For the `finalize()` method of JavaSim's `SimulationProcess` class, shown in figure 3.3.4, and a particular exception e, the property `finalize` $\Rightarrow \nu X.(\neg(e \wedge r)) \wedge [-]X$ specifies that exceptions of type e are

```
public void finalize () {
    if (!Terminated) {
           Terminated = true; Passivated = true;
           wakeuptime = SimulationProcess.Never;
           if (!idle()) Scheduler.unschedule(this);
           if (this == SimulationProcess.Current) {
               try { Scheduler.schedule(); }
               catch (SimulationException e) { } }
           SimulationProcess.allProcesses.Remove(this); }}
```

Figure 3.4: The `finalize()` method of JavaSim's `SimulationProcess` class

caught locally. The instructions in the `finalize()` method that may raise an exception are the calls to the virtual method `idle()`, the static methods `unschedule()`, `schedule()`, `Remove()` and accesses to the fields `Never`, and `Current`. All but one of these instructions raise only the `NullPointerException`: the call to method `schedule()` might raise `NullPointerException` and `SimulationException`, an application-defined exception. Model checking the property succeeded for all exceptions e except for `NullPointerException`, showing that not all exceptions are caught locally.

3.3.5 Interface Characterization of Flow Graphs with Exceptions

Given an interface for a flow graph with exceptions I, we can characterize the flow graphs with this interface by the formula σ_I. We state by this characterization that all initial control points are normal, and that after a transition, either the control point is normal again, or it is an exceptional point where all outgoing edges from this point are internal transfer edges leading to a normal control point:

$$\begin{array}{rcl} \sigma_I & = & \bigvee_{m\in I^+}(\nu X.\ P_m \wedge \bigwedge_{e\in I^\varepsilon} \neg e \wedge \\ & & \qquad [I^-,\varepsilon](X \vee (\bigwedge_{m\in I^+}[m]\mathsf{ff} \wedge\ P_m \wedge \bigvee_{e\in I^\varepsilon} P_e \wedge [\varepsilon]X))) \\ P_m & = & m \wedge \bigwedge_{m'\in I^+\setminus\{m\}} \neg m' \qquad\qquad P_e \ = \ e \wedge \bigwedge_{e'\in I^\varepsilon\setminus\{e\}} \neg e' \end{array}$$

The following result establishes that σ_I characterizes all flow graphs with exceptions with interface I, thus condition (ii) of section 3.2.5 holds.

Theorem 3.14. *Let I be an interface for flow graphs with exceptions. For any specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ over labels $L = I^- \cup \{\varepsilon\}$ and atomic propositions $A = I^+ \cup \{r\} \cup E$ we have (where $\mathcal{R}$ denotes the reachable part of a specification, as defined on page 56): $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of theorem 31 in [53]. □

Because of this result and theorem 3.13, the compositional verification principle (3.1) also applies to flow graphs with exceptions.

3.4 Multi-threaded Control Flow

As a second example, we extend the basic program model to capture multi-threaded control flow. As with exceptions, a set of thread and lock names are to be added to the model. Therefore, we take the set of control values to be the set of lock and thread names, *i.e.,* $\mathcal{C}ontVal = \mathcal{L}ock \times \mathcal{T}id$, where $\mathcal{L}ock$ and $\mathcal{T}id$ are infinite sets of lock and thread names, respectively. Given an interface I, we use $I^{\mathcal{L}}$ and $I^{\mathcal{T}}$ to extract the legal lock and thread names, respectively.

Our program model supports all basic thread constructs as provided by Java: thread creation, monitors, a wait-notify mechanism, and the possibility to join a thread (i.e., wait for its completion). The behavior of this model maintains a configuration for each thread. We assume that (the interleaving behaviors of) programs do not contain data races and thus, by virtue of the Java Memory Model [82], assume an interleaving semantics. Notice that the program model described in this section can be easily combined with the program model described above into a single program model with multi-threading and exceptions.

3.4.1 Program Model with Multi-threading

To define method specifications for multi-threaded programs, we introduce edge labels that correspond to the instructions specific to multi-threading. Following the Java semantics, the body of a method is taken to execute sequentially, possibly starting new threads, interleaved with other threads. Let $L_{M,L,T}$ abbreviate the set of labels $M \cup \{\varepsilon\} \cup \{c\, l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}, l \in L\} \cup \{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in T, m \in M\} \cup \{\mathsf{join}\ t \mid t \in T\}$.

Definition 3.15. (Method Specification with Multi-threading) A *flow graph with multi-threaded control flow* for $m \in \mathcal{M}eth$ over sets $M \subseteq \mathcal{M}eth$, $L \subseteq \mathcal{L}ock$ and $T \subseteq \mathcal{T}id$ is a finite model $\mathcal{M}_m = (V_m, L_{M,L,T}, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, $A_m = \{m, r\}$, and $m \in \lambda_m(v)$ for all $v \in V_m$. A *method specification with multi-threaded control flow* for $m \in \mathcal{M}eth$ over M, L and T is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ with $\mathcal{M}_m$ a method graph with multi-threaded control flow for m over M, L and T, and $\mathbb{E}_m \subseteq V_m$ a non-empty set of entry points of m.

3.4.2 Extracting Flow Graphs from Multi-threaded Java Classes

To extend the Program Analyzer to multi-threaded classes, we generate edges with appropriate labels for all Java primitives and native methods still used in relation to concurrency, with the exception of the timed wait and the interrupt mechanism. For instance, calling the `start` (or `fork`) method on a thread object, is modeled by an edge labeled spawn, while a call to `join` leads to an edge labeled join. Special

[exec.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma')), \mathsf{L}, \mathsf{W})$	if	$t \notin \mathsf{W}$, $\Sigma(t) \xrightarrow{a}_{bs} (v', \sigma')$
[coord.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma)), \mathsf{L}', \mathsf{W}')$	if	$\Sigma(t) = (v, \sigma)$,$t \notin \mathsf{W}$, $v \xrightarrow{a}_m v'$, $m \in I^+$, $(\mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_c (\mathsf{L}', \mathsf{W}')$
[resume]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,\ \mathsf{resume}\ l)}_{bm} (\Sigma, \mathsf{L}', \mathsf{W}')$	if	$(t, n, \mathsf{tt}) \in \mathsf{W}(l)$, $\mathsf{L}' = \mathsf{L}(l{:=}(t, n))$, $\mathsf{L}(l) = \perp$, $\mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \backslash (t, n, \mathsf{tt}))$
[thr.-ops.]	$(\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma'(t{:=}((v', \sigma)), \mathsf{L}, \mathsf{W})$	if	$\Sigma(t) = (v, \sigma)$,$t \notin \mathsf{W}$, $v \xrightarrow{a}_m v'$ $m \in I^+$, $\Sigma \xrightarrow{a}_t \Sigma'$

Table 3.1: Transition rules $\rightarrow_{bm}$ for multi-threaded behavior

care is taken for calls to synchronized methods: they are preceded and followed by edges labeled lock and unlock on the appropriate object, i.e. the synchronization is made explicit.

Special care has to be taken to ensure that the extracted sets of thread and lock names are finite. For threads, a safe over-approximation is to use the declared class name of the thread as thread name in the model. Using a more precise analysis can help to distinguish different threads that are instances of the same class. For locks, abstracting with the class name might under-approximate the program behavior. To overcome this problem, we require that the program has only a finite number of lock objects with the same class name.

3.4.3 Flow Graph Behavior with Multi-threading

The behavior specification follows closely the Java Specification [81]. Instead of only maintaining a single call stack as was done in the basic behavior, we maintain a configuration (i.e. control point and call stack) per active thread. If a thread is not active, we map it to the value $\perp$. We also maintain a *lock map* and a *wait map*. The lock map returns for each lock the identity of the thread holding the lock and the lock counter (i.e. how many times the lock is held, necessary to correctly model the reentrant locking behavior of Java). The wait map returns for each lock the set of threads that are waiting for it, the number of times the thread was holding the lock when it started waiting, and a flag whether the thread has been notified. This ensures that the thread resumes in the exact same state as when it issued a wait, thus making sure a correct number of unlock operations is executed to release a lock. We explicitly require that a thread be active for it to wait for a lock.

We assume that execution always starts with a special thread called main, and that any closed flow graph contains such a thread. Labels and atomic propositions are paired with thread identifiers. Further, we introduce the atomic proposition $\mathsf{haslock}(t, l)$ to hold in any state where thread t holds lock l.

Definition 3.16. (Behavior with Multi-threading) Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed multi-threaded flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The *multi-threaded behavior* of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bm}, A_b, \lambda_b)$ is defined as follows:

- $S_b = \{s \in \ (I^{\mathcal{T}} \rightarrow (V \times V^*)_\perp) \times (I^{\mathcal{L}} \rightarrow (I^{\mathcal{T}} \times \mathbb{N})_\perp) \times$
  $(I^{\mathcal{L}} \rightarrow \mathcal{P}(I^{\mathcal{T}} \times \mathbb{N} \times \mathbb{B})) \mid \forall l, t, n, b.(t, n, b) \in \pi_3(s)(l) \Rightarrow \pi_1(s)(t) \neq \perp\}$,
- $L_b = T \times (\ \{m_1\ c\ m_2 \mid c \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\} \cup$
  $\{c\, l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}, \mathsf{resume}\}, l \in I^{\mathcal{L}}\} \cup$
  $\{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in I^{\mathcal{T}}, m \in I^+\} \cup \{\mathsf{join}\ t \mid t \in I^{\mathcal{T}}\})$,
- $\rightarrow_{bm}$ is defined in table 3.1[3] (using auxiliary rules $\rightarrow_c$ and $\rightarrow_t$ of table 3.2) ,
- $A_b = (T \times A) \cup \{\mathsf{haslock}(t, l) \mid t \in I^{\mathcal{T}}, l \in I^{\mathcal{L}}\}$, and
- $\lambda_b(s) \ = \ \{(t, p) \mid t \in I^{\mathcal{T}} \wedge \pi_1(s)(t) \neq \perp \wedge p \in \lambda(\pi_1(\pi_1(s)(t)))\} \cup$
  $\{\mathsf{haslock}(t, l) \mid \pi_2(s)(l) \neq \perp \wedge \pi_1(\pi_2(s)(l)) = t\}$.

The set of initial states $\mathbb{E}_b$ is defined as $\mathbb{E}_b = \{(\Sigma_I^v, \lambda l.\ \perp, \lambda l.\varnothing) \mid v \in \mathbb{E}\}$ where $\Sigma_I^v(\mathsf{main}) = (v, \epsilon, \perp)$ and $\Sigma_I^v(t) = \perp$ for all $t \in I^{\mathcal{T}}$.

The transition rules should be understood as follows.

- Rule [exec.] lifts the standard rules for sequential applets, repeated for completeness in Def. 3.7, to the multi-threaded case. In this rule and several others, the thread progresses only if it is not in the wait set W. Note that this is the only rule that (possibly) changes the call stack of the current thread.
- Rule [coord.] models the coordination of threads via locks, i.e. the operations lock, unlock, wait, notify and notifyAll: the current thread changes control point if the lock and wait maps can be updated appropriately (as defined by the auxiliary transition rules $\rightarrow_c$ in table 3.2).

  The first lock is obtained (i.e. [lock] transition succeeds) only if the lock l is not held by any thread (i.e. l maps to $\perp$ in L). As a result, in the updated lock map l maps to $(t, 1)$, where t is the current thread identifier. Subsequent lock operations by t increase this counter, while unlock operations by t decrease it until the value becomes $\perp$, i.e. until the lock is released.

  The wait notification mechanism is modeled using the wait map W. When a thread t decides to wait on a lock l (which it holds n times), it releases the lock, and (t, n, ff) is added to $\mathsf{W}(l)$. A notify(All) for l sets the notification flag of some (all) thread waiting for l to tt (where rule [notify-cont] handles the special case where no thread is waiting for the lock l). If lock l becomes

[3] We abbreviate $\exists n, b, l.(t, n, b) \in \mathsf{W}(l)$ as $t \in \mathsf{W}$. We use $f(i{:=}x)$ to denote function update. Further, $\Sigma(i) = (v, \sigma)$ implicitly implies that $\Sigma(i) \neq \perp$.

[lock]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W})$	if	$\mathsf{L}(l)=\bot$, $\mathsf{L}'=\mathsf{L}(l{:=}(t,1))$
[re-lock]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W})$	if	$\mathsf{L}(l)=(t,n)$, $\mathsf{L}'=\mathsf{L}(l{:=}(t,n))$
[unlock]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{unlock}\ l)}_c (\mathsf{L}',\mathsf{W})$	if	$\mathsf{L}'=\mathsf{L}(l :=$ $(\mathsf{L}(l)=\bot \vee \mathsf{L}(l)=(t,1)?$ $\bot:$ $(\pi_1(\mathsf{L}(l)),\pi_2(\mathsf{L}(l))-1)))$
[wait]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{wait}\ l)}_c (\mathsf{L}',\mathsf{W}')$	if	$\mathsf{L}(l)=(t,n)$, $\mathsf{L}'=\mathsf{L}(l{:=}\bot)$, $\mathsf{W}'=\mathsf{W}(l{:=}\mathsf{W}(l)\cup\{(t,n,\mathsf{ff})\})$
[notify]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W}')$	if	$\mathsf{L}(l)=(t,n)$,$(t',n,\mathsf{ff})\in\mathsf{W}(l)$, $\mathsf{W}'=\mathsf{W}(l :=$ $\mathsf{W}(l)\backslash\{(t',n,\mathsf{ff})\}\cup\{(t',n,\mathsf{tt})\})$
[notify-cont]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W})$	if	$\mathsf{L}(l)=(t,n)$,$\forall t'.(t',n,\mathsf{ff})\notin\mathsf{W}(l)$
[notifyAll]	$(\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notifyAll}\ l)}_c (\mathsf{L},\mathsf{W}')$	if	$\mathsf{L}(l)=(t,n)$, $\mathsf{W}'=\mathsf{W}(l :=$ $\{(t',n,\mathsf{tt}) \mid (t',n,r)\in\mathsf{W}(l)\})$
[spawn]	$\Sigma \xrightarrow{\mathsf{spawn}\ t'\ \mathsf{with}\ m'}_t \Sigma'$	if	$\Sigma(t')=\bot$, $m'\in I^+$, $v''\in E$, $v''\models m'$, $\Sigma'=\Sigma(t'{:=}(v'',\epsilon))$
[join]	$\Sigma \xrightarrow{\mathsf{join}\ t'}_t \Sigma$	if	$\Sigma(t')=(v'',\epsilon)$, $v''\models r$

Table 3.2: Auxiliary transition rules $\rightarrow_c$ and $\rightarrow_t$

available, any notified element in $\mathsf{W}(l)$ can acquire it, and resume execution as specified by rule [resume] (in table 3.1). When a thread re-acquires a lock l after a wait, it re-acquires it exactly as many times as it was holding the lock, when starting to wait. This is recorded as the value n in the wait map. This ensures that the thread has to execute exactly n releases to release the lock.

- Rule [thread-ops] models the operations of creating and joining a thread (using the auxiliary transition rules $\rightarrow_t$). A thread can be spawned only if it was not active before, starting with an empty call stack in one of the entry points of the method it was spawned with (i.e. the method `run` in a Java program). Only a thread executed to completion can be joined, i.e. it is blocked at a return point, with an empty call stack.

- Rule [resume] handles the case where a thread is waiting on an object, has been notified, and now continues execution as was described above.

Also in the case of multi-threaded flow graphs, there is a direct correspondence between flow graph structure and behavior, and thus structural simulation implies behavioral simulation.

Theorem 3.17. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be flow graphs with multi-threading. If $\mathcal{G}_1 \leq_s \mathcal{G}_2$ then $\mathcal{G}_1 \leq_b \mathcal{G}_2$.*

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define

$$\begin{array}{l}(\Sigma, \mathsf{L}, \mathsf{W}) R_b (\Sigma', \mathsf{L}', \mathsf{W}') \Leftrightarrow \\ (\forall t \in T.\ \mathsf{if}\ \Sigma(t) = (v, \sigma) \\ \qquad \mathsf{then}\ \Sigma'(t) = (v', \sigma') \wedge v R v' \wedge |\sigma| = |\sigma'| \wedge \forall i.i < |\sigma|.\sigma(i) R \sigma'(i) \\ \qquad \mathsf{else}\ \Sigma'(t) = \bot) \wedge \mathsf{L} = \mathsf{L}' \wedge \mathsf{W} = \mathsf{W}'\end{array}$$

It is easy to check that R_b is a behavioral simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

3.4.4 Properties over Flow Graphs with Multi-threading

The instantiation of the generic flow graph model with multi-threaded control flow allows us to express properties that are related to the multi-threaded character of the flow graph. Given a flow graph $\mathcal{G} : I$ with multi-threaded control flow, typical (behavioral) properties expressible in our logic are:

- Method $m \in I^+$ can only be called by thread t, if t has lock l: $\nu X. \bigwedge t \in I^{\mathcal{T}} (\mathsf{haslock}(t, l) \vee \bigwedge_{m' \in I^+} [(t, m'\ \mathsf{call}\ m)]\mathsf{ff}) \wedge [-]X$. If method m is the only method accessing some data, this means that data is lock protected.
- Locks are acquired in a particular order, for example lock l_2 can only be acquired by a thread that already has lock l_1: $\nu X. \bigwedge_{t \in I^{\mathcal{T}}} (\mathsf{haslock}(t, l_1) \vee [(t, \mathsf{lock}\ l_2)]\mathsf{ff})] \wedge [-]X$. This guarantees absence of deadlocks by synchronization (however, it does not guarantee absence of deadlocks, caused by the wait-notify mechanism, or by joining a non-terminating thread).
- No more than n threads are created in an application. This is an important resource property. Formally, this can be expressed as $MaxThr\,(n)$, inductively defined as follows:

$$\begin{array}{rcl} MaxThr\,(1) & = & \nu X_1. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m]\mathsf{ff} \wedge [-]X_1 \\ MaxThr\,(k+1) & = & \nu X_{k+1}. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m] MaxThr\,(k) \wedge [-]X_{k+1} \end{array}$$

3.4.5 Interface Characterization of Flow Graphs with Multi-threading

Given an interface for a flow graph with multi-threaded control flow I, the flow graphs with this interface can be characterized by the formula σ_I, where $L_{M,L,T}$ is as defined on Page 69:

$$\sigma_I = \bigvee_{m \in I^+} (\nu X. P_m \wedge [L_{I^-, I^{\mathcal{C}}, I^{\mathcal{T}}}] X) \qquad P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m$$

Theorem 3.18. *Let I be an interface for multi-threaded flow graphs. For any specification $\mathcal{S} = (\mathcal{M}, E)$ over labels $I^- \cup \{\varepsilon\} \cup L_{M,L,T}$ and atomic propositions $A = I^+ \cup \{r\}$ we have : $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of theorem 31 in [53]. □

Thus, the compositional verification principle (3.1) also applies to flow graphs with multi-threading. However, applying the verification principle poses a problem to model checking, since the verification problem resulting from the second premise, $\theta_I(\psi) \uplus \mathcal{G}_2 \models_b \phi$, is not decidable in general for the case of pushdown systems with multiple stacks. This is a consequence of a basic undecidability result due to Ramalingam [99], which is related to the undecidability of the problem of emptiness of intersection of context-free languages. Hence, every such model checking algorithm must use an under- or over-approximation of the program behavior. Different approaches have been proposed, see e.g. [17, 45, 98]. It is future work to study whether and how these solutions can be integrated into our framework.

3.5 Related Work

The maximal model technique for compositional verification was originally developed by Grumberg and Long [51] for the universal fragment of CTL, and was later generalized by Kupferman and Vardi [73] for ACTL*. A more detailed background on maximal model construction and its relationship with compositional verification was already presented in section 2.2.

Gurov *et al.* have introduced a maximal model construction for the fragment of the modal μ-calculus without least fixed-points and diamond modalities.This program model has been inspired by the one of Besson *et al.* [15], who address the problem of verifying stack invariants of Java programs. The model of Gurov *et al.* is also close to that of Recursive State Machines, proposed by Alur *et al.* [6], while somewhat courser. However, Recursive State Machines have not been used to address compositional verification of programs with recursion. Still other models exist for capturing the control flow of applications in Java-like languages, see e.g., [87]. However, because of the specific requirements of our compositional verification technique, we cannot directly reuse these models, and instead rely on our own.

We create models of implementations in Java bytecode using the SOOT Framework. When exceptional behavior is taken into account, the control flow graphs SOOT creates are similar to ours. SOOT control graphs do not have explicit nodes for exceptional program points however. Because in these graphs each node matches an instruction in the Jimple program text.

Several tools exist for the (non-compositional) verification of behavioral program properties. The behavior of programs with recursion is usually represented as pushdown systems and model checked against temporal logic properties [16, 42, 44].

Moped [68, 43] and Alfred [97] are examples of tools that follow this scheme. In particular, a variant of Moped, jMoped [105], translates Java bytecode to a pushdown system extended with a set of variables, where instructions are directly mapped to transitions of the system. Also closely related is the two-step extraction technique of Obdržálek [91], where a control flow graph of the program is produced first, and the pushdown system is then generated from this graph. However, neither of these translations addresses multi-threading. Further, existing model checkers for multi-threaded Java (such as Bogor[4] and JavaPathFinder[5]) typically use an implicit program representation that is close to the program itself. Then, abstraction is applied to make verification feasible. In contrast, our program model directly abstracts the program behavior; without this abstraction a maximal flow graph cannot be constructed.

3.6 Conclusion

In this section we summarize this chapter and propose some future work.

3.6.1 Summary and Contribution

In this part of the thesis, we show how the previously developed method of Gurov *et al.* for compositional verification of control flow properties of sequential flow graphs with procedures can be adapted to richer program models. In particular, we propose extensions to the original program model that allow exceptional control-flow and multi-threading to be captured. We show that for both extensions the compositional verification principle still applies, by identifying the conditions under which the compositional verification principle is sound and complete with respect to a given program model, and proving that these conditions hold for the proposed extensions. It is important to note, however, that in the case of multi-threaded flow graphs, the model checking problem is not decidable due to a general undecidability result for pushdown systems with multiple stacks. The restrictions on the instantiations required to ensure soundness and completeness of the principle are not severe, and the resulting models are intuitive and standard and can thus be used for other analysis as well.

3.6.2 Future Work

The methodology we provide in this chapter can be employed for making further extensions to the program model underlying the compositional verification technique. Especially of interest is to add data (from finite domains), and access control information. We are currently adapting the tool set to handle multi-threaded models and plan to integrate a suitable pushdown model checking algorithm for the verification of these models.

[4] See `http://bogor.projects.cis.ksu.edu`.
[5] See `http://javapathfinder.sourceforge.net`.

Chapter 4

Provably Correct Runtime Monitoring

4.1 Introduction

As mobile devices gain capabilities, the demand for new applications increases. However, running third-party applications on mobile devices is considered risky. These devices contain personal information and provide access to costly functionality (e.g. GSM services and GPRS connections), which not only makes the security issue involved in running third-party applications more critical, but also results in more complex requirements on applications by both the device platform and the user. The user may, for example, want to allow a chat application more freedom regarding connections than a gaming application. Therefore, crude sandboxes that hide functionality from third-party code are not an option in this setting. Such imposed handicaps would simply make the applications uninteresting.

Most mobile device users currently permit only signed third-party software on their devices due to security concerns. This process goes as follows. After production, the application is handed to a certifying party. The certifying party (ideally) performs certain analysis that checks the adherence of the program to the policy of the mobile platform provider. (The policy is usually the security policy of a fixed platform, hence the certification has to be repeated for each platform.) After it approves the application, the certifier signs it with its private key. In the deployment phase, the key attached to the application is used to designate the origin of the application and the application is installed only if this party is recognized as a trusted party by the platform developers or by the mobile operator. This security model depends on mutual agreement and trust between various parties, namely the application developer, the certifier party, the platform provider, and the mobile operator. Therefore, this type of certification significantly prolongs the overall time it takes for a product to reach the market and raises the production costs. An efficient security mechanism that provides controlled access to the resources of the

device would be a superior alternative.

Many practical mobile device policies such as limiting the number of short messages that are sent by an application per hour, or disallowing connections to unsecure domains after access to personal information are *access control policies*. Access control policies fall into the class of policies enforceable by *monitors* [102, 55]. A monitor operates by observing the behavior of the target program at runtime and intervenes to prevent any policy violation. Runtime monitoring is a firmly established and efficient mechanism [64, 41, 69, 59, 80, 79], thus a candidate for being included in mobile security frameworks.

The use of monitoring as a mechanism for enforcement of security policies in a sensitive context such as mobile devices requires that the monitor is *sound*, i.e. the monitor guarantees that the executions of the program adheres to the desired policy. Soundness depends essentially on the ability of the monitor to intervene with the execution of the program (*target control*), and to intercept all the security relevant actions performed by the program, i.e. that the untrusted program can not perform security relevant actions by circumventing the monitor (*uncircumventability*). In *explicit* monitoring, target program actions are intercepted and approved by some external monitoring agent [64, 69, 59]. Therefore, both target control and uncircumventability are issues that need to be addressed when this type of monitoring is employed.

A variant of monitoring is *monitor inlining*, in which target programs are rewritten to include the desired monitor functionality, thus making them self-monitoring. General purpose monitor inlining has been pioneered in Evans and Twyman's Naccio [46] and in Erlingsson and Schneider's SASI [41] tools. Security automata introduced by Schneider [102] are a suitable formalism for expressing policies. These are in essence Büchi automata where all the states are accepting. Inlining for a policy given by a security automaton can be performed by adding new variables to the target program in order to record the current automaton state and by inserting code to perform necessary checks and corresponding updates on these variables at each security relevant action, according to the transition relation of the automaton. The program is terminated if there is no transition for the next security relevant action in the current state. As a result, the monitor becomes part of the program, so target control ceases to be an issue. What is more, uncircumventability can be based directly on the semantics of the language and type safety [13].

Monitor inlining is especially suitable for mobile devices as it is simple, and eliminates the need for a complicated security enforcement infrastructure which may be costly. In terms of runtime overhead, inlining has an advantage over explicit monitoring in that the information to decide on security issues is directly accessible to the monitor from within the program. An inliner is typically a small program [39] and can be incorporated into the device platform. There are several advantages, however, of inlining in development time over performing the inlining on device. The practical use of any security enforcement mechanism depends on the functionality of the program not being unnecessarily hindered. To this end, the monitor should be *transparent* so that those executions of the target program that adhere to the policy

are not altered by the monitor. Performing inlining prior to deployment enables developers to preserve full control over their applications. The code producer is free to decide how the inlining is to be performed so to provide a transparent monitor, not trading off functionality for security. Furthermore, the availability of resources at development time allows for more optimizations on the inlined code.

Using "off-device" inlining to enforce policies does not solve the mobile code security problem. A program may contain a sound monitor for the device policy, yet it is as untrusted as any other program if the device does not have the means to check this. Proof-carrying code (PCC) [90] has been introduced as a means to establish trust in mobile code on the host that executes the code. A PCC setting involves two parties, the code producer (e.g. the code developer) and the code consumer (e.g. the platform in which the code executes). The code producer ships an application with an additional piece of information that *simplifies* the check on the consumer side that the application obeys the security policy.

The framework we support in this chapter (see figure 4.1) uses monitor inlining in a proof-carrying code setting to facilitate quick certification and deployment of applications[1]. In this framework, the software production process is extended by monitor inlining and proof generation steps. The program is inlined by the software producer or a third-party to adhere to a policy, the security requirements on the program that are envisioned by the developer. The inlining step serves to guarantee this behavior by inserting a monitor into the program that enforces the requirements. The producer policy and the inlined program are then passed to the *proof generator* which produces a proof of this guarantee. Finally, the inlined program is packaged together with the producer policy and the proof of its compliance to this policy, ready to be shipped to the code consumer.

The deployment phase on the mobile device consists of policy matching and proof checking. The *matcher* checks that the security relevant behavior of the program, represented by the producer policy, obeys the security requirements of the platform. If matching returns a negative answer, the program is rejected. Otherwise, it should be further checked that the program indeed complies to the policy it was shipped with. To this end, the *proof checker* considers the shipped policy and the proof and approves the program only if this proof is correct for the program and the producer policy.

This framework has a number of advantages over the current certification process based on code-signing. The policy matcher of the above framework may relieve the developers from having to know the exact security requirements of the platform. Instead, the developer guarantees a certain behavior for the program. This declared behavior may cause the acceptance or the rejection of the program on the basis of not only the platform requirements but also of the policy of the particular mobile device user. Hence, it also provides a means for users to customize the requirements for each application. Assuming that the proof checker component is a less complex

[1]The framework was designed within the IST programme of the EC, under the IST-STREP-27004 Security of Software and Services for Mobile Systems (S3MS) project, URL: www.s3ms.org.

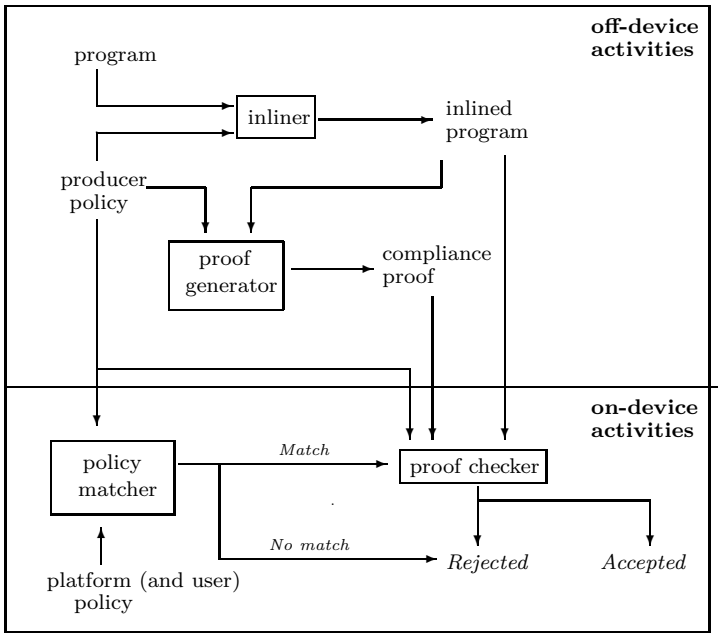


Figure 4.1: Security Framework

software than the monitor inliner, the set of entities that build the security architecture of the device and hence need to be trusted (i.e. the *trusted computing base* of the device) becomes smaller. Most important of all, in this framework, employing trust in a certifying party is no longer a necessity, which decreases the time and cost of mobile application development considerably.

In order to support the above framework, we propose in this chapter a new policy language and formalization of monitoring and monitor inlining. We focus on policies expressed as security automata [102] that operate on calls to some fixed API from a target program given as a Java Virtual Machine (JVM) class file. Automaton transitions are allowed to depend locally on argument values, heap at time of call and (normal or exceptional) return, and return value. We introduce the language ConSpec for writing such policies, as a successor to earlier policy languages devised for runtime monitoring such as PSLang [39]. It is designed to render a clean semantics and tractability for the formal treatment of various security related activities while retaining the intuitiveness of PSLang. In particular, when policy matching is taken as language inclusion, the problem is decidable for ConSpec policies. The language also allows a seamless treatment of inheritance, passing constraints on methods to non-overriding subclasses.

We present a formalization of monitoring using, as monitors, security automata derived from ConSpec policies. A *co-execution* of a program and an automaton is

an interleaving of a program execution with an automaton run, where the program execution is not affected by the monitor except for possible termination. Whereas, the monitor transits on the security relevant actions of the program. A basic result established here is monitor soundness and transparency.

The main goal of this work is to develop a suitable notion of proof to facilitate automatic proof generation and efficient proof checking for the purpose of certifying correctly inlined programs. By correctness, we mean here soundness and leave dealing with transparency to future work. As in the original PCC framework by Necula [90], we use annotated programs as proofs, which enables the reuse of some existing machinery. In order to handle a large class of inlined programs, we do not fix the inlining procedure that is to be used by the producer. Instead, we use annotations to characterize self-monitoring programs, which would include any correctly inlined program.

Our main contributions are characterizations, in terms of JVM class files annotated by formulae in a suitable Floyd-like program logic, of the following two conditions on a program relative to a given policy:

1. that the program is policy-adherent;
2. that the program contains a method-local monitor for the policy, in the sense that updates to the monitor state do not cross method call boundaries.

Note that the second condition entails the first condition.

The annotations serve as an important intermediate step towards a decidable annotation validity problem, once the inliner is suitably instantiated. The method-local nature of the embedded monitor enables compositional analysis: validity can be checked per method. Being method-local is not an overly restrictive condition on embedded monitors and is satisfied by all general purpose inliners we know of.

By these characterizations, the problem of showing correctness of inlined monitors reduces to proving the validity of the corresponding annotations. We illustrate the practicality of this approach by describing a monitor inliner for which we prove correctness. We also sketch how, for this inliner, the annotations can be completed to produce a fully annotated program for which validity can be efficiently decided using a bytecode weakest precondition checker. Such a full annotation can thus be used as the proof element in the setting of figure 4.1. The full annotations are sufficient to decide whether the program complies to the contract, provided that:

- the full annotations imply the partial annotations that guarantee contract adherence, and
- the fully annotated program is valid with respect to the axiomatic semantics of the bytecode instructions,

both of which can be decided efficiently on device.

Organization This chapter is structured as follows. Section 4.2 presents the JVM model used in this paper. The next two sections introduce the automaton model

in concrete and symbolic forms, the ConSpec language, and relations between the three. Section 4.5 gives an account of monitoring by interleaved (co-)execution of a target program with a monitor, and establishes the equivalence of policy adherence and co-execution. In section 4.8, we present a two-level annotation scheme that characterizes the two conditions mentioned above. In section 4.9 the inliner and its correctness proof are sketched. We also sketch how to produce, for this inliner, fully annotated programs with a decidable validity problem. We summarize related approaches in section 4.10. Finally, in section 4.11 we conclude and discuss future work.

4.2 Program Model

We assume the reader to be familiar with Java bytecode syntax, and the Java Virtual Machine (JVM). Here we only present components of the JVM that are essential for the definitions in the rest of the text.

4.2.1 Notation

We first introduce some notation that we will use in the rest of the chapter. For a partial function $f : S_1 \rightharpoonup S_2$, the domain of f is all elements x of S_1 for which $f(x)$ is defined: $Dom(f) = \{x \mid x \in S_1 \ \wedge \ f(x) \in S_2\}$.

Function update, written $f[x \mapsto v]$, is defined as usual as

$$(f[x \mapsto v])[y] = \begin{cases} v & \text{if } x = y \\ f[y] & \text{otherwise} \end{cases}$$

for all $y \in Dom(f)$.

We use sequences to model stacks, as well as lists of values and types. The empty sequence is ϵ and $v \cdot s$ places v at the front of sequence s. Sequence concatenation is written $s1 \bullet s2$, and the substitution $s[b/a]$ replaces all occurrences of a in s with b.

4.2.2 Types and Values

We denote bytecode programs with T. We fix a set of class names $c \in \mathbb{C}$, a set of method names $m \in \mathbb{M}$, and a set of field names $f \in \mathbb{F}$. Primitive type set *PrimType* consist of the types `int` and `string`. The type `Unit` is used for methods that do not return a value. A type $\tau \in$ *Type* is either a primitive type, a class c, the type `Unit` or the type `Null`. We let $\gamma \in (Type)^*$ range over tuples of types.

Each type $\tau \in$ *Type* determines a set $\|\tau\|$ of values. *Val* denotes the set of all values. Values of types `int` and `string` are integers and strings, respectively, and make up the values of type *PrimType*, which are called the primitive values *PrimVal*. The sets determined by the types `Unit` and `Null` are singletons consisting of the values *void* and `null`, respectively.

Values of object type are (typed) locations $\ell \in Loc$, mapped to objects by a heap $h \in \mathbb{H} = Loc \rightharpoonup \mathbb{O}$. The partial function $type : (\ell, h) \mapsto \mathbb{C}$ returns the type

of location ℓ in heap h, if $\ell \in \mathit{Dom}(h)$, and is otherwise undefined (i.e. $\perp$). The structure of objects in $\mathbb{O}$ is not further specified here. It suffices to assume that if $h : \ell \mapsto o \in \mathbb{O}$ then $h(\ell)$ determines a field $h(\ell).f$ whenever the class which this object is a member of, declares f.

We also introduce a static heap $\mathit{sh} : c \times f \rightharpoonup \mathit{Val}$ that stores the values of the static variables of a class. We assume $\mathit{sh}_0^{\mathrm{T}}$ to be the initial mapping which maps each static variable of a class reachable through the program T to its initial value as given by its class definition. In this sense, we make two assumptions: the static variables of all reachable classes are assumed to be initialized to constant values (i.e. we disregard static initializers) and the initialization is assumed to have been done *before* the program starts executing. The first assumption can be dropped by extending our approach to handle static initializers, which is straightforward to perform.

Each class determines a set of fields and methods defined for that type through its declaration. The class declarations induce a hierarchy given by the subclassing partial preorder $<:$ on the set $\{\texttt{Null}\} \cup \mathbb{C}$. We write $c_1 <: c_2$ if c_1 is a subclass of (or extends) c_2. `Null` is the bottom element with respect to this ordering: $\forall \tau \in \{\texttt{Null}\} \cup \mathbb{C}.\ \texttt{Null} <: \tau$. If c defines m (declares f) explicitly, then c defines (declares) $c.m$ ($c.f$). We say that c defines $c'.m$ (declares $c'.f$) if c is the smallest superclass of c' that contains an explicit definition (declaration) of $c.m$ ($c.f$). Single inheritance ensures that definitions/declarations are unique, if they exist.

4.2.3 Methods

Method definitions are modeled through an environment Γ taking method references to their definitions. The environment Γ is elided where possible. We assume furthermore a partitioning on the set of methods which divides the set into API methods (*ApiMet*) and application methods (*AppMet*): $\mathbb{M} = \mathit{ApiMet} \uplus \mathit{AppMet}$. To simplify notation, method overloading is not considered, so a method is uniquely identified by a method reference of the form $M = (c, m)$. For a method $(c.m)$, $(c.m) : \gamma \to \tau$ when γ is the list of argument types and τ is the return type of the method. A method definition is a pair (P, H) consisting of a method body P and an exception handler array H. The method body (the exception handler array) of M is denoted P_M (H_M) when the environment Γ is clear from the context. For each program, we assume that there exists a main method method which does not have a class defining it. We identify this method with the special reference $\langle\texttt{main}\rangle$.

A method body P is a partial function from ω to the set of instructions such that $\mathit{ADDR}_P = \mathit{Dom}(P)$ has the form $\{1, \ldots, n\}$ for some $n \in \omega$. We use the notation $M[L] = I$ to indicate that $\Gamma(M) = (P, H)$ and $P(L)$ is defined and equal to the instruction I. The exception handler array H is a partial map from integer indices to exception handlers. An exception handler is a four-tuple (L_1, L_2, L_3, c) consisting of three program labels ($L_1, L_2, L_3 \in \mathit{Dom}(P)$) and a class which is a subtype of class `Throwable`, ($c <: \texttt{Throwable}$).

4.2.4 Operational Semantics

A *configuration* of the JVM is a triple $C = (R, h, sh)$ of a stack R of activation records, a normal heap h and a static heap sh. For normal execution, the activation record at the top of the execution stack has the shape (M, pc, s, f), where

- The method reference M is the currently executing method.
- The *program counter* pc is an index into the currently executing instruction array, i.e. it is a member of $Dom(P)$ where P is the body of M. The configuration C is *calling*, if $P(pc)$ is an invoke instruction, and it is *returning normally*, if $P(pc)$ is a return instruction.
- The *operand stack* s is the stack of values (i.e. primitive values or locations) currently being operated on.
- The *local variables* lv is a mapping of variables to values, preserving types.

For exceptional configurations C the top frame has the form $(b)_e$ where b is the location of an exceptional object. For exceptional configurations, the current program counter and executing method is given by the frame below the exceptional frame. Then, C is *returning exceptionally* if there is no handler for this exception and the current instruction label in the currently executing method. Configuration C is *returning* if C is either returning normally or exceptionally. Finally, if C is exceptional and there is a single frame in the activation record, then the program is exiting exceptionally.

We assume a transition relation $\longrightarrow_{\text{JVM}}$ on JVM configurations, presented in tables 4.1, 4.2 and 4.3 for a subset of instructions of the JVM. This operational semantics is a simplified version of the semantics by Freund and Mitchell [48] and does not consider, for instance, details of object initialization[2]. Each row in these tables describes the conditions under which a program represented by the environment Γ can move from configuration C_0 to configuration C_1, i.e. $C_i \longrightarrow_{\text{JVM}} C_{i+1}$. The first column of the first two tables indicates the instruction form captured by the rule. If the instruction about to be executed matches that form and all conditions in the 'Condition' column are satisfied, then a transition occurs from configuration C_0 to configuration C_1. The initial configuration is $((\langle \texttt{main} \rangle, 1, \epsilon, lv_0), h_0, sh_0^{\text{T}})$, consisting of a single, normal activation record with an empty stack, an arbitrary mapping to local variables (lv_0), an empty heap ($Dom(h_0) = \varnothing$) and the initial static heap.

For example, table 4.1 contains the rule for the `getstatic` instruction, which pushes the value stored in the specified static field of the specified class, by accessing the static heap. The rule indicates that the execution may proceed from configuration C_0 to C_1 in a single step if for the currently executing method M, $M[pc] =$ `getstatic` $c.f$ and C_0 and C_1 match the patterns in the table. In this

[2]Initialization information is not included in the configurations for reasons of clarity and brevity.

$M[pc]$	Condition	C	C'
`goto` L		$((M, pc, s, lv) \cdot R, h, sh)$	$((M, L, s, lv) \cdot R, h, sh)$
`ifeq` L	$v = 0$	$((M, pc, v \cdot s, lv) \cdot R, h, sh)$	$((M, L, s, lv) \cdot R, h, sh)$
`ifeq` L	$v \neq 0$	$((M, pc, v \cdot s, lv) \cdot R, h, sh)$	$((M, pc + 1, s, lv) \cdot R, h, sh)$
`getstatic` $c.f$	$sh(c.f) = v$	$((M, pc, s, lv) \cdot R, h, sh)$	$((M, pc + 1, v \cdot s, lv) \cdot R, h, sh)$
`putstatic` $c.f$		$((M, pc, v \cdot s, lv) \cdot R, h, sh)$	$((M, pc + 1, s, lv) \cdot$ $R, h, sh[c.f \mapsto v])$
`iload/aload` rx		$((M, pc, s, lv) \cdot R, h, sh)$	$((M, pc + 1, lv(x) \cdot s, lv) \cdot R, h, sh)$
`istore/astore` rx		$((M, pc, v \cdot s, lv) \cdot R, h, sh)$	$((M, pc + 1, s, lv[x \mapsto v]) \cdot$ $R, h, sh)$
`athrow`	$type(h, b) <: \texttt{Throwable}$	$((M, pc, b \cdot s, lv) \cdot R, h, sh)$	$(b)_{\text{exc}} \cdot ((M, pc, s, lv) \cdot R, h, sh)$
`instanceof` c	$type(h, d) <: c$	$((M, pc, 1 \cdot s, lv) \cdot R, h, sh)$	$((M, pc, s, lv) \cdot R, h, sh)$
`instanceof` c	$\neg(type(h, d) <: c)$	$((M, pc, 0 \cdot s, lv) \cdot R, h, sh)$	$((M, pc, s, lv) \cdot R, h, sh)$
`ireturn`	$\Gamma(M) = \gamma \to \texttt{int}$ $\lvert\gamma\rvert = \lvert s'\rvert$	$((M, pc, v \cdot s, lv) \cdot$ $(M', pc', s' \bullet (d \cdot s''), lv') \cdot$ $R, h, sh)$	$((M', pc' + 1, s'', lv') \cdot R, h, sh)$

Table 4.1: Operational Semantics for a Subset of Instructions of the JVM

rule, and all others, if we apply a function g to an argument x, we have the implicit requirement that $x \in Dom(g)$. The `athrow` instruction raises an exception by taking an object which is of a subtype of the type `Throwable` off the top of the stack and pushing a new activation record containing that reference. The rule for `ireturn` is understood better after considering table 4.2.

An *execution* E of a program (class file) T is then a (possibly infinite) sequence of JVM configurations $C_1C_2C_3\ldots$ where C_1 = is the initial configuration, Γ set up according to T, and for each $i \geq 1$, $C_i \longrightarrow_{\text{JVM}} C_{i+1}$. We restrict attention to configurations that are *type safe*, in the sense that heap contents match the types of corresponding locations, and that arguments and return/exceptional values for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions (cf. [78]).

Method Calls The rule for invokevirtual uses the type of the object stored in the heap at the address provided by the stack in order to determine which method is called in effect, i.e. the method for the new activation record. The notation $g[1\ldots n \mapsto v_n \cdot v_{n-1} \cdot \ldots \cdot v_1]$ is an abbreviation for $g[1 \mapsto v_1][2 \mapsto v_2]\ldots[n \mapsto v_n]$. The function lv_0 maps the local variables to arbitrary values.

Table 4.2 shows the rules for virtual method calls, i.e. for the case where $M[pc] =$`invokevirtual` $c.m$. We use the following abbreviations in the table:

$ApiCallCond \stackrel{\triangle}{=} M' = c'.m, M' \in ApiMet, CallCond$

$AppCallCond \stackrel{\triangle}{=} M' = c'.m, M' \in AppMet, CallCond$

$CallCond \stackrel{\triangle}{=} d \neq \texttt{null}, type(h, d) <: c, c'$ *defines* $type(h, d).m, c'.m : \gamma \to \tau, |s| = |\gamma|$

Condition	C	C'
AppCallCond	$((M, pc, s \bullet (d \cdot s'), lv) \cdot R, h, sh)$	$((M', 1, \epsilon, lv_0[0 \mapsto d, 1 \ldots \lvert\gamma\rvert \mapsto s]) \cdot$ $(M, pc, s \bullet (d \cdot s'), lv) \cdot R, h, sh)$
ApiCallCond $type(h', v) = \tau$	$((M, pc, s \bullet (d \cdot s'), lv) \cdot R, h, sh)$	$((M, pc + 1, v \cdot s', lv) \cdot R, h', sh')$
ApiCallCond $type(h', b) <: \texttt{Throwable}$	$((M, pc, s \bullet (d \cdot s'), lv) \cdot R, h, sh)$	$((b)_{\text{exc}} \cdot (M, pc, s', lv) \cdot R, h', sh')$
$d = \texttt{null}$ $\lvert s\rvert = \lvert\gamma\rvert$ $b \notin Dom(h)$	$((M, pc, s \bullet (d \cdot s'), lv) \cdot R, h, sh)$	$((b)_{\text{exc}} \cdot (M, pc, s', lv) \cdot$ $R, h[e \mapsto new(\texttt{Throwable})], sh)$

Table 4.2: Rules for Virtual Method Calls

The only non-standard aspect of $\longrightarrow_{\text{JVM}}$ is the treatment of API methods. We assume that we have access only to the signature of methods in *ApiMet*, but not the implementation. We therefore treat API method calls as atomic instructions with a non-deterministic semantics as seen in rules of table 4.2. This is similar to the approach taken, e.g., in [100]. In this sense, we do not practice *complete mediation* as defined by Saltzer in [101]. When an API method is called either the *pc* is incremented and arguments popped from the operation stack and replaced by an arbitrary return value of appropriate type, or else an arbitrary exceptional activation record is returned. Similarly, the return configurations for API method invocations contain an arbitrary heap, since we do not know how API method bodies change heap contents.

Our approach hinges on our ability to recognize such method calls. This property is destroyed by the *reflect* API, which is left out of consideration. Among the method invocation instructions, we discuss here only `invokevirtual`; the remaining invocation instructions are treated similarly.

Exception Handling An exception handler (L_1, L_2, L, c) catches exceptions of type c and its subtypes raised by instructions in the range $[L_1, L_2)$ and transfers control to address L, if it is the topmost handler in the exception handler array that covers the instruction for this exception type.

The predicates *GoodHandler* and *Handles* formalize the constraints for being a handler for an exception:

$$GoodHandler(L, c, (L_1, L_2, L', c')) \Leftrightarrow L_1 \leq L < L_2 \wedge c <: c'$$

$$\begin{aligned} Handles(L, M, L', c) \quad \Leftrightarrow \quad & \exists i \in Dom(H_M), L_1, L_2, c'.\ H_M[i] = (L_1, L_2, L', c') \wedge \\ & GoodHandler(L, c, H_M[i]) \wedge \\ & (\forall j \in Dom(H_M).j < i \Rightarrow \neg GoodHandler(L, c, H_M[j]) \end{aligned}$$

Table 4.3 shows how exceptions are handled. If a valid handler is found in the topmost activation record, control is transferred to the destination of that handler. Otherwise, the topmost activation record is popped off the stack, and the exception is attempted to be caught again in the next activation record.

Condition	C	C'
$type(h,d) = c$ $Handles(L, M, pc, c)$	$((d)_{\text{exc}} \cdot (M, pc, s, lv) \cdot R, h, sh)$	$((M, L, \epsilon, lv) \cdot R, h, sh)$
$type(h,d) = c$ $\neg\exists L \in Dom(P_M).$ $Handles(L, M, pc, c)$	$((d)_{\text{exc}} \cdot (M, pc, s, lv)\cdot$ $(M', pc', s', lv')\cdot, h, sh)$	$((d)_{\text{exc}} \cdot (M', pc', s', lv')\cdot, h, sh)$
	$((d)_{\text{exc}} \cdot \epsilon, h, sh)$	(ϵ, h, sh)

Table 4.3: Exception Handling Rules

The fourth rule of table 4.2 gives an example of an instruction that raises an exception as the conditions for it to execute normally is not satisfied. In this invocation rule, the method can not be invoked since the provided address is not of a class type (it is of type `Null`), so it is not mapped to an object in the heap. Therefore, instead of an activation record for a new method call, an exceptional activation record is pushed on top of the stack. In the semantics, rules with the exception of `athrow` create a new `Throwable` object without calling its constructor, which enables us to treat the process of generating an exception for a run-time error as an atomic operation. The creation of an object is denoted by the function *new* which returns for a class c, an arbitrary object of this type.

4.3 Policies and Security Automata

Security policies specify the acceptable executions of programs. Typical policies met in applications are:

- *Access control policies.* These define restricted access to resources. Examples include "Only user A can read file `foo`" (e.g. on a multi-user system), "An applet can allocate at most 100KB memory" (e.g. on a smart card platform), and "A game may send at most 3 SMS per game" (e.g. on a mobile device).
- *Information flow policies.* These policies capture that confidential information does not flow to a location where this confidentiality is not preserved. Examples are: "Information about a patient should not leak from the hospital database", and "Value of key should stay confidential".
- *Availability policies.* These restrict continuous denial of a resource. An example is "If the web page is requested, then it will eventually be available", and "Bandwidth can not reduce by more than %30 of its peak value".

All these policies can be phrased and specified as sets of (acceptable) executions. Hence, security policies are naturally defined as predicates on sets of executions. Let T be a program for which we identify a set of *security relevant actions* A. Let $\Psi = A^* \cup A^\omega$ denote the set of all executions, where executions are finite or infinite sequences of actions. The executions of T determines a corresponding set $\Pi(\text{T}) \subseteq \Psi$ of finite or infinite traces of actions.

Definition 4.1 (Security Policy [102])**.** A *security policy* is a predicate $\mathcal{P}$ on 2^{Ψ}. Program T *satisfies* a policy $\mathcal{P}$ if $\mathcal{P}(\Pi(\mathrm{T}))$ holds.

In the previous section, we defined executions as sequences of JVM configurations, while the above definition refers to them as sequences of (security relevant) actions. In section 4.5, we describe how, given an execution of the former kind, to obtain the corresponding execution of the latter kind, by mapping each two consecutive configurations to a (possibly empty) sequence of actions.

The various types of policies introduced informally above are all of interest in mobile code security. In our work, however, we use monitoring as the security enforcement mechanism and hence focus on policies that can be effectively enforced by monitoring. Therefore, we first describe this class.

4.3.1 Policies Enforceable by Monitors

Monitors enforce policies by checking, at each step of the target program, whether the current execution satisfies the policy and halting the target program if the policy is about to be violated. Though this definition is simple, different monitoring implementations use different policy languages, thus making it difficult to compare the enforced classes. Below we summarize previous work on enforcing capabilities of monitors, which helps to understand the limitations of the approach and is used in proofs later on in the text.

We let τ and σ range over executions, Ψ is the set of all executions, and $\sigma[..i]$ denotes the prefix of σ consisting of its first i steps.

We start by presenting four necessary conditions for a policy to be enforceable by monitoring. The first observation is that the monitoring mechanism decides whether a given execution is acceptable or not based on this execution alone, and so can only enforce *properties*.

Definition 4.2 (Property [102])**.** A security policy $\mathcal{P}$ is a *property* if and only if it is induced by a characteristic predicate $\hat{\mathcal{P}}$ over executions such that for every subset Π of Ψ,

$$\mathcal{P}(\Pi) \iff \Pi \subseteq \{\sigma \in \Psi \mid \hat{\mathcal{P}}(\sigma)\} \tag{4.1}$$

Properties are defined through their characteristic predicates and hence do not depend on any relationship between the executions they render acceptable. This definition then includes access control policies but leaves out information flow policies [86]. Information flow policies are clearly not enforceable by monitoring single executions, since they state the indistinguishability of program executions with respect to a certain aspect, for instance the values assigned to a subset of program variables during the execution.

Monitors operate by terminating the execution if it the execution of the next action will result in a violation of the the policy, hence if an execution is acceptable

with respect to a policy then all its prefixes are also acceptable, that is the predicate $\hat{\mathcal{P}}$ is *prefix-closed*:

$$\hat{\mathcal{P}}(\sigma[..j]) \Rightarrow \forall i : 1 \leq i < j : \hat{\mathcal{P}}(\sigma[..i]) \tag{4.2}$$

Finally, any execution rejected by a monitor must be rejected in finite time:

$$\neg\hat{\mathcal{P}}(\sigma) \Rightarrow \exists i : 1 \leq i : \neg\hat{\mathcal{P}}(\sigma[..i]) \tag{4.3}$$

A predicate $\mathcal{P}$ that satisfies (4.1-4.3) is a *safety property*. A safety property is a property that stipulates that no "bad thing" happens during an execution [75]. Schneider shows by these observations (4.1-4.3) that safety properties are an upper bound on the set of policies enforceable by simple monitors.

Viswanathan tightens this bound by making explicit the computational constraint involved in the problem [112]. Given a finite execution, the monitor must decide whether to reject it in finite time:

$$\hat{\mathcal{P}}(\sigma) \text{ is decidable when } \sigma \text{ is finite} \tag{4.4}$$

The class defined by (4.1-4.4) corresponds to the class of *co-recursively enumerable* (coRE) properties of programs [112]. A property $\mathcal{P}$ is co-recursively enumerable if a Turing machine $M_{\mathcal{P}}$ can be constructed for this property such that $M_{\mathcal{P}}$ takes an arbitrary Turing Machine M as input and rejects it in finite time if M *does not* have property $\mathcal{P}$, and loops forever otherwise. $M_{\mathcal{P}}$ simulates M to yield each of its finite executions σ and uses $\hat{\mathcal{P}}$ as a decision procedure to determine if σ satisfies the property. $M_{\mathcal{P}}$ semi-decides $\neg\mathcal{P}$, by taking a Turing machine encoding of target program T and rejecting the program in finite time if some execution of T is not in $\mathcal{P}$.

It is important to note that there exist properties in class coRE that can not be enforced by monitors [55]. The policy that "No execution should terminate" is not enforceable by a monitor, since the only way a monitor can intervene to a violating execution is by terminating it. This policy satisfies (4.1-4.4), showing that coRE is a strict upper bound on the policies enforceable by simple monitors.

Hamlen, Schneider, and Morrisett compare the classes of properties enforceable by static analysis, monitoring, and program rewriting [55]. For the comparison of static analysis and monitoring, the interested reader is directed to this work. The technique we use for enforcement is monitor inlining which is both a monitoring and a rewriting technique, therefore we include here a summary of the discussion about these two methods.

The authors define policies enforceable by rewriting through the existence of a total, computable rewriter function, which rewrites a program so that all executions of the resulting program satisfy the policy and if the original program already satisfies the policy, then the rewritten version will be equivalent to the original program. Program equivalence is defined through some arbitrary, but decidable equivalence relation on executions of the original and the rewritten program. Although these

properties capture soundness as introduced for monitors in section 4.1, it is important to note that not all rewriter functions are monitor inliners. The equivalence relation on programs is meant to impose transparency on the rewriter only if the original program already obeys the policy. According to the above definition, a rewriter function is free to change the program in an arbitrary way, in case the program has even a single violating execution. For instance, all security relevant actions may be taken out or remedying actions may be inserted before an action which would otherwise cause a violation.

Rewriting is a clearly a more powerful enforcement mechanism then monitoring. The secret file policy constitutes an example of a policy that is not coRE but still RW-enforceable [55]. In order to satisfy the policy, the program should display information about all files in a directory, except the secret ones. The monitor can terminate the program if it is about to display information about a secret file, but can not enforce that the information on the other files is actually displayed. The program can easily be rewritten to enforce the policy though, by inserting a guard before each information display action that skips the action if the current file is a secret one.

There are policies not enforceable by RW-enforceability due to the computability restriction on the rewriter function. There are even policies that are in the coRE class, but that are not enforceable by rewriting. Thus, the set of policies that are enforceable by an monitoring mechanism constitute the intersection of coRE with RW-enforceable properties [55].

Our notion of monitoring takes termination as the only means available to the monitor for interfering with program execution. We have seen above that this restriction limits the enforcement capabilities of monitors. If different means of intervention are available to the enforcement mechanism, then a larger class of policies can be enforced [80]. Ligatti *et al.* [79] introduce a notion of monitoring where the executions can be altered by the monitor by inserting and suppressing actions besides truncation. This enables monitors to enforce properties that do not satisfy (4.2). In fact, policies enforceable by these monitors are not limited to safety properties. But since we focus on the former definition of monitoring (monitors with only termination capability), we do not elaborate on the latter type and direct the interested reader to [80] for further reading.

4.3.2 Security Automata

Any formalism that allows sets of executions consisting of (finite as well as infinite) sequences of actions to be specified can be used to express security policies. The most prominent classes of specification formalisms for this purpose are regular expressions, automata and temporal logics.

The notion of security automata was introduced by Schneider [102] to define security policies. Security automata capture safety properties. Here, we view a *security automaton* over alphabet A as an automaton $\mathcal{A} = (Q, \delta, q_0)$ where Q is a countable set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times A \rightharpoonup Q$ is

a (partial) transition function. All $q \in Q$ are viewed as accepting. A security automaton $\mathcal{A}$ induces a security policy $\mathcal{P}_\mathcal{A} \subseteq 2^{A^* \cup A^\omega}$ through its language $L_\mathcal{A}$ by $\mathcal{P}_\mathcal{A}(X) \Leftrightarrow X \subseteq L_\mathcal{A}$.

While security automata are a useful formalism, their direct use in the specification of security policies is in many ways inconvenient. For example, security automata are not necessarily finite objects. Therefore, languages have been proposed to aid policy writing. PSLang (Policy Specification Language) is one such policy language, introduced by Erlingsson and Schneider [40, 39]. PSLang policies consist of a set of variable declarations, followed by a list of security relevant events, where each event is accompanied by a piece of Java-like code that specifies how the security state variables should be updated in case the event is encountered in the current state. A policy text is intended to encode a security automaton: the state variables represent the automaton states and updates represent transitions. While the intuition is given, the exact way to extract an automaton from a PSLang policy is not provided by the authors. Such a task is not trivial due to the power of the programming language constructs that can be used in the updates.

In this study, we use the PSLang inspired language ConSpec (see section 4.4) to express policies. We focus on security automata that are induced by policies in ConSpec and are therefore named *ConSpec automata*. The security relevant actions are method calls, represented by the class name and the method name of the method, along with a sequence of values that represent the actual arguments. We partition the set of security relevant actions into *pre-actions* $A^\flat \subseteq \mathbb{C} \times \mathbb{M} \times Val^* \times \mathbb{H}$ and *post-actions* $A^\sharp \subseteq RVal \times \mathbb{C} \times \mathbb{M} \times Val^* \times \mathbb{H} \times \mathbb{H}$, corresponding to method invocations and returns. Both types of actions may refer to the heap prior to method invocation, while the latter may also refer to the heap upon termination and to a return value from $RVal = Val \cup \{\text{exc}\}$ where exc is used to mark exceptional return from a method call[3]. The partitioning on security relevant actions induces a corresponding partitioning on the transition function δ of ConSpec automata.

4.4 ConSpec

In this section, we introduce the policy specification language ConSpec [3]. ConSpec is intended for programs written in intermediate object-oriented languages such as the bytecode languages of Java and .NET. Security relevant actions are taken as method invocations, more specifically system calls or invocations of API methods. ConSpec is designed to be used for both specification of requirements and the description of the security-relevant behavior of systems, e.g. for specifying the producer policy in the framework introduced in section 4.1. For this reason, the formalism selected is based on automata, which have been used for both purposes. For instance, the SPIN tool [60] inputs system specifications as models written in the guarded-command language Promela and performs model checking on the

[3]We disregard the exceptional value since we do not, as yet, put constraints on these in ConSpec policies.

Büchi automata extracted from these models. Security properties and security relevant behavior of systems are expressed as automata in various other approaches (e.g. [102, 103]).

ConSpec is strongly inspired by the policy specification language PSLang, which was developed by Erlingsson and Schneider [39] for runtime monitoring. However, ConSpec is more restricted than PSLang. This is a design decision taken in order to enable automatic matching of contracts against policies, and not just runtime monitoring. More specifically, ConSpec does not allow arbitrary types in representing the security state and restricts the way the security state variables are updated. We have used a guarded-command language for the updates where the guards are side-effect free and commands do not contain loops. The simplicity of the language then allows for a comparatively simple semantics.

Example 4.3. Assume method `Open` of class `File` is used for creating files (when argument `mode` has value "CreateNew") or for opening files (`mode` is "Open"), either for reading (argument `access` is "OpenRead") or for writing. Assume further that method `Open` of class `Connection` is used for opening connections, that method `AskConnect` is used for asking the user for permission to open a connection and that this latter method returns true in case of approval. Now, consider the security policy, which allows applications to access existing files for reading only, and requires, once such a file has been accessed, applications to obtain approval from the user each time a connection is to be opened. This policy can be specified in ConSpec as follows:

```
SCOPE Session

SECURITY STATE
     bool accessed= false;
     bool permission = false;

BEFORE File.Open(string path, string mode, string access)
PERFORM
     mode.equals("CreateNew")             -> { skip; }
     mode.equals("Open") && access.equals("OpenRead")
                                -> { accessed= true; }

BEFORE Connection.Open(string type, string address)
PERFORM
     !accessed              -> { permission = false; }
     accessed && permission -> { permission = false; }

AFTER bool answer= GUI.AskConnect() PERFORM
     answer  -> { permission=true; }
     !answer -> { permission=false; }
```

```
MAXINT M
MAXLEN N
SCOPE <Object ClassName PersistentStateDec
      | Session
      | Multisession PersistentStateDec
      | Global PersistentStateDec>

SECURITY STATE
    PrimType SecVar1 = InitVal1
            ⋮
    PrimType SecVarN = InitValN

Clause1
            ⋮
ClauseK
```

(a) Policy Syntax

```
<BEFORE
 |EXCEPTIONAL
 |AFTER [Type Name = ]> Signature

    PERFORM
    Guard1          -> {UpdateBlock1}
                ⋮
    GuardM          -> {UpdateBlockM}
    [ ELSE          -> {UpdateBlock} ]
```

(b) Event Clause Syntax

Figure 4.2: ConSpec Syntax

We specify by setting the scope to `Session` that the policy applies to each single execution of an application. Scope declaration is followed by a *security state declaration*: the security state of the example policy is represented by the boolean variables `accessed` and `permission`, which are both false initially to mark, respectively, that no file has been accessed and that no permissions are granted when the program begins executing. The example policy contains three *event clauses* that state the conditions for and effect of the security relevant actions: call to the method `File.Open`, call to the method `Connection.Open` and return from the method `GUI.AskConnect`. The types of the method arguments are specified along with representative names, which have the event clause as their scope. The *modifiers* BEFORE and AFTER mark whether the call of or the normal return from the method specified in the event clause is security relevant (exceptional returns can be specified by the modifier EXCEPTIONAL). Event clauses contain guards and associated updates to the security state variables.

4.4.1 Syntax

Figure 4.2 summarizes the syntax of ConSpec. Before the actual policy, ConSpec policies set a limit on values of the type `int` which consist of some initial segment of natural numbers. Similarly, a maximum length for strings is specified. (We have skipped these in the example policy.) This aims to limit the state space of the corresponding automata, in order to enable matching. The `Scope` construct is used for expressing security requirements on different levels and explained in more detail below.

States. The security state variables of ConSpec are restricted to the primitive types (*`PrimType`*): booleans, integers, and strings.

Event Clauses. An event clause (figure 4.2(b)) gives us a security relevant action and its modifier. Security relevant events are bound to the API methods in the program (we support programs delivered in either Java or .NET intermediate bytecode language). In order to resolve which method is of interest in case of overloading, the argument types of the method is to be specified as part of the action specification. The security relevant action is then fully specified by its *signature* which consists of the name of the method, the class to which the method belongs and the types of its arguments. The signature of an event clause is defined as the signature of the method associated with it. In ConSpec policies, all event clauses with the same modifier have a unique signature. This restriction has been imposed in order to ensure determinism and means that one can not, for example, have two `BEFORE` clauses for the same method. Notice that since the signature does not include the type of the return variable, it is not possible to have two `AFTER` event clauses for the same method, even if they do not agree on the return variable types. The modifier states when the update to the state will be performed: before the event, after the event or immediately after the throwing of an exception by the event.

Guards and Update Blocks. The event specification is followed by a sequence of pairs of guards and update blocks. The update block specifies how a state will be updated for the security relevant action while the guard selects the states, which the particular update will apply to. The guards are evaluated top to bottom and the update corresponding to the first guard that holds is performed. In case none of the guards evaluates to true, there is no transition for that action from the current state, unless an `ELSE` block is present, in which case the update of this block is executed. The guard is a side-effect free boolean expression which can only mention argument values (and the return value when the `AFTER` modifier is used) and the security state. The update block begins with declarations of the local variables, which have the current block as their scope. A list of assignments to local variables and security state variables follow the declarations. If no assignments are present, the update block consists of the statement `skip`. The expression language used for forming guards and right hand side of assignments are explained below.

Expressions. The expression language of ConSpec has been designed to ensure that checking language containment of the induced automata (i.e. the matching problem) is decidable. The sets of expressions and boolean expressions of ConSpec are *Exp* and *BoolExp*, respectively. Variables except security state and local variables can be of a primitive type or an object. The expressions on integers are built using basic arithmetic and comparison operators. Strings can be checked for equality and the prefix relation using the functions `equals` and `beginsWith` respectively. The expression language of ConSpec can potentially be extended with calls to other side-effect free functions. Expressions can also include field accesses using object references, expressed by the "`.`" operator. Regardless of the modifier used, accesses to fields of method arguments are interpreted on the heap at the time of call, while accesses to fields of the return value are interpreted at the time of return. Therefore, it is not, as of yet, possible to put constraints on the fields of a method

argument at the time of return. The last field accessed in a field access expression should always be of one of the primitive types.

Example 4.4. The following policy specifies executions where at most one message is sent per day, and where all messages are sent to a single phone number, determined by the first message sent.

```
SCOPE Session
SECURITY STATE
     int lastmessageday = 1;
     string usednumber = "";

BEFORE WindowsMobile.PocketOutlook.SmsMessage.Send()
PERFORM
    usednumber == "" &&
    this.To.Count == 1     -> { lastmessageday = Now.GetDay();
                                usednumber = this.To[0]; }
    this.To.Count == 1 &&
    this.To[0] == usednumber &&
    !Now.GetDay().equals(lastmessageday)
                           -> { lastmessageday = Now.GetDay(); }
```

The security state of the policy includes the integer variable `lastmessageday` that stores the date of the last text message, and the string `usednumber`, which is used to record the phone number of the recipient. The example contains a single event clause, bound to the call of the .NET API method `WindowsMobile.PocketOutlook.SmsMessage.Send`. This method does not have any arguments, and all parameters about the message are stored in the fields of the `SmsMessage` object. In particular, the `Count` field of the `To` field of a message states how many times the message will be sent and the first recipient address is provided in the first member of this field.

Scopes. Case studies show that many interesting real-life policies concern the entire execution history rather than a single run of the application [115]. However, most policy languages (including PSLang) do not contain the feature of distinguishing between events in the current run and in the previous runs. ConSpec is expressive enough to write policies on multiple executions of the same application (scope `Multisession`) and on executions of all applications of a system (scope `Global`), in addition to policies on a single execution of the application (scope `Session`) and on lifetimes of objects of a certain class (scope `Object`). The syntax of persistent state declaration is similar to security state declaration and aims to specify the state that is preserved across single executions when the scope is `Multisession` or `Global`. When the scope is `Object`, the security state declaration specifies the state local to each object of the class, while the persistent state is equivalent to the security state of scope `Session`, that is the security state of the application for a single execution.

Example 4.5. The personal information manager (PIM) saves personal information (e.g. phonebook) in mobile devices. Secure connections are established by connecting to destinations starting with "https://". The policy below specifies that an application must not access the PIM while unsecure connections are open and can only open secure connections after the PIM is accessed.

```
SCOPE Object Connection

PERSISTENT SECURITY STATE
     bool opened = false;

SECURITY STATE
     bool secure = false;
     bool active = false;

BEFORE PIM.open()
PERFORM
     secure || !active -> { opened = true; }

BEFORE Connection.open(string url)
PERFORM
     !opened &&  url.startsWith("https")
                    -> { active = true; secure = true; }
     !opened && !url.startsWith("https")
                    -> { active = true; secure = false; }
      opened &&  url.startsWith("https")
                    -> { active = true; }

AFTER Connection.close()
PERFORM
     true -> { active = false;}
```

4.4.2 Semantics

The formal semantics of ConSpec policies is defined in terms of *symbolic security automata*, which in turn induce ConSpec automata.

Definition 4.6 (Symbolic Security Automaton)**.** Given a set *Svar* of security state variables and a set *Var* of variables, a *symbolic security automaton* is a tuple $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$, where:

(i) $q_s = Svar$ is the initial and only state;

(ii) $Init_s : q_s \to Val$ is an *initialization function*;

(iii) $A_s = A_s^\flat \cup A_s^\sharp$ is a countable set of *symbolic actions*, where:
$A_s^\flat \subseteq \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ are *symbolic pre-actions*, and
$A_s^\sharp \subseteq \{(\{PrimType \cup \mathbb{C}\} \times Var) \cup \texttt{Unit} \cup \{exc\}\} \times \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ are *symbolic post-actions*;

(iv) $\delta_s = \delta_s^\flat \cup \delta_s^\sharp$ is a *symbolic transition relation*, where:
$\delta_s^\flat \subseteq A_s^\flat \times BoolExp \times (q_s \to Exp)$ and
$\delta_s^\sharp \subseteq A_s^\sharp \times BoolExp \times (q_s \to Exp)$
are the symbolic pre- and post-transitions, respectively.

ConSpec policies and symbolic automata are two very similar representations. The set of security state variables of a ConSpec policy is the state of the symbolic automaton. Each event clause clause gives rise to one symbolic action, and each guarded command of the clause gives rise to a symbolic transition consisting of the security relevant action itself, the guard of the guarded command in conjunction with negations of the guards that lie above it in the clause, and the effect of the guarded command. The updates to security state variables, which are presented as a sequence of assignments in ConSpec, are captured in the automaton as functions that return one ConSpec expression per symbolic state variable, determining the value of that variable after the update.

Symbolic Automaton Extraction From Policy Text

The semantics of a ConSpec policy $\mathcal{P}$ is given in terms of a symbolic automaton $\mathcal{A}_s = (Q_s, A_s, \delta_s, q_s, Init_s)$ as described below. The set of variables Var and the set of types Ω are fixed for ConSpec policies as the set of all variable names and the set of types: $\{\texttt{int}, \texttt{bool}, \texttt{string}\} \cup \mathbb{C}$.

States. The set of states Q_s of the symbolic automaton $\mathcal{A}_s$ is determined by the declarations in the `SECURITY STATE` block of the policy $\mathcal{P}$. Consider the security state declaration of $\mathcal{P}$:

$$\begin{array}{ll} \texttt{SECURITY STATE} & \tau_{s_1}\ s_1 = v_1 \\ & \vdots \\ & \tau_{s_k}\ s_k = v_k \end{array}$$

The set of variable names that are induced by such a state declaration is the set of *security state variables* which give us the only state of the automaton: $q_s = \{s_1, \ldots, s_k\}$ and $Q_s = \{q_s\}$. The initialization function simply maps the variables in state q_s to their initial values: $\forall s_i \in q_s.\ Init_s(s_i) = v_i$.

Actions. The actions A_s of the automaton are the events mentioned in event clauses of the policy.

- The action $(\mathbf{c}.\mathbf{m}(\tau_1\, x_1, \ldots, \tau_n\, x_n)) \in A_s^\flat$, if and only if the $\mathcal{C}$ contains an event clause with the modifier `BEFORE` followed by the event $\mathbf{c}.\mathbf{m}\,(\tau_1\, x_1,\, \ldots,\, \tau_n\, x_n)$
- The action $(r, \mathbf{c}, \mathbf{m}, ((\tau_1, x_1),\, \ldots,\, (\tau_n, x_n))) \in A_s^\sharp$ and $r = (\tau, x)$ or $r = \mathit{void}$, if and only if $\mathcal{C}$ contains an event clause with the modifier `AFTER` followed by $\tau\, x$ and the event $\mathbf{c}.\mathbf{m}\,(\tau_1\, x_1,\, \ldots,\, \tau_n\, x_n)$ or the modifier `AFTER` followed by only the event $\mathbf{c}.\mathbf{m}\,(\tau_1\, x_1,\, \ldots,\, \tau_n\, x_n)$, respectively.
- The action $(\mathrm{exc}, \mathbf{c}, \mathbf{m}, ((\tau_1, x_1),\, \ldots,\, (\tau_n, x_n))) \in A_s^\sharp$, if and only if $\mathcal{C}$ contains an event clause with the modifier `EXCEPTIONAL` followed by the event $\mathbf{c}.\mathbf{m}\,(\tau_1\, x_1,\, \ldots,\, \tau_n\, x_n)$.

Transitions. Each event clause of the policy induces a partial transition function. The transition functions $\delta_s^\flat$ and $\delta_s^\sharp$ of the automaton are the union of the partial functions corresponding to event clauses with the `BEFORE` and `AFTER/EXCEPTIONAL` modifiers, respectively. The definition of the partial functions is similar for both types of event clauses. For brevity, here we only the case for an `AFTER` clause.

Consider an `AFTER` event clause $\phi^\sharp$:

```
AFTER   τ x = c.m (τ1 x1, ..., τn xn)
PERFORM
   G1  ->  U1
   ⋮
   Gm  ->  Um
```

Let $\mathit{Avar} = \{x_1, \ldots, x_n\}$ be the set of formal arguments of the event and $\mathit{Pvar} = \{x\} \cup \mathit{Avar}$ be the set of all program variables of the event clause, where $\mathit{Pvar} \subseteq \mathit{Var}$. Below, let states $q \in Q$ be valuations for the security state variables that respect their types, $q : \mathit{Svar} \to \mathit{PrimVal}$, and let $\sigma : \mathit{Pvar} \to \mathit{Val}$ range over the set Σ of valuations of program variables that respect the declared types of the variables. We assume for any ConSpec expression $E \in \mathit{Exp}$ occurring in this AFTER event clause, the semantic function:

$$[\![E]\!] : Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \to \mathit{PrimVal}$$

and every update block U_j of $\phi^\sharp$, we assume the semantic function:

$$[\![U_j]\!] : Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \to Q$$

where the two heaps in the function types refer to the heap of the program before and after the execution of the method call, respectively.

Semantics of field access expressions occurring in guards and update blocks are relativized on heaps. The heap is not changed by the automata, but is used to look up fields of object references. To fetch the field values of method arguments, the heap at the time of call is accessed. In order to get the field values of the return

value, however, heap at the time of return is used. Below, we denote the heap before the call with $h^\flat$, and the heap after the call with $h^\sharp$. The value of a field access expression with depth k is as follows:

$$x.\mathbf{f_1}.\mathbf{f_2}.\ldots.\mathbf{f_k} = \begin{cases} (h^\flat(\ldots(h^\flat(h^\flat(\sigma(x)).\mathbf{f_1}).\mathbf{f_2})\ \ldots).\mathbf{f_k}) & \text{if } x \in \mathit{Avar} \\ (h^\sharp(\ldots(h^\sharp(h^\sharp(\sigma(x)).\mathbf{f_1}).\mathbf{f_2})\ \ldots).\mathbf{f_k}) & \text{if } x \in \mathit{Pvar}\backslash\mathit{Avar} \end{cases}$$

The updates performed by an update block U can be captured by a function $f_U : \mathit{Svar} \to \mathit{Exp}$ which maps each security state variable to a ConSpec expression such that the execution of U and the application of f_U have the same effect on the security state:

$$\forall q \in Q, \sigma \in \Sigma, h, h' \in \mathbb{H}.\ [\![U]\!](q, \sigma, h, h') = \lambda s \in \mathit{Svar}.\ [\![f_U(s)]\!](q, \sigma, h, h')$$

In appendix B.1, a method for obtaining f_U using syntactic transformations on U is presented.

The set of transitions induced by the event clause $\phi^\sharp$ above is denoted $\delta^\sharp_\phi$ and is the least set for which:

(i) $(a^\sharp_s, G_1, f_{U_1}) \in \delta^\sharp_\phi$

(ii) $\forall 1 < i < m.\ (a^\sharp_s, \neg G_1 \wedge \ldots \neg G_{i-1} \wedge G_i, f_{U_i}) \in \delta^\sharp_\phi$

Notice that ϕ induces all transitions of the automaton with the action $a^\sharp_s = ((\tau, x), \mathbf{c}, \mathbf{m}, ((\tau_1, x_1), \ldots, (\tau_n, x_n)))$. This definition captures that the guards are evaluated top to bottom in order to select the right update block.

Finally, the post-transition function $\delta^\sharp_s$ is the union of the transitions induced by each event clause (with disjoint domains):

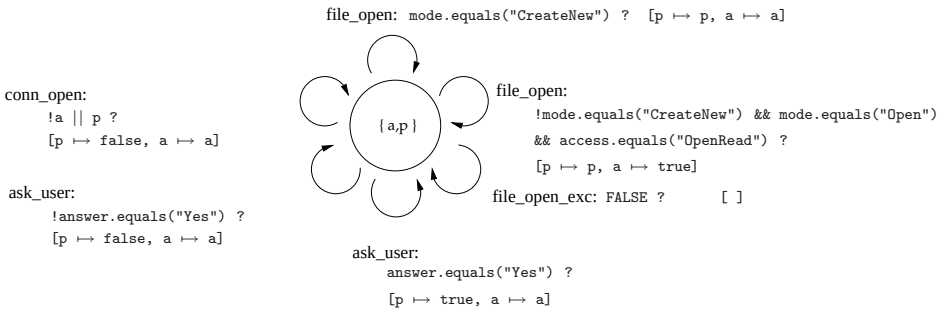
$$\delta^\sharp_s = \biguplus_{\phi^\sharp \in \mathcal{P}} \delta^\sharp_\phi$$

Example 4.7. In figure 4.3 we present the symbolic automaton corresponding to the ConSpec policy of example 4.3, using "a" for `accessed` and "p" for `permission`.

ConSpec Automaton Induced by Symbolic Automaton

Symbolic automata determine ConSpec automata in the following way: Let $\mathcal{A}_s = (q_s, A_s, \delta_s, \mathit{Init}_s)$ be a symbolic automaton. The ConSpec automaton induced by $\mathcal{A}_s$ is the automaton $\mathcal{A} = ((q_s \to \mathit{Val})_\perp, \delta, \mathit{Init}_s)$ over alphabet A, determined as follows:

- The post-actions of A are all tuples $(v, c, m, v_1 \cdots v_n, h^\flat, h^\sharp)$ such that there is a symbolic post-action $a^\sharp_s = (r, c, m, ((\tau_1\ x_1), \ldots, (\tau_n\ x_n)))$ with $v_i : \tau_i$ for all $i : 1 \leq i \leq n$, and either $r = \tau x$ and $v : \tau$ or else $x = r \in \{\mathit{void}, \mathrm{exc}\}$. The pre-actions are defined similarly.



$A_s^\flat$={file_open, conn_open}
file_open=(File,Open,(string path, string mode, string access))
conn_open=(Connection,Open,(string type, string address))
$A_s^\sharp$={ask_user, file_open_exc}
ask_user=(string answer, GUI,AskConnect,())
file_open_exc=(*exc*,File,Open,(string path, string mode, string access))
$Init_s$ =[p ↦ false, a ↦ false]

Figure 4.3: Symbolic Automaton for the Example Policy

- The post-transition function $\delta^\sharp$ is defined indirectly, by referring to the standard denotational semantic functions for expressions $e \in Exp$ and boolean expressions $b \in BoolExp$ such that $[\![e]\!] : (Svar \to Val) \to (Var \to Val) \to \mathbb{H} \to \mathbb{H} \to Val$ and $[\![b]\!] : (Svar \to Val) \to (Var \to Val) \to \mathbb{H} \to \mathbb{H} \to Val$, defined as expected. Then, if $\delta_s^\sharp(a_s^\sharp, b, E)$ in $\mathcal{A}_s$, we define $\delta^\sharp(q, a^\sharp) = q'$ in $\mathcal{A}$ if and only if there exists an interpretation I and heaps $h^\flat$ and $h^\sharp$ such that $[\![a_s^\sharp]\!] I h^\flat h^\sharp = a^\sharp$, $[\![b]\!] q I h^\flat h^\sharp = true$, and $[\![E(v)]\!] q I h^\flat h^\sharp = q'(v)$ for all $v \in Svar$. The pre-transition function $\delta^\flat$ is defined similarly. In addition, given post-action $a_s^\sharp$, let B be the set of boolean expressions b such that $\delta_s^\sharp(a_s^\sharp, b, E)$ for some E. Then, for every state $q \in Q$, interpretation I, and heaps $h^\flat$ and $h^\sharp$, we define $\delta^\sharp(q, a^\sharp) = \bot$ if $[\![a_s^\sharp]\!]\ I\ h^\flat\ h^\sharp = a^\sharp$ and $[\![b]\!]\ q\ I\ h^\flat\ h^\sharp = false$ for all $b \in B$.

The language of a symbolic automaton $\mathcal{A}_s$ is defined as the language $L_\mathcal{A}$ of the induced ConSpec automaton $\mathcal{A}$.

Converting ConSpec Automata to Büchi Automata

Each ConSpec automaton extracted from a policy can be written as a deterministic Büchi automaton by adding a self loop to all states with a new action that represents all security irrelevant steps of the program. This new action is also used for completing finite traces of the program to infinite traces in order to meet

the Büchi acceptance condition. Finally, a non-accepting state is added and all missing transitions of the original automaton are directed to this new state; the idea is that the automaton is stuck in this state if it violates the policy. The automaton is deterministic as all ConSpec automata induced by ConSpec policies are deterministic.

Matching

Although we do not address the policy matching problem in this work, we give some insight on this issue and present some pointers to related work. One way to match a producer policy against a device policy when both are expressed in ConSpec is to check that the language of the producer policy automaton is included in the language of the policy automaton. Since the domains of the security state variables are bounded, the extracted automata have finitely many states (but possibly infinitely many transitions) and standard methods for checking language inclusion for automata (see for instance [28]) can be facilitated for matching. Such an approach is taken in [85] for matching producer policies (referred as *contracts*) against device policies, when both are expressed as *automata modulo theory* ($\mathcal{AMT}$), a type of symbolic Büchi automata. The idea is essentially to check, given the producer policy automaton A_P and the device policy automaton A_D, whether the language of the product automaton $A_P \times \overline{A_D}$ is empty. ConSpec policies can be converted to $\mathcal{AMT}$s in order to make use of the matching algorithms provided in [85].

4.5 Monitoring with ConSpec Automata

In this section, we first formalize the infinite or finite sequence of security relevant actions induced by a target program execution. Each target transition can give rise to zero, one, or two security relevant actions, namely, in the latter case, a preaction followed by a postaction. Given the action set A, and the configurations C_1 and C_2, we define the security relevant preaction, $act^{\flat}_A(C_1)$, of the configuration C_1, and the corresponding postaction $act^{\sharp}_A(C_1, C_2)$, as in the table below. If none of the conditions of the table hold, the corresponding action is ϵ.

$act^{\flat}_{\mathcal{P}}(C)$	Condition
(c, m, s, h_b)	$C = ((M, pc, s \cdot [d] \cdot s', lv) \cdot R, h^{\flat})$ $M[pc] =$ `invokevirtual` $c'.m$, $\quad c$ *defines* $type(h^{\flat}, d).m$, $\quad type(h^{\flat}, d) <: c'$ $(c, m, s, h^{\flat}) \in A^{\flat}$

$act^{\sharp}_{\mathcal{P}}(C_1, C_2)$	Condition
$(void, c, m, s, h^{\flat}, h_a)$	$C_1 = ((M, pc, s \cdot d \cdot s', lv) \cdot R, h^{\flat})$, $\quad C_2 = ((M, pc + 1, s', lv) \cdot R, h^{\sharp})$, $M[pc] =$ `invokevirtual` $c'.m$, $\quad c$ *defines* $type(h^{\flat}, d).m$, $\quad type(h^{\flat}, d) <: c'$, $(void, c, m, s, h^{\flat}, h^{\sharp}) \in A^{\sharp}$
$(v, c, m, s, h^{\flat}, h^{\sharp})$	$C_1 = ((M, pc, s \cdot d \cdot s', lv) \cdot R, h^{\flat})$, $\quad C_2 = ((M, pc + 1, v \cdot s', lv) \cdot R, h^{\sharp})$, $M[pc] =$ `invokevirtual` $c'.m$, $\quad c$ *defines* $type(h^{\flat}, d).m$, $\quad type(h^{\flat}, d) <: c'$, $(v, c, m, s, h^{\flat}, h^{\sharp}) \in A^{\sharp}$
$(\mathrm{exc}, c, m, s, h^{\flat}, h^{\sharp})$	$C_1 = ((M, pc, s \cdot d \cdot s', lv) \cdot R, h^{\flat})$, $\quad C_2 = ((b)_e \cdot (M, pc, s', lv) \cdot R, h^{\sharp})$, $M[pc] =$ `invokevirtual` $c'.m$, $\quad c$ *defines* $type(h^{\flat}, d).m$, $\quad type(h^{\flat}, d) <: c'$, $(\mathrm{exc}, c, m, s, h^{\flat}, h^{\sharp}) \in A^{\sharp}$

We obtain the *security relevant trace*, $srt_A(E)$, of an execution E by lifting the operations $act^\flat_A$ and $act^\sharp_A$ co-inductively to executions in the following way:

$$\begin{array}{rr} srt_A(\epsilon) = \epsilon & srt_A(C) = act^\flat_A(C) \\ srt_A(C_1C_2 \cdot E) = act^\flat_A(C_1) \cdot act^\sharp_A(C_1, C_2) \cdot srt_A(C_2 \cdot E) & \end{array}$$

Then a target program T *adheres* to a policy $\mathcal{P}$, if the security trace of each execution of T is in the language of the corresponding ConSpec automaton $\mathcal{A}_\mathcal{P}$, i.e.

$$\forall E \in \Pi(\mathrm{T}).\, srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}$$

Monitor co-execution A basic application of a ConSpec automaton is to execute it alongside a target program to monitor for policy compliance. We can view such an execution as an interleaving $w = (C_0, q_0)(C_1, q_1)\cdots$ such that C_0 and q_0 is the initial configuration and state of T and $\mathcal{A}$, respectively, and such that for each consecutive pair $(C_i, q_i)(C_{i+1}, q_{i+1})$, either the target (only) progresses:

$$C_i \longrightarrow_{\mathrm{JVM}} C_{i+1} \text{ and } q_{i+1} = q_i$$

or the automata (only) progresses:

$$C_{i+1} = C_i \text{ and } \exists a \in A.\, \delta(q_i, a) = q_{i+1}.$$

In the former case we write $(C_i, q_i) \longrightarrow_{\mathrm{JVM}} (C_{i+1}, q_{i+1})$, and in the latter case we write $(C_i, q_i) \longrightarrow_{\mathrm{AUT}} (C_{i+1}, q_{i+1})$. We can, without loss of generality, assume that at most one of these cases apply, for instance by tagging each interleaving step.

The first projection function $w \downarrow 1$ on interleavings $w = (C_1, q_1)(C_2, q_2)\cdots$ extracts the underlying execution as follows:

$$\begin{array}{rcl} ((C_1, q_1)(C_2, q_2) \cdot w') \downarrow 1 & = & \begin{cases} C_1(((C_2, q_2) \cdot w') \downarrow 1) & C_1 \longrightarrow_{\mathrm{JVM}} C_2 \\ ((C_2, q_2) \cdot w') \downarrow 1) & \mathit{otherwise} \end{cases} \\ (C, q) \downarrow 1 & = & C \end{array}$$

To similarly extract automata derivations we use the (co-inductive) function *extract* such that

$$\mathit{extract}((C_1, q_1)(C_2, q_2)w) = q_1 q_2\, \mathit{extract}((C_2, q_2)w)$$

if $(C_1, q_1) \longrightarrow_{\mathrm{AUT}} (C_2, q_2)$,

$$\mathit{extract}((C_1, q_1)(C_2, q_2)w) = act^\flat_A(C_1)\, act^\sharp_A(C_1, C_2)\, \mathit{extract}((C_2, q_2)w),$$

if $(C_1, q_1) \longrightarrow_{\mathrm{JVM}} (C_2, q_2)$, $\mathit{extract}(C, q) = act^\flat_A(C)$, and $\mathit{extract}(\epsilon) = \epsilon$. We call such an extracted sequence of automaton states and security relevant action a *potential derivation*. Note that $\mathit{extract}(w)$ may well be finite even if w is infinite.

Definition 4.8 (Co-Execution)**.** Let $E^\flat = \{qq'a^\flat \mid q, q' \in Q, a^\flat \in A^\flat, \delta^\flat(q, a^\flat) = q'\}$, $E^\sharp = \{a^\sharp qq' \mid q, q' \in Q, a^\sharp \in A^\sharp, \delta^\sharp(q, a^\sharp) = q'\}$. An interleaving w is a co-execution if

$$extract(w) \in (E^\flat \cup E^\sharp)^* \cup (E^\flat \cup E^\sharp)^\omega$$

In other words, an interleaving is a co-execution, if the potential derivation it extracts corresponds to a real derivation.

A monitor is *conservative* if all monitored executions are also executions of the original program, i.e. if the monitor does not introduce new behavior. When monitoring is done by ConSpec automata in the sense captured by the notion of co-execution, the monitor is sound, transparent and conservative.

Theorem 4.9 (Correctness of Monitoring by Co-execution)**.** *Let* T *be a program, and $\mathcal{P}$ a policy. The following holds, where A is the action set of $\mathcal{A}_\mathcal{P}$:*

$$\{w \downarrow 1 \mid w \textit{ is a co-execution of } \mathrm{T} \textit{ and } \mathcal{A}_\mathcal{P}\} = \{E \in \Pi(\mathrm{T}) \mid srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}\}$$

Proof. The proof can be found in Appendix B.3.1. □

A corollary of this result is that the set of executions of a program that obeys the policy are identical to the set of executions of the program monitored for the policy.

Corollary 4.10. *Program* T *adheres to policy $\mathcal{P}$ if, and only if, for each execution $C_0C_1\cdots$ of* T *there is a co-execution w of* T *and the automaton $\mathcal{A}_\mathcal{P}$ such that $w \downarrow 1 = C_0C_1\cdots$.*

4.6 Annotation Language

We specify self-monitoring using annotations in a Floyd-style logic for bytecode. We build on the program logic of Bannwart and Müller [10]. As an extension to their logic, our annotation language makes use of "ghost" variables. These are essentially specification variables that can be assigned values by a multi-assignment statement.

Methods are equipped with annotations consisting of assertions on the extended state (current configuration and current ghost variable environment), and ghost variable assignments. We first introduce the syntax of this annotation language.

Assertions Let g range over ghost variables, i over natural numbers, and let *Op* and *Bop* range over a standard, not further specified, collection of unary and binary operations on strings and integers. Expressions e and assertions a have the following shape:

$$\begin{array}{lcl} e & ::= & \bot \mid v \mid g \mid e.f \mid \mathrm{s}[i] \mid \mathrm{r}i \mid Op\ e \mid e\ Op\ e \\ a & ::= & e\ Bop\ e \mid e : c \mid e <: c \mid \neg a \mid a \wedge a \mid a \vee a \end{array}$$

Here, s[i] is the value at the i^{th} position of the current operation stack, if defined, and $\perp$ otherwise, $e : c$ is a class membership test and $e <: c$ is a subclass membership test. The notation ri denotes the i^{th} local variable. The assertions are evaluated with respect to extended states. Extended states consist of a program configuration C and a ghost environment σ that maps ghost variables to integer values, addresses or the value $\perp$, which captures that the variable is undefined. Referring to standard denotational semantics, we assume a semantic function $\|a\|(C, \sigma)$ that returns, for assertion a and extended state (C, σ), a truth value.

Example 4.11. The following example assertion states that if the address at the top of the stack points to an object of type `GUI` in the heap, then the ghost variables g_{a} and g_{p} are both defined.

$$\mathrm{s}[0] : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\perp, \perp)$$

Ghost Variable Instructions Ghost variables are assigned using a single, guarded multi-assignment of the form

$$\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} \mid \cdots \mid a_m \rightarrow \overrightarrow{e_m} \tag{4.5}$$

where $\overrightarrow{gs}$ is a vector of ghost variables and $\overrightarrow{e_i}$ $(1 \leq i \leq m)$ are vectors of expressions, such that the arities (and types) of $\overrightarrow{gs}$ and the $\overrightarrow{e_i}$ match. The multi-assignment is performed with vector $\overrightarrow{e_i}$ if guard a_i is the first guard (from left) that holds in the current extended state. If no guard is true, the ghost state is assigned the constant vector with all elements $\perp$ and the arity matching to that of $\overrightarrow{gs}$. This is the case, in particular, when $m = 0$ in (4.5) above. This case is written as follows: $\overrightarrow{gs} := ()$.

Method Annotations A target program is annotated by an extended environment, Γ^*, which maps method references M to tuples $(P, H, A, \mathit{Requires}, \mathit{Ensures})$ such that A is an assignment to each program point $n \in \mathit{Dom}(P)$ of a sequence, ψ, of atomic annotations, i.e. assertions and ghost variable assignments. *Requires* also consists of a sequence of atomic annotations, while *Ensures* is a single assertion. These two clauses do not mention method arguments or return values. The precondition *Requires* is allowed to contain ghost assignments, since we occasionally use these clauses to initialize ghost variables.

Annotation Semantics In the absence of ghost variable assignments the notion of annotation validity is the expected one, i.e. the assertions annotating a given program point (or the point of exceptional return) hold whenever control is at that program point. To extend this account to ghost variables, the ghost variable assignments should be given a suitable semantics. We present such a semantics in this section, which essentially treats ghost variables as program variables. For this purpose, the program state is extended by a store for ghost variables which is altered only by ghost variable assignments, method calls and returns.

The rewrite semantics we use for annotated programs is built on top of the transition relation $\longrightarrow_{\text{JVM}}$ of section 4.2 and is shown on table 4.4. The semantics uses extended configurations that are quintuples of the form $(\psi, C, \sigma, \Sigma)$ such that ψ is the sequence of annotations remaining to be evaluated for the current program point of C, and σ is the ghost environment introduced above, mapping ghost variables to values. Each ghost environment σ can be partitioned to the global and local ghost environments σ_l and σ_g where the domain of σ_g is the variables of the ghost state, which are declared and initialized in the beginning of $\langle\texttt{main}\rangle$ and the domain of σ_l is all other ghost variables that have been set values in the current method. Finally Σ is a sequence of local ghost (variable) environments. The top element of Σ is the local ghost environment that belongs to the caller of the current method. Each method call causes a new local ghost environment σ_l^0 to be created, which is defined as $[g_{\text{pc}} \mapsto 0]$. Note that local ghost variables are not allowed to occur in *Requires* or *Ensures* clauses, like it is the case for local program variables.

We overload M, pc, A, *Requires*, *Ensures*, to refer to the first, second, third, fourth, and fifth projections on configurations, respectively. *MethodRet* holds of a configuration if the program counter of the top frame points to a return instruction. *MethodCall* holds of a configuration if the program counter of the top frame points to a method invocation instruction, which resolves to an *application* method call. Notice that these predicates can not be satisfied simultaneously. The predicate *Unhandled* holds of a configuration if it has an exceptional frame on top of the frame stack, and Γ^* does not contain a handler for that exception in the current method. Finally, *Exc* holds of a configuration that has an exceptional frame on the top of the stack.

The condition $\mathit{Assert}(a, C, \sigma)$ in Rule (1) always returns true, and does not effect the execution. But as a side-effect causes the predicate argument a to be "asserted", e.g. to appear on some output channel. The asserted predicate is valid if $\|a \parallel (C, \sigma)\|$ returns *true*. Rules (2), (3) and (4) capture the ghost variable assignment semantics as described above. Rule (5) is for intra-procedural execution, and applies to exception handling steps, but not to exception raising steps, which are handled by rule (8). Rule (6) causes any assertions in *Ensures* to be asserted at times of method exit, which are method returns and exceptional exits. Note also that the values assigned to the local ghost variables σ_l by the current method are discarded as the method terminates, and instead the environment is updated to use the assignments σ_l' of the calling method. Similarly, if the current instruction is a method call to an application method and executes without raising exceptions, rule (7) causes all assertions in the *Requires* clause of the called method to be asserted. When a new method starts executing, the local ghost environment of the caller method are pushed to the stack and the ghost environment uses the environment σ_l^0. Finally, when the execution of an instruction raises an exception, no predicates are asserted, as captured by rule (8).

The initial extended configuration $(\psi_0, C_0, \sigma_0, \Sigma_0)$ of program T is as follows: ψ_0 is $\mathit{Requires}(\Gamma^*(\langle\texttt{main}\rangle)) \cdot A_{\langle\texttt{main}\rangle}[1]$, C_0 is the initial configuration of T, $\sigma_0 = \sigma_l^0 = [g_{\text{pc}} \mapsto 0]$, $\Sigma_0 = \epsilon$.

$$(1)\quad \frac{\mathit{Assert}(a, C, \sigma)}{\Gamma^* \vdash (a\psi, C, \sigma, \Sigma) \rightarrow (\psi, C, \sigma, \Sigma)}$$

$$(2)\quad \frac{\|a_1\|(C,\sigma) = \mathit{true}, \quad m > 0}{\begin{array}{c}\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma) \rightarrow \\ (\psi, C, \sigma[\overrightarrow{gs} \mapsto \| \overrightarrow{e_1} \| (C,\sigma)], \Sigma)\end{array}}$$

$$(3)\quad \frac{\|a_1\|(C,\sigma) \neq \mathit{true}, \quad m > 0}{\begin{array}{c}\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma) \rightarrow \\ ((\overrightarrow{gs} := a_2 \rightarrow \overrightarrow{e_2} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma)\end{array}}$$

$$(4)\quad \frac{\cdot}{\Gamma^* \vdash ((\overrightarrow{gs} := ())\psi, C, \sigma, \Sigma) \rightarrow (\psi, C, \sigma[\overrightarrow{gs} \mapsto \overrightarrow{\bot}], \Sigma)}$$

$$(5)\quad \frac{C \longrightarrow_{\mathrm{JVM}} C' \quad \neg(\mathit{MethodCall}(C) \vee \mathit{MethodRet}(C)) \wedge \neg \mathit{Exc}(C'))}{\Gamma^* \vdash (\epsilon, C, \sigma, \Sigma) \rightarrow (A(\Gamma^*(M(C')))(pc(C')), C', \sigma, \Sigma)}$$

$$(6)\quad \frac{C \longrightarrow_{\mathrm{JVM}} C', \quad \mathit{MethodRet}(C) \vee \mathit{Unhandled}(C)}{\begin{array}{c}\Gamma^* \vdash (\epsilon, C, \sigma_g \uplus \sigma_l, \sigma_l' \cdot \Sigma) \rightarrow \\ (\mathit{Ensures}(\Gamma^*(M(C))), C', \sigma_g \uplus \sigma_l', \Sigma)\end{array}}$$

$$(7)\quad \frac{C \longrightarrow_{\mathrm{JVM}} C', \quad \mathit{MethodCall}(C) \wedge \neg \mathit{Exc}(C')}{\begin{array}{c}\Gamma^* \vdash (\epsilon, C, \sigma_g \uplus \sigma_l, \Sigma) \rightarrow \\ (\mathit{Requires}(\Gamma^*(M(C'))) \cdot A_{M(C')}[1], C', \sigma_g \uplus \sigma_l^0, \sigma_l \cdot \Sigma)\end{array}}$$

$$(8)\quad \frac{C \longrightarrow_{\mathrm{JVM}} C' \quad \neg \mathit{Exc}(C) \wedge \mathit{Exc}(C')}{\Gamma^* \vdash (\epsilon, C, \sigma, \Sigma) \rightarrow (\epsilon, C', \sigma, \Sigma)}$$

Table 4.4: Operational Semantics of Annotated Programs

Definition 4.12 (Validity of an Annotated Program)**.** A program annotated according to the rules set up above is *valid* for the extended environment Γ^*, if all predicates asserted as a result of a Γ^*-derivation $(\psi_0, C_0, \sigma_0, \Sigma_0) \rightarrow \cdots \rightarrow (\psi_n, C_n, \sigma_n, \Sigma_n) \rightarrow \cdots$ are valid, where $(\psi_0, C_0, \sigma_0, \Sigma_0)$ is the initial extended configuration of the program.

4.7 Checking Validity

For a fully annotated program, checking validity can be reduced to the simpler problem of checking "local" validity by referring to the axiomatic semantics of instructions. Local validity can be checked by generating verification conditions for each instruction and for method entry and exit points, by using the corresponding pre- and post-conditions and checking that these verification conditions hold. As mentioned before, our logic is an adaptation of the logic of Bannwart and Müller [9, 10], which in turn is a specialization of the logic of Poetzsch-Heffter and Müller [96] to bytecode. In this section, we first introduce this logic briefly, noting the differences it has with the one presented in the previous section. Then we define a notion of local validity, the correctness of which is based on the results of Bannwart and Müller.

4.7.1 Bannwart-Müller Logic

The logic is for a bytecode language with object-oriented features such as classes and objects, inheritance, fields, and virtual method resolution, as well as unstructured control flow with conditional and unconditional jumps, which makes it suitable for our purposes. Instead of using triples for instruction specifications as in classic Hoare logic, programs are annotated by associating a single assertion with each instruction, interpreted as its precondition. For an instruction I, its precondition has to be established by all predecessors of I, which usually includes the instruction that precedes I in the program text as well as all instructions that jump to I. We also follow this approach for specifying instructions.

Bannwart and Müller allow different specifications to be attached to the method implementation and the method body. What is more, it is possible to pose a common pre- and post-condition to all methods that can be invoked using the same method invocation instruction. Properties of methods are expressed by Hoare triples of the form $\{P\}comp\{Q\}$, where P and Q are first-order formulae and *comp* is a method body, a method implementation, or a "virtual" method, explained in more detail below. The triple $\{P\}comp\{Q\}$ expresses the following refined partial correctness property: if the execution of *comp* starts in a state satisfying P, then (1) this computation terminates in a state in which Q holds, or (2) *comp* aborts due to errors beyond the semantics of the programming language (for instance, internal JVM errors), or (3) *comp* does not terminate.

In both our logic and this logic, individual instruction specifications can be combined at the level of method bodies. This is due to the following guarantees on the structure of Java bytecode: the instruction sequence constituting a method body is always entered at the first instruction and left after the last instruction[4] and all jumps are local within a method body. The body of the method $c.m$ is denoted with $body(c.m)$. A *method body specification* is then written as $\{P\}body(c.m)\{Q\}$,

[4]Note that methods in which return instructions occur earlier can be rewritten so to redirect all returns to the last return instruction in order to satisfy this condition.

where the precondition P is the precondition of the first instruction and the postcondition Q is the precondition of the return instruction. *Method implementation specifications* play a similar role to that of *Requires* and *Ensures* in our logic, except that the postcondition of a method implementation need not hold at an exceptional exit from the method. These are denoted with $\{P\}imp(c.m)\{Q\}$ for method $c.m$.

The proofs are constructed in this logic using rules that combine specifications on a lower level to infer specifications on a higher level and language independent rules. Here we only present a few rules from the version of this logic where exceptions not considered (i.e. [10]) in order to give an intuition to the user. The details of the logic, including reasoning on programs that contain exception handling can be found in [9]. A sequent in this proof system has the following form:

$$\Phi \vdash \{P\}\, comp\{Q\}$$

where Φ is a set of method specifications needed for dealing with recursive methods. The rule about method body specifications, which combines the assertions occurring in the method body as explained above is as follows:

$$\frac{\forall i \in \{|body(c.m)|\}.\ (\Phi \vdash \{A[i]\}Li : I_i)}{\Phi \vdash \{A[1]\}body(c.m)\{A[|body(c.m)|]\}}$$

To infer the method implementation specifications, the following rule is used:

$$\frac{\Phi, \{P\}imp(c.m)\{Q\} \vdash \{P \wedge \mathrm{r0} \neq \texttt{null}\}body(c.m)\{Q\}}{\Phi \vdash \{P\}imp(c.m)\{Q\}}$$

The assertion r0 $\neq$ `null` guarantees that, at the point where the method body starts executing, the address to the object the method is called on is stored in the local variable r0 and that it is not `null`. Notice that this is indeed the case, as captured by the first rule of figure 4.2 in the language semantics. The rule also shows how assertions such as $\{P\}imp(c.m)\{Q\}$ are added to the set of assumptions.

Specifications on virtual methods are meant to capture method specifications imposed by the specification of an `invokevirtual` instruction on any method that can be called as a result of the execution of this instruction. In order to prove the precondition P for the instruction `invokevirtual` $c.m$, it has to be proven that (i) $\{P'\}virtual(c.m)\{Q'\}$, i.e. the methods that can be called by this instruction satisfy their method specification, (ii) that P implies the precondition P' of the virtual method specification, with actual arguments substituted for the formal parameters, and that (iii) the postcondition Q' of the method specification implies the precondition of the instruction following `invokevirtual`. In turn, to be able to prove $\{P\}virtual(c.m)\{Q\}$, it has to be proven that the specification holds for each method that can be called. This is done through proving $\{P \wedge \mathrm{r0} : c'\}imp(c'.m)\{Q\}$ for the implementation of each method $c'.m$ where $c' <: c$ and where the assertion r0 $: c'$ guarantees that the method is called on an object of type c'.

Our method specifications and the related rules are simplifications of this logic. While Bannwart-Müller logic is both sound and complete [9] with respect to the lan-

guage presented above, we only aim at soundness for program specifications of a particular shape. We will introduce these specifications in detail in section 4.8.1, 4.8.2 and 4.9. Here we only note that when the program is fully annotated in our scheme, all methods (with the exception of ⟨`main`⟩ which is not to be called from inside the program) have the same specification. Furthermore, both method specifications (the pre- and post-condition of methods) and the pre- and post-condition of method invocation instructions mention the same (invariant) assertion. Finally, this invariant does not mention formal arguments. Our *Requires* and *Ensures* clauses correspond to method implementation pre- and post-conditions of Bannwart-Müller logic, respectively. Notice that method body specifications do not correspond to our *Requires* and *Ensures* clauses, as a method body pre-condition is asserted each time there is a jump to the first instruction from within the body, since it is identical to the precondition of the first instruction. Therefore the rules of Bannwart-Müller logic become superfluous. Our only extension to this logic is the use of ghost variables and ghost assignments. Ghost variables can be seen as regular variables which are not affected by program code. We treat ghost assignments the same way as program instructions as these are not boolean expressions.

4.7.2 Local Validity

In section 4.9, we describe how a program inlined for a policy with a simple inliner can be fully annotated so that the validity of the annotations implies adherence of the program to the policy. Consequently, the problem of policy adherence for an inlined program is reduced to checking local validity. In this section, we introduce a suitable notion of fully annotated programs and conditions for a fully annotated program to be locally valid.

A *fully annotated program* is, then, a program where a sequence of annotations γ are associated with each instruction and with the *Requires* clause, and where only a single assertion is associated with the *Ensures* clause; each instruction specification and *Requires* clause consists of a single boolean expression or an alternating sequence of ghost assignments and boolean expressions α, with the first and last elements being a boolean expression. This definition guarantees that each ghost assignment is preceded and succeeded by a boolean expression. The expression before a ghost assignment can then be used as its specification, like it is done for program instructions.

In the definition of local validity, we use the function $wp(M[L])$ for computing the local weakest precondition of an instruction, which is the weakest precondition of the instruction with respect to all its possible successors. This function is exemplified in table 4.5 and is adapted for JVM instructions from the weakest precondition function of Bannwart-Müller [10]. In the definition, the function $\mathit{shift}(A)$ denotes the substitution, for all i, of $\mathrm{s}[i]$ by $\mathrm{s}[i+1]$ in assertion A, while function $\mathit{unshift}(A)$ denotes the inverse function.

The notion of local validity is defined as expected, namely that (i) the method precondition implies the annotation associated with the first instruction, (ii) the

$M[L]$	$wp(M[L])$
`dup`	$\mathit{unshift}((\mathit{head}(A_M[L+1]))[\mathrm{s}[1]/\mathrm{s}[0]])$
`iload r` / `aload r`	$\mathit{unshift}((\mathit{head}(A_M[L+1]))[\mathtt{r}/\mathrm{s}[0]])$
`istore r` / `astore r`	$(\mathit{shift}(\mathit{head}(A_M[L+1])))[\mathrm{s}[0]/\mathtt{r}]$
`putstatic m`	$(\mathit{shift}(\mathit{head}(A_M[L+1])))[\mathrm{s}[0]/\mathtt{m}]$
`goto` L'	$\mathit{head}(A_M[L'])$
`ifeq` L'	$(\mathrm{s}[0]=0 \Rightarrow \mathit{shift}(\mathit{head}(A_M[L'])))\wedge$ $(\neg(\mathrm{s}[0]=0) \Rightarrow \mathit{shift}(\mathit{head}(A_M[L+1])))$
`instanceof` c	$\mathrm{s}[0] <: c \Rightarrow (\mathit{head}(A_M[L+1]))[1/\mathrm{s}[0]]\wedge$ $\neg(\mathrm{s}[0] <: c) \Rightarrow (\mathit{head}(A_M[L+1]))[0/\mathrm{s}[0]]$

Table 4.5: Weakest precondition function $wp(M[L])$

precondition of the return instruction implies the method post-condition, (iii) the pre-condition of an instruction implies the weakest precondition of the instruction provided it is not a method invocation, (iv) the last assertion before a ghost assignment implies the first assertion after the ghost assignment where the ghost values are replaced with the conditional expression of the assignment, (v) the pre-condition of an instruction implies the pre-condition of any handler that covers the instruction and it implies the post-condition of the method if it can raise an exception not covered by any handler of the method, (vi) for all method invocation instructions L, there exists an assertion α such that the pre-condition of the instruction implies the conjunction of the pre-condition of any method, which can be called by the instruction and α, while the conjunction of α and the post-condition of any method, which can be called by the instruction, imply the pre-condition of $L+1$, furthermore if a method that can be called by this instruction raises exceptions, then the post-condition of the called method implies the pre-condition of the corresponding handler if any, or implies the postcondition of the caller method, (vii) finally, the initializations to the static variables done by the initial static heap is sufficient to make the pre-condition of $\langle\mathtt{main}\rangle$ valid.

In the definition below, the function *head* returns the first element of an annotation sequence and *last*, the last element. $\mathit{Statics}_{\mathrm{T}}$ denotes the set of static variables $c.f$ of T and for all $c.f \in \mathit{Statics}_{\mathrm{T}}$, $v_{c.f}$ is equal to $sh_0^{\mathrm{T}}(c.f)$. We let $\models$ denote standard first-order logic validity.

Definition 4.13 (Local Validity)**.** A fully annotated program T is *locally valid* if for every virtual method $M = (P, H, A, \mathit{Requires}, \mathit{Ensures})$ the following holds:

(i) $\models \mathit{last}(\mathit{Requires}) \Rightarrow \mathit{head}(A[1])$,

(ii) $\models \mathit{last}(A[|P|]) \Rightarrow \mathit{Ensures}$,

(iii) for all $L \in \mathit{Dom}(P)$ where $M[L]$ is not a method invocation instruction: $\mathit{last}(A[L]) \Rightarrow wp(M[L])$,

(iv) whenever $\gamma \cdot \alpha \cdot (\overrightarrow{g} := ce) \cdot \alpha' \cdot \gamma'$ is an instruction specification,

$$\models \alpha \Rightarrow \alpha'[ce/\overrightarrow{g}]$$

(v) for all $L \in Dom(P)$, if $M[L]$ can raise an exception with type c, one of the following holds:

a) There exists a handler (L_1, L_2, L', c') that handles this exception, and

$$\models last(A[L]) \Rightarrow head(A[L'])$$

b) There does not exist a handler for label L and exception c, and

$$\models last(A[L]) \Rightarrow Ensures$$

(vi) for a label $L \in Dom(P)$ where $M[L] =$ `invokevirtual` $c.m$, and for all methods $c'.m$ that can be invoked as a result of the execution of this instruction and virtual method resolution, let $c'.m = (P', H', A', Requires', Ensures')$ and $c'.m$ be of arity n. Then there exists the assertion α which mentions only local and (local) ghost variables, such that:

- $\models last(A[L]) \Rightarrow (head(Requires') \wedge \alpha)$,
- $\models \alpha \wedge Ensures' \Rightarrow head(A[L+1])$,
- If $c'.m$ raises an exception then either there exists a handler with destination L' and $\models Ensures' \Rightarrow head(A[L'])$ or there is no such handler and $\models Ensures' \Rightarrow Ensures$.

(vii)

$$\models \bigwedge_{c.f \in Statics_{\mathrm{T}}} c.f = v_{c.f} \;\Rightarrow\; head(Requires_{\langle \texttt{main} \rangle})$$

The assertion α mentioned in item (vi) is used for assertions that are preserved by the method call, that is assertions on local program and ghost variables. Note that this notion of local validity includes recursive methods, since we do not require that the called method be a different method from the caller method. This, in effect, means that the method specification can be assumed at the point of the recursive call, which is in line with [10].

Showing that a locally valid program is valid also in the sense of definition 4.12 can be done based on the soundness result[5] of [10], detailed and extended to exceptions in [9]. Such a proof consists of extending Bannwart-Müller logic with a rule

[5]Their soundness result states that whenever $\vdash \{P\} body(c.m) \{Q\}$, it is the case that for all initial configurations C that $c.m$ can start running with (i.e. configurations with the right number of values for arguments are on the stack etc.) and all configurations C', if C' is a terminating configuration with the current method $c.m$, C is related to C' with the reflexive, transitive closure of the transition relation of the operational semantics and P holds at C, then Q holds at C'. In order to prove this result, they prove a similar result for single steps of the machine.

for ghost assignments, extending the soundness proof they present with this rule and showing that a proof tree can be constructed to infer $\{Requires_M\}imp(M)\{Ensures_M\}$ for each method M of the program in this logic. The proof construction is straightforward as our local validity definition is simply an application of the proof rules in a restricted setting. A rule for handling ghost variables is not hard to develop either, guided by the fact that ghost assignments are very much like ordinary assignment statements that do not alter the machine configuration, and would resemble rule (iv) above.

4.8 Specification of Self-monitoring

The idea behind our annotation scheme is the following. In a first annotation, referred to as *policy annotation* (or level I), we specify a correct monitor for the given policy by means of "ghost" variables, updated before or after every security relevant action according to the symbolic automaton induced by the given security policy. In a second annotation, referred to as *synchronisation annotation* (or level II), we add assertions that state at all relevant program points that the actual inlined monitor (represented by global program variables) is "in sync" with the correct monitor (represented by the ghost variables).

4.8.1 Policy Annotations (Level I)

The *policy annotations* define a monitor for the given policy by means of ghost variables. The ghost variables, which constitute the *specified security state*, are initialized in the precondition of the ⟨`main`⟩ method and updated at relevant points by annotating all the methods defined by the classes of the target program. We call each such method a *target method*.

When adding the level I annotations, we make the following assumptions. We assume that ⟨`main`⟩ is not called by any target method (including itself) and that all exceptions that may be raised by a security relevant instruction (i.e. an instruction that may lead to a security relevant action) are covered by an exception handler. We also assume that the exception handling is structured such that unexceptional execution can not "fall through" to an exception handler, i.e. the only way an instruction in an exception handler gets executed is when an exception has been raised previously in the execution and caught by the handler that the instruction belongs to. We assume that the first instruction of a program can not be a handler instruction.

Updating the Specified Security State The updates to the specified security state are done according to the transitions of the symbolic automaton. If the automaton does not have a transition for a security relevant method call, the call is violating and the corresponding annotation sets the value of the specified state to undefined. Such a program should terminate without executing the next security

relevant action in order to adhere to the policy. This is specified by asserting, as a precondition to each security relevant method invocation and before each update to the specified state, that the specified state is not undefined.

If the execution of a method invocation instruction of a target method may lead to a preaction of the automaton, then an annotation is inserted as a precondition to this instruction, which updates the specified security state. If a method invocation instruction may lead to a postaction, we record the object the method is called on, values of the method arguments (and possibly a part of the heap) by assigning them to ghost variables as the precondition to the instruction. The updates to the specified state are done in the postcondition of the instruction, if the method invocation can lead to a normal (unexceptional) postaction. If the instruction can cause an exceptional postaction, however, the update to the specified security state is inserted as a precondition to the first instruction of each exception handler that cover the instruction. The recorded label is used then at the handler to resolve which instruction has caused the exception, so that the correct update (or no update if the exception was raised by an irrelevant instruction) is performed.

Preliminary Definitions In the definitions below, assume given a program T and a policy $\mathcal{P}$. Let $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$ be the symbolic automaton induced by $\mathcal{P}$, and let $q_s = \{s_1, \ldots, s_n\}$. We define the set $A_s^e \subseteq A_s^\sharp$ of exceptional symbolic post-actions as those post-actions which have the value exc as their first component. Given a symbolic action set A'_s, the function $RS((c, m), A'_s)$ returns those subclasses c' of c for which the method (c', m) is defined by a class c'' such that A'_s has an action with the reference (c'', m). In the annotations, the ghost variables that represent the security state are named identically with the security state variables of the automaton, and we use the tuple $\overrightarrow{gs} = (s_1, \ldots, s_n)$ in guarded multi-assignments. We use the ghost variable g_{pc} to record labels of security relevant instructions. Ghost variables g also used for recording stack values. For an expression mapping $E : q_s \to Exp$, let $\overrightarrow{e_E}$ denote the corresponding expression tuple and for a boolean ConSpec expression $b \in BoolExp$, let a_b denote the corresponding assertion.

Level I Annotation Further below, we define an initializing ghost annotation IA, and for every method M, three arrays of annotations: a pre-annotation array $A_M^\flat[i]$, a post-annotation array $A_M^\sharp[i][j]$, and an exceptional annotation array $A_M^e[i][k]$, where i ranges over the instructions of method M. The second index $j \in \{0, 1\}, k \in \{0, 1, 2\}$ indicates whether the annotation will be placed as a precondition of the instruction $(j, k = 0)$, as a precondition to the next instruction $(j, k = 1)$, or as a precondition to all the exception handlers of the instruction $(k = 2)$. The predicate *Handler* holds for a label L and a method M if L is a destination of some exception handler, i.e. $(L_1, L_2, L, c) \in H_M$ for some labels L_1, L_2, and class name c. In addition, we define $Exc(L, M)$ as the sequence of all annotations $A_M^e[L'][2]$ where L' is a security relevant instruction and there exists an exception handler $(L_1, L_2, L, c) \in H_M$ such that $L_1 \leq L' < L_2$, and as ϵ if such an

L' does not exist.

Given these annotations, the *level I annotation* of program T is given for each application method M as a precondition $Requires^{\mathrm{I}}_M$ and an array A^{I}_M of annotation sequences defined as follows (where $L > 0$):

$$Requires^{\mathrm{I}}_M = \begin{cases} (\overrightarrow{gs} := \overrightarrow{e_{Init_s}}) & \text{if } M = \langle \texttt{main} \rangle \\ \epsilon & \text{otherwise.} \end{cases}$$

$$\begin{aligned} A^{\mathrm{I}}_M[1] &= A^{\flat}_M[1] \cdot A^{\sharp}_M[1][0] \cdot A^{e}_M[1][0] \\ A^{\mathrm{I}}_M[L] &= \begin{cases} Exc(L,M) \cdot A^{\flat}_M[L] \cdot A^{\sharp}_M[L][0] \cdot A^{e}_M[L][0] & \text{if } Handler(L,M) \\ A^{e}_M[L-1][1] \cdot A^{\sharp}_M[L-1][1] \cdot A^{\flat}_M[L] \cdot A^{\sharp}_M[L][0] \cdot A^{e}_M[L][0] & \text{otherwise} \end{cases} \end{aligned}$$

The annotation $Requires_{\langle \texttt{main} \rangle}$ initializes the ghost state using function $Init_s$ of the automaton. The following paragraphs define the annotation arrays mentioned in the above definition.

After Annotations For every method M, the elements of the post-annotation array $A^{\sharp}_M[i][j]$ are defined for each label L as follows:

(i) If the instruction at label L is not an `invokevirtual` instruction or is of the form $M[L] = \texttt{invokevirtual}\ (c.m)$ where $RS((c,m), A^{\sharp}_s \setminus A^{e}_s) = \varnothing$, we define the pre- and postconditions to be empty:

$$A^{\sharp}_M[L][0] = A^{\sharp}_M[L][1] = \epsilon$$

(ii) Otherwise, if the instruction at label L is of the form $M[L] = \texttt{invokevirtual}\ (c.m)$ with $(c.m) : (\gamma \to \tau)$ and $|\gamma| = n$ and $RS((c,m), A^{\sharp}_s \setminus A^{e}_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction saves the arguments and the object in ghost variables:

$$A^{\sharp}_M[L][0] = ((g_0, \ldots, g_{n-1}, g_{\mathrm{this}}) \ := \ (\mathrm{s}[0], \ldots, \mathrm{s}[n])) \cdot \mathit{Defined}^{\sharp}$$

The assertion $\mathit{Defined}^{\sharp}$ checks if the ghost variables are defined:

$$\mathit{Defined}^{\sharp} = (g_{\mathrm{this}} : c'_1 \vee \ldots \vee g_{\mathrm{this}} : c'_p) \Rightarrow (\overrightarrow{gs} \neq \overrightarrow{\bot})$$

while the postcondition of the instruction uses these saved values to compute the new security state:

$$A^{\sharp}_M[L][1] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha)$$

where the α_k are the guarded expressions

$$(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge g_{\mathrm{this}} : c'_i \wedge a_b \rho_i \ \rightarrow \ \overrightarrow{e_E} \rho_i$$

$A^I[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\left\{ \begin{array}{l} g_{\text{this}} := \text{s}[0] \cdot \\ g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \end{array} \right\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\left\{ \begin{array}{l} (g_{\text{a}}, g_{\text{p}}) := \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \text{s}[0] \rightarrow (g_{\text{a}}, \mathit{true}) \quad \mid \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s}[0] \rightarrow (g_{\text{a}}, \mathit{false}) \quad \mid \\ \neg(g_{\text{this}} : \texttt{GUI}) \rightarrow (g_{\text{a}}, g_{\text{p}}) \end{array} \right\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure 4.4: An application method with level I annotations for the example policy

where class c'' defines (c'_i, m) and there exists $a^\sharp_s = (r, c'', m, (\tau_0 x_0, \ldots \tau_{n-1}\ x_{n-1})) \in A^\sharp_s \setminus A^e_s$ such that $(a^\sharp_s, b, E) \in \delta^\sharp_s$. The substitution ρ_i is defined as $[s[0]/x,\ g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{\text{this}}/\texttt{this}]$ if $r = (\tau\ x)$ and as $[g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{\text{this}}/\texttt{this}]$ if $r = \mathit{void}$. Finally, $\alpha = \neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \rightarrow \overrightarrow{gs}$.

Example 4.14. An application method annotated with level I annotations for the example policy 4.3 is shown in figure 4.4. The ghost state is represented by the ghost variables g_{a} and g_{p}, i.e. $\overrightarrow{gs} = (g_{\text{a}}, g_{\text{p}})$. (The setting of the ghost variable g_{pc} is ignored since the policy does not include an exceptional clause.) The annotations are valid if the class `GUI` does not have any subclasses. The annotations are identical as long as all subclasses of this class overrides `AskConnect`.

Let us suppose `GUI` has a single subclass `MyGUI` and that it inherits `AskConnect`. Then the annotated program is as in figure 4.5.

The annotation array $A^\flat_M$ is defined similar to $A^\sharp_M$ except that the transitions of the automaton on pre-actions are considered. The values of the arguments of the security relevant instruction can be obtained by accessing the stack directly, so the argument names in the guards and update expressions of the symbolic automaton should be substituted with corresponding stack positions in this case.

The exceptional annotation array $A^e_M[i][k]$ is defined considering transitions of the automaton on exceptional post-actions (the set A^e). The precondition $A^e_M[L][0]$ of an instruction $M[L]$ that may cause an exceptional post-action saves its label L in the ghost variable g_{pc}, in addition to recording the object and arguments to the method. The conditions of the ghost variable update placed in the precondition to the corresponding handler $A^e_M[L][2]$, then include the conjunct $g_{\text{pc}} = L$ to check that the exception was indeed raised by this instruction. The ghost variable g_{pc} is

$A^I[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\left\{ \begin{array}{l} g_{\text{this}} := \text{s}[0] \cdot \\ (g_{\text{this}} : \texttt{GUI} \vee g_{\text{this}} : \texttt{MyGUI}) \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \end{array} \right\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\left\{ \begin{array}{l} (g_{\text{a}}, g_{\text{p}}) := \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \text{s}[0] \rightarrow (g_{\text{a}}, \mathit{true}) \quad \mid \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s}[0] \rightarrow (g_{\text{a}}, \mathit{false}) \quad \mid \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{MyGUI} \wedge \text{s}[0] \rightarrow (g_{\text{a}}, \mathit{true}) \quad \mid \\ (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{MyGUI} \wedge \neg\text{s}[0] \rightarrow (g_{\text{a}}, \mathit{false}) \quad \mid \\ \neg(g_{\text{this}} : \texttt{GUI} \vee g_{\text{this}} : \texttt{MyGUI}) \rightarrow (g_{\text{a}}, g_{\text{p}}) \end{array} \right\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure 4.5: A application method with level I annotations for the example policy

set back to 0 after the update in the annotation (i.e. in $A^e_M[L][2]$) or if the method never raises an exception (i.e. in $A^e_M[L][1]$).

The formalizations are presented here for completeness.

Before Annotations For every method M, the elements of the pre-annotation array $A^\flat_M[i]$ are defined for each label L as follows:

(i) If the instruction at label L is not an `invokevirtual` instruction or is of the form $M[L] =$ `invokevirtual` $(c.m)$ where $RS((c,m), A^\flat_s) = \varnothing$, we define the precondition to be empty: $A^\flat_M[L] = \epsilon$.

(ii) Otherwise, if the instruction at label L is of the form $M[L] =$ `invokevirtual` $(c.m)$ with $(c.m) : (\gamma \rightarrow \tau)$ and $|\gamma| = n$ and $RS((c,m), A^\flat_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction computes the new security state using the arguments and the object of the called method and updates the ghost variables:

$$A^\flat_M[L] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha) \cdot \mathit{Defined}^\flat$$

The assertion $\mathit{Defined}^\flat$ checks if the ghost variables are defined:

$$\mathit{Defined}^\flat = (\text{s}[n] : c'_1 \vee \ldots \vee \text{s}[n] : c'_p) \Rightarrow (\overrightarrow{gs} \neq \overrightarrow{\bot})$$

The α_k are the guarded expressions

$$(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge s[n] : c'_i \wedge \ a_b\rho_i \ \rightarrow \ \overrightarrow{e_E}\rho_i$$

where class c'' defines (c'_i, m) and there exists $a^\flat_s = (c'', m, (\tau_0 x_0, \ldots \tau_{n-1} x_{n-1}))$ $\in A^\flat_s$ such that $(a^\flat_s, b, E) \in \delta^\flat_s$. The substitution ρ_i is defined as $[s[0]/x_0, \ldots,$ $s[n-1]/x_{n-1}, s[n]/\texttt{this}]$. Finally, $\alpha = \neg(\text{s}[n] : c'_1 \vee \ldots \vee \text{s}[n] : c'_p) \ \rightarrow \ \overrightarrow{gs}$.

Exceptional Annotations For every method M, the elements of the exceptional annotation array $A^e_M[i][j]$ are defined for each label L as follows:

(i) If the instruction at label L is not an `invokevirtual` instruction or is of the form $M[L] =$ `invokevirtual` $(c.m)$ where $RS((c,m), A^e_s) = \varnothing$, we define the pre- and post-conditions to be empty: $A^e_M[L][0] = A^e_M[L][1] = A^e_M[L][2] = \epsilon$.

(ii) Otherwise, if the instruction at label L is of the form $M[L] =$ `invokevirtual` $(c.m)$ with $(c.m) : (\gamma \to \tau)$ and $|\gamma| = n$ and $RS((c,m), A^e_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction saves the arguments, the object and the label of the instruction in ghost variables:

$$A^e_M[L][0] = ((g_0, \ldots, g_{n-1}, g_{\text{this}}, g_{\text{pc}}) \;:=\; (\text{s}[0], \ldots, \text{s}[n]), L) \cdot \mathit{Defined}^e$$

The assertion $\mathit{Defined}^e$ checks if the ghost variables are defined:

$$\mathit{Defined}^e = (g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \Rightarrow (\overrightarrow{gs} \neq \overrightarrow{\perp})$$

The postcondition of the instruction resets the value of g_{pc} to 0. Notice that this annotation gets executed only if the method invocation did not return with an exception.

$$A^e_M[L][1] = g_{\text{pc}} := 0$$

The precondition of each handler that covers this instruction uses g_{pc} to check whether the exception catched was thrown by a security relevant instruction. If the exception was raised by a method called by the instruction with the relevant label, the annotation uses the saved values to compute the new security state:

$$A^e_M[L][2] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha) \cdot (g_{\text{pc}} := 0)$$

where the α_k are the guarded expressions

$$(g_{\text{pc}} = L) \wedge (\overrightarrow{gs} \neq \overrightarrow{\perp}) \wedge g_{\text{this}} : c'_i \wedge \; a_b\rho_i \;\to\; \overrightarrow{e_E}\rho_i$$

where class c'' defines (c'_i, m) and there exists $a^e_s = (\text{exc}, c'', m, (\tau_0 x_0, \ldots \tau_{n-1} x_{n-1})) \in A^e_s$ such that $(a^e_s, b, E) \in \delta^e_s$. The substitution ρ_i is defined as $[g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{this}/\texttt{this}]$. Finally, $\alpha = \neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \to \overrightarrow{gs}$.

Correctness Each execution of a program that is valid with respect to level I annotations for policy $\mathcal{P}$ is a co-execution of the program and the automaton for $\mathcal{P}$, where the automaton states are given by the ghost state; hence the program adheres to $\mathcal{P}$.

Theorem 4.15 (Correctness of Level I Annotations)**.** *The level I annotated program* T *for policy* $\mathcal{P}$ *is valid if, and only if,* T *adheres to* $\mathcal{P}$*.*

Proof. The proof can be found at Appendix B.3.2. □

4.8.2 Synchronisation Annotations (Level II)

An inlined program can be expected to contain an explicit representation of the security state, an *embedded state*, which is updated in synchrony with the execution of security relevant actions. The level II annotations aim to capture this idea in a generic enough form that it is independent of many design choices a specific inliner may make. In particular, it seems natural to require of an inlined monitor that it maintains agreement between the ghost state and the embedded state immediately prior to execution of a security relevant action. That is, program and monitor state are both tested and, where necessary, updated whenever a security relevant action is about to be performed. This is by no means a necessary condition: For instance, a monitor implementation may in advance determine that some fixed sequence of security relevant actions is permissible without necessarily reflecting this through an explicit sequence of updates to the embedded state. Thus, in the middle of such a sequence, the embedded state and the ghost state may disagree. In this paper, however, we assume that this type of optimized inlining is not performed.

The second assumption we make in this section is that updates to the embedded state are made locally, by the same method that executes the security relevant method call. This allows correctness to be expressed by asserting equality of the ghost state and the embedded state for every method at point of entry, at normal and exceptional exit, and at each method invocation. This compositionality property has the important advantage that virtual call resolution can be avoided for the level II annotations: The specified and the embedded states are synchronized at all call points, not just at the points where a security relevant action is invoked.

For simplicity we assume that the embedded state is determined as a fixed vector $\overrightarrow{ms}$ of global static variables of the target program, of types corresponding pointwise to the type of ghost state vector $\overrightarrow{gs}$. The *synchronisation assertion* is the equality $\overrightarrow{gs} = \overrightarrow{ms}$, and the *level II annotations* are formed by appending the synchronization assertion to the level I annotations of each method M of the target program at the following points:

1. Each annotation $A(\Gamma^*(M))(i)$ such that $P(\Gamma^*(M))(i)$ is an invoke or a return instruction.

2. The annotation $\mathit{Ensures}(\Gamma^*(M))$.

We explain a sense in which the level II annotations can be argued to characterize exactly the two conditions assumed in this section (the synchronous update assumption, and the method-local update assumption).

Consider a level II annotated program T with the extended environment Γ^*. Consider an execution $E = C_0C_1\cdots$ from the initial configuration C_0 of T. We sample the embedded state $\overrightarrow{ms}$ at all configurations that are either invoke instructions, return instructions, the first instruction of a method, or an unhandled exception. More precisely, the index i is a *sampling point* if one of the following two conditions hold:

$A^{II}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$g_{\text{this}} := \text{s}[0]\ \cdot$ $g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)\ \cdot$ $(g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})$	L5	`invokevirtual GUI/AskConnect()Z`
$(g_{\text{a}}, g_{\text{p}}) :=$ $(g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \text{s}[0] \rightarrow (g_{\text{a}}, \mathit{true})$ | $(g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s}[0] \rightarrow (g_{\text{a}}, \mathit{false})$ | $\neg(g_{\text{this}} : \texttt{GUI}) \rightarrow (g_{\text{a}}, g_{\text{p}})$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
$(g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})$	L13	`ireturn`

Figure 4.6: An application method with level II annotations for the example policy

1. The top frame of C_i has the shape (M, pc, s, f), and $M[pc]$ is either an `invokevirtual` instruction, a return instruction, or else $pc = 1$.

2. Alternatively, the top frame of C_i is exceptional, of the form $(b)_e$.

We can then construct a sequence $w = (C_0, q_0)(C_1, q_1)\cdots$ (or, $w(E, \overrightarrow{ms})$ if the underlying execution and embedded state needs emphasis) such that:

- q_0 is the initial automaton state,
- for all sampling points $i > 0$, $q_i = C_i(\overrightarrow{ms})$, the value of $\overrightarrow{ms}$ in configuration C_i, and
- for any two consecutive sampling points i and i', for all $j : i \leq j < i'$, $q_j = q_i$.

In other words, the embedded state is sampled at the sampling points and maintained constant in between.

Example 4.16. An application method annotated with level II annotations for the example policy of section 4.4 is shown in figure 4.6. This is an augmented version of figure 4.4, where the embedded state consists of the static fields `accessed` and `permission` of the `SecState` class. The *Ensures* clause is the synchronization assertion

$$(g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})$$

and *Requires* $= \epsilon$. The annotated method is valid since the embedded state is updated as is described by the policy, after a call to the method `GUI.AskConnect`.

The role of the sequence w is roughly similar to the role of interleavings in section 4.5. However, a slightly different treatment is needed here since the sequence $q_0 q_1 \cdots$ may not necessarily correspond to an automaton derivation. This is so for the case of post-actions followed by pre-actions where the intermediate automaton state is not sampled, as there is no well-defined point where this might be done. Also, the construction needs to account for the method-local nature of embedded state updates.

For this reason define the operation $extract_{\mathrm{II}}$ taking sequences w to strings over the alphabet $Q \cup A \cup \{\texttt{brk}\}$ where `brk` is a distinguished symbol by the following conditions:

- $extract_{\mathrm{II}}((C_1, q_1)(C_2, q_2)w) = q_1\ act^\flat(C_1)\ act^\sharp(C_1, C_2) q_2\ extract_{\mathrm{II}}((C_2, q_2)w)$, if C_1 is an API method call.
- $extract_{\mathrm{II}}((C, q)w) = \texttt{brk}q\ extract_{\mathrm{II}}(w)$, if C is entering to or returning (possibly exceptionally) from an application method call.
- $extract_{\mathrm{II}}((C, q)) = q\ act^\flat(C)$
- $extract_{\mathrm{II}}(\epsilon) = \epsilon$

Definition 4.17 (Method-local Co-execution)**.** Let $\Sigma_1 = \{\texttt{brk}\} \cup Q \cup E^\flat \cup E^\sharp \cup \{a^\sharp q q' a^\flat \mid q, q' \in Q, a^\flat \in A^\flat, a^\sharp \in A^\sharp, \exists q''. \delta^\flat(q, a^\flat) = q'', \delta^\sharp(q'', a^\sharp) = q'\}$, and $\Sigma_2(q) = \{q\texttt{brk}q\} \cup \{qq\} \cup \{qaq \mid a \in A\} \cup \{(qa^\flat a^\sharp q) \mid a^\flat \in A^\flat, a^\sharp \in A^\sharp\}$. A sequence w is a method-local co-execution, if $extract_{\mathrm{II}}(w) \in (\Sigma_1^* \cup \Sigma_1^\omega) \cap ((\bigcup_{q \in Q} \Sigma_2(q))^* \cup (\bigcup_{q \in Q} \Sigma_2(q))^\omega)$.

We can then extend theorem 4.15 to the situation where a target program T has a monitor for the given policy inlined into it.

Theorem 4.18 (Level II Characterization)**.** *The level II annotation of* T *with embedded state* $\overrightarrow{ms}$ *is valid if, and only if, for each execution* E *of* T*, the sequence* $w(E, \overrightarrow{ms})$ *is a method-local co-execution.*

The idea of the proof is to sample pre- and post-actions from E, immediately preceded and followed by a sample of the embedded state $\overrightarrow{ms}$. The sequence extracted in this way is almost a potential derivation, but in the case of a post-action followed, some time later, by a pre-action, an intermediate automaton state may be missing. It is not clear, however, how to sample this state. Also, it is necessary to ensure that embedded state updates do not cross method boundaries. To this end, extracted sequences need to be completed by (a) missing intermediate automaton states, and (b) indicators of method boundary crossings at: method invocations that are not security relevant actions, return instructions, configurations that contain an exceptional frame at the top of the frame stack, and at the first instruction of each method.

First, we note that the embedded state $\overrightarrow{ms}$ is equal to the ghost state $\overrightarrow{gs}$ at sampling points if and only if the synchronisation assertions added at level II hold. We show in the proof of theorem 4.15 that the ghost state and machine configurations constitute a co-execution if and only if the program annotated with level I annotations is valid. If the program annotated with level II annotations is valid then the sampling of the embedded stated as described above amounts to taking the co-execution of the ghost state and the program and "skipping" some ghost updates, which the embedded state does not follow (as the sampling of the embedded state is not done as frequently). Then $extract_{\mathrm{II}}$ applied to this sequence falls in the set stated in definition 4.17.

4.9 Correctness of Inlining

As an application of the annotation scheme described in the previous section, we characterize the correctness of a class of inliners in the flavor of PoET/PSLang [40, 39]. We first describe the operation of a simple inliner that embeds, in target programs, a method-local monitor for a ConSpec policy. Then we show how level II annotations for these inlined programs can be efficiently completed to produce fully annotated code, thus reducing the policy adherence problem to checking the validity of these annotations.

4.9.1 A Simple Inliner

The inliner inputs a ConSpec policy and a program, and inserts code for (i) storing the security state and (ii) for updating it according to the policy clauses at calls to security relevant methods.

Storing the Security State The inliner adds a single class definition to the program. The class stores the embedded state in its static fields. Since this new class is not in the previous name space, the embedded state is safe from interference by the target program. Here, we assume that `SecState` is a fresh name for the target program, and use this name for referring to the class storing the embedded state.

Compiling Policy Body to Bytecode The first part of the transformation compiles the policy to bytecode, and is independent of the method(s) for which inlining is to be performed. For each clause in the policy, a code fragment is produced. These fragments are inlined in application methods in the rewriting stage.

Each clause of a policy consists of an event modifier, an event specification and a list of guarded commands. The created code evaluates, in turn, the guards and

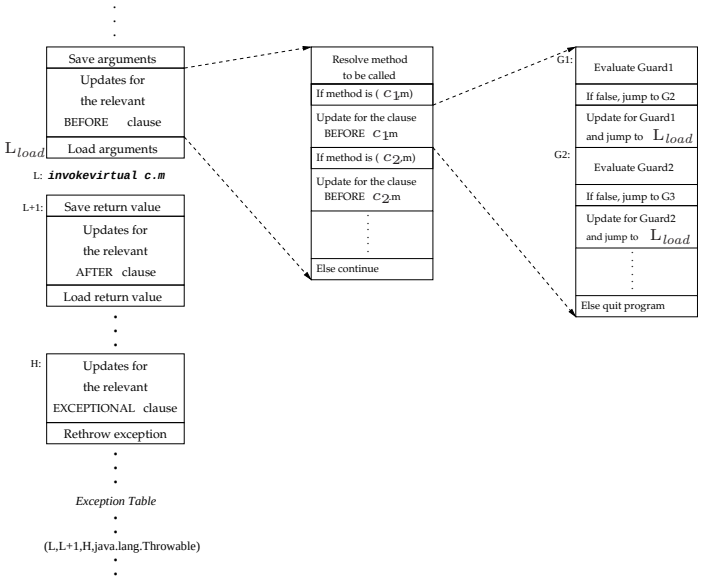


Figure 4.7: Inlining of an Instruction

either updates the security state according to the update block associated with the first condition that holds or quits the program if none of them hold[6].

The variables that occur in an event clause are of three kinds: security state variables, method arguments and fields of method arguments. Security state variables are stored in the `SecState` class. Since the fields storing the embedded state are static, they are created and initialized as soon as the class is loaded to the JVM. Actual arguments of a method (including the reference to the object it operates on) reside on top of the stack immediately before the method invocation. In order to use argument values while computing the new values of the security state variables, the arguments are copied from the stack to local variables that are not used by the original program. Since local variables of the calling method are not affected by the execution of the callee, argument values stored in this manner can be accessed even after the control returns to the calling method. Finally, fields of arguments are accessed using the arguments and the heap. Notice that ConSpec policies specify only updates to security state variables. The compiled code uses fresh variables for both the security state and storing arguments. Therefore inlining does not modify the original program behavior beyond forcing its exit upon violation.

Rewriting Methods According to Policy The rewriting process consists of identifying method invocation instructions that lead to security relevant actions (se-

[6]In order to abort the program, the method `System.exit()` can be used in standard API's, so that a security violation is distinguished from a normal return.

	Method before inlining	Method after inlining
L1	`aload r0`	`aload r0`
L2	`getfield gui`	`getfield gui`
L3	`invokevirtual GUI/AskConnect()Z`	`dup`
L4	`ireturn`	`astore r1`
L5		`invokevirtual GUI/AskConnect()Z`
L6		`istore r2`
L7		`aload r1`
L8		`instanceof GUI`
L9		`ifeq L12`
L10		`iload r2`
L11		`putstatic SecState/permission`
L12		`iload r2`
L13		`ireturn`

Figure 4.8: An application method inlined with the example policy

curity relevant instructions), and for each such instruction, inserting code produced by policy compilation in an appropriate manner. The inlined code is depicted for a single instruction in figure 4.7. The inliner inserts, immediately before the security relevant instruction, code that records the object the method is called for, and the arguments (and possibly parts of the heap) in local variables. Then, code for the relevant BEFORE clauses of the policy (if any) is inserted. Next, the object and the method arguments are restored on the stack. If there are AFTER clauses in the policy for the instruction, first the return value (if there is any) is recorded in a local variable, the code compiled from the AFTER clauses is inlined, followed by code to restore the return value on the stack. Finally, if there are EXCEPTIONAL clauses for the instruction, an exception handler is created that covers only the method invocation instruction and catches all types of exceptions. It is placed highest amongst the handlers for this label in the handler list, so that whenever the instruction throws an exception, this handler will be executed. The code of this exception handler consists of code created for the related EXCEPTIONAL clauses and ends by rethrowing the caught exception. All (original) exception handlers of the program that cover the security relevant instruction are redirected to cover this last throw instruction instead.

Method Resolution Due to virtual method call resolution, execution of an invocation instruction can give rise to different security relevant actions. The inliner inserts code to resolve, at runtime, the signature of the method that is called, using the type of the object that the method is invoked on, and information on which methods have been overridden. A check to compare this signature against the signature of the event mentioned in the clause is prepended to code compiled for the clause[7].

We have implemented such an inliner to be used in a mobile code context; the tool is available at [2].

[7] This can be accomplished using the `instanceof` instruction or methods of the Reflect API.

Example 4.19. Figure 4.8 shows an inlined method example. The original method is given on the left, while the method inlined for the example policy is presented on the right. The method is inlined assuming that the class `GUI` does not have any subclasses. Notice that the inlined method is the one we have been employing in the annotation examples in sections 4.8.1 and 4.8.2.

4.9.2 Correctness of Inlining

We first describe how, for programs inlined with an inliner such as the one described above, level II annotations can be efficiently completed to an (equivalent full) level III annotation in the sense of section 4.7, and then show that local validity of level III annotations – and thus policy adherence – holds for such programs and is efficiently checkable.

The completion of level II annotations to level III annotations consists of backwards propagation of assertions using the weakest precondition function for inlined instructions, and insertion of the synchronization annotation to the other program points which have not already been annotated in level II. Method specifications assert that the synchronization annotation holds at points of entry and exit.

The completion also includes placing a boolean expression before and after each ghost assignment. We use the normalizing function *norm* on annotations for this purpose, with the combined effect of conjuncting consecutive logical assertions and backward weakest precondition propagation:

$$\begin{aligned} norm(\alpha) &= \alpha \\ norm(\gamma \cdot \alpha_0 \cdot \alpha_1) &= norm(\gamma \cdot (\alpha_0 \wedge \alpha_1)) \\ norm(\gamma \cdot (\overrightarrow{g} := ce) \cdot \alpha) &= norm(\gamma \cdot \alpha[ce/\overrightarrow{g}]) \cdot (\overrightarrow{g} := ce) \cdot \alpha \end{aligned}$$

where α, α_0, and α_1 range over logical assertions, γ over annotation sequences, and where $\alpha[ce/\overrightarrow{g}]$ denotes the substitution in α of each ghost variable $g_i \in \overrightarrow{g}$ by the conditional expression ce_i, obtained from ce by replacing each expression vector $\overrightarrow{e_E}$ occurring in ce with its i-th component. The normalizing procedure is in line with our treatment of ghost assignments as instructions.

The completion uses the weakest precondition function $wp(M[L])$ introduced in table 4.5 of section 4.7. To deal with side-effect free API methods that are used in inlining (i.e. the string operations), we add two more rules to the table. In particular, the weakest precondition of the static method `System.exit` is taken to be *true*. In both rows, n denotes the arity of method $c.m$, and $f_{c.m}$ denotes the operation implemented by method $c.m$ (which is of arity $n+1$ in the case of `invokevirtual`, with the reference to the object as an implicit argument).

We can now define how to construct a level III annotated program from a level II annotated one.

Level III Annotation A level III (or "full") annotation is obtained as follows.

$M[L]$	$wp(M[L])$
`invokevirtual` $c.m$	$(shift^n(head(A_M[L+1])))[f_{c.m}(\mathrm{s}[0],\ldots,\mathrm{s}[\mathrm{n}])/\mathrm{s}[\mathrm{n}]]$
`invokestatic` $c.m$	$(shift^{n-1}(head(A_M[L+1])))[f_{c.m}(\mathrm{s}[0],\ldots,\mathrm{s}[\mathrm{n}-1])/\mathrm{s}[\mathrm{n}-1]]$

Table 4.6: Weakest precondition function for side-effect free API method calls

1. *Requires*$(\Gamma^*(M))$ and *Ensures*$(\Gamma^*(M))$ are both defined as the synchronisation assertion, $\overrightarrow{gs} = \overrightarrow{ms}$.

2. For all non-inlined instructions $M[L]$, not (level II) annotated with the synchronisation assertion, define

$$A_M^{III}[L] \;=\; norm(A_M^{II}[L] \cdot (\overrightarrow{gs} = \overrightarrow{ms}))$$

3. For all (non-inlined) potentially post-security relevant instructions $M[L]$, define

$$A_M^{III}[L] \;=\; norm(A_M^{II}[L] \cdot (g_0 = r_0) \cdot \ldots \cdot (g_{n-1} = r_{n-1}) \cdot (g_{this} = r_{this}))$$

   where $r_0, \ldots, r_{n-1}, r_{this}$ are the local variables used by the inliner to store the values of the parameters and the reference to the object with which the method is invoked.

4. For all remaining non-inlined instructions $M[L]$, define

$$A_M^{III}[L] \;=\; norm(A_M^{II}[L])$$

5. For all blocks of inlined code, we apply the weakest precondition function $wp(M[L])$ defined in table 4.5 and table 4.6 to propagate backwards the head assertion of the first instruction following the block (which is the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$). Notice that these blocks are cycle-free and do not contain jumps to any other instruction outside of the block, thus the backward *wp*-propagation is well-defined (and in effect computes the weakest precondition of the whole block with respect to the synchronisation assertion).

   Thus, if $M[L]$ is an inlined instruction immediately following a potential (nonexceptional) post-security relevant instruction or the first instruction of a handler for a potential (exceptional) post-security relevant instruction, define

$$\begin{aligned} A_M^{III}[L] \quad = \quad norm((g_0 = r_0) \cdot \ldots \cdot (g_{n-1} = r_{n-1}) \cdot (g_{this} = r_{this}) \cdot \\ \overrightarrow{gs} = \overrightarrow{ms} \cdot \\ A_M^{II}[L] \cdot \\ wp(M[L])) \end{aligned}$$

   and otherwise define

$$A_M^{III}[L] \;=\; wp(M[L])$$

$A^{III}[L]$	L	$M[L]$
$\{\ INV\ \}$	L1	`aload r0`
$\{\ INV\ \}$	L2	`getfield gui`
$\{\ (\mathrm{s}[0] : \texttt{GUI} \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot)) \wedge INV \wedge (\mathrm{s}[0] = \mathrm{s}[0])\ \}$	L3	`dup`
$\{\ ((\mathrm{s}[1] : \texttt{GUI} \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot)) \wedge INV \wedge (\mathrm{s}[1] = \mathrm{s}[0]))\ \}$	L4	`astore r1`
$\{\ ((\mathrm{s}[0] : \texttt{GUI} \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot)) \wedge INV \wedge (\mathrm{s}[0] = \mathrm{r1})) \cdot$ $(g_\mathrm{this} := \mathrm{s}[0]) \cdot$ $((g_\mathrm{this} : \texttt{GUI} \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot)) \wedge INV \wedge (g_\mathrm{this} = \mathrm{r1}))\ \}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{\ (g_\mathrm{this} = \mathrm{r1}) \wedge INV \wedge (\mathrm{r1} <: \texttt{GUI} \Rightarrow (\Phi = (\texttt{SecState.accessed}, \mathrm{s}[0]))) \wedge$ $(\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow (\Phi = (\texttt{SecState.accessed}, \texttt{SecState.permission}))) \cdot$ $((g_\mathrm{a}, g_\mathrm{p}) := \Phi \cdot$ $((\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{s}[0]))) \wedge$ $(\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow INV)))\ \}$	L6	`istore r2`
$\{\ \neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow INV \wedge$ $(\mathrm{r1} <: \texttt{GUI}) \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{r2})\ \}$	L7	`aload r1`
$\{\ \neg(s[0] <: \texttt{GUI}) \Rightarrow INV \wedge$ $(s[0] <: \texttt{GUI}) \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{r2})\ \}$	L8	`instanceof GUI`
$\{\ (s[0] = 0) \Rightarrow INV \wedge$ $\neg(s[0] = 0) \Rightarrow (g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{r2})\ \}$	L9	`ifeq L12`
$\{\ (g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{r2})\ \}$	L10	`iload r2`
$\{\ (g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \mathrm{s}[0])\ \}$	L11	`putstatic SecState/permission`
$\{\ INV\ \}$	L12	`iload r2`
$\{\ INV\ \}$	L13	`ireturn`

Figure 4.9: An application method with level III annotations for the example policy

Example 4.20. For the method presented in the example of section 4.8.2, level III annotations of the method body are shown in figure 4.9 below. The synchronization assertion $(g_\mathrm{a}, g_\mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})$ is denoted by INV. The post-condition and last element of the pre-condition of the method are the synchronization assertion:

$$Ensures = Requires = INV$$

The symbol Φ denotes the following multi-assignment expression:

$$\begin{array}{l} (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot) \wedge g_\mathrm{this} : \texttt{GUI} \wedge \mathrm{s}[0] \rightarrow (g_\mathrm{a}, \mathit{true}) \quad | \\ (g_\mathrm{a}, g_\mathrm{p}) \neq (\bot, \bot) \wedge g_\mathrm{this} : \texttt{GUI} \wedge \neg\mathrm{s}[0] \rightarrow (g_\mathrm{a}, \mathit{false}) \quad | \\ \neg(g_\mathrm{this} : \texttt{GUI}) \rightarrow (g_\mathrm{a}, g_\mathrm{p}) \end{array}$$

Details of the computation are found in Appendix B.2.

We assume that inliners as described above have the following property:

Property 4.21. Given a policy $\mathcal{P}$, and a program T, let T′ be the program inlined for the policy. Then the following holds for each post-security relevant instruction $M[L]$ of T′: Let $M[L] =$ `invokevirtual` $(c.m)$ for some c and m, $\alpha_1, \ldots, \alpha_m$ be the guarded expressions $g_\mathrm{this} : c_i' \wedge\ a_b\rho_i \rightarrow \overrightarrow{e_E}\rho_i$, $1 \leq i \leq m$, and α be $\neg(g_\mathrm{this} : c_1' \vee \ldots \vee g_\mathrm{this} : c_p') \rightarrow \overrightarrow{gs}$, induced, by the policy, for $M[L]$ as described in section 4.8.1. Furthermore, let r_this be the local variable used by the inliner

to record the reference of the object $M[L]$ operates on. Then the weakest precondition of the block of code inlined immediately after the instruction $M[L]$ in T' with respect to the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$ is the logical assertion

$$\begin{array}{ll} & \bigwedge_{1\leq i\leq m} r_{\mathrm{this}} : c'_i \wedge\ a_b\rho'_i\ \rightarrow\ \overrightarrow{gs} = \overrightarrow{e_E\rho'_i} \\ \wedge & \neg(r_{\mathrm{this}} : c'_1 \vee \ldots \vee r_{\mathrm{this}} : c'_p)\ \rightarrow\ \overrightarrow{gs} = \overrightarrow{ms} \end{array}$$

The blocks inlined above and at the exception handlers of security relevant instructions can be specified similarly.

We conjecture that the inliner of [2] has this property. Let I range over inliners described as above and satisfying property 4.21 and let I(T,$\mathcal{P}$) denote the program T inlined by I for the policy $\mathcal{P}$.

The following result uses the full annotation to show that programs inlined for a policy (rewritten as described above and having property 4.21) contain a monitor for the policy as characterized by theorem 4.18.

Theorem 4.22. *Let* T *be a program, and* $\mathcal{P}$ *a ConSpec policy. The inlined program I(*T,$\mathcal{P}$*), fully annotated as above, is locally valid, and validity is efficiently checkable.*

Proof. The proof can be found in Appendix B.3.4. □

Hence by the above result, the level III annotation of I(T,$\mathcal{P}$) is also valid in terms of definition 4.12. As a consequence, the inlined program is valid with respect to the level I annotation for policy $\mathcal{P}$, and therefore, by theorem 4.15, adheres to the policy.

Corollary 4.23 (Correctness of Inlining)**.** *Let* $\mathcal{P}$ *be a ConSpec policy and* T *be a program. The inlined program I(*T,$\mathcal{P}$*) adheres to the policy.*

Notice that a level III annotation as described above can be used for on-device checking of inlining correctness in a proof-carrying code setting.

Another corollary of theorem 4.22 is that for any program T and policy $\mathcal{P}$ the inlined program I(T,$\mathcal{P}$) yields only method-local co-executions. This is so since programs that validate level III annotations validate also level II annotations and thus theorem 4.18 applies to inlined programs.

4.10 Related Work

In this section, we review related work in policy languages, monitor inliners, specifying policy adherence and several other security frameworks for mobile code.

4.10.1 Policy Languages

There exist a number of automata-based languages for security policy specification. Amongst these, ConSpec is closest to PSLang [39] which has also introduced the

modifiers used in ConSpec. The language is intended solely for runtime monitoring and freely uses programming language constructs such as loops. The expressive power of the language enables a larger class of policies to be specified than can be done with ConSpec, but complicates the task of providing a formal semantics and policy matching becomes undecidable. Since the authors do not provide a formal semantics for PSLang, their monitor inlining algorithm is to be trusted on intuition as no proof of its correctness can be constructed.

The Polymer language [12] has the same drawback. Polymer policies consist of Java classes which, when inlined, may trigger various actions in case of violation. For instance it is possible to execute some recovery action as a response to the violation, after which the application is allowed to progress. Polymer policies implement *edit automata* [79], which extend security automata. But the correctness of the Polymer policy inlining cannot be proven either, as its semantics is not formally presented. In a recent work [13], the same authors present a simpler version of the Polymer language with its semantics. The Polymer semantics is given in the context of a lambda calculus, which is used to present the programming language semantics. Using this formalization, the authors prove uncircumventability of the monitor and type safety of the monitored programs.

Many logic-based formalisms are also used to express security properties for monitoring purposes (e.g. [57, 58, 106, 84, 113, 11]). We only note here that temporal logic formulae expressing safety properties can be translated to automata and vice versa (see e.g. [66]).

4.10.2 Monitor Inliners

Monitor inlining has been employed as a security enforcement mechanism in a number of application areas. We account here the basic *types* of monitor inlining implementations offered in the context of language-based security, in terms of where the code is inlined. We focus on method calls as security relevant actions, although it is possible to monitor many other events with existing tools such as PoET [40].

The inliners that input policies in the form of security automata can be categorized according to where they insert code to perform the inquiry on whether the security relevant action is safe to perform in the current state and the update on the security state. The inlining style of PoET and our tool creates code where security checks and updates are *scattered* in the program: the code is inserted around the security relevant method call at the caller side. An alternative is to inline the program by altering the methods of the untrusted program only through rewriting method invocation instructions so that calls to security relevant API methods are redirected to new methods added in the inlining process. These new methods are then dedicated and consist of code for performing necessary security checks for the particular security relevant API method, the call to the method and the update to the security state. Such a *wrapping* approach is taken in Naccio, one of the first tools for monitor inlining [46]. A rather clean way of implementing inlining is through *centralizing*, i.e. using a dedicated component that implements the policy.

The information about the method call (or return), like the name of the method and the values of arguments are passed to this module which performs the necessary operations. The inliner of Vanoverberghe and Piessens is an example [110]. The Polymer system also practices a centralized inlining where a policy is specified as a Java class[8] but API method bodies are altered in the course of the inlining.

All of these approaches have advantages and disadvantages. Scattered inlining allows better optimizations to be performed. For instance, security checks and updates may be found unnecessary, depending on the information on method arguments at a certain call point obtained from static analysis. On the other hand, a scattered implementation of a policy is more difficult to understand than a centralized implementation. We are not concerned here with evaluating different implementations. What is important is that our annotation scheme can handle wrapping and scattered implementations. For handling centralized implementations, however, some adaptation is needed as the updates to the security state do not occur in the caller method boundaries. We do not support the inlining of code in API methods, either.

Monitor inlining can also be implemented through *aspect-oriented programming* (AOP) [67, 47]. In such an approach, the security policy is programmed as an aspect, which gets inserted into the program at the compilation stage (*aspect weaving*) resulting in scattered or wrapped code. In [36], an example can be found for contract enforcement for a Java application in AspectJ [94], an AOP environment. In this example, the contract, provided through assertions, is programmed as an aspect. In this manner the two concerns, development and contract declaration, are elegantly separated. We have not carried out this study in the context of aspect-orientation as this makes reasoning compositionally on the level of program methods more difficult.

4.10.3 Specifying Policy Adherence

Alpern and Schneider propose in earlier work a method for showing that a program adheres to a temporal property by producing proof obligations [5]. This work is more general than ours in a number of aspects: (i) concurrent, as well as sequential, programs are considered, (ii) properties are not just safety properties but the set of properties expressible by the conjunction of a set of Büchi automata (i.e. includes liveness properties) and (iii) the program is not expected to be inlined with a monitor. The advantage of our method, on the other hand, is that it is decidable.

The problem that we address here can be seen as a special case of the above method. Note that our policies correspond to safety properties and that a ConSpec automaton induced by such a policy can be converted to a deterministic Büchi automaton (see section 4.4.2 for details). Since, in this case, the desired property can be expressed by a single deterministic Büchi automaton, coming up with proof

[8] An advantage is that this particular implementation allows different types of policy compositions to be implemented as operations on these objects.

obligations for a program using the method of Alpern and Schneider becomes trivial and amounts to asserting that the automaton is at an accepting state throughout the execution of the program. The drawback of this approach is that the resulting verification conditions are not presented as assertions on the program text but rather as conditions on the state space of the program, presented as a transition system. Therefore, in order to apply this method, such a model of the program should be extracted. Considering that the states of this model includes a program counter value along with variable values, the models are expected to be very large for bytecode programs, thus making this method unattractive in our setting.

A result closely related to ours is the recent work on type-based monitor certification by Hamlen *et al.* [54]. Policies supported by their language *Mobile* are attached to security relevant classes and restrict the sequences of security relevant actions exhibited by instances of these classes. (ConSpec policies with scope `Object` are Mobile policies when a persistent, global state is not present.) Thus the focus is on per-object monitoring, as compared to the "per-session" model we consider in our annotation scheme. Mobile programs are essentially programs in the intermediate language of the .NET framework with additional typing annotations that track an abstract representation of program execution history. The important property of Mobile programs is that a program that types correctly with respect to a security policy is guaranteed to adhere to the policy. Thus the authors reduce the problem of correct inlining to that of type-checking, which is an efficient, well-studied procedure. However, their results are restricted to one particular inliner, whereas we give a characterization of a whole class of compositional inliners.

4.10.4 Security Frameworks for Mobile Code

Many frameworks have been developed in recent years for ensuring safe execution of mobile code. We focus here on those concerned with security as defined by safety properties and that are, at least partially, based on the PCC idea.

Abstraction-Carrying Code In [4], Albert *et al.* introduce a PCC framework using abstract interpretation as enabling technology. In the Abstraction-Carrying Code (ACC) approach, an abstract model of the program's behavior is computed by a static analyzer based on abstract interpretation and shipped together with the program. On the consumer side, the abstraction is first checked to be faithful to the program, after which policy adherence can be checked by proving the arising verification conditions. The former step is performed on the consumer side by an efficient checker, which simply verifies that the abstraction is loyal to the program, in contrast to the fixpoint calculation needed to perform the abstraction on the producer side. The class of policies handled by the framework are restricted to the class of stateless ConSpec policies, as the abstract interpretation proceeds by abstraction on the infinite domains of method arguments and return values. This approach can handle a larger class of programs however, as the program does not need to have been inlined for the proof to be generated.

Model-Carrying Code The S3MS framework presented in section 4.1 is quite

similar to that of model-carrying code (MCC) introduced by Sekar *et al.* [103]. In the MCC setting, the program is shipped together with a model of its security relevant behavior. To extract such a model would be too costly to be performed with precision on the consumer side. On the producer side, however, more resources are available and the process can potentially be performed on the source code rather than the bytecode. In order to be assured of the safety of the code, the consumer should perform two tasks in the MCC framework: the model should be checked to satisfy the device policy and the model should be checked to be a safe approximation of the program. Since the latter may be a costly task due to the size of the model extracted from the program, an under-approximation of the program behavior is captured by the model instead and this model is enforced on the program execution at runtime by monitoring. (Note that if the model is precise and therefore has a small number of spurious misbehavior, the runtime aborts due to this enforcement are expected to occur infrequently.)

The first task mentioned above to be performed on the consumer side is policy matching. *Extended finite state automata* (EFSA) are used to represent program models and policies in MCC. These are finite state automata extended with a set of variables for recording argument values. EFSA are similar to ConSpec automata, except that the variables used for recording arguments can range over infinite domains. (Recall that to enable the matching of two ConSpec policies, we require that the domains of security state variables be finite.) In order to enable matching in the presence of infinite domains, the current MCC framework allows only equality conditions of the variables, while our language allows more sophisticated expressions, including basic arithmetic operations and comparisons of numeric values. For the program automaton M and the policy automaton P, the matching step is performed on the consumer side by checking that the language of the automaton $M \times \bar{P}$ is empty, i.e. that all the behaviors of the program are contained in the behaviors allowed by the policy.

As noted also by the authors, in the MCC approach, the code producers need not know anything about the security policy of the code consumer, except for the security relevant action set. With a precise enough model, it will be possible to check whether the program violates the user policy on the consumer side even without prior knowledge on the policy. This is not the case in our framework, as the producer policy is not constructed as a model of the program but rather preconceived and then enforced on the program. In this aspect, the MCC framework is superior to ours. What is more, MCC does not employ a proof checker component, but only a policy matcher. On the other hand, a runtime monitoring component is used to enforce the model on the program and the model for each program needs to be saved for this purpose. This may be a drawback of this approach when storage space is scarce as these models are admittedly rather large. The main difficulty in applying MCC in practice is to develop a suitable model extraction scheme. In [103], the authors suggest a method of "learning" the model through executing the program on test cases. It may not be even possible to produce a model when more precise models than EFSA (e.g. pushdown automata (PDA)) are used, as the problem

becomes undecidable in this case.

Developers of MCC view runtime aborts as unpleasant and some of their design choices are taken to avoid them. This, however, may result in programs to be rejected although they obey the security policy in *most cases*. The reason is that, in this framework the program is not run at all on the consumer side if its model contains violating behavior. Whereas, it may be the case that, these execution paths are seldom taken in practice (e.g. a certain feature may not be used at all by some users). In our framework, on the other hand, the producer policy is enforced on the program by monitor inlining, therefore it is enough that the producer policy is in line with the consumer policy and that the program is correctly inlined to run the program safely. In other words, in our framework, the consumer gets to try out the program and the decision on whether the runtime aborts are too many is left to the consumer. This is a general advantage of using runtime monitoring over static approaches as the program is not disqualified based on one "bad" execution but rather such executions are prevented from doing harm in the hope that the program has enough "good" executions to be of use.

4.11 Conclusion

4.11.1 Summary and Contributions

The contributions of this chapter can be summarized as follows:

- *The policy language* ConSpec is a guarded-command language with bounded domains and a restricted language for updates, rendering the policy matching problem decidable. Policies in this language are sets of sequences of security relevant actions which are considered safe. Calls to and returns from API methods are taken as the security relevant actions. Constraints can be put on the actual arguments and return value of such an action and heap at time of call/return. These are propagated to subclasses to handle inheritance in a seamless manner. Semantics of "per-session" policies written in this language is presented as symbolic and as concrete security automata.

- *Monitoring as co-execution* We present a formalization of monitoring with security automata induced from ConSpec policies. The notion of co-execution is introduced as a predicate on interleavings of (possibly infinite) program executions with automata runs. In a co-execution, the automaton makes transitions on the security relevant actions of the program and the program terminates when no transition is possible. A program execution complies to the policy if there exists a corresponding co-execution with the policy induced automaton.

- *Level I annotations* We give a characterization of policy-adherence for programs, in terms of JVM class files annotated by formulae in a Floyd-like logic.

In order to specify policy adherence, we insert a correct monitor into the program using annotations. The monitor state is represented by "ghost" variables, essentially specification variables that can be assigned values through a conditional update statement. The annotations are constructed to mimic updates to the monitor state as specified in the policy and inserted at relevant method call instructions in the program. Finally, the ghost variables representing the monitor state is asserted to be defined immediately before any security relevant action. We prove in particular that a level I annotated program is valid if and only if the program is policy adherent. This result reduces the problem of showing policy-adherence to establishing the validity of the level I annotated program.

- *Level II annotations* We define how level I annotations can be extended to level II annotations. These annotations capture the existence of a concrete representation of the monitor state in the target program, by asserting the equality of the ghost state and the concrete monitor state at method boundaries. As a consequence, level II annotations characterize correct, compositional inlining and can be used to show the correctness of a particular inliner after the concrete monitor state is suitably instantiated as the inlined state.

- *Correctness of Inlining* In order to show the practical use of the annotations, we present a simple monitor inlining scheme and show correctness for this inliner. We sketch how, for the inliner, the level II annotations can be completed to produce a fully annotated program for which validity can be efficiently decided using a weakest precondition checker. Hence, full annotations that imply level II annotations can be used as proof of correct inlining to certify compliance to a third party such as a mobile device.

4.11.2 Future Work

There are many possibilities for extending our work, we enumerate most significant of these below.

- *Semantics of Policies with Different Scopes* In section 4.4, we have introduced four scopes (`Global`, `Multisession`, `Session` and `Object`) for policies, out of which we have presented semantics, introduced inlining techniques and developed an annotation scheme for only one, the `Session` scope. We would like to extend our framework for policies of scope `Multisession` and `Object`. (The `Global` scope is of less interest as this type of policy restricts all applications on a platform and hence could be viewed as a "global" `Multisession` policy.)

- *Transparency* Transparency is a prerequisite for the practical use of any monitor for the purpose of security enforcement. Showing transparency is a subtle issue when the monitor is embedded into the program. In this case, the monitor is part of the program state so transparency can not be defined as the

identity of the executions of the original and the inlined program, on the same input, provided the original program execution is non-violating. Transparency of the monitor is not the concern of the code consumer, however, but rather a concern to the code producer and therefore would be checked off-device.

- *Monitoring multi-threaded programs* Provided an interleaving semantics for multi-threaded bytecode programs, the notion of co-execution can be used even for the monitoring of this type of program. However, monitor inliners for multi-threaded programs have to implement mechanisms to avoid races such as those which occur when several threads attempt to execute security relevant actions at the same time.

- *Handling other self-monitoring programs* As we stated earlier, level I annotations characterize self-monitoring programs. This is a much larger class than correctly inlined programs. A program may obey the policy simply as a result of security-aware development. However, checking validity of such a level I annotated program is not straightforward and would require heuristics to instantiate the security state in order for level II (and gradually level III) annotations to be created.

Appendix A

Part I Appendix

A.1 Proofs for Part I

A.1.1 Correctness of Maximal Model Construction

Preliminaries Before we proceed to the proof of the correctness, we give some definitions. We extend the notion of simulation for "labeled" EMTSs, that is for EMTS $\mathcal{E}$ equipped with a labeling function $\lambda : S_{\mathcal{E}} \to 2^{PropVar}$ that labels the states of $\mathcal{E}$ with propositional variables.

Definition A.1 (Simulation)**.** $R \subseteq S_{\mathcal{T}} \times S_{\mathcal{E}}$ is a *simulation* with respect to valuation $\mathrm{V} : PropVar \to 2^{S_{\mathcal{T}}}$ if whenever tRs, $a \in A$ and $Z \in PropVar$:

1. if $s_1 \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S_1$, then there is a set of states S_2 such that $s_2 \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} S_2$ and for each $s'_1 \in S_1$, there exists a $s'_2 \in S_2$ such that $s'_1 R s'_2$;

2. if $s_2 \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S_2$, then there is a set of states S_1 such that $s_1 \overset{a}{\longrightarrow}^{\Box}_{\mathcal{E}} S_1$ and for each $s'_1 \in S_1$, there exists a $s'_2 \in S_2$ such that $s'_1 R s'_2$;

3. if the run $\rho_{s_2} = s_2 \xrightarrow{a_1}_{\mathcal{E}} s_2^1 \xrightarrow{a_2}_{\mathcal{E}} s_2^2 \xrightarrow{a_3}_{\mathcal{E}} \ldots$ is in $W_{\mathcal{E}}$ then every infinite run $\rho_{s_1} = s_1 \xrightarrow{a_1}_{\mathcal{E}} s_1^1 \xrightarrow{a_2}_{\mathcal{E}} s_1^2 \xrightarrow{a_3}_{\mathcal{E}} \ldots$ such that $s_1^i R\, s_2^i$ for all $i \geq 1$ is also in $W_{\mathcal{E}}$;

4. if $Z \in \lambda(s)$, then $t \in \mathrm{V}(Z)$.

We write $tR^{\mathrm{V}}s$ when tRs with respect to valuation V. We say that abstract state s simulates state t with respect to valuation V, denoted $t \preceq_{\mathrm{V}} s$, if there is a simulation relation R such that $tR^{\mathrm{V}}s$.

We define the notion of denotation of an EMTS $\mathcal{E}$ with respect to a valuation as follows:

Definition A.2 (EMTS Denotation)**.** Let $\mathcal{E}$ an EMTS with $(S_{\mathcal{E}}, A, \longrightarrow^{\Diamond}_{\mathcal{E}}, \longrightarrow^{\Box}_{\mathcal{E}}, c)$, $S \subseteq S_{\mathcal{E}}$ a set of start states of $\mathcal{E}$, and $\lambda : S_{\mathcal{E}} \to 2^{PropVar}$ a labeling function.

The denotation of a state $s \in S_{\mathcal{E}}$ with respect to an LTS $\mathcal{T}$ and valuation function $\mathrm{V} : PropVar \to 2^{S_{\mathcal{T}}}$ is defined as $[\![s]\!]^{\mathcal{T}}_{\mathrm{V}} \triangleq \{t \mid t \preceq_{\mathrm{V}} s\}$. The notion of denotation is lifted to sets of states in the natural way.

The denotation of a triple $(\mathcal{E},\ S,\ \lambda)$, then, is defined as the denotation of the start states:

$$[\![(\mathcal{E}, S, \lambda)]\!]^{\mathcal{T}}_{\mathrm{V}} \triangleq [\![S]\!]^{\mathcal{T}}_{\mathrm{V}}$$

Proof Using these definitions, we restate the theorem 2.13 to include valuation V. Notice that the valuation does not have any effect when the states of $\mathcal{E}$ are not labeled.

Theorem 2.13 $[\![\varepsilon(\Phi)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\Phi||^{\mathcal{T}}_{\mathrm{V}}$.

Proof. The proof proceeds by induction on the structure of Φ. Let $\varepsilon(\Psi_1) = ((S_{\mathcal{E}_1},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}_1},\ \longrightarrow^{\Box}_{\mathcal{E}_1},\ W_1),\ S_1,\ \lambda_1)$ and $\varepsilon(\Psi_2)= ((S_{\mathcal{E}_2},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}_2},\ \longrightarrow^{\Box}_{\mathcal{E}_2},\ W_2),\ S_2,\ \lambda_2)$ be constructed as defined in Figure 2.4.

- $\Phi \equiv \mathrm{tt}$
  $[\![S]\!]_{\mathcal{U}} = S_{\mathcal{U}} = ||\mathrm{tt}||^{\mathcal{U}}_{\mathrm{V}_0}$
- $\Phi \equiv \mathrm{ff}$
  $[\![S]\!]_{\mathcal{U}} = \varnothing = ||\mathrm{ff}||^{\mathcal{U}}_{\mathrm{V}_0}$
- $\Phi \equiv Z$
  $[\![S]\!]_{\mathcal{U}} = [\![\mathbf{0}]\!]_{\approx} = ||Z||^{\mathcal{U}}_{\mathrm{V}_0}$
- $\Phi \equiv \langle a \rangle\ \Psi_1$

  $t \preceq S$
  $\iff$ $t \preceq s_{new}$ (Construction)
  $\iff$ $\exists t'.t \xrightarrow{a}_{\mathcal{T}} t'$ where $t' \preceq S_1$ (Def. A.1)
  $\iff$ $\exists t'.t \xrightarrow{a}_{\mathcal{T}} t'$ where $t' \models^{\mathcal{U}}_{\mathrm{V}_0} \Psi_1$ (Induction Hyp.)
  $\iff$ $t \models^{\mathcal{U}}_{\mathrm{V}_0} \langle a \rangle\ \Psi_1$ (Def., page 2.5)

- $\Phi \equiv [a]\ \Psi_1$

  $t \preceq S$
  $\iff$ $t \preceq s_{new}$ (Construction)
  $\iff$ if there exists a transition $t \xrightarrow{a}_{\mathcal{T}} t'$, then $t' \preceq S_1$ (Def. A.1)
  $\iff$ $\forall t'.t \xrightarrow{a}_{\mathcal{T}} t' \Rightarrow t' \models^{\mathcal{U}}_{\mathrm{V}_0} \Psi_1$ (Induction Hyp.)
  $\iff$ $t \models^{\mathcal{U}}_{\mathrm{V}_0} [a]\ \Psi_1$ (Def., page 2.5)

- $\Phi \equiv \Psi_1 \wedge \Psi_2$

  Assume $[\![\varepsilon(\Psi_i)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\Psi_i||^{\mathcal{T}}_{\mathrm{V}}$ for $i \in \{1, 2\}$ (Induction Hyp.)

  We show $[\![\varepsilon(\Psi_1 \wedge \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\Psi_1 \wedge \Psi_2||^{\mathcal{T}}_{\mathrm{V}}$ in two parts.

  ($\subseteq$) Assume $t \in [\![\varepsilon(\Psi_1 \wedge \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$, then there is a simulation relation $R_1 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1 \wedge \Psi_2)}$ for V such that $tR^{\mathrm{V}}_1(s, r)$ for some $(s, r) \in (S_{\varepsilon(\Psi_1)} \times S_{\varepsilon(\Psi_2)})|_{\Box} \cap$

$(S_1 \times S_2)$ by definition of construction. We define $R'_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1)}$ as tR'_2s if and only if there exists $r \in S_{\varepsilon(\Psi_2)}.tR_1(s,r)$. We prove R'_2 is a simulation relation below. Hence, $t \in [\![\varepsilon(\Psi_1)]\!]^{\mathcal{T}}_{\mathrm{V}}$. By a similar argument we can show $t \in [\![\varepsilon(\Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$. Therefore, $t \in ||\Psi_1 \wedge \Psi_2||^{\mathcal{T}}_{\mathrm{V}}$.

We now prove that R'_2 is a simulation relation. Assume $t\ R'_2\ s$.

1. Whenever $a \in A$,
   a) If $t \xrightarrow{a}_{\mathcal{T}} t'$, by definition of simulation there exists S such that $(s,r) \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S$ and $t'R_1(s',r')$ for some $(s',r') \in S$. By construction $(s,r) \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S$ if and only if there exists S' and R' such that $s \xrightarrow{a}^{\Diamond}_{\mathcal{E}_1} S'$, $r \xrightarrow{a}^{\Diamond}_{\mathcal{E}_2} R'$ and $S = S' \times R'$. Thus $s \xrightarrow{a}^{\Diamond}_{\mathcal{E}_1} S'$ where $t'R'_2s'$ for some $s' \in S'$.
   b) If $s \xrightarrow{a}^{\Box}_{\mathcal{E}_1} S'$, then $(s,r) \xrightarrow{a}^{\Box}_{\mathcal{E}} S$ where $S = S' \times \cup\partial^{\Diamond}_a(r)$. (The set $\partial^{\Diamond}_a(r)$ can not be empty since in this case this state would not be a part of the constructed EMTS) Since $tR_1(s,r)$, $t \xrightarrow{a}_{\mathcal{T}} t'$ such that $t'R'_2(s',r')$ where $s' \in S'$ and $r' \in \cup\partial^{\Diamond}_a(r)$, so $t'R'_2s'$.
2. If $\rho_s = s \xrightarrow{a_1}_{\mathcal{E}_1} s_2 \xrightarrow{a_2}_{\mathcal{E}_1} s_3 \xrightarrow{a_3}_{\mathcal{E}_1} \ldots$ is in W, then no infinite run $\rho_t = t \xrightarrow{a_1}_{\mathcal{T}} t_2 \xrightarrow{a_2}_{\mathcal{T}} t_3 \xrightarrow{a_3}_{\mathcal{T}} \ldots$ of t such that $t_i\ R'_2\ s_i$ for all $i \geq 1$ is possible. Such an infinite run ρ_t is not possible because the run $\rho_{(s,r)} = (s,r) \xrightarrow{a_1}_{\mathcal{E}} (s_2,r_2) \xrightarrow{a_2}_{\mathcal{E}} (s_3,r_3) \xrightarrow{a_3}_{\mathcal{E}} \ldots \in W$ by construction.

$(\supseteq)$ Assume $t \in [\![\varepsilon(\Psi_1)]\!]^{\mathcal{T}}_{\mathrm{V}}$ and $t \in [\![\varepsilon(\Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$, then there are simulation relations $R_1 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1)}$ and $R_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_2)}$ for V such that tR^{V}_1s and tR^{V}_2s for some $s_1 \in S_1$ and $r_1 \in S_2$ by definition of construction. We define $R \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1 \wedge \Psi_2)}$ as $tR(s,r)$ if and only if tR_1s and tR_1r. We prove R is a simulation relation below. Hence, $t \in [\![\varepsilon(\Psi_1 \wedge \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$.

We now prove that R is a simulation relation. Assume $t\ R\ (s,r)$.

1. Whenever $a \in A$,
   a) If $t \xrightarrow{a}_{\mathcal{T}} t'$, by induction hypothesis and definition of simulation there exists S' and R' such that $s \xrightarrow{a}^{\Diamond}_{\mathcal{E}_1} S'$, $r \xrightarrow{a}^{\Diamond}_{\mathcal{E}_1} R'$ and $t'R_1s'$, $t'R_2r'$ for some $s' \in S'$ and $r' \in R'$. Then by construction there exists S where $(s,r) \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S$ and $(s',r') \in S$ and $t'R(s',r')$.
   b) If $(s,r) \xrightarrow{a}^{\Box}_{\mathcal{E}} S$, then either there exists S' such that $s \xrightarrow{a}^{\Box}_{\mathcal{E}} S'$ and $S = \{(s',r') \mid s' \in S' \wedge r' \in \cup\partial^{\Diamond}_a(r)\}$ or there exists R' such that $r \xrightarrow{a}^{\Box}_{\mathcal{E}} R'$ and $S = \{(s',r') \mid s' \in \cup\partial^{\Diamond}_a(s) \wedge r' \in R'\}$.
   If it is the first case, by the definition of of simulation there exists t' such that $t \xrightarrow{a}_{\mathcal{T}} t'$, and $t'R_1s'$ for some s' in S'. Since t is also simulated by r, there exists some $R' \in \cup\partial^{\Diamond}_a(r)$ such that $t'R_2r'$ for

some $r' \in R'$. Then $(s', r') \in S$ and $t' R (s', r')$.
The second case is similar.

2. If $\rho_{(s,r)} = (s,r) \xrightarrow{a_1}_{\mathcal{E}} (s_2, r_2) \xrightarrow{a_2}_{\mathcal{E}} (s_3, r_3) \xrightarrow{a_3}_{\mathcal{E}} \ldots$ is in W, then the infinite run $\rho_t = t \xrightarrow{a_1}_{\mathcal{T}} t_2 \xrightarrow{a_2}_{\mathcal{T}} t_3 \xrightarrow{a_3}_{\mathcal{T}} \ldots$ such that $t_i \; R \; (s_i, r_i)$ for all $i \geq 1$ should not be possible.
Assume that such an infinite run ρ_{t_1} exists. By construction, $\rho_{(s,r)} \in W$ if and only if $\rho_s = s \xrightarrow{a_1}_{\mathcal{E}_1} s_2 \xrightarrow{a_2}_{\mathcal{E}_1} s_3 \xrightarrow{a_3}_{\mathcal{E})_1} \ldots \in W_1$ or $\rho_r = r \xrightarrow{a_1}_{\mathcal{E}_2} r_2 \xrightarrow{a_2}_{\mathcal{E}_2} r_3 \xrightarrow{a_3}_{\mathcal{E})_2} \ldots \in W_2$. If it is the first case and $\rho_s \in W_1$, by definition of simulation there is no infinite run $\rho_t = t \xrightarrow{a_1}_{\mathcal{T}} t'_2 \xrightarrow{a_2}_{\mathcal{T}} \longrightarrow_{\mathcal{T}} t'_3 \xrightarrow{a_3}_{\mathcal{T}} \ldots$ where $t'_i \; R_1 \; s_i$ for all $i \geq 1$. However, ρ_t is such a run hence we reach a contradiction. The argument is similar if $\rho_r \in W_2$.

- $\Phi \equiv \Psi_1 \vee \Psi_2$

  Assume $[\![\varepsilon(\Psi_i)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\Psi_i||^{\mathcal{T}}_{\mathrm{V}}$ for $i \in \{1, 2\}$ (Induction Hyp.)

  We show $[\![\varepsilon(\Psi_1 \vee \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\Psi_1 \vee \Psi_2||^{\mathcal{T}}_{\mathrm{V}}$ in two parts.

  ($\subseteq$) Assume $t \in [\![\varepsilon(\Psi_1 \vee \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$, then there is a simulation relation $R_1 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1 \vee \Psi_2)}$ for V such that $t R_1^{\mathrm{V}} s$ for some $s \in S_1 \cup S_2$ by definition of construction. We define $R'_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1)}$ as $t R'_2 s$ if and only if $t R_1 s$ and $s \in S_{\varepsilon(\Psi_1)}$ and similarly $R''_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_2)}$ as $t R''_2 s$ if and only if $t R_1 s$ and $s \in S_{\varepsilon(\Psi_2)}$. These relations are well defined as $S_{\varepsilon(\Psi_1)}$ and $S_{\varepsilon(\Psi_2)}$ are disjoint sets. As the sets are disjoint, showing these relations are indeed simulation relations is trivial as the conditions are satisfied by R_1. Hence $t \in [\![\varepsilon(\Psi_1)]\!]^{\mathcal{T}}_{\mathrm{V}}$ or $t \in [\![\varepsilon(\Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$ and by (Induction Hypothesis) $t \in ||\Psi_1||^{\mathcal{T}}_{\mathrm{V}}$ or $t \in ||\Psi_2||^{\mathcal{T}}_{\mathrm{V}}$. Therefore, $t \in ||\Psi_1 \vee \Psi_2||^{\mathcal{T}}_{\mathrm{V}}$.

  ($\supseteq$) Assume $t \in [\![\varepsilon(\Psi_1]\!]^{\mathcal{T}}_{\mathrm{V}}$. Then there is a simulation relation $R'_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1)}$ for V such that $t R'^{\mathrm{V}}_2 s$ for some $s \in S_1$. We define $R_1 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi_1 \vee \Psi_2)}$ as $t R_1 s$ if and only if $t R'_2 s$. It is again trivial to show that R_1 is a simulation relation since it is identical to R'_2. Hence $t \in [\![\varepsilon(\Psi_1 \vee \Psi_2)]\!]^{\mathcal{T}}_{\mathrm{V}}$. The case for $t \in [\![\varepsilon(\Psi_2]\!]^{\mathcal{T}}_{\mathrm{V}}$ is similar.

- $\Phi \equiv \nu Z.\Psi_1$
  Dual to least fixed point case.

- $\Phi \equiv \mu Z.\Psi$
  Let $\varepsilon(\Psi) = ((S_{\varepsilon(\Psi)}, A, \longrightarrow^{\Diamond}_{\mathcal{E}}, \longrightarrow^{\Box}_{\mathcal{E}}, c), S_1, \lambda)$ and $\varepsilon(\mu Z.\Psi) = ((S_{\varepsilon(\mu Z.\Psi)}, A, \longrightarrow^{\Diamond}_{\mathcal{E}'}, \longrightarrow^{\Box}_{\mathcal{E}'}, c'), S'_1, \lambda')$ be constructed as defined in figure 2.4. We will prove

  $$[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}} = ||\mu Z.\Psi||^{\mathcal{T}}_{\mathrm{V}}$$

  in two steps.

  ($\subseteq$) We make use of ordinal approximants and the unfolding theorem for fixed point formulae (i.e. theorem 2.6).

We show

$$[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}} \subseteq ||\mu Z.\Psi||^{\mathcal{T}}_{\mathrm{V}}$$

Note first that

$$||(\mu Z.\Psi)^{\kappa}||^{\mathcal{T}}_{\mathrm{V}} = \bigcup_{\beta<\kappa} ||(\mu Z.\Psi)^{\beta}||^{\mathcal{T}}_{\mathrm{V}}$$

and so

$$||(\mu Z.\Psi)^{\kappa}||^{\mathcal{T}}_{\mathrm{V}} = ||\Psi||^{\mathcal{T}}_{\mathrm{V}[\bigcup_{\beta<\kappa} ||(\mu Z.\Psi)^{\beta}||^{\mathcal{T}}_{\mathrm{V}}/Z]}$$

Using the induction hypothesis then, we can replace the formula with its EMTS:

$$||(\mu Z.\Psi)^{\kappa}||^{\mathcal{T}}_{\mathrm{V}} = [\![\varepsilon(\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}[\bigcup_{\beta<\kappa} ||(\mu Z.\Psi)^{\beta}||^{\mathcal{T}}_{\mathrm{V}}/Z]}$$

Assume $t \in [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$. Then there is a simulation relation $R_1 \in S_{\mathcal{T}} \times S_{\varepsilon(\mu Z.\Psi)}$ for V such that tR_1s for some state state $s \in S_1$.

Then there is a mapping $ord : S_{\mathcal{T}} \to Ord_+$ such that whenever $t'R_1q'$, $t' \xrightarrow{a}_{\mathcal{T}} t''$, $q' \xrightarrow{a}^{\Diamond}_{\mathcal{E}} q''$ and $t''R_1q''$, then $ord(t'') < ord(t')$ whenever $Z \in \lambda_1(\cup q'')$ and $ord(t'') \leq ord(t')$ otherwise.

Define $S^{\kappa}_{\mathcal{T}} \stackrel{def}{=} \{t' \in S_{\mathcal{T}} \mid ord(t') \leq \kappa\}$ and $R^{\kappa}_1 \stackrel{def}{=} R_{1|_{S^{\kappa}_{\mathcal{T}}}}$

We claim that R^{κ}_1 is a simulation with respect to V. We show the following by induction on $ord(t')$.

$$\forall t' \in S_{\mathcal{T}}.(\exists q' \in S_{\varepsilon(\mu Z.\Psi)}.q' \cap S_1 \wedge t'R_1^{ord(t')}q') \Rightarrow t' \in ||(\mu Z.\Psi)^{ord(t')}||^{\mathcal{T}}_{\mathrm{V}}$$

Assume it holds for all $t'' \in S_{\mathcal{T}}$ such that $ord(t'') \leq \kappa$ by ordinal induction hypothesis. Assume $t' \in S_{\mathcal{T}}$ is such that $ord(t') = \kappa$, and there exists $q' \in S_{\varepsilon(\mu Z.\Psi)}.\exists q' \in S_{\varepsilon(\mu Z.\Psi)}.q' \cap S_1 \neq \varnothing \wedge t'R_1^{ord(t')}q')$. we show $t' \in [\![\varepsilon(\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}^{\kappa}}$ where $\mathrm{V}^{\kappa} = \mathrm{V}[\bigcup_{\beta<\kappa} ||(\mu Z.\Psi)^{\beta}||^{\mathcal{T}}_{\mathrm{V}}/Z]$, and then by the original induction hypothesis and the unfolding of formula illustrated above we obtain $t' \in ||(\mu Z.\Psi)^{ord(t')}||^{\mathcal{T}}_{\mathrm{V}}$ since $\kappa = ord(t')$.

Define $R^{\kappa}_2 \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi)}$ as $\{(t', s') \mid \exists q' \in S_{\varepsilon(\mu Z.\Psi)}.(s' \in real(q') \wedge t'R^{\kappa}_1q')\}$, where $real(q)$ is defined inductively as follows:

- $real(q) = q$ if q is a singleton,

– $real(q) \subseteq q$ is defined otherwise as the set of derivatives of $\bigcup Q$ in q, where the set Q in turn is defined as follows, $q'' \in Q$ if and only if such that $\exists Q' \subseteq S_{\varepsilon(\mu Z.\Psi)}.\ q'' \stackrel{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} Q'$ where $q \subseteq Q'$.

We show that $R_2^\kappa \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi)}$ is a simulation with respect to V^κ. Then, by the assumption on t', we have $t' R_2^\kappa s'$ for some $s' \in S_1$, and hence $t' \in [\![\varepsilon(\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}^\kappa}$.

($\supseteq$) We show

$$[\![\varepsilon(\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}[[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}/Z]} \subseteq [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}(*)$$

Then by the induction hypothesis

$$||\Psi||^{\mathcal{T}}_{\mathrm{V}[[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}/Z]} \subseteq [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$$

The result holds since $[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$ is a prefixed point of the function $\lambda S.||\Psi||^{\mathcal{T}}_{[S/X]}$, and since $||\mu Z.\Psi||^{\mathcal{T}}_{\mathrm{V}}$ is the least such prefixed point by the semantics of modal μ-calculus.

Let $\mathrm{V}^* = \mathrm{V}[[\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}/Z]$. Proof of $[\![\varepsilon(\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}^*} \subseteq [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$ is as follows:

We show that if there exists an $s \in S_1$ such that $t R^{\mathrm{V}^*} s$ for some simulation relation $R \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\Psi)}$, then there exists a simulation relation $R' \subseteq S_{\mathcal{T}} \times S_{\varepsilon(\mu Z.\Psi)}$ and a state $\{q\} \in S_1'$ such that $t R'^{\mathrm{V}} \{q\}$.

We define R' for V using R for V^* as the least relation that satisfies the following:

– If tRs and $s \in S_1$, $tR'\{s\}$

– If $t'R'\{q\}$ and $t' \stackrel{a}{\longrightarrow}_{\mathcal{T}} t''$ since $t'Rq$ there exists q' such that $q' \in Q'$ for some Q' where $q \stackrel{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} Q'$ and $t''Rq'$. Then,

  1. if $Z \in \lambda(q')$, then we know by the definition of simulation that $t \in [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$. Then there exists a simulation relation R'' such that and for some $s'' \in S_1$ $t''R''^{\mathrm{V}}\{s''\}$ by construction and the def. of denotation. Then $t''R'\{q', s''\}$.
  2. if $Z \notin \lambda(q')$, then $t''R'^{\mathrm{V}}\{q'\}$.

– If $t'R'^{\mathrm{V}}\{q_1, \ldots, q_n\}$, $n > 1$ and $t' \stackrel{a}{\longrightarrow}_{\mathcal{T}} t''$
since there exists

  * $q_i \notin S_1$ such that $t'Rq_i$, there exists q' such that $q' \in Q'$ for some Q' where $q \stackrel{a}{\longrightarrow}_{\mathcal{E}}^{\Diamond} Q'$ and $t''Rq'$. This is justified through an invariant of the construction, namely that for every EMTS in the range of ε, the start states of the EMTS do not have incoming transitions.

* $Q_1, \ldots, Q_m \in S_{(\varepsilon(\mu Z.\Psi))}$ such that $(\bigcup_{1 \leq j \leq m} Q_j) \cup q_i = \{q_1, \ldots, q_n\}$ and for $1 \leq j \leq m$, $t' {R'_j}^{\mathrm{V}} Q_j$ for some simulation relation $R'_j \subseteq S_{\mathcal{T}} \times S_{(\varepsilon(\mu Z.\Psi))}$. (INV 2.) There exists $Q'_1, \ldots, Q'_m$ such that for $1 \leq j \leq m$ $Q'_j \in S_{Qj}$ for some S_{Qj} where $Q_j \longrightarrow^{\Diamond a}_{\varepsilon(\mu Z.\Psi)} S_{Qj}$ and $t'' {R'_j}^{\mathrm{V}} Q'_j$.

1. If $Z \in \lambda(q')$, then we know by the definition of simulation that $t \in [\![\varepsilon(\mu Z.\Psi)]\!]^{\mathcal{T}}_{\mathrm{V}}$, then there exists a simulation relation R such that and for some $s'' \in S_1$ $t'' {R''}^{\mathrm{V}} \{s''\}$ by construction and the def. of denotation. In this case $t'' R' (\bigcup_{1 \leq j \leq m} Q'_j \cup \{q'\} \cup \{s''\})$.
2. If $Z \notin \lambda(q')$, then $t'' R' (\bigcup_{1 \leq j \leq m} Q'_j \cup \{q'\})$.

That the resulting states are actually reachable in $\varepsilon(\mu Z.\Psi)$ is by the definition of ε.

The proof that R' is a simulation for V is as follows:

Assume $t R' \{q_1, \ldots, q_n\}$ where $n > 1$, $a \in A$ and $X \in PropVar$, then by the invariants 1 and 2:

- there exists $q_i \in \{q_1, \ldots, q_n\}$ such that $q_i \notin S_1$ and $t R q_i$,
- $Q_1, \ldots, Q_m \in S_{(\varepsilon(\mu Z.\Psi))}$ such that $(\bigcup_{1 \leq j \leq m} Q_j) \cup q_i = \{q_1, \ldots, q_n\}$ and for $1 \leq j \leq m$, $t {R'_j}^{\mathrm{V}} Q_j$ for some simulation relation $R'_j \subseteq S_{\mathcal{T}} \times S_{(\varepsilon(\mu Z.\Psi))}$.

1. Straightforward.
2. If $\{q_1, \ldots, q_n\} \longrightarrow^{\Box a}_{\varepsilon(\mu Z.\Psi)} S$, then by construction there exists a $q \in \{q_1, \ldots, q_n\}$ such that $q \longrightarrow^{\Box a}_{\varepsilon(\Psi)} S'$ and $S = \partial_{\mathcal{P}}((\cup \partial^{\Diamond}_a(q_1), \ldots, S', \ldots, \cup \partial^{\Diamond}_a(q_n)), S_1, \lambda, Z)$.
   - If $q = q_i$, then by the definition of simulation there exists $t \longrightarrow^{a}_{\mathcal{T}} t'$ such that $t' R q'_i$ for some $q'_i \in S'$. Again by the definition of simulation, if $t \longrightarrow^{a}_{\mathcal{T}} t'$, then there should exist $Q'_1 \in \cup \partial^{\Diamond}_a(Q_1), \ldots, Q'_{Q-1} \in \cup \partial^{\Diamond}_a(q_{i-1}), q'_{i+1} \in \cup \partial^{\Diamond}_a(q_{i+1}), \ldots, q'_n \in \cup \partial^{\Diamond}_a(q_n)$
   - If $q \in Q_j$ for some j, then let $Q_j = \{s_1, \ldots, s_k\}$. By construction $Q_j \longrightarrow^{\Box a}_{\varepsilon(\mu Z.\Psi)} S'_j$ where $S = \partial_{\mathcal{P}}((\cup \partial^{\Diamond}_a(s_1), \ldots, S', \ldots, \cup \partial^{\Diamond}_a(s_k)), S_1, \lambda, Z)$.
3. If for $\rho_{\mu Z.\Psi} = \{q_1, \ldots, q_n\} \xrightarrow{a_1}_{\mathcal{E}} S_1^{\mu} \xrightarrow{a_2}_{\mathcal{E}} S_2^{\mu} \xrightarrow{a_3}_{\mathcal{E}} \ldots max(inf(c_{\mu}(\rho_{\mu Z.\Psi})(j)))$ is odd, where $1 \leq j \leq k$, then there does not exist an infinite run $\rho_t = t \xrightarrow{a_1}_{\mathcal{T}} t_1 \xrightarrow{a_2}_{\mathcal{T}} t_2 \xrightarrow{a_3}_{\mathcal{T}} \ldots$ such that $t_i R' S_i^{\mu}$ for all $i \geq 0$.
4. If $X \in \lambda'(\{q_1, \ldots, q_n\})$ then there is at least one member q_j of this set for which $X \in \lambda(q_j)$ and $X \neq Z$ by construction. By the invariants 1 and

2, tRq_i for some $q_i \in \{q_1, \ldots, q_n\}$ and there exists a set $Q \subseteq \{q_1, \ldots, q_n\}$ where $q_j \in Q$ and $tR''Q$ for some R'':

- If $q_j = q_i$ then $t \in \mathrm{V}^*(X)$ but since $\mathrm{V}(X) = \mathrm{V}^*(X)$ when $Z \neq X$, so $t \in \mathrm{V}(X)$, or
- Then by construction, $X \in \lambda'(Q)$, so $t \in \mathrm{V}(X)$.

□

A.1.2 Correctness of Construction for Process Terms

Proposition A.3. *Let* fix $X.E$ *be a guarded term.* $E'[\text{fix } X.E/X]\rho_A\rho_0 \xrightarrow{a}$ $E''[\text{fix } X.E/X]\rho_A\rho_0$ *if and only if* $E'[\mathbf{0}/X] \xrightarrow{a} E''[\mathbf{0}/X]$ *and* $E' \not\equiv X$.

Proof. This can be proved by induction on the structure of E'.
We will illustrate by giving one case, the others are similar. Suppose $E' \equiv a.F$. By transition rules of CCS $a.F[\mathit{fix}\ X.E/X] \xrightarrow{a}_{\mathcal{T}} F[\mathit{fix}\ X.E/X]$. Then $a.F[\mathbf{0}/X] \xrightarrow{a}_{\mathcal{T}} F[\mathbf{0}/X]$.

Suppose $E' \equiv F+G$. By transition rules of CCS either $(F+G)[\mathit{fix}\ X.E/X] \xrightarrow{a}_{\mathcal{T}} F'[\mathit{fix}\ X.E/X]$ or $(F+G)[\mathit{fix}\ X.E/X] \xrightarrow{a}_{\mathcal{T}} G'[\mathit{fix}\ X.E/X]$ and this is the case if $F[\mathit{fix}\ X.E/X] \xrightarrow{a}_{\mathcal{T}} F'[\mathit{fix}\ X.E/X]$ or $G[\mathit{fix}\ X.E/X] \xrightarrow{a}_{\mathcal{T}} G'[\mathit{fix}\ X.E/X]$ respectively. And since $G, F \not\equiv X$, we can use the induction hypothesis. Suppose $E' \equiv F \mid G$. This is the case if F and G does not contain occurrences of X, because of the assumption on the syntax of the algebra terms in the theorem. Then $(F \mid G)[\mathit{fix}\ X.E/X] \equiv (F \mid G)[\mathbf{0}/X]$. So the claim holds trivially. □

Lemma A.4. *Let* $\Gamma \rhd E$ *be a guarded OTA without composition and let* $\varepsilon(\Gamma \rhd E) = (\mathcal{E},\ S,\ \lambda)$. *Then* $[\![S]\!]_{\mathcal{U}}$ *is equal to the set* $[\![\Gamma \rhd E]\!]_{\rho_0}$ *up to bisimulation denoted* $[\![S]\!]_{\mathcal{U}} \backsimeq [\![\Gamma \rhd E]\!]_{\rho_0}$ *where* ρ_0 *maps each recursion process variable* X *to* $\mathbf{0}$.

Proof. The proof proceeds on the structure of E. Note that in the proofs below $t \approx t'$ denotes that t is bisimilar to t'. Let $\varepsilon(\Gamma \rhd E_1)=((S_{\mathcal{E}_1},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}_1},\ \longrightarrow^{\Box}_{\mathcal{E}_1},\ W_1),\ S_1,\ \lambda_1)$ and $\varepsilon(\Gamma \rhd E_2)= ((S_{\mathcal{E}_2},\ A,\ \longrightarrow^{\Diamond}_{\mathcal{E}_2},\ \longrightarrow^{\Box}_{\mathcal{E}_2},\ W_2),\ S_2,\ \lambda_2)$ be constructed as defined in figure 2.8. In the cases below, Induction Hypothesis stands for induction hypothesis and TR for transition rules of CCS.

- $E \equiv \mathbf{0}$

  $t \preceq S$
  $\iff$ $t \preceq s$ (Construction)
  $\iff$ for no $a \in A$ exists t' such that $t \xrightarrow{a}_{\mathcal{T}} t'$. (Def. A.1)
  $\iff$ $t \approx \mathbf{0}$ (Def. 2.7), $\{\mathbf{0}\} = [\![\Gamma \rhd \mathbf{0}]\!]_{\rho_0}$

- $E \equiv X$

1. $X \in AssProcVar$
   $[\![\Gamma \rhd X]\!]_{\rho_0} = \varepsilon(\bigwedge_{X:\Psi \in \Gamma} \Psi)$ and we can directly use theorem 2.13. Note that since the logical formulae Ψ are closed, the returned labeling function labels all states with the empty set.
2. $X \in RecProcVar$
   $[\![\Gamma \rhd X]\!]_{\rho_0} = \rho_0(X) = \{\mathbf{0}\}$ by Def. 2.7.

- $E \equiv a.E_1$

$t \preceq S$
$\iff$ $t \preceq s_{new}$ (Construction)
$\iff$ $\exists t'.t \xrightarrow{a}_{\mathcal{T}} t'$ where $t' \preceq S_1$ and
$\forall k \in A$ where $k \neq a$ there exists no t'' s.t. $t \xrightarrow{k}_{\mathcal{T}} t''$ (Def. A.1)
$\iff$ $\exists t'.t \xrightarrow{a}_{\mathcal{T}} t'$ where $t' \approx u'$ where $u' \in [\![\Gamma \rhd E_1]\!]_{\rho_0}$ and $\forall k \in A$
where $k \neq a$ there exists no t'' s.t. $t \xrightarrow{k}_{\mathcal{T}} t''$ (Induction Hyp.)
$\iff$ $\exists u'.t \approx a.u'$ and $u' \in [\![\Gamma \rhd E_1]\!]_{\rho_0}$
$\iff$ $\exists u.t \approx u$ and $u \in [\![\Gamma \rhd a.E_1]\!]_{\rho_0}$ (Def. 2.7)

- $E \equiv E_1 + E_2$

$t \preceq S$
$\iff$ $\exists s_1 \in S_{v_1}, r_1 \in S_{v_2}.\ t \preceq (s_1, r_1)$
$\iff$ $\exists t_1, u_1, s_1 \in S_{v_1}, r_1 \in S_{v_2}.t \approx t_1 + u_1 \wedge t_1 \preceq s_1$
$\wedge u_1 \preceq r_1$ (See below)
$\iff$ $\exists t_1, u_1.t \approx t_1 + u_1 \wedge (\exists v_1.t_1 \approx v_1 \wedge v_1 \in [\![\Gamma \rhd E_1]\!]_{\rho_0})$
$\wedge(\exists v_2.u_1 \approx v_2 \wedge v_2 \in [\![\Gamma \rhd E_2]\!]_{\rho_0})$ (Induction Hyp.)
$\iff$ $\exists v_1, v_2.t \approx v_1 + v_2 \wedge v_1 \in [\![\Gamma \rhd E_1]\!]_{\rho_0} \wedge v_2 \in [\![\Gamma \rhd E_2]\!]$
$\iff$ $\exists v_1, v_2.t \approx v_1 + v_2 \wedge v_1 + v_2 \in [\![\Gamma \rhd E_1 + E_2]\!]_{\rho_0}$
$\iff$ $\exists v.t \approx v \wedge v \in [\![\Gamma \rhd E_1 + E_2]\!]_{\rho_0}$ (Def. 2.7)

The second equivalence is established by separate proofs of the two directions:

($\Rightarrow$)Consider some t such that $t \preceq (s_1, r_1)$ where $s_1 \in S_{v_1}$ and $r_1 \in S_{v_2}$. By the Expansion theorem and by our construction definition, we take $t \approx (a_0.t_0 + \ldots + a_k.t_k) + (a_{k+1}.t_{k+1} + \ldots + a_n.t_n)$ for some processes $t_i \in \mathcal{U}$ and where for each $a_i.t_i$ in this sum, $\exists S'.(s_1, r_1) \xrightarrow{a_i}{}^{\Diamond}_{\mathcal{E}} S'$ and $t_i \preceq S'$ where $S' \subseteq S_{\mathcal{E}_1}$ if $0 < i \leq k$, and $S' \subseteq S_{\mathcal{E}_2}$ if $k < i \leq n$ by the definition of simulation. The arguments for $(a_0.t_0 + \ldots + a_k.t_k) \preceq S_{v_1}$ and $(a_{k+1}.t_{k+1} + \ldots + a_n.t_n) \preceq S_{v_2}$ are then trivial.

($\Leftarrow$) By Induction Hypothesis there exists simulation relations R_1 and R_2 between elements of $[\![\Gamma \rhd E_1]\!]_{\rho_0}$ and $\mathcal{E}_{v_1}$, $[\![\Gamma \rhd E_2]\!]_{\rho_0}$ and $\mathcal{E}_{v_2}$ respectively. We define the relation $R' \subseteq S_{\mathcal{U}} \times S_{\mathcal{E}}$ using R_1 and R_2 as follows:

$$tR'q \triangleq \begin{cases} t_1R_1s_1 \wedge t_2R_2r_1 & \text{if } t = t_1 + t_2 \text{ and } q = (s,r) \text{ for } s_1 \in S_{\upsilon_1}, r_1 \in S_{\upsilon_2} \\ tR_1q & \text{if } q \in S_{\mathcal{E}_1} \\ tR_2q & \text{if } q \in S_{\mathcal{E}_2} \end{cases}$$

We show that R' is a simulation relation.

Assume $tR'q$. The argument is obvious if $q \notin S_{\upsilon_1} \times S_{\upsilon_2}$. So we assume $q = (s_1, r_1)$ for some $s_1 \in S_{\upsilon_1}$ and some $r_1 \in S_{\upsilon_2}$.

(i)(a) $\exists t'.(t_1 + u_1) \xrightarrow{a}_{\mathcal{T}} t'$

$\Rightarrow$ $\exists t'.(t_1 \xrightarrow{a}_{\mathcal{T}} t' \vee u_1 \xrightarrow{a}_{\mathcal{T}} t')$ (TR)

$\Rightarrow$ $\exists S'.(s_1 \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S' \wedge t'R_1S')$ or $\exists S'.(r_1 \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S' \wedge t'R_2S')$ (Def A.1)

$\Rightarrow$ $\exists S'.((s_1, r_1) \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S') \wedge t'RS'$ (Construction)

(i)(b) $\exists S'.(s_1, r_1) \xrightarrow{a}^{\Box}_{\mathcal{E}} S'$

$\Rightarrow$ $\exists S'.s_1 \xrightarrow{a}^{\Box}_{\mathcal{E}_1} S'$ or $\exists S'.r_1 \xrightarrow{a}^{\Box}_{\mathcal{E}_2} S'$ (Construction)

$\Rightarrow$ $\exists t'.t_1 \xrightarrow{a}_{\mathcal{T}} t' \wedge t'R_1S'$ or $\exists t'.u_1 \xrightarrow{a}_{\mathcal{T}} t' \wedge t'R_2S'$ (Def. 2.9)

$\Rightarrow$ $t_1 + u_1 \xrightarrow{a}_{\mathcal{T}} t' \wedge t'RS'$

(ii) We show that if $(s_1, r_1) \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots \in W$, then $\rho_t = t \xrightarrow{a_1}_{\mathcal{T}} t_1 \xrightarrow{a_2}_{\mathcal{T}} t_2 \ldots$ where t_iRq_i for all $i > 1$ is not a run of t.

Assume such a ρ_t exists, and let $t = u_1 + u_2$ where $u_1R_1s_1$ and $u_2R_2r_1$. By the construction either $s_1 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots \in W_{\upsilon_1}$ or $r_1 \xrightarrow{a_0} q_1 \xrightarrow{a_2} q_2 \ldots \in W_{\upsilon_2}$. Assume it is the first case. Then t_iRq_i where $q_i \in S_{\mathcal{E}_1}$ for all $i > 0$. For this to be the case, $t_iR_1q_i$ for all $i > 0$. Then $u_1 \xrightarrow{a_0}_{\mathcal{T}} t_1 \xrightarrow{a_2}_{\mathcal{T}} t_2 \ldots$ is an infinite run where $t_iR_1q_i$ for all $i > 0$ but such an infinite run can not exist by the definition of simulation.

- $E \equiv \mathit{fix}\ X.E_1$

$v \in [\![\Gamma \rhd \mathit{fix}\ X.E_1]\!]_{\rho_0}$

$\iff$ $v \equiv (\mathit{fix}\ X.E_1\rho_A\rho_0)$ for some ρ_A (Def. 2.7)

$\iff$ $v \equiv X[\mathit{fix}\ X.E_1/X]\rho_A\rho_0$ for some ρ_A

$\iff$ $\exists t.t \equiv E_1[0/X]\rho_A\rho_0 \wedge t \in [\![\Gamma \rhd E_1]\!]_{\rho_0}$ (Def. 2.7)

$\iff$ $\exists s_1 \in S_1.t \equiv E_1[0/X]\rho_A\rho_0 \wedge t \preceq s_1$ (Def. 2.9)

$\iff$ $\exists s_1 \in S_1.v \preceq s_1$ (See below)

We establish the last equivalence as follows. By induction hypothesis, there exists a simulation relation R_1 between processes in $[\![\Gamma \rhd E_1]\!]_{\rho_0}$ and states of $\mathcal{E}_1$. Using this relation R_1, we define $R \subseteq S_{\mathcal{U}} \times S_{\mathcal{E}}$ as follows:

$$(E_1'[\mathit{fix}\ X.E_1/X])\rho_A\rho_0Rs \triangleq \begin{cases} E_1'[\mathbf{0}/X]\rho_A\rho_0\ R_1\ s & \text{if } E_1' \not\equiv X \\ E_1[\mathbf{0}/X]\rho_A\rho_0\ R_1\ s & \text{if } E_1' \equiv X \end{cases}$$

We show that R is a simulation relation.
Assume $E_1'[\mathit{fix}\ X.E_1/X]\rho_A\rho_0 R s$
(i)(a) First let us take the case where $E_1' \equiv X$, then
$E_1'[\mathit{fix}\ X.E/X]\rho_A\rho_0 \equiv (\mathit{fix}\ X.E_1)\rho_A\rho_0$.

$(\mathit{fix}\ X.E_1)\rho_A\rho_0 t \xrightarrow{a}_{\mathcal{T}} E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0$
$\iff E_1[\mathit{fix}\ X.E_1/X]\rho_A\rho_0 t \xrightarrow{a} E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0$ (TR)
$\iff E_1[0/X]\rho_A\rho_0 \xrightarrow{a}_{\mathcal{T}} E_1''[0/X]\rho_A\rho_0$ ($\mathit{fix}\ X.E_1$ is guarded, Prop. A.3)
$\iff E_1[0/X]\rho_A\rho_0 \in [\![\Gamma \rhd E_1]\!]_{\rho_0}$
$\iff s \in S_1 \wedge E_1[0/X]\rho_A\rho_0 R_1 s$ (Def. 2.7)
$\iff \exists S'.s \xrightarrow{a}{}^{\Diamond}_{\mathcal{E}} S'$ where $E_1''[0/X]\rho_A\rho_0 R_1 s'$ for some s' in S' (Def. 2.9)
$\iff E_1''[\mathit{fix}\ X.E/X]\rho_A\rho_0 R s'$

Argument for $E_1' \not\equiv X$ is similar.

b)First let us take the case where $s \in S_1$.

$s \xrightarrow{a}{}^{\Box}_{\mathcal{E}} S'$
$\iff$ $E_1[0/X]\rho_A\rho_0 \xrightarrow{a}_{\mathcal{T}} E_1''[0/X]\rho_A\rho_0$
and $E_1''[0/X]\rho_A\rho_0 R_1 s'$ for some $s' \in S'$(Def. 2.9)
$\iff$ $E_1[\mathit{fix}\ X.E_1/X]\rho_A\rho_0 \xrightarrow{a}_{\mathcal{T}} E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0$
and $E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0 R_1 s'$ for some $s' \in S'$(Prop. A.3, $E_1 \not\equiv X$)
$\iff$ $\mathit{fix}\ X.E_1\rho_A\rho_0 \xrightarrow{a} E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0$ and $E_1''[\mathit{fix}\ X.E_1/X]\rho_A\rho_0 R s'$
for some $s' \in S'$ (TR)

Argument for $s \notin S_1$ is similar.

2) Take some prohibited run of s_1, $s_1 \xrightarrow{a_1}_{\mathcal{E}} s_2 \ldots \xrightarrow{a_{n-1}}_{\mathcal{E}} s_n).\rho_{s_n}$ where $n \geq 1$ and $\rho_{s_n} \in W_1$. For all mentioned states s_m in this run, where $m > n$, $s_m \notin S_1$. Suppose there is an infinite run of $t = E'[\mathit{fix}\ X.E/X]\rho_A\rho_0$: $(t \xrightarrow{a_1}_{\mathcal{T}} t_2 \ldots \xrightarrow{a_{n-1}}_{\mathcal{T}} t_n).\rho_{t_n}$, such that for all $i > 1$, $t_i R s_i$.

Assume $s_n \in S_1$. We have $E[\mathbf{0}/X]\rho_A\rho_0 R' s_n$ and so
$t_n = X[\mathit{fix}\ X.E/X]\rho_A\rho_0 R s_n$ by Induction Hypothesis. Then, the infinite run ρ_{t_n} has the form $X[\mathit{fix}\ X.E/X]\rho_A\rho_0 \xrightarrow{a_n}_{\mathcal{T}} E_{n+1}[\mathit{fix}\ X.E/X]\rho_A\rho_0 \xrightarrow{a_{n+1}}_{\mathcal{T}} \ldots$ where for all $i > 1$, $E_{n+i}[\mathit{fix}\ X.E/X]\rho_A\rho_0 R s_{n+i}$ and $s_{n+i} \notin S_1$. By the way we defined R, this is possible if for all $i > 1$, $E_{n+i}[\mathbf{0}/X]\rho_A\rho_0 R' s_{n+i}$ and $E_{n+1} \not\equiv E$. Then we can construct a run of $E[\mathbf{0}/X]\rho_A\rho_0$, if we replace the *fix* expression substitution at each process with a $\mathbf{0}$ substitution: $E[\mathbf{0}/X]\rho_A\rho_0 \xrightarrow{a_n}_{\mathcal{T}} E_{n+1}[\mathbf{0}/X]\rho_A\rho_0 \xrightarrow{a_{n+1}}_{\mathcal{T}} \ldots$ where $E[\mathbf{0}/X]\rho_A\rho_0 R' s_n$ and for all $i > 1$, $E_{n+i}[\mathbf{0}/X]\rho_A\rho_0 R' s_{n+i}$. This run is a legal run of $E[\mathbf{0}/X]\rho_A\rho_0$ in $\mathcal{E}_1$, since the transitions between these terms exist also in $\mathcal{E}_1$ (Proposition. A.3, TR) This results in an infinite run of $E[\mathbf{0}/X]\rho_A\rho_0$ which is simulated by ρ_{s_n}, but this is impossible by Def. 2.9 since $\rho_{s_n} \in W_1$. Hence no such infinite run simulated by ρ_{s_1} is possible. The case for $s_n \notin S_1$ is similar.

□

Lemma A.5. *Let $\mathcal{T}$ be a transition-closed LTS, $\Gamma \rhd E_1 \parallel E_2$ be a guarded linear OTA where every recursion process variable in the scope of parallel composition is bound by a* fix *operator in the same scope, and let $\varepsilon(\Gamma \rhd E) = (\mathcal{E}, S, \lambda)$. Then the set $[\![S]\!]_{\mathcal{T}}$ includes $[\![\Gamma \rhd E_1 \parallel E_2]\!]_{\rho_0}$ up to bisimulation.*

Proof. Let $\varepsilon(\Gamma \rhd E_1)$=$((S_{\mathcal{E}_1}, A, \longrightarrow^{\Diamond}_{\mathcal{E}_1}, \longrightarrow^{\Box}_{\mathcal{E}_1}, W_1), S_1, \lambda_1)$ and $\varepsilon(\Gamma \rhd E_2)$= $((S_{\mathcal{E}_2}, A, \longrightarrow^{\Diamond}_{\mathcal{E}_2}, \longrightarrow^{\Box}_{\mathcal{E}_2}, W_2), S_2, \lambda_2)$ be constructed as defined in figure 2.8.

$$
\begin{array}{rl}
& v \in [\![\Gamma \rhd E_1 \| E_2]\!]_{\rho_0} \\
\iff & \exists t_1, u_1.\, v \approx t_1 \| u_1 \wedge\ t_1 \in [\![\Gamma \rhd E_1]\!]_{\rho_0} \wedge u_1 \in [\![\Gamma \rhd E_2]\!]_{\rho_0} \text{ (linearity)} \\
\Rightarrow & \exists t_1, u_1, s_1, r_1.\, v \approx t_1 \| u_1 \wedge\ s_1 \in S_1 \wedge r_1 \in S_2 \wedge \\
& t_1 \preceq s_1 \wedge u_1 \preceq r_1 \text{ (Induction Hyp.)} \\
\Rightarrow & \exists s_1, r_1, x_1.\ \ v \preceq (s_1, r_1, x_1) \text{(See below)}
\end{array}
$$

In order to show the last implication we define a relation $R \subseteq S_{\mathcal{U}} \times S_{\mathcal{E}}$ (see below). By Induction Hypothesis, simulation relations R_1 and R_2 exist between elements of $[\![\Gamma \rhd E_1]\!]_{\rho_0}$, $[\![\Gamma \rhd E_2]\!]_{\rho_0}$ and states of $\mathcal{E}_1$, $\mathcal{E}_2$ respectively.

R is the least relation that satisfies the following:

- if tR_1s_1 and uR_2r_1 where $s_1 \in S_1$ and $r_1 \in S_2$, then $(t\|u)R(s_1, r_1, x)$ for all $x \in \{1, 2\}$
- if tR_1s and uR_2r and $t \xrightarrow{a}_{\mathcal{T}} t'$ and $t'R_1s'$, then $(t'\|u)R(s', r, 1)$
- if tR_1s and uR_2r and $u \xrightarrow{a}_{\mathcal{T}} u'$ and $u'R_2r'$, then $(t\|u')R(s, r', 2)$.

We claim that the relation R is a simulation relation.
Assume $(t\|u)\ R\ (s, r, x)$ for some $x \in \{1, 2\}$.

1. Whenever $a \in A$,
   a) If $(t\|u) \xrightarrow{a}_{\mathcal{T}} (t'\|u')$, then by TR either $t \xrightarrow{a}_{\mathcal{T}} t'$ and $u' = u$ or $u \xrightarrow{a}_{\mathcal{T}} u'$ and $t' = t$. In the first case, tR_1s by assumption. Then, by the Def. 2.9 there is a $S_1' \subseteq S_{\mathcal{E}_1}$ such that $s \xrightarrow{a}^{\Diamond}_{\mathcal{E}_1} S_1'$ and $t'R_1s'$ for some $s' \in S_1'$. So $(s, r) \xrightarrow{a}^{\Diamond}_{\mathcal{E}} S$, where $S = \{(s', r, 1) \| s_1' \in S_1'\}$. Again by assumption uR_2r, then $(t'\|u)\ R\ (s', r, 1)$. The second case is similar.
   b) If $(s, r, x) \xrightarrow{a}^{\Box}_{\mathcal{E}} S$, then either $s \xrightarrow{a}^{\Box}_{\mathcal{E}_1} S_1'$ and $S = \{(s', r, 1) \| s_1' \in S_1')\}$ or $r_1 \xrightarrow{a}^{\Box}_{\mathcal{E}_2} S_2'$ and $S = \{(s, r', 2) \| r' \in S_2'\}$. In the first case, tR_1s by assumption. Then, Def. 2.9 there is t' such that $t \xrightarrow{a}_{\mathcal{T}} t'$ and $t'R_1s'$ for some $s' \in S_1'$. Then by TR, $(t \mid u) \xrightarrow{a}_{\mathcal{T}} (t'\|u)$. Again by assumption uR_2r. Then $(t'\|u)\ R\ (s', r, 1)$. The second case is similar.
2. If $\rho_{(s,r,x)} = (s, r, x) \xrightarrow{a_1}_{\mathcal{E}} (s_1, r_1, x_1) \xrightarrow{a_2}_{\mathcal{E}} (s_2, r_2, , x_2) \xrightarrow{a_3}_{\mathcal{E}} \ldots$ is in W, then no infinite run $\rho_{(t\|u)} = t\|u \xrightarrow{a_1}_{\mathcal{E}} t_1\|u_1 \xrightarrow{a_2}_{\mathcal{E}} t_2\|u_2 \xrightarrow{a_3}_{\mathcal{E}} \ldots$ such that $(t_i\|u_i)\ R\ (s_i, r_i, x_i)$ for all $i \geq 1$ can exist. Suppose that such a run of $(t\|u)$ exists. This run would clearly be an interleaving of runs of t and u, where at least one

of these runs are infinite. Since $\rho_{(s,r,x)}$ is prohibited and $S_{\varepsilon(\Gamma \rhd E_1 \| E_2)}$ is finite, there exists an infinitely occurring state (s', r', x') in this run such that for some $1 \leq j \leq k$, the color entry of this state $c(s', r', x')(j)$ is odd and larger than the other infinitely occurring integers in the $j^t h$ entry of $c(\rho_{(s,r,x)})$.

If $1 \leq j \leq k_1$, then we know that the infinitely occurring state (s', r', x') is a state where the last transition was performed by the first component, so $x = 1$. By just selecting from the run $\rho_{(s,r,x)}$ those transitions which are followed by some state labeled with 1 and the first component of these states, we can extract a run ρ_s of the first component. By a similar selection of the first component from the same positions of $\rho_{(t\|u)}$, we can build an infinite run of t. By our assumption these two runs follow each other, although ρ_s is in W, ρ_t is not. But again by assumption tR_1s, so we reach a contradiction. Same argument then applies to u if $k_1 < j \leq k_2 + k_1$.

□

Theorem 2.14 Let $\mathcal{T}$ be a transition-closed LTS, $\Gamma \rhd E \| t$ be a guarded linear OTA where E does not contain parallel composition, and t is closed, and let $\varepsilon(\Gamma \rhd E \parallel t) = (\mathcal{E}, S, \lambda)$. Then $[\![S]\!]_{\mathcal{T}}$ is equal to the set $[\![\Gamma \rhd E \parallel t]\!]_{\rho_0}$ up to bisimulation, where ρ_0 maps each recursion process variable X to $\mathbf{0}$.

Proof. This is similar to the proof of lemma A.4 with the additional case of the parallel composition.

Let $\varepsilon(\Gamma \rhd E)=((S_{\mathcal{E}_1}, A, \longrightarrow^{\Diamond}_{\mathcal{E}_1}, \longrightarrow^{\Box}_{\mathcal{E}_1}, W_1), S_1, \lambda_1)$ and $\varepsilon(\Gamma \rhd t)= ((S_{\mathcal{E}_2}, A, \longrightarrow^{\Diamond}_{\mathcal{E}_2}, \longrightarrow^{\Box}_{\mathcal{E}_2}, W_2), S_2, \lambda_2)$ be constructed as defined in figure 2.8. The *may* and *must* transitions of $\varepsilon(\Gamma \rhd t)$ coincide, and all colors are 0 since no maximal model construction is done for this case.

$$
\begin{array}{ll}
v \preceq S & \\
\iff & \exists s_1 \in S_1, r_1 \in S_2, x_1 \in \{1,2\}.\ v \preceq (s_1, r_1, x_1) \text{ (Construction)} \\
\iff & \exists t_1, u_1, s_1 \in S_1, r_1 \in S_2.\ v \approx t_1 \| u_1 \wedge t_1 \preceq s_1 \wedge u_1 \preceq r_1 \text{ (See below)} \\
\iff & \exists t_1, u_1.\ v \approx t_1 \| u_1 \wedge\ t_1 \in [\![\Gamma \rhd E]\!]_{\rho_0} \wedge \\
 & u_1 \in [\![\Gamma \rhd t]\!]_{\rho_0} \text{ (Induction Hypothesis)} \\
\iff & v \in [\![\Gamma \rhd E \| t]\!]_{\rho_0}
\end{array}
$$

We show the second equality in two directions:

($\Rightarrow$) For this case, t_1 and u_1 can be constructed inductively, using the state space of (s_1, r_1, x_1). The process u_1 is to be chosen bisimilar to t. At each step we know which component will make a transition through the last entry of the tuple, x.

($\Leftarrow$) Corollary of lemma A.5. □

Theorem 2.15 Let $\mathcal{T}$ be a transition-closed LTS, $\Gamma \rhd E$ be a guarded linear OTA where every recursion process variable in the scope of parallel composition is bound by a *fix* operator in the same scope, and let $\varepsilon(\Gamma \rhd E) = (\mathcal{E}, S, \lambda)$. Then the set $[\![S]\!]_{\mathcal{T}}$ includes $[\![\Gamma \rhd E]\!]_{\rho_0}$ up to bisimulation.

Proof. This is a direct corollary of lemma A.4 and lemma A.5. □

A.1.3 Soundness and Completeness of the Proof System

Proposition A.6. *For any $s \in S_{\mathcal{E}}$ and LTS $\mathcal{T}$, $\partial_a([\![s]\!]_{\mathcal{T}}) \subseteq [\![\cup \partial_a^{\Diamond}(s)]\!]_{\mathcal{T}}$*

Proof. Assume $t' \in \partial_a([\![s]\!]_{\mathcal{T}})$. Then, by the notion of a-derivatives for LTS, there is a $t \in [\![s]\!]_{\mathcal{T}}$ s.t. $t \xrightarrow{a}_{\mathcal{T}} t'$. Then, by the definition of denotation, $t \preceq s$ and hence, by the definition of simulation, there are S' and $s' \in S'$ such that $s \xrightarrow{a}_{\mathcal{E}}^{\Diamond} S'$ and $t' \preceq s'$. Therefore $t' \in [\![s']\!]_{\mathcal{T}}$ and hence, by the definition of a-derivatives for EMTS, $t' \in [\![\cup \partial_a^{\Diamond}(s)]\!]_{\mathcal{T}}$. □

Lemma 2.22 *For each rule instance in $\Sigma_{\mathcal{E}}$, translation π assigns a correct proof tree in $\Sigma_{\mathcal{T}}$, so that each premise (resp. conclusion), $s \vdash_{\mathrm{V}}^{\mathcal{E}} \Phi$, of the rule is matched by a leaf (resp. root), $[\![s]\!]_{\mathcal{T}} \vdash_{\mathrm{V}}^{\mathcal{T}} \Phi$, and all unmatched leaves of the constructed proof tree are successful terminals.*

Proof. We consider each rule of $\Sigma_{\mathcal{E}}$ in turn. The cases for **$\wedge$-rule**, **$\vee$-rule**, **σ-rule**, and **Z-rule** are trivial.

For the **$\Box_a$-rule**, in the first case we have $\cup\partial_a^{\Diamond}(s) = \varnothing$ Then $\Box_a$ rule is applied to node $[\![r]\!]_{\mathcal{T}} \vdash_{\mathrm{V}}^{\mathcal{T}} [a]\,\Psi'$ in $\Sigma_{\mathcal{T}}$. Since none of the states implementing r have a-derivatives, the resulting node is $\varnothing \vdash_{\mathrm{V}}^{\mathcal{T}} \Psi'$, which is a successful terminal. For the second case, we use $\partial_a([\![s]\!]_{\mathcal{T}}) \subseteq [\![\cup\partial_a^{\Diamond}(s)]\!]_{\mathcal{T}}$ by Proposition A.6. Then, the Thin rule is only applied if the subset relation between the range of f_a and $[\![\{s_1 \ldots s_n\}]\!]_{\mathcal{T}}$ is proper. Finally, a sequence of Cut-rules are applied.

For the **$\Diamond_a$-rule**, by the side condition we have $s \xrightarrow{a}_{\mathcal{E}}^{\Box} \{s_1, \ldots, s_n\}$, and hence, by the definitions of simulation and denotation, for every $t \in [\![s]\!]_{\mathcal{T}}$ there must be a $s' \in \{s_1, \ldots, s_n\}$ and $t' \in [\![s']\!]_{\mathcal{T}}$ such that $t \xrightarrow{a}_{\mathcal{T}} t'$. Then the mapping f_a mapping each $t \in [\![s]\!]_{\mathcal{T}}$ to such a corresponding t' is a valid choice function for rule $\Diamond_a$ of $\Sigma_{\mathcal{T}}$. Again the Thin rule is applied only when the subset relation is a proper one. The applications of the Cut-rules follow to split $[\![\{s_1, \ldots, s_n\}]\!]_{\mathcal{T}}$ to the denotation of individual states. □

Corollary 2.23 For proof tree $A_{\mathcal{E}}$ with root sequent $s \vdash_{\mathrm{V}}^{\mathcal{E}} \Phi$ in $\Sigma_{\mathcal{E}}$, $A_{\mathcal{T}} = \pi_{\mathcal{T}}(A_{\mathcal{E}})$ is a proof tree with root sequent $[\![s]\!]_{\mathcal{T}} \vdash_{\mathrm{V}}^{\mathcal{T}} \Phi$ in $\Sigma_{\mathcal{T}}$, such that each leaf $s_i \vdash_{\mathrm{V}}^{\mathcal{E}} \Phi_i$ of $A_{\mathcal{E}}$ is matched by a leaf $[\![s_i]\!]_{\mathcal{T}} \vdash_{\mathrm{V}}^{\mathcal{T}} \Phi_i$ in $A_{\mathcal{T}}$, and all unmatched leaves in $A_{\mathcal{T}}$ are successful terminals.

Proof. Follows directly from lemma 2.22 by induction on the depth of $A_{\mathcal{E}}$. □

Canonical Proof Constructions and the Matching Function γ. Let $\mathcal{E}$ be a finite-state EMTS, $s \in S_{\mathcal{E}}$, and Φ have prime subformulas only. If $s \models_{\mathrm{V}}^{\mathcal{E}} \Phi$, then the construction process of the proof trees $A_{\mathcal{U}}$ and $A_{\mathcal{U}}^*$ in the weakened version of $\Sigma_{\mathcal{T}}$ for the goal $[\![s]\!]_{\mathcal{U}} \vdash_{\mathbf{V}}^{\mathcal{U}} \Phi$ is presented below. The construction is described through the matching function $\gamma : \Gamma_N \to \Gamma_M \cup \{\bot\}$, where Γ_N and Γ_M are the node spaces

of $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$, respectively and $\perp$ stands for the "undefined value". The definition of γ is given as the construction progresses.

Construction

1. The root nodes n_0 of $A_{\mathcal{U}}$ and m_0 of $A^*_{\mathcal{U}}$ both contain the sequent: $[\![s]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi$. $(\gamma(n_0) \triangleq m_0)$

2. Thin rule is applied to n_0 to produce the subgoal $n_1 : ||\Phi||^{\mathcal{U}}_{\mathbf{V}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi$. $(\gamma(n_1) \triangleq m_0)$

3. If the current subgoal in $A_{\mathcal{U}}$ is $n : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$:

   a) $\gamma(n) = m$, where $m : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$

   - $\Psi = \Psi_1 \wedge \Psi_2$: Apply $\wedge$-rule to n and $\wedge$-macro to m to get the new subgoals $n^1 : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_1$,$n^2 : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_2$ and $m^1 : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_1$, $m^2 : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_2$ respectively. $(\gamma(n^1) \triangleq m^1$ and $\gamma(n^2) \triangleq m^2)$.
   - $\Psi = \Psi_1 \vee \Psi_2$: Pick Ψ_i where $i \in 1, 2$ so that Ψ implies Ψ_i. Apply $\vee$-rule to n and $\vee$-macro to m to get the new subgoals $n' : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_i$ and $m' : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi_i$ respectively. $(\gamma(n') \triangleq m')$
   - $\Psi = [a]\,\Psi'$: Apply $\Box$-rule to n and the corresponding $\Box$-macro[1] to m according to whether $\partial^{\Diamond}_a(s_m) = \varnothing$. Let the *immediate* subgoals of n and m be n' and m', respectively. $(\gamma(n') \triangleq m')$
     If $\cup\partial^{\Diamond}_a(s_m) = \{s_1 \dots s_k\}$ where $k > 1$, then apply $k-1$ consecutive Cut-rules to n'. For every application, if the current subgoal is n_j, then the newly produced subgoals are $n_{j+1} : \partial_a(S_n) \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ and $n_{j+2} : \partial_a(S_n) \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ with the subgoals of $\gamma(n_j)$ being $m_{j+1} : [\![\{s_1 \dots s_{k'-1}\}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ and $m_{j+2} : [\![s_{k'}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ where $1 \leq k' \leq k$. Apply the next Cut rule to n_{j+1}. $(\gamma(n_{j+1}) \triangleq m_{j+1}$ and $\gamma(n_{j+2}) \triangleq m_{j+2})$
   - $\Psi = \langle a \rangle\,\Psi'$: Pick $S' \in \partial^{\Box}_a(s_m)$ with for all $s' \in S'$, $s' \models^{\mathcal{E}}_{\mathrm{V}} \Psi'$. Apply $\Diamond$-rule to n and $\Diamond$-macro[2] to m using the choice functions f_a and $f^{\mathcal{E}}_a$ respectively, where f_a is some choice function that preserves validity with $f_a|_{[\![s_m]\!]_{\mathcal{U}}} = f^{\mathcal{E}}_a$ and $f^{\mathcal{E}}_a$ maps each $t \in [\![s_m]\!]_{\mathcal{U}}$ to some $t' \in [\![S']\!]_{\mathcal{U}}$ such that $t \xrightarrow{a} t'$ and there exists $s' \in S'$ where s' simulates t'. Let the *immediate* subgoals of n and m be n' and m', respectively. $(\gamma(n') \triangleq m')$
     If $S' = \{s_1 \dots s_k\}$ where $k > 1$, then apply $k-1$ consecutive Cut-rules to n'. For every application, if the current subgoal is n_j, then the newly produced subgoals are $n_{j+1} : f_a(S_n) \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$

[1] No application of the Thin rule is needed in this macro. See Proof of Proposition A.6
[2] No application of the Thin rule is needed in this macro. See proof of Proposition A.7

and $n_{j+2} : f_a(S_n) \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ with the subgoals of $\gamma(n_j)$ being $m_{j+1} : [\![\{s_1 \ldots s_{k'-1}\}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ and $m_{j+2} : [\![s_{k'}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ where $1 \leq k' \leq i$. Apply the next Cut rule to n_{j+1}. ($\gamma(n_{j+1}) \stackrel{\Delta}{=} m_{j+1}$ and $\gamma(n_{j+2}) \stackrel{\Delta}{=} m_{j+2}$)

- $\Psi = \sigma Z.\Psi'$: Apply the Thin rule to n to get the new subgoal n''. ($\gamma(n'') \stackrel{\Delta}{=} m$)
  Apply σZ rule to n'' and σZ macro to m to get the new subgoals n' and m', respectively. ($\gamma(n') \stackrel{\Delta}{=} m'$)
- $\Psi = Z$ where Z identifies the fixed point formula $\sigma Z.\Psi'$: Apply Z rule to n and Z macro to m to get the new subgoals n' and m', respectively. ($\gamma(n') \stackrel{\Delta}{=} m'$)

Let $n^1 \ldots n^i$ where $i \in \{1, 2\}$ be the current subgoals in $A_{\mathcal{U}}$:
Test for each subgoal n^j if it is a terminal using the conditions below. If n^j is a terminal than no further rules are applied to $\gamma(n^j)$. If n^j is not a terminal, repeat Step (iii) for each n^j.

b) $\gamma(n) = m$, where $m : \varnothing \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$, which means m is a successful terminal of $A^*_{\mathcal{U}}$ and will not be applied any further rules.
Apply to n the corresponding rule according to the structure of Ψ as in a). If the rule to be applied is $\Diamond$, the choice function f can be picked as any that preserves validity.
Let $n^1 \ldots n^i$ where $i \in \{1, 2\}$ be the subgoals of n. A subgoal n^j of n is a terminal if it obeys one of the termination conditions for the original proof system, stated on page 36. For the subgoals that are not terminals the process is repeated beginning from Step (iii).($\gamma(n^j) \stackrel{\Delta}{=} \perp$ where $1 \leq j \leq i$)

c) $\gamma(n) = \perp$ Apply corresponding rule according to the structure of Ψ as in a). If the rule to be applied is $\Diamond$, the choice function f can be picked as any that preserves validity.
Let $n^1 \ldots n^i$ where $i \in \{1, 2\}$ be the subgoals of n. A subgoal n^j of n is a terminal if it obeys one of the termination conditions for the original proof system, stated on page 36. For the subgoals that are not terminals the process is repeated beginning from Step (iii).($\gamma(n^j) \stackrel{\Delta}{=} \perp$ where $1 \leq j \leq i$)

Successful Termination for node $n' : S'_n \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi'$ of $A_{\mathcal{U}}$:

1. $\Psi' = \mathrm{tt}$, or else $\Psi' = Z$, Z is free in the initial formula, and $S'_n \subseteq \mathbf{V}(Z)$
2. $S'_n = \varnothing$
3. $\Psi' = Z$ where Z identifies a fixed point formula $\sigma Z.\Phi$, then this sequent is a σ-terminal if node $\gamma(n') = m'$ in $A^*_{\mathcal{U}}$ where $m' : [\![s_{m'}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ is a σ-terminal

with companion node m'', which mentions same sequent $m'' : [\![s_{m'}]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi'$ and the companion node of n is $\gamma^{-1}(m'')$. The terminal is successful when $\sigma = \nu$. If $\sigma = \mu$, then the terminal is successful if there is no infinite chain of composable trails $T_0 \circ T_1 \circ T_2 \ldots$ of $\gamma^{-1}(m'')$ and n'.

It can be shown that the matching function γ is surjective, that is for each node m of $A^*_{\mathcal{U}}$ there exists at least one node n of $A_{\mathcal{U}}$ such that $\gamma(n) = m$. Furthermore, one can say that for all nodes m of $A^*_{\mathcal{U}}$, the set $\gamma^{-1}(m)$ is either a singleton or $\gamma^{-1}(m)$ has two elements n_1 and n_2, when the rule applied to n_1 is a Thin, with n_2 as the resulting subgoal.

Proposition A.7. *Let $\mathcal{E}$ be an EMTS. For universal LTS $\mathcal{U}$, $s \in S_{\mathcal{E}}$, for every $S' \in \partial^{\Box}_a(s)$, there exists a choice function f_a such that $f_a([\![s]\!]_{\mathcal{U}}) = [\![S']\!]_{\mathcal{U}}$.*

Proof. By definition 2.9, for every $t \in [\![s]\!]_{\mathcal{U}}$ there exists $t' \in [\![S']\!]_{\mathcal{U}}$ such that $t \xrightarrow{a}_{\mathcal{T}} t'$. So we know that at least one choice function f_a exists which takes the elements of $[\![s]\!]_{\mathcal{U}}$ to a subset of $[\![S']\!]_{\mathcal{U}}$. What is more, we can construct such an f_a whose range covers all elements of $[\![S']\!]_{\mathcal{U}}$: Take any $t \in [\![s]\!]_{\mathcal{U}}$ and let $t' \in \partial_a(t)$ and $t \in [\![S']\!]_{\mathcal{U}}$. For each state $t'' \in [\![S']\!]_{\mathcal{U}}$, define $f_a(t_{new}) = t''$ where t_{new} is a state that has all transitions of t plus the transition $t_{new} \xrightarrow{a}_{\mathcal{T}} t''$. It is clear that each t_{new} defined in this manner is simulated by s and exists in $\mathcal{U}$.

□

Proposition A.8. *Let $\mathcal{E}$ be an EMTS. For universal LTS $\mathcal{U}$ and $s \in S_{\mathcal{E}}$, if $s \vDash^{\mathcal{E}}_{\mathrm{V}} \langle a \rangle \Psi$, then there exists $S' \in \partial^{\Box}_a(s)$ such that for all $s' \in S'$, $s' \vDash^{\mathcal{E}}_{\mathrm{V}} \Psi$.*

Proof. First, we prove that $\partial^{\Box}_a(s)$ is not empty. Assume $\partial^{\Box}_a(s) = \varnothing$. Then, the state t_{new} which has all the transitions of some $t \in [\![s]\!]_{\mathcal{U}}$, with the exception of a-transitions would still be simulated by s. Then by definition 2.11, $t_{new} \models^{\mathcal{U}}_{\mathbf{V}} \langle a \rangle \Psi'$, but this is clearly not the case so we reach a contradiction.

Next, we prove that there exists $S' \in \partial^{\Box}_a(s)$ such that for all $s' \in S'$, $s' \vDash^{\mathcal{E}}_{\mathrm{V}} \Psi$. Assume for all $S' \in \partial^{\Box}_a(s)$, S' does not satisfy Ψ'. In such a case the state $t_{brandnew}$ which has all the transitions of a state $t \in [\![s]\!]_{\mathcal{U}}$ excluding a-transitions of t and with the addition of the transition $t_{brandnew} \xrightarrow{a}_{\mathcal{T}} t'$ for some $t' \in [\![S']\!]_{\mathcal{U}}$, is still simulated by s. But then $t_{brandnew}$ does not satisfy $\langle a \rangle \Psi'$, which is a contradiction. □

Lemma A.9. *Let $\mathcal{E}$ be a finite-state EMTS and $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$ be proof trees constructed as described above. If for node $n : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi_n$ in $A_{\mathcal{U}}$, $\gamma(n) = m$ where $m : S_m \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi_m$ in $A^*_{\mathcal{U}}$, then $S_m \subseteq S_n$.*

Proof. In order to make our proof, we can use induction on the depth of the rule applications in $A_{\mathcal{U}}$.

As a base case, n_0 and $\gamma(n_0)$ are the roots of the trees and mention the same sets, so initially $S_m = S_n$.

Suppose node $n : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$ of $A_{\mathcal{U}}$ matches $m : S_m \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$ of $A^*_{\mathcal{U}}$ and $S_m \subseteq S_n$ we show the same subset condition holds for each subgoal produced by rule induction. If the rule applied to n is:

- Thin, then let the immediate successor of n be $n' : ||\Psi||^{\mathcal{U}}_{\mathbf{V}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$. We know that $\gamma(n') = m$ by definition and that $S_n \subseteq ||\Psi||^{\mathcal{U}}_{V}$, hence $S_m \subseteq ||\Psi||^{\mathcal{U}}_{V}$
- $\vee$, $\wedge$, σZ or Z, the sets S_n and S_m also occur in the immediate successors, so the property is preserved.
- Cut, then the set S_n occurs in both subgoals n^1 and n^2, meanwhile the sets mentioned in $\gamma(n^1)$ and $\gamma(n^2)$ are both subsets of S_m, so they are also subsets of S_n.
- $\Box_a$, then $S_m = [\![s_m]\!]_{\mathcal{U}}$ for some state s_m of $\mathcal{E}$. Then $\partial_a([\![s_m]\!]_{\mathcal{U}}) = [\![\cup\partial^{\Diamond}_a(s_m)]\!]_{\mathcal{U}}$ by Proposition A.8. Since $[\![s]\!]_{\mathcal{U}} \subseteq S_n$, $\partial_a([\![s]\!]_{\mathcal{U}}) \subseteq \partial_a(S_n)$. Hence $[\![\cup\partial^{\Diamond}_a(s_m)]\!]_{\mathcal{U}} \subseteq \partial_a(S_n)$.
- $\Diamond_a$ This is the case since we have selected $f_a|_{[\![s_m]\!]_{\mathcal{U}}} = f^{\mathcal{E}}_a$ in the construction. □

Lemma A.10. *Let $\mathcal{E}$ be a finite-state EMTS, $s \in S_{\mathcal{U}}$, and Φ have prime subformulas only. If $s \models^{\mathcal{E}}_{\mathrm{V}} \Phi$, then let $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$ be proof trees constructed as described above. If $A_{\mathcal{U}}$ is a finite proof tree, than $A^*_{\mathcal{U}}$ is also a finite proof tree.*

Proof. First, we have to show the correctness of the application of the rules involved in macro applications, i.e. $A^*_{\mathcal{U}}$ is indeed a proof tree. Next, we show that validity is preserved with each macro application. Finally. we show that the application of macros guided by the rule applications in $A_{\mathcal{U}}$ can not go on forever, i.e. $A^*_{\mathcal{U}}$ is finite.

Let $m : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$ be a node of $A^*_{\mathcal{U}}$ where $[\![s]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} \Psi$. If the macro applied to m is:

- $\wedge$, σZ or Z, the application is identical to the corresponding rule application in $\Sigma_{\mathcal{T}}$.
- $\vee$ and $\Psi = \Psi_1 \vee \Psi_2$, the goal is guaranteed to be reduced to a single subgoal because Ψ is a subformula of Φ and hence prime.
- $\Box_a$ where $\Psi = [a]\,\Psi'$ and $[\![s]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} [a]\,\Psi'$. We first have to show that no application of the Thin rule is needed. This is the case since $\partial_a([\![s]\!]_{\mathcal{U}}) = [\![\cup\partial^{\Diamond}_a(s)]\!]_{\mathcal{U}}$ by the definition of satisfaction, definition 2.11. This automatically shows that $[\![\cup\,\partial^{\Diamond}_a(s)]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} \Psi'$ by the preservation of validity in $A_{\mathcal{U}}$.
- $\Diamond_a$ where $\Psi = \langle a\rangle\,\Psi'$ and $[\![s]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} \langle a\rangle\,\Psi'$. The existence of a proper choice function $f^{\mathcal{E}}_a$ is a result of Proposition A.7 and Proposition A.8.

This concludes the proof of the correctness of the macro applications.

The fact that validity is preserved comes from lemma A.9 and that validity is preserved in canonical proofs and hence is preserved in $A_{\mathcal{U}}$.

The fact that the tree is finite is obvious, because a macro is applied to a sequent of $A^*_{\mathcal{U}}$, if a rule is applied to the matching sequent in $A_{\mathcal{U}}$. Cut-rules are applied in $A_{\mathcal{U}}$ when a macro includes them so that after the whole construction each rule application in $A_{\mathcal{U}}$ is matched by a rule application in $A^*_{\mathcal{U}}$ except for Thin. The possible causes of nontermination could be the modifications we made on the proof system: the addition of the Cut-rule applications and the new termination condition.

The modified proof system allows for infinitely many Cut-rule applications since the sequent is not changed with each application, but the number of consecutive Cut-rule applications allowed in the construction is equal to the number of states the particular state of the $\mathcal{E}$ has transition to. So the number of applications is guaranteed to be finite since $\mathcal{E}$ is finite state.

The new termination condition used in the construction requires that for a repeating node in $A_{\mathcal{U}}$ to be a terminal, its matching node in $A^*_{\mathcal{U}}$ must be a σ-terminal which mentions exactly the same sequent with its companion. The set mentioned in repeating nodes is always the denotation of a single set and the number of states of the $\mathcal{E}$ is finite, so eventually we are guaranteed to reach the identical sequent in $A^*_{\mathcal{U}}$. Therefore the modified termination condition does not cause infinite proof trees. □

Definition A.11 (Trail Translation)**.** Let $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$ be constructed as described above, and $T = (t_i, m_i), \ldots, (t_k, m_k)$ be a trail in $A^*_{\mathcal{U}}$ of the terminal node m_k and its companion m_i. The function δ converts T to a trail of the matching σ-terminal and its companion in $A_{\mathcal{U}}$ by replacing each node with the matching one(s) in $A_{\mathcal{U}}$:

$$\begin{aligned} \delta(\varepsilon) &\triangleq \varepsilon \\ ((t,m)\cdot T) &\triangleq \begin{cases} ((t,n)\cdot\delta(T)) & \gamma^{-1}(m) = \{n\} \\ ((t,n)\cdot(t,n')\cdot\delta(T)) & \gamma^{-1}(m) = \{n,n'\} \text{ with } n \text{ above } n' \text{ in } A_{\mathcal{U}} \end{cases} \end{aligned}$$

It is possible to use the same state t_i for the trail of $A_{\mathcal{U}}$ because for each m_i that is matched by a n_i, a state t_i that occurs in node m_i is also in n_i, by lemma A.9.

Lemma A.12. *If $A_{\mathcal{U}}$ is a successful proof tree, then $A^*_{\mathcal{U}}$ is also a successful proof tree.*

Proof. We know by lemma A.10 that we will get a correct tableaux $A^*_{\mathcal{U}}$. It remains to prove that if the first tableaux is successful, then the second is also successful.

Let $m : S_m \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$ be a leaf of $A^*_{\mathcal{U}}$, and $n : S_n \vdash^{\mathcal{U}}_{\mathbf{V}} \Psi$ where $\gamma(n) = m$ and the rule applied to n is not Thin. Assume n is a successful terminal.

If $S_m = \varnothing$, then it is trivially a successful terminal. For all other cases, since the termination is checked only after (possible) Cut-rule applications, $S_m = [\![s_m]\!]_{\mathcal{U}}$ for some state s_m of $\mathcal{E}$.

If Ψ is tt, m is trivially successful. If Ψ is a variable Z which is free in the initial formula, then it should be the case that $S_n \subseteq V(Z)$, and by lemma A.9, $[\![s]\!]_{\mathcal{U}} \subseteq V(Z)$.

In the case where Ψ is a variable Z that identifies $\sigma Z.\Psi'$, let n' be the companion node of n. By construction, $\gamma(n') = m'$ where $m' : [\![s_m]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} Z$ in $A^*_{\mathcal{U}}$, and m' a predecessor of m. Then m is a successful terminal if Z identifies $\nu Z.\Psi'$.

If Z identifies $\mu Z.\Psi'$, it has to be shown that there is no infinite chain of composable trails of the companion node m'. Suppose there is such an infinite chain of composable trails, $\kappa_{m'}{=}T_0 \circ T_1 \circ T_2 \ldots$ in $A^*_{\mathcal{U}}$. Then there is a corresponding infinite chain of composable trails in $A_{\mathcal{U}}$ given by $\kappa_{n'}{=}\delta(T_0) \circ \delta(T_1) \circ \delta(T_2) \ldots$. However, n is a successful terminal, therefore no such infinite chain of n' exists. Hence we reach a contradiction. □

Lemma A.13. *If proof trees $A_{\mathcal{U}}$ and $A^*_{\mathcal{U}}$ constructed as described above are successful, then $A_{\mathcal{E}} = \pi^{-1}_{\mathcal{U}}(A^*_{\mathcal{U}})$ is also a successful proof tree.*

Proof. If $A_{\mathcal{U}}$ is a successful proof tree, $A^*_{\mathcal{U}}$ is a successful proof tree by lemma A.12. The side conditions of the rules of $\Sigma_{\mathcal{E}}$ are satisfied by the correctness of the macro applications in $A^*_{\mathcal{U}}$. Thus, to establish the rest of the result, it suffices to show that each leaf of $A_{\mathcal{E}} = \pi^{-1}_{\mathcal{U}}(A^*_{\mathcal{U}})$ is a successful terminal.

By the definition of $\pi_{\mathcal{T}}$, it is easy to observe that each leaf of $A_{\mathcal{E}}$ is matched by a leaf of $A^*_{\mathcal{U}}$, except the case where the leaf node is $m : r \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$ with the matching node being $n : [\![r]\!]_{\mathcal{U}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$ with $\cup\, \partial^{\Diamond}_a(r) = \varnothing$. Since r does not have *may*-transitions, m is a successful terminal.

Consider the successful terminal $n : [\![r]\!]_{\mathcal{U}} \vdash^{\mathcal{T}}_{\mathrm{V}} \Psi$ in $A^*_{\mathcal{U}}$ and the matching node of $A_{\mathcal{E}}$, $m : r \vdash^{\mathcal{E}}_{\mathcal{V}} \Psi$. The proof that the latter is also a successful terminal is trivial when $\Psi = \mathrm{tt}$, when $\Psi = Z$ and Z is free in the initial formula, or when Z identifies the formula $\nu Z.\Psi'$.

For the terminal node m to be successful when $\Psi = Z$ and Z identifies $\mu Z.\Psi'$, all runs that correspond to an infinite sequence of $\mathcal{E}$-trails of m and its companion node m' should be in W, that is there is no $w_{m'} = E_1 \circ E_2 \ldots$, where for all $i \geq 1$, E_i begins with (r, m') and ends with (r, m), and $\alpha(w_{m'})$ is not in W.

Assume that such an infinite sequence, $w_{m'} = E_1 \circ E_2 \ldots$ exists, where $\alpha(w_{m'})$ is not in W. Then an LTS, $\mathcal{T}$, can be constructed such that for each r_i in $S_{\mathcal{E}}$, there exists $r_{i'}$ in $S_{\mathcal{T}}$ and there exists a transition $r_{i'} \xrightarrow{a}_{\mathcal{T}} r_{j'}$, if and only if $r_i \overset{a}{\longrightarrow}^{\Diamond}_{\mathcal{E}} r_j$.

It is clear that each state, $r_{i'}$, of $\mathcal{T}$ is simulated by a state r_i of $\mathcal{E}$, so $r_{i'} \in [\![r_i]\!]_{\mathcal{T}}$. Since $\mathcal{U}$ contains $\mathcal{T}$, there is an infinite sequence of composable trails in $A^*_{\mathcal{U}}$ of the terminal node n and its companion n' that mentions the same states, $r_{i'}$. But since n is a successful terminal, there can be no such infinite sequence of trails, hence we reach a contradiction.

□

Theorem 2.27 (Completeness) *Let $\mathcal{E}$ be a finite–state EMTS, $s \in S_{\mathcal{E}}$, and let Φ have prime subformulas only. Then $s \models^{\mathcal{E}}_{\mathrm{V}} \Phi$ implies $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$.*

Proof. Assume $s \models^{\mathcal{E}}_{\mathrm{V}} \Phi$. Then, by definition 2.11, $[\![s]\!]_{\mathcal{T}} \models^{\mathcal{T}}_{\mathbf{V}} \Phi$ for any $\mathcal{T}$, and hence also $[\![s]\!]_{\mathcal{U}} \models^{\mathcal{U}}_{\mathbf{V}} \Phi$. By completeness of $\Sigma_{\mathcal{T}}$, there exists a family of canonical proofs for the goal $[\![s]\!]_{\mathcal{U}} \vdash^{\mathcal{U}}_{\mathbf{V}} \Phi$. $A_{\mathcal{U}}$, which can be viewed as a combination of several of these canonical proofs, can be constructed with $A^*_{\mathcal{U}}$ as described above. Then, by lemma A.13, $\pi^{-1}_{\mathcal{U}}(A^*_{\mathcal{U}})$ is a proof of $s \vdash^{\mathcal{E}}_{\mathcal{V}} \Phi$ in $\Sigma_{\mathcal{E}}$. □

Appendix B

Part III Appendix

B.1 ConSpec Semantics Annex

In section 4.4.2, we described how ConSpec policies induce symbolic automata. We assumed in this description a semantic function f_U for each update block U of the policy. While update blocks U are comprised of sequences of assignments to security state variables, the function f_U returns for each security state variable. The property of f_U is that the value of a security state variable s has the same value as a result of the execution of U on the initial state σ of security state variables and the assignment to s of the expression $f_U(s)$ interpreted on σ. Here we describe how to obtain an f_U from the update block U.

We consider update blocks in the context of event clauses. Here we describe the case where the update blocks to be translated belong to an AFTER event clause:

```
AFTER  τ x_ret = c.m (τ_1 x_1, ..., τ_n x_n)
PERFORM
  G_1  ->  U_1
  ⋮
  G_m  ->  U_m
```

Let $Avar = \{x_1, \ldots, x_n\}$ be the set of formal arguments of the event and $Pvar = \{x_ret\} \cup Avar$ be the set of all program variables of the event clause.

Below, let states $(q : Svar \to PrimVal) \in Q$ be as mappings from security state variables of the policy to primitive values, and let $\sigma : Pvar \to Val$ range over the set Σ of mappings from program variables to values which respect the declared types of the variables. We assume for any ConSpec expression E occurring in an AFTER event clause, the semantic function:

$$[\![E]\!] : Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \to PrimVal$$

and every update block $UpdateBlock$, or U for short, of $\varphi^\sharp$, we assume the semantic

function:

$$[\![U]\!] : Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \to Q$$

where the two heaps in the function types refer to the heap of the program before and after the execution of the method call, respectively.

The function f_U for a ConSpec update block U has the following property:

$$\forall q \in Q, \sigma \in \Sigma, h, h' \in \mathbb{H}.[\![U]\!](q, \sigma, h, h') = \lambda s \in Svar.[\![f_U(s)]\!](q, \sigma, h, h')$$

The goal is to come up with expressions that use the original values of the security variables, i.e. the state of the automaton before the action has occurred. The function f_U can be produced by going through the sequence of assignments in the update block from top to bottom and replacing the occurrences of security state variables by their *latest assigning expressions*. The auxiliary function f_{aux} takes as argument a mapping from security state to ConSpec expressions and the update block; with each assignment statement, the function updates the latest assigning expressions recorded in the mapping. Let $UVar$ be the set of all variables occurring in the update block (i.e. the union of $Svar$, $Avar$, the return value x_{ret} and any local variables), and its cardinality be m. We let y range over the variables of this set. Furthermore, we take substitution ρ to be $[y_1/g(y_1), \ldots, y_m/g(y_m)]$. The function f_{aux} is defined recursively as follows:

$$\begin{array}{l} f_{aux}(g)(y \ = \ E;\ Rest) = f_{aux}(g[y \mapsto (E\rho)])(Rest) \\ f_{aux}(g)(\texttt{local}\ \tau\ y \ = \ E;\ Rest) = f_{aux}(g[y \mapsto (E\rho)])(Rest) \\ f_{aux}(g)(\texttt{skip;}\ Rest) = f_{aux}(g)(Rest) \\ f_{aux}(g)(\epsilon) = g \end{array}$$

The function f_U then uses this auxiliary function by calling it with an initial mapping f_0, which maps each security state variable to some fixed symbol as a representative of their original value. These symbols are exchanged with the variables' names when the auxiliary function returns.

$$f_U = (f_{aux}(f_0)(U))\rho_0 \text{ where } \rho_0 = [f_0(s_1)/s_1, \ldots, f_0(s_n)/s_n]$$

Example B.1. Suppose the following is an update block U for a policy with security state variables s1, s2, s3 and the program variable x:

```
local int foo= 0;
foo = s1;
s1 = s2;
s2 = foo;
s3 = x;
```

Then the calls to the auxiliary method with f_0=[s1 $\mapsto$ X_1, s2 $\mapsto$ X_2, s3 $\mapsto$ X_3] is as follows:

$f_{aux}(f_0)(U)$
$= f_{aux}([\text{s1} \mapsto X_1, \text{s2} \mapsto X_2, \text{s3} \mapsto X_3])(\text{local int foo = 0;foo = s1;s1 = s2;s2 = foo;s3 = x;})$
$= f_{aux}([\text{s1} \mapsto X_1, \text{s2} \mapsto X_2, \text{s3} \mapsto X_3, \text{foo} \mapsto 0])(\text{foo = s1;s1 = s2;s2 = foo;s3 = x;})$
$= f_{aux}([\text{s1} \mapsto X_1, \text{s2} \mapsto X_2, \text{s3} \mapsto X_3, \text{foo} \mapsto X_1])(\text{s1 = s2;s2 = foo;s3 = x;})$
$= f_{aux}([\text{s1} \mapsto X_2, \text{s2} \mapsto X_2, \text{s3} \mapsto X_3, \text{foo} \mapsto X_1])(\text{s2 = foo;s3 = x;})$
$= f_{aux}([\text{s1} \mapsto X_2, \text{s2} \mapsto X_1, \text{s3} \mapsto X_3, \text{foo} \mapsto X_1])(\text{s3 = x;})$
$= f_{aux}(\text{s1} \mapsto X_2\ , \text{s2} \mapsto X_1, \text{s3} \mapsto \text{x}, \text{foo} \mapsto X_1)()$
$= \text{s1} \mapsto X_2\ , \text{s2} \mapsto X_1, \text{s3} \mapsto \text{x}, \text{foo} \mapsto X_1$

f_U
$= f_{aux}(f_0)(U)\rho_0$
$= (\text{s1} \mapsto X_2, \text{s2} \mapsto X_1, \text{s3} \mapsto \text{x}, \text{foo} \mapsto X_1)[X_1/\text{s1}, X_2/\text{s2}, X_3/\text{s3}]$
$= \text{s1} \mapsto \text{s2}, \text{s2} \mapsto \text{s1}, \text{s3} \mapsto \text{x}$

B.2 Example from Part III

The text contains a complete example. The Java source code of the program is as follows. The program contains the `Ask` class, which has a single field called `gui` of type `GUI`. The `ask` method of this class calls `GUI.AskConnect()` on the object stored in the `gui` field and returns the return value of this method.

```
public class MyClass {
   GUI gui;
   public bool ask() {
      return gui.AskConnect();
   }
 }
```

The policy is the example policy from section 4.4. The corresponding bytecode program is presented on the left of figure 4.8. The program is inlined for the example policy; the inlined program is as presented on the right of figure 4.8. The level I annotated program for the example policy is presented in figure 4.4. Both these steps are performed assuming that `GUI` does not have any subclasses. The level II annotated program for the program is given in figure 4.6. Finally, the level III (fully) annotated program is given in figure 4.9. We now show in detail how to compute level III annotations from the level II annotated program.

Requires and *Ensures* are determined by rule 1
Requires = *Ensures* = *INV*

Annotations of $L1 - L2$ are computed using rule 2.
$A^{III}[L1] = norm(INV) = INV$
$A^{III}[L2] = norm(INV) = INV$

Annotation of $L5$ (used for computing the annotations $L3 - L4$) is computed using rule 3.

$$\begin{array}{lcl}
A^{III}[L5] & = & norm((g_{\text{this}} := \text{s}[0]) \cdot \\
 & & (g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \cdot \\
 & & ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \cdot \\
 & & (g_{\text{this}} = r1)) \\
 & = & ((\text{s}[0] : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (\text{s}[0] = \text{r1})) \cdot \\
 & & (g_{\text{this}} := \text{s}[0]) \cdot \\
 & & ((g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (g_{\text{this}} = \text{r1}))
\end{array}$$

Annotations of $L3 - L4$ are computed using rule 5 using wp computation.

$$\begin{array}{lcl}
A^{III}[L4] & = & wp(M[L4]) \\
 & = & (shift(head(A_M^{III}[L5])))[\text{s}[0]/\text{r1}] \\
 & = & ((\text{s}[1] : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (\text{s}[1] = \text{r1}))[\text{s}[0]/\text{r1}] \\
 & = & ((\text{s}[1] : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (\text{s}[1] = \text{s}[0])) \\
A^{III}[L3] & = & wp(M[L3] \\
 & = & unshift((head(A_M^{III}[L+1]))[\text{s}[1]/\text{s}[0]]) \\
 & = & unshift((\text{s}[1] : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (\text{s}[1] = \text{s}[1])) \\
 & = & (\text{s}[0] : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot)) \wedge INV \wedge (\text{s}[0] = \text{s}[0])
\end{array}$$

Annotation of $L13$ (used for computing the annotations of $L6 - L12$) is computed using rule 4.

$$\begin{array}{lcl}
A^{III}[L13] & = & norm(INV) \\
 & = & INV
\end{array}$$

Annotations of $L6 - L12$ are computed using rule 5 using wp computation.

$$\begin{array}{lcl}
A^{III}[L12] & = & wp(M[12]) \\
 & = & unshift(head(A_M^{III}[L13])[\text{s}[0]/\text{r2}]) \\
 & = & unshift(INV) \\
 & = & INV
\end{array}$$

$$\begin{array}{lcl}
A^{III}[L11] & = & wp(M[11]) \\
 & = & (shift(head(A_M^{III}[L12])))[\text{s}[0]/\texttt{SecState.permission}] \\
 & = & INV[\text{s}[0]/\texttt{SecState.permission}] \\
 & = & (g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{s}[0])
\end{array}$$

$$\begin{array}{lcl}
A^{III}[L10] & = & wp(M[10]) \\
 & = & unshift(head(A_M^{III}[L13])[\text{r2}/\text{s}[0]]) \\
 & = & unshift((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2})) \\
 & = & (g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2})
\end{array}$$

$$\begin{array}{lcl}
A^{III}[L9] & = & wp(M[9]) \\
 & = & (\text{s}[0] = 0 \Rightarrow shift(head(A_M^{III}[L12]))) \wedge \\
 & & (\neg(\text{s}[0]) = 0 \Rightarrow shift(head(A_M^{III}[L10]))) \\
 & = & (\text{s}[0] = 0 \Rightarrow INV) \wedge \\
 & & (\neg(\text{s}[0]) = 0 \Rightarrow (g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2}))
\end{array}$$

$$\begin{array}{lcl}
A^{III}[L8] & = & wp(M[8]) \\
 & = & (\text{s}[0] <: \texttt{GUI} \Rightarrow (head(A_M^{III}[9]))[1/\text{s}[0]]) \wedge \\
 & & (\neg(\text{s}[0] <: \texttt{GUI}) \Rightarrow (head(A_M^{III}[9]))[0/\text{s}[0]]) \\
 & = & (\text{s}[0] <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2}))) \wedge \\
 & & (\neg(\text{s}[0] <: \texttt{GUI}) \Rightarrow (INV))
\end{array}$$

$$\begin{array}{lcl}
A^{III}[L7] & = & wp(M[L7]) \\
 & = & unshift((head(A^{I}II_{M}[L8]))[\text{r1}/\text{s}[0]]) \\
 & = & unshift((\text{r1} <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2}))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow (INV))) \\
 & = & (\text{r1} <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2}))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow INV)
\end{array}$$

We denote with Φ below the right hand side of the ghost assignment of $A_M^{II}[L6]$.

$$\begin{array}{lcl}
A^{III}[L6] & = & norm(g_{\text{this}} = \text{r1} \cdot INV \cdot A_M^{II}[L6] \cdot wp(M[L6])) \\
 & = & norm(g_{\text{this}} = \text{r1} \cdot INV \cdot A_M^{II}[L6] \cdot \\
 & & (shift(head(\text{r1} <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{r2}))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow INV)))[\text{s}[0]/\text{r2}]) \\
 & = & norm(g_{\text{this}} = \text{r1} \cdot INV \cdot \\
 & & ((g_{\text{a}}, g_{\text{p}}) := \Phi \cdot \\
 & & ((\text{r1} <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{s}[0]))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow INV))) \\
 & = & (g_{\text{this}} = \text{r1}) \wedge INV \wedge (\text{r1} <: \texttt{GUI} \Rightarrow (\Phi = (\texttt{SecState.accessed}, \text{s}[0]))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow (\Phi = (\texttt{SecState.accessed}, \texttt{SecState.permission}))) \cdot \\
 & & ((g_{\text{a}}, g_{\text{p}}) := \Phi \cdot \\
 & & ((\text{r1} <: \texttt{GUI} \Rightarrow ((g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \text{s}[0]))) \wedge \\
 & & (\neg(\text{r1} <: \texttt{GUI}) \Rightarrow INV)))
\end{array}$$

B.3 Proofs for Part III

B.3.1 Proof of Theorem 4.10

Theorem 4.9 (Correctness of Monitoring by Co-execution) *Let* T *be a program, and* $\mathcal{P}$ *a policy. The following holds, where* A *is the action set of* $\mathcal{A}_\mathcal{P}$*:*

$$\{w \downarrow 1 \mid w \text{ is a co-execution of T and } \mathcal{A}_\mathcal{P}\} = \{E \mid E \in \Pi(\mathrm{T}) \wedge srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}\}$$

Proof. ($\supseteq$) We prove that for all executions E of the program such that the security relevant trace of E is in the language of the policy automaton, there is an execution w of the program and the policy automaton, where $w \downarrow 1 = E$.

Let $q_0 q_1 \ldots$ be the run of the automaton for $srt_A(E)$. We (1) construct a configuration-automaton state pair sequence w for E, using the automaton run and (2) prove that w is a co-execution with $w \downarrow 1 = E$.

(1) Intuitively, we begin the construction with the initial configuration C_0 of E and the initial state q_0. We add the following configurations, paired with this state until a security relevant action (s.r.a.) is produced. Whenever an s.r.a. is produced, the state component of the added pair is changed with the next automaton state in the run. This process is repeated until both the end of the execution and of the automaton run is reached, for infinite executions the process is repeated infinitely many times. Security relevant actions are detected by using the act_A functions on consecutive configurations of the execution.

Formally, let w_n denote the sequence constructed for the (finite) prefix $C_0 C_1 \ldots C_n$ of E. The sequence w_0 is defined as (C_0, q_0) if $act^\flat(C_0) = \epsilon$ and $(C_0, q_0)(C_0, q_1)$ if $act^\flat(C_n) \in A^\flat$. When constructing the sequence w_n for longer executions, we use the current state as the state component of the last pair of w_{n-1}, denoted below by q_k. The sequence w_n for $n > 0$ is defined as follows:

$$w_n = \begin{cases} w_{n-1} \cdot (C_n, q_k) & \text{if } act^\sharp(C_{n-1}, C_n) act^\flat(C_n) = \epsilon \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) & \text{if } act^\sharp(C_{n-1}, C_n) act^\flat(C_n) \in A^\flat \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) & \text{if } act^\sharp(C_{n-1}, C_n) act^\flat(C_n) \in A^\sharp \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) \cdot (C_n, q_{k+2}) & \text{if } act^\sharp(C_{n-1}, C_n) \in A^\sharp, act^\flat(C_n) \in A^\flat \end{cases}$$

(2) We prove that w_i is a co-execution and $w_i \downarrow 1 = C_0 C_1 \ldots C_i$ for all finite prefixes $C_0 \ldots C_i$ of E. The result then follows since this is a continuous predicate on configuration sequences with respect to the immediate prefix ordering and $\Pi(\mathrm{T})$ is prefix-closed.

In the proof, we use the fact that ConSpec automata are deterministic (by definition) and their language is prefix-closed (since each ConSpec automaton is a security automaton as defined by Schneider [102]). We can then conclude for each prefix E' of E that E' is in the language of the automaton and the run of the automaton which accepts E' is a prefix of the run accepting E.

(*Base Case*) Consider the execution consisting of the initial configuration C_0. If $act^\flat(C_0) \in A^\flat$, then the security relevant trace of C_0 is $act^\flat(C_0)$. Then $w_0 =$

$(C_0, q_0)(C_0, q_1)$ by construction. This sequence is an interleaving since : $(C_0, q_0) \longrightarrow_{\text{AUT}} (C_0, q_1)$. By definition then, $w_0 \downarrow 1 = C_0$ and $extract(w_0) = q_0 q_1\ act^\flat(C_0)$. Clearly $extract(w_0) \in E^\flat$. On the other hand, if $act^\flat(C_0) = \epsilon$, the security relevant trace is empty. The accepting run then consists of q_0 and the constructed sequence w_0 of (C_0, q_0). Again by definition, $w_0 \downarrow 1 = C_0$, and $extract(w_0) = \epsilon$.

(*Induction Hypothesis*) Assume that w_i is a co-execution and $w_i \downarrow 1 = C_0 C_1 \ldots C_i$ for all $i < n$.

(*Inductive Step*) Consider the sequence w_n constructed for the prefix $C_0 \ldots C_{n-1} C_n$ using the automaton run $q_0 \ldots q_m$ where m is the number of security relevant actions of $C_0 \ldots C_{n-1} C_n$. By definition, the following holds:

$$srt_A(C_0 \ldots C_{n-1} C_n) = srt_A(C_0 \ldots C_{n-1})\ act^\sharp(C_{n-1}, C_n)\ act^\flat(C_n)$$

We consider the most difficult case where $act^\sharp(C_{n-1}, C_n) \in A^\sharp, \quad act^\flat(C_n) \in A^\flat$. (The other cases are similar) Since $q_0 \ldots q_m$ is an accepting run for this execution:

$$\begin{array}{rcll} \delta(q_{m-2}, act^\sharp(C_{n-1}, C_n)) & = & q_{m-1} & (i) \\ \delta(q_{m-1}, act^\flat(C_n)) & = & q_m & (ii) \end{array}$$

Then the sequence w_{n-1}, constructed (as described above) for E_{n-1} using the run $q_0 \ldots q_{m-2}$, is a co-execution by the induction hypothesis. Note that the last component of this co-execution is C_{n-1}, q_{m-2} by the construction. Again by construction, the sequence w_n is an extension of w_{n-1} (last case):

$$w_n = w_{n-1}(C_n, q_{m-2})(C_n, q_{m-1})(C_n, q_m)$$

We prove that:

- *w_n is an interleaving:* The sequence w_{n-1} is an interleaving by the induction hypothesis. Since E_n is an execution, there is a machine transition from C_{n-1} to C_n. There are transitions between the consecutive states $q_{m-2} q_{m-1} q_m$ of the automaton run. Thus the extension to w_{n-1} consists of one machine transition followed by the automaton transitions:

  $$(C_{n-1}, q_{m-2}) \longrightarrow_{\text{JVM}} (C_n, q_{m-2}) \longrightarrow_{\text{AUT}} (C_n, q_{m-1}) \longrightarrow_{\text{AUT}} (C_n, q_m)(*)$$

- *w is a co-execution:* By assumption w_{n-1} is a co-execution. Then $extract(w_{n-1}) \in (E^\flat \cup E^\sharp)^{m-2}$ since there $m-2$ s.r.a's in E_{n-1}. By definition of the *extract* function and using (*):

  $$extract(w_n) = extract(w_{n-1}) act^\sharp_A(C_{n-1}, C_n) \cdot\ q_{m-2} q_{m-1} q_{m-1} q_m\ \cdot act^\flat_A(C_n)$$

  By (i), $act^\sharp_A(C_{n-1}, C_n) q_{m-2} q_{m-1} \in E^\sharp$ and by (ii), $q_{m-1} q_m act^\flat_A(C_n) \in E^\flat$. Hence $extract(w_n) \in (E^\flat \cup E^\sharp)^m$.

- *$w \downarrow 1 = E_n$:* This simply follows from the induction hypothesis and applying the first projection function to w_n.

($\subseteq$) We prove that for all co-executions w of the program and the policy automaton, the projection to the first component is (i) an execution of the program and that (ii) its security relevant trace is in the language of the policy automaton.

(i) We prove this by induction on the length of w.

(*Base Case*) If $w = (C, q)$, since w is an interleaving, $C = C_0$ and $q = q_0$. Then, $w \downarrow 1 = C_0$, which is an execution.

(*Induction Hypothesis*) We assume the statement for w_n of length n.

(*Inductive Case*) We prove the statement for w_{n+1}, where $w_{n+1} = w_n \bullet [(C_{n+1}, q_{n+1})]$ for some q_{n+1}. Let the last element of w_n be (C_n, q_n) and $w_n \downarrow 1 = E_n$. By the definition of $\downarrow 1$ (page 4.5), $E_n = E' \bullet [C_n]$ for some sequence of configurations E'. Again by the definition of $\downarrow 1$, $w_{n+1} \downarrow 1 = E'C_n \bullet [C_{n+1}]$ if $C_n \longrightarrow_{\text{JVM}} C_{n+1}$ and $w_{n+1} \downarrow 1 = E' \bullet [C_{n+1}]$ otherwise.

1. If the first case applies, $w_{n+1} \downarrow 1 = (w_n \downarrow 1) \bullet [C_{n+1}]$ and everything but the last element is an execution by the induction hypothesis and the last two configurations are related with the JVM transition relation. Hence this is an execution of T. We also note the observation here that $\Pi(\text{T})$ is closed under the transitive closure of the suffix relation built using the JVM transition relation.

2. If the second case applies, by the definition of interleaving, $C_{n+1} = C_n$ and therefore $w_{n+1} \downarrow 1 = w_n \downarrow 1$. The result follows from the inductive hypothesis.

(ii) We prove this also by induction on the length of w.

(*Base Case*) If $w = (C, q)$, since w is an interleaving, $C = C_0$ and $q = q_0$. Then, $w \downarrow 1 = C_0$. According to the table of page 4.5, $srt_A(C_0) = a^\flat \in A^\flat$ for some pre-action $a^\flat$, or $srt_A(C_0) = \epsilon$. The statement trivially holds in the latter case, as ϵ is in the language of all security automata with at least one state. Let us assume the first case. We will prove that such a co-execution does not exist, thus reaching a contradiction and hence the statement will hold vacuously for the first case. By the definition of *extract*, $extractw = a^\flat$ if the first case applies. But by the definition of being a co-execution $a^\flat$ should be in the set $(E^\flat \cup E^\sharp)^* \cup (E^\flat \cup E^\sharp)^\omega$. This is not possible as there is no string of length 1 in this set. Hence we reach a contradiction.

(*Induction Hypothesis*) We assume the statement for all w_i of length i, where $i < n + 1$.

(*Inductive Case*) We prove the statement for w_{n+1}. We have to consider the cases of how a co-execution is produced by extending another co-execution. We prove the statement for w_{n+1}, where $w_{n+1} = w_n \bullet [(C_{n+1}, q_{n+1})]$ for some q_{n+1}. Notice that since both are co-executions, the function *extract* maps both to the same set. It can not be however that $extractw_{n+1} = extractw_n \bullet E'$ for some $E^\flat$ or $E^\sharp$, for these extensions contain always three elements, two automata states and an action, and therefore can not be extracted when the co-execution is extended with only one pair. This means that the security relevant trace of E_{n+1} is the same

with that of E_n and the result holds by induction hypothesis. The other cases are proved similarly. □

B.3.2 Proof of Theorem 4.15

The proof of this theorem is quite complicated as it brings together many concepts of the paper such as the symbolic and the ConSpec automaton, co-execution, and operational semantics of annotations.

Let us first assume that the level I annotated program is valid. Intuitively, our goal is to show that any execution of the program can be "completed" with automaton states to form a co-execution of the program with the policy automaton (the ConSpec automaton for the policy). This means for each configuration in the execution such an automaton state should be found that the pair sequence that is formed is a co-execution. We use the value of the ghost state as the automaton state. Remember the conditions for a configuration-automaton state pair sequence to be a co-execution. The first condition is that the automaton component of the first pair is the initial automaton state. When execution begins, the ghost state is the initial state, thus satisfies this condition. The second is that for a pre-action, the automaton state is updated sometime before the action takes place, but after the previous action in the series. The ghost state is updated immediately before the execution of a pre-action, since a ghost assignment is placed before each instruction which may yield a preaction when executed. (Similarly for post-actions but with an update to automaton state/ghost state immediately after.) We call a co-execution where the monitor updates are done immediately before (or after) a s.r.a. a *closest updating co-execution*. For the ghost state to be a monitor for the program, it should also be updated to the correct automaton state. We prove this using the way a ConSpec automaton is induced by a symbolic automaton and the way the (same) symbolic automaton induces the level I annotations. There is one catch: if at some configuration of the execution, the ghost state is undefined and a security relevant action is performed, the ghost state remains undefined; but such a sequence can never be co-execution. The reason is that the "undefined" state of the automaton does not have any outgoing transitions, thus the automaton sequence extracted from such a pair sequence would not be a run of the automaton. The validity assumption is used to rule out this possibility. So we also show in the course of the proof that if the annotated program is valid, then a security relevant action is not executed when the ghost state is undefined.

We first present some new definitions that will be used in the course of the proof like extended execution and closest updating co-execution.

Preliminaries In the text below, the program T annotated with level I annotations for policy $\mathcal{P}$ is $\mathrm{T}_{\mathcal{P}}$. Furthermore, $pc(C)$ denotes the value of the program counter and $M(C)$ the method at the top frame of configuration C. Finally, $\sigma(\overrightarrow{gs})$ denotes the value of the ghost state given at the environment σ.

The following property follows from the definition of level I annotations.

Property B.2. Let C be an unexceptional configuration of program T. If $A^{\flat}_{M}[pc(C)] = \epsilon$ in $\mathrm{T}_{\mathcal{P}}$, then $act^{\flat}_{A}(C) = \epsilon$. Let C' be configuration following C in an execution of program T. If C' is unexceptional and $A^{\sharp}_{M}[pc(C)] = \epsilon$ in $\mathrm{T}_{\mathcal{P}}$, then $act^{\sharp}_{A}(C, C') = \epsilon$.

Definition B.3 (Extended Execution)**.** Given an annotated program T_A, a sequence of extended configurations
$(\psi_0, C_0, \sigma_0, \Sigma_0)(\psi_1, C_1, \sigma_1, \Sigma_1)\ldots$ is termed an *extended execution* of T_A, if:

- $(\psi_0, C_0, \sigma_0, \Sigma_0)$ is the initial extended configuration as defined on page 4.6, and
- $\forall i.\, \Gamma^* \vdash (\psi_i, C_i, \sigma_i, \Sigma_i) \rightarrow (\psi_{i+1}, C_{i+1}, \sigma_{i+1}, \Sigma_{i+1})$

That is, any Γ^*-derivation that definition 4.12 refers to is an extended execution.

The projection of an extended execution to its second component isolates the execution of the JVM program, and is described similar to the definition of the first projection function in section 4.5. An extended execution is called *complete* if it executes the precondition (if any) of the instruction at the program counter of its last configuration to completion.

Definition B.4. (Complete Extended Execution) Given a finite execution $E = C_0 \ldots C_{n-1}C_n$ of program T, the extended execution
$X_E = (\psi_0, C'_0, \sigma_0, \Sigma_0)\ldots(\psi_{m-1}, C'_{m-1}, \sigma_{m-1}, \Sigma_{m-1})(\psi_m, C'_m, \sigma_m, \Sigma_{m-1})$ of the annotated program $\mathrm{T}_{\mathcal{P}}$ is the *complete extended execution* of E if $X_E \downarrow 2 = E$ and $\psi_m = \epsilon$.

Given a finite execution $E_n = C_0 \ldots C_n$ of program T, notice that the following hold for the execution $E_{n+1} = C_0 \ldots C_n C_{n+1}$, where $(\epsilon, C_n, \sigma, \Sigma)$ is the last element of X_{E_n}:

1. If C_n is not an application method call or a return, and C_{n+1} is not exceptional, i.e. rule (5) of table 4.4 applies:

$$X_{E_{n+1}} = X_{E_n} \bullet (A_{M(C_{n+1})}[pc(C_{n+1})], C_{n+1}, \sigma, \Sigma) \ \ldots\ (\epsilon, C_{n+1}, \sigma', \Sigma) \quad \text{(B.1)}$$

   for some σ'.

2. If C_n is a return or is exceptional with an exception that can not be handled in the current method, i.e. rule (6) applies:

$$X_{E_{n+1}} = X_{E_n} \bullet (\mathit{Ensures}(\Gamma^*(M(C_n))), C_{n+1}, \sigma_g \uplus \sigma'_l, \Sigma')\ldots\ (\epsilon, C_{n+1}, \sigma', \Sigma') \quad \text{(B.2)}$$

   where $\sigma = \sigma_g \uplus \sigma_l$ for some σ_g and σ_l and $\Sigma = \sigma'_l\Sigma'$.

3. If C_n is an application method call and C_{n+1} is not exceptional, i.e. rule (7) applies:

$$\begin{array}{l} X_{E_{n+1}} = X_{E_n} \bullet \\ \mathit{Requires}(\Gamma^*(M(C_{n+1}))) \cdot A_{M(C_{n+1})}[1], C_{n+1}, \sigma_g \uplus \sigma^0_l, \sigma_l \cdot \Sigma) \\ \quad \ldots\ (\epsilon, C_{n+1}, \sigma', \sigma_l \cdot \Sigma) \end{array} \quad \text{(B.3)}$$

where $\sigma = \sigma_g \uplus \sigma_l$ for some σ_g and σ_l.

4. Finally, if C_n was not exceptional but C_n is exceptional, i.e. rule (8) applies:

$$X_{E_{n+1}} = X_{E_n} \bullet (\epsilon, C_{n+1}, \sigma, \Sigma) \tag{B.4}$$

Constructing the Co-execution A sequence of configuration-automaton state pairs are constructed from a sequence of extended configurations using the function *subw*. This function forms a sequence by sampling the machine configuration and the ghost state whenever one of the two is updated. If the machine configuration changes in consecutive extended configurations, the sequence is extended with the machine configuration and the ghost state of this second If the current extended configuration is the last in the sequence, then the sequence is not extended further. If a configuration induces a preaction, the annotated program $\mathrm{T}_{\mathcal{P}}$ updates the ghost state immediately before transiting to the next configuration (that is "executing the method"). If two consecutive configurations induce a non-exceptional postaction, the ghost state is updated immediately after transiting to the second configuration (that is upon return). However, in the case of an exceptional postaction the update is not immediate. When two consecutive configurations C and C' induce an exceptional action, the new state can not be obtained by sampling the ghost state some time during the extended execution that ends with C'. The reason is that there is no annotation associated with exceptional configurations and the ghost update is done in this case at the precondition of the first instruction of the handler. This precondition is executed *after* at the extended execution of the configuration following C'. In order to sample the ghost value in such a situation, we consider a maximal execution of which the finite execution is a prefix of. This way we get to "peek" to the new value of the ghost state.

Let $E = C_0 \dots C_{j-1} C_j$ be a finite execution and let $X_E = (\psi_0, C'_0, \sigma_0, \Sigma_0) \dots (\psi_k, C'_k, \sigma_k, \Sigma_k)$ be its corresponding extended execution. Notice that the first extended configuration correspond to the execution of $\mathit{Requires}^I_{\langle \texttt{main} \rangle}$. If the last two configurations $(C_{j-1},\ C_j)$ of E do not induce an exceptional action, the sequence of configuration-automaton state pairs corresponding to this extended execution is defined as

$$w(X_E) = (C_0, q_0)\ \mathit{subw}((\psi_1, C'_1, \sigma_1, \Sigma_1) \dots (\psi_k, C'_k, \sigma_k, \Sigma_k))$$

where q_0 is the initial state of $\mathcal{A}_{\mathcal{P}}$ and *subw* is defined below. If C_{j-1} and C_j induce an exceptional action, we extract the co-execution using the complete extended execution X' of $E' = C_0 \dots C_j C_{j+1}$. The value of the ghost state at the last element of X' is taken in this case.

- $\mathit{subw}((\psi_1, C_1, \sigma_1, \Sigma_1) \cdot (\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X') = (C_2, \sigma_2(\overrightarrow{gs})) \cdot \mathit{subw}((\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X')$ if $C_1 \longrightarrow_{\mathrm{JVM}} C_2$
- $\mathit{subw}((\psi_1, C_1, \sigma_1, \Sigma_1) \cdot (\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X') = (C_2, \sigma_2(\overrightarrow{gs})) \cdot \mathit{subw}((\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X')$ if $\psi_1 = (\overrightarrow{gs} := \alpha_1 | \dots | \alpha_k) \cdot \psi_2$ for some $k \neq 1$

- $subw((\psi_1, C_1, \sigma_1, \Sigma_1) \cdot (\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X') = subw((\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X')$ otherwise.
- $subw(\psi, C, \sigma, \Sigma) = \epsilon$

In the definition above, the update of the ghost state causes a sampling only if the update is not done by the last condition of the conditional update. The reason is that when the program is annotated with level I annotations, an update on the ghost state using the last condition of the conditional expression is a stutter.

Definition 4.8 captures all interleavings of the monitor and the program, for a monitor that updates the security state every time a s.r.a. occurs. If a configuration induces a preaction, the update should happen before the transition to the next configuration. If two consecutive configurations induce a postaction, the update should be done after the transition to the latter configuration. The definition aims to specify the interval where the update may be done for the interleaving to be a co-execution. A co-execution is a *closest updating co-execution* if the monitor makes a corresponding transition at the latest possible point when the update is for a preaction and at the earliest possible point when the update is for a postaction.

Definition B.5 (Closest Updating Co-execution)**.** A co-execution is *closest updating co-execution* if the following holds for consecutive pairs $(C_1, q_1)(C_2, q_2)(C_3, q_3)$ (C_4, q_4):

- $act^{\flat}_A(C_1) \in A^{\flat} \wedge (C_2, q_2) \longrightarrow_{\text{JVM}} (C_3, q_3) \Rightarrow (C_1, q_1) \longrightarrow_{\text{AUT}} (C_2, q_2)$
- $act^{\sharp}_A(C_1, C_2) \in A^{\sharp} \wedge \neg Exc(C_2) \Rightarrow (C_2, q_2) \longrightarrow_{\text{AUT}} (C_3, q_3)$
- $act^{\sharp}_A(C_1, C_2) \in A^{\sharp} \wedge Handled(C_2) \Rightarrow (C_2, q_2) \longrightarrow_{\text{JVM}} (C_3, q_3) \wedge$ $(C_3, q_3) \longrightarrow_{\text{AUT}} (C_4, q_4)$

The Proof We now prove that, the configuration-automaton state pairs extracted from a level I annotated program is a co-execution, provided that the annotations are valid and vice versa. What is more, due to the shape of the annotations, we prove that these co-executions are closest updating.

Lemma B.6. *$\mathrm{T}_{\mathcal{P}}$ is valid, if and only if, for every maximal execution E of T, the extracted sequence $w(X_E)$ of the complete extended execution X_E of $\mathrm{T}_{\mathcal{P}}$ is closest updating and $w(X_E) \downarrow 1 = E$.*

Proof. There are two aspects to the proof. First, we are showing that ghost assignments follow security relevant method executions and are performed according to the way described in the policy. Second, that no security relevant action execution happens when the ghost state is undefined if and only if the annotated program is valid.

We proceed by induction on the length of E.

(*Base Case*) When the number of configurations in E is 1, the complete extended execution is the execution of $Requires^{I}_{\langle \texttt{main} \rangle}$ and the precondition of the first instruction of $\langle \texttt{main} \rangle$. The more involved case arises if this precondition is not empty. Otherwise, $w(C_0) = (C_0, q_0)$ by construction. Similarly, if $A^{I}_{\langle \texttt{main} \rangle}[1]$ includes a ghost assignment then the constructed sequence depends on which condition the assignment was done for. Let us consider the case when $k \neq 1$. In this case, the constructed sequence is $w(C_0) = (C_0, q_0)(C_0, \sigma_0(\overrightarrow{gs}))$, where σ_0 is the mapping at the end of the extended execution. By definition, this is a co-execution if $act^{\flat}_{\langle \texttt{main} \rangle}(C_0) \in A^{\flat}$ and $\delta^{\flat}(q_0, act^{\flat}_{\langle \texttt{main} \rangle}(C_0)) = \sigma_0(\overrightarrow{gs})$. This can be proven using the definition of before annotations and the way ConSpec automaton is extracted from symbolic automaton.

(*Induction Hypothesis*) For all executions $E_i = C_0 \dots C_{i-2}C_{i-1}$ of length i such that $i \leq n$ and $act^{\sharp}_{A}(C_{i-2}, C_{i-1})$ is not an exceptional post action, we assume that $w(X_i)$ is a co-execution where $w(X_i) \downarrow 1 = E_i$ if and only if all boolean formulae asserted in the complete extended execution X_i holds except possibly the assertions $Defined^{\sharp}$ and $Defined^{e}$ asserted in the course of the execution of the precondition of $pc(C_{i-1})$.

Notice that this induction hypothesis is sufficient, since no maximal execution can end with an exceptional configuration that is immediately preceded by an exceptionally security relevant API method call. Similarly, for no maximal execution $Defined^{\sharp}$ or $Defined^{e}$ is asserted in the course of the execution of the precondition of $pc(C_{i-1})$. If the maximal execution is one which returns from the $\langle \texttt{main} \rangle$, then $pc(C_{i-1})$ is `return` and hence no definedness precondition. If the maximal execution is one which ends exceptionally, then this exception is not one thrown by a security relevant API method.

(*Inductive Step*) Consider the execution $E_{n+1} = C_0 \dots C_{n-1}C_n$ of T and its corresponding extended execution $X_{E_{n+1}}$.

We consider the different forms of the pair C_{n-1}, C_n:

- *C_{n-1} and C_n are both not exceptional, and C_{n-1} is not an application method call:*
  We have assumed that the statement holds for $E_n = C_0 \dots C_{n-1}$. Since $X_{E_{n+1}}$ is an extension of X_{E_n}, the assertions met in $X_{E_{n+1}}$ hold if and only if assertions met in X_{E_n} and X hold where $X_{E_{n+1}} = X_{E_n} \cdot X$. By the induction assumption, the assertions met in X_{E_n} of $\mathrm{T}_{\mathcal{P}}$ hold if and only if $w(X_{E_n})$ is a co-execution and $w(X_{E_n}) \downarrow 1 = E_n$.

  Let the last element of X_{E_n} be $(\epsilon, C_{n-1}, \sigma, \Sigma)$ for some σ and Σ, executing method of C_n be M and $\mathrm{pc}(C_n)$ be L. Notice that since C_{n-1} is not exceptional, L is not a handler instruction. By the definition of a complete extended execution, the first element of the suffix X is $(A_M[L], C_n, \sigma, \Sigma)$, and its last element is $(\epsilon, C_n, \sigma', \Sigma)$ for some σ' that is determined by the assignments in $A_M[L]$. That is X corresponds to the execution of the annotation sequence

that is associated with L in M: $A_M[L]$. By the definition of *subw* and w:

$$w(X_{E_{n+1}}) \quad = \quad w(X_{E_n})(C_n, \sigma(\overrightarrow{gs})) \cdot subw(X)$$

By the definition of level I annotations,

$$A_M[L] = \quad A^e_M[L-1][1] \cdot A^\sharp_M[L-1][1] \cdot A^\flat_M[L] \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0]$$

In the rest of the argument of this case, we take $A^e_M[L-1][1] = \epsilon$ for simplicity. This annotation otherwise would set g_{pc} to 0, which does not change the argument.

Notice that, again by the definition of level I annotations, $A_M[L]$ contains at most two assignments to the ghost state in this case. (For all L', $A^\sharp_M[L'][1]$ and $A^\flat_M[L']$ can contain at most one ghost assignment (to the ghost state), while $A^\sharp_M[L'][0]$, $A^e_M[L'][0]$ and $A^e_M[L'-1][1]$ can not contain any.) In order to go through all shapes the suffix $subw(X)$ can have, we consider the possible ghost assignments in X:

- $A^\sharp_M[L-1][1] = \epsilon$, $A^\flat_M[L] = \epsilon$
  In this case there are no ghost assignments in $A_M[L]$ and so $subw(X) = \epsilon$ by definition. Then, $w(X_{E_{n+1}}) = w(X_{E_n})(C_n, \sigma(\overrightarrow{gs}))$.
  From this and the induction hypothesis, the following can be concluded: (i) $w(X_{E_{n+1}})) \downarrow 1 = E_{n+1}$ by the definition of $\downarrow$, (ii) $w(X_{E_{n+1}})$ is an interleaving, since the last element of $w(X_{E_n})$ is $(C_{n-1}, \sigma(\overrightarrow{gs}))$ and $C_{n-1} \longrightarrow_{\text{JVM}} C_n$.
  By the definition of the *extract* function:

  $$extract(w(X_{E_{n+1}})) = extract(w(X_{E_n}))act^\sharp_A(C_{n-1}, C_n)act^\flat_A(C_n)$$

  By property B.2, $act^\flat_A(C_n) = \epsilon$ and $act^\sharp_A(C_{n-1}, C_n) = \epsilon$. By the induction hypothesis, $w(X_{E_{n+1}})$ is a co-execution.
  Assume that $w(X_{E_{n+1}})$ is a closest updating co-execution The only way this is possible is that $w(X_{E_n})$ is itself a closes updating (c.u.) co-execution, and $act^\flat_A(C_n) = \epsilon$, $act^\sharp_A(C_{n-1}, C_n) = \epsilon$. (Otherwise there would be ghost updates executed in $w(X_{E_{n+1}}))$). The latter we have already shown to hold. By the induction hypothesis, if $w(X_{E_n})$ is a c.u. co-execution then all assertions (except possibly the definedness assertions $Defined^\sharp$ and $Defined^e$ executed for $pc(C_{n-1})$) hold. Since $A^\flat_M[L] = \epsilon$, there is no $Defined^\flat$ that is asserted in the precondition of $pc(C_{n-1})$, hence the only assertions that should be shown to hold are $Defined^\sharp$ and $Defined^e$ of $pc(C_{n-1})$. If $pc(C_{n-1})$ is not a method invocation instruction, there is no definedness assertions in its precondition, and we are done. If $pc(C_{n-1})$ is a method invocation instruction, either

C_{n-1} is either an application method call or an API method call. In the former case, both *Defined*$^\sharp$ and *Defined*^{e} hold vacuously since the premise of the boolean formula does not hold, that is the object that the method is invoked on is not one of those mentioned in these assertions.

Let us consider the case where C_{n-1} is an API method call. Since there are no jumps to instructions after method calls, $pc(C_{n-1})$ should be $L-1$. By the definition of AFTER annotations, $A^\sharp_M[L-1][1] = \epsilon$ implies that $A^\sharp_M[L-1][0] = \epsilon$, so there is no *Defined*$^\sharp$ for $pc(C_{n-1})$. If it is also the case that $A^e_M[L-1][0] = \epsilon$, we are done. If there is a *Defined*^{e} however, we have to show that this also holds.

Suppose that *Defined*^{e} which comes from $A^e_M[L-1][0]$ does not hold. Then an alternative execution of the program can be constructed by replacing C_n with C'_n where C'_n is exceptional. Since $L-1$ is exceptionally security relevant (otherwise there would be no *Defined*^{e} asserted for C_{n-1}), there is a handler H for $L-1$. Now consider the alternative execution that is archived by extending the execution with C'_{n+1} where $pc(C_{n-1}) = H$: $E' = C_0 \ldots C_{n-1} C'_n C'_{n+1}$. Then $w(X_{E'})$ can not be a co-execution. We reach a contradiction.

– $A^\sharp_M[L-1][1] \neq \epsilon$, $A^\flat_M[L] = \epsilon$
Then the suffix X is as follows:

$$
\begin{array}{ll}
 & ((\overrightarrow{gs} := ce) \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma, \Sigma) \quad (1)\\
\rightarrow_* & ((\overrightarrow{gs} := \alpha_1 | \cdot \cdots \cdot | \alpha_k) \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma, \Sigma) \quad (2)\\
\rightarrow & (A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma', \Sigma) \quad (3)\\
\rightarrow_* & (\epsilon, C_n, \sigma'', \Sigma) \quad (4)
\end{array}
$$

By definition, $subw(X) = \epsilon$ if $k = 1$ and $subw(X) = (C_n, \sigma'(\overrightarrow{gs}))$ otherwise. Notice that $\sigma''(\overrightarrow{gs}) = \sigma'(\overrightarrow{gs})$ since there are no assignments to the ghost state in the steps between (3) and (4) and furthermore if $k = 1$, $\sigma'(\overrightarrow{gs}) = \sigma(\overrightarrow{gs})$, by the definition of level I annotations.

By the definition of AFTER annotations, $A^\sharp_M[L-1][1] \neq \epsilon$ if $M[L-1] =$ `invokevirtual` $(c.m)$ for some class c and method m. That is the instruction at above the current program counter is a method invocation instruction. By the assumption that there are no direct jumps to instructions immediately below method calls, the previous configuration is either a method call (to an API method) or a method return (from an application method).

($\Leftarrow$) This direction is similar to the argument for the case above.

($\Rightarrow$) As is apparent from the execution of X, $subw(X)$ is determined by the value of k above:

1. $k = 1$:
   This corresponds to the case where we have a stuttering if the

ghost state is defined when the assignment begins executing. This type of stuttering is meant to occur when the current call is not to a security relevant action, in order to not to update the state unnecessarily with this assignment. This last condition can be satisfied also if the ghost state is not defined when the assignment begins executing. In this case, for the extracted sequence to be a co-execution, the method return should not be a postaction.

By the definition of *subw*, $subw(X) = \epsilon$ and $w(X_{E_{n+1}})$ is the same as case 1 above. The argument that this is an interleaving and that its first projection is E_n is also identical. The equation B.3.2 also holds. For $w(X_{E_{n+1}})$ to be a co-execution then, we should show that no security relevant actions are induced by the addition of configuration C_n to the execution E_{n-1}. By property B.2, $act^{\flat}_A(C_n) = \epsilon$. If C_{n-1} is a return from an application method, $act^{\sharp}_A(C_{n-1}, C_n) = \epsilon$. The case where C_{n-1} is a method call to an API is more complicated. This case is to prove that, although this instruction has been annotated, in this case the method called as a result of virtual method resolution turned out not to be security relevant.

Let C_{n-1} be $((M, L-1, s \cdot d \cdot s', lv) \cdot R, h^{\flat})$ and C_n be $((M, L, v \cdot s', lv) \cdot R, h^{\sharp})$ for some actual arguments s, some location d, some stack s' and return value v. Notice that there exists a class c' such that c' *defines* $type(h^{\flat}, d).m$ and $type(h^{\flat}, d) <: c$. (If this was not the case, C_n would be exceptional.) Now suppose $(v, c', m, s, h^{\flat}, h^{\sharp})$ is a postaction of the induced automaton $A_{\mathcal{P}}$. Then there should exist, for some names $x, x_1, \ldots x_n$, a symbolic postaction $a^{\sharp}_s = (\tau x, c', m, ((\tau_1\, x_1), \ldots, (\tau_n\, x_n)))$ of A_s such that the type of v is τ, the type of $s[0]$ is τ_1 etc. It would then be the case that $type(h^{\flat}, d) \in RS((c, m), A^{\sharp}_s \setminus A^e_s)$, by the definition of RS. Notice that $\sigma(g_{\text{this}}) = d$ by the execution of $A^{\sharp}[L-1][0]$ in $X_{E_{n-1}}$.

Since $k = 1$, either $\neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p)$ or $\overrightarrow{gs} = \overrightarrow{\perp}$ or both of them holds at (2) where $RS((c, m), A^{\sharp}_s \setminus A^e_s) = \{c'_1, \ldots, c'_p\}$. If only the first holds, at (2), d is not an object of one of these classes, $type(h^{\flat}, d) \notin RS((c, m), A^{\sharp}_s \setminus A^e_s)$. (We assume that $type(h^{\flat}, d) = type(h^{\sharp}, d)$, that is an API call does not change the type of the object it is called on) We reach a contradiction, showing that $act^{\sharp}_A(C_{n-1}, C_n) = \epsilon$. Hence, $extract(w(X_{E_{n+1}}))$ is a co-execution. If both holds, then the return is again not security relevant and $extract(w(X_{E_{n+1}}))$ is a co-execution.

If only the second holds however $act^{\sharp}_A(C_{n-1}, C_n) \in A^{\sharp}$ and $extract(w(X_{E_{n+1}}))$ can not be a co-execution since there is no

outgoing transitions from the undefined state in a ConSpec automaton induced from the symbolic automaton of the policy. In order to rule out this case, we should prove that $\sigma(\overrightarrow{gs}) \neq \bot$. Now we use the assumption that all assertions in $X_{E_{n+1}}$ holds. This is only the case if all assertions of X_{E_n} holds. By the definition of AFTER annotations, $A^\sharp[L-1][0]$ asserts that if g_{this} is of a class which is a member of $RS((c,m), A^\sharp_s \setminus A^e_s$, then $\sigma(\overrightarrow{gs}) \neq \bot$. Hence it can not be the case only the second conjunct holds.

2. $k > 1$:
Let $\sigma(\overrightarrow{gs}) = q$ and $\sigma'(\overrightarrow{gs}) = q'$, by the definition of *subw* and of *extract*:

$$\begin{array}{rcl} w(X_{E_{n+1}}) & = & w(X_{E_n})(C_n, q)(C_n, q') \\ \mathit{extract}(w(X_{E_{n+1}})) & = & \mathit{extract}(w(X_{E_n}))act^\sharp_A(C_{n-1}, C_n)qq'act^\flat_A(C_n) \end{array}$$

In order to show that $w(X_{E_{n+1}})$ is an interleaving, we should prove that there exits an action $a \in A$ such that $\delta(q,a) = q'$. From this, it will also follow that $w(X_{E_{n+1}}) \downarrow 1 = E_{n+1}$. To prove that $w(X_{E_{n+1}})$ is a co-execution, however, we should prove a stronger statement, namely that $\delta^\sharp(q, act^\sharp_A(C_{n-1}, C_n)) = q'$. (This is the only possibility since by property B.2, $act^\flat_A(C_n) = \epsilon$) This is the case when one of the conditions (other than the last condition) of the conditional assignment is satisfied and the ghost state is set accordingly. We show that this is the case only if C_{n-1} is a return from a post security relevant method call and that the ghost state is set correctly.
Since $k > 1$, in the execution segment above, α_1 has the following form: $(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge g_{\text{this}} : c'_i \wedge a \rightarrow \overrightarrow{e}$, where $c'_i \in RS((c,m), A^\sharp_s \setminus A^e_s)$. Note that α_1 holds at (2). This implies that $\overrightarrow{gs} \neq \overrightarrow{\bot}$ at σ. We first show that C_n can not be a return from an application method. (If this was the case the return would be from an application method, hence not security relevant). Assume that this is the case, let this method which is returning be c', m and the object it was called on to be d. (That is the second frame in the activation stack of C_{n-1} is $(M, L-1, s \cdot d \cdot s')$ for some actual arguments s, and some stack s') Since the call was made by the instruction `invokevirtual` $c.m$, it should be the case that c' *defines* $(\mathit{type}(d,h), m)$ where h is the heap at the time of the method call. Notice that $g_{\text{this}} = d$, since it was set to this value by $A^\sharp[L-1][0]$ just before the method call was made and since it is local so could not have been changed during the execution of the application method. (We further assume that the application method does not change the type of the object it is called on) This means that $c' \in RS((c,m), A^\sharp_s \setminus A^e_s)$, which

can not be the case since it is an application method. Hence we reach a contradiction, showing that C_n can not be a return from an application method.
The only possibility left is that C_n is a return from an API method. Let C_{n-1} be $((M, L-1, s \cdot d \cdot s', lv) \cdot R, h^\flat)$ and C_n be $((M, L, v \cdot s', lv) \cdot R, h^\sharp)$ for some actual arguments s, some location d, some stack s', return value v and heaps $h^\flat, h^\sharp$. Let $(c.m) : (\gamma \to \tau)$. Since α_1 is a part of the ghost assignment, the symbolic automaton should include the action
$a_s^\sharp = (\tau x, c_i', m, ((\tau_1\, x_1), \ldots, (\tau_{|\gamma|}\, x_{|\gamma|})))$ for some names $x, x_1, \ldots$ and types $\tau, \tau_1, \ldots$ such that the type of v is τ, the type of $s[0]$ is τ_1 etc. What is more there exists a predicate b and an expression tuple E such that $(a_s^\sharp, b, E) \in \delta_s^\sharp$ and $a = a_b\rho$ where a_b is the boolean formula for predicate b and $\overrightarrow{e_E}$ as defined in section 4.8.1. The substitution $\rho = [v/x, g_0/x_0, \ldots g_{k-1}/x_{n-1}, g_{this}/\texttt{this}]$ by construction. Notice that $\sigma(g_{\text{this}}) = d$ by the execution of $A^\sharp[L-1][0]$ in $X_{E_{n-1}}$ and hence c_i' *defines type*$(h^\flat, d).m$. Thus $(v, c', m, s, h^\flat, h^\sharp)$ is a postaction of the induced automaton $A_{\mathcal{P}}$. We have proven that $act_A^\sharp(C_{n-1}, C_n) \in A^\sharp$.
We are left to prove that $\delta^\sharp(q, act_A^\sharp(C_{n-1}, C_n)) = q'$. Since α_1 holds at (2),

$$\parallel a_b\rho \parallel (C_n, \sigma) = \mathit{true} \Leftrightarrow \parallel b \parallel qIh^\flat h^\sharp = \mathit{true}$$

where $I = [x \mapsto v, x_1 \mapsto s[0], \ldots]$. Using the same interpretation,

$$\parallel \overrightarrow{e_E}\rho \parallel (C_n, \sigma) = q' \Leftrightarrow \parallel E(s_i) \parallel qIh^\flat h^\sharp = q'(s_i)$$

for all security state variables s_i of $\overrightarrow{gs}$. The result then follows from the way a ConSpec automaton is induced by a symbolic automaton.

3. $k = 0$:

Let $\sigma(\overrightarrow{gs}) = q$ and $\sigma'(\overrightarrow{gs}) = q'$, by the definition of *subw* and of *extract*:

$$\begin{aligned} w(X_{E_{n+1}}) &= w(X_{E_n})(C_n, q)(C_n, q') \\ \mathit{extract}(w(X_{E_{n+1}})) &= \mathit{extract}(w(X_{E_n}))act_A^\sharp(C_{n-1}, C_n)qq'act_A^\flat(C_n) \end{aligned}$$

In order to show that $w(X_{E_{n+1}})$ is an interleaving, we should prove that there exits an action $a \in A$ such that $\delta(q, a) = q'$. From this, it will also follow that $w(X_{E_{n+1}}) \downarrow 1 = E_{n+1}$. To prove that $w(X_{E_{n+1}})$ is a co-execution, however, we should prove a stronger statement, namely that $\delta^\sharp(q, act_A^\sharp(C_{n-1}, C_n)) = q'$. (This is the only possibility since by property B.2, $act_A^\flat(C_n) = \epsilon$)

It is possible to show in this case that C_n is a return from a security relevant method call by a similar argument. The idea is that if C_n was a return from an application method call, the last condition of the conditional assignment would instead have been satisfied, hence k would have been 1. Since this is not the case, we know that C_n is a return from an API call. What is more, let this method be $c'.m$. Then $c' \in RS((c,m), A^\sharp_s \setminus A^e_s$. Notice that none of the conditions in the assignment hold, that is $k = 0$, if either $\sigma(\overrightarrow{gs}) = \bot$ or $\sigma(\overrightarrow{gs}) \neq \bot$ but the guards are not satisfied. In both cases, after this assignment the ghost state is undefined: $\sigma'(\overrightarrow{gs}) = \bot$.
The case that $k = 0$ may only occur if the ghost state becomes undefined since the return from the API method was a violation. Since the last condition does not hold, we know that the ghost state was not undefined at σ and we know that the object the method was called is of one of the classes in $RS((c,m), A^\sharp_s \setminus A^e_s$. This means that the call is security relevant. Since none of the conditions before the last was satisfied, this is a violating postaction. By the definition of the way a ConSpec automaton is extracted from a symbolic automaton, any such state has a transition to the undefined state. Hence $\delta^\sharp(q, act^\sharp_A(C_{n-1}, C_n)) = q'$, where $q' = \overrightarrow{\bot}$ and we are done.
Hence, $w(X_{E_{n+1}})$ is a co-execution.

  - The cases where $A^\sharp_M[L{-}1][1] = \epsilon$, $A^\flat_M[L] \neq \epsilon$ and where $A^\sharp_M[L{-}1][1] \neq \epsilon$, $A^\flat_M[L] \neq \epsilon$ are proved similar to the case above.

- C_{n-1} and C_n are both not exceptional, and C_{n-1} is an application method call:
  This case is similar to the one above when C_{n-1} is not an application method call.

- C_{n-1} is exceptional, while C_n is not exceptional: The only interesting subcase of this case is when C_{n-2} is an API method call and $act^\sharp_A(C_{n-2}, C_{n-1}) \neq \epsilon$. In this case, notice that $w(X_{E_{n+1}})$ is not an extension of $w(X_{E_n})$, but rather of $w(X_{E_{n-1}})$, by the definition of w function.

- C_{n-1} is not exceptional, while C_n is exceptional: The only interesting subcase of this case is when C_{n-2} is an API method call and $act^\sharp_A(C_{n-2}, C_{n-1}) \neq \epsilon$. Then the special construction described for $w(X)$ when X has an exceptional configuration as last element and the element before the last is an API call is used.

□

Proposition B.7. *Given a program* T *and a policy* $\mathcal{P}$*, if for every execution* E *of* T *there exists a co-execution* w *of* T *and* $\mathcal{A}_\mathcal{P}$ *such that* $w \downarrow 1 = E$*, then the sequence* $w(X_E)$ *extracted from the extended execution* X_E *corresponding to this execution is also a co-execution such that* $w(X_E) \downarrow 1 = E$ *and* $w(X_E)$ *is closest updating.*

Proof. For each co-execution, a closest updating co-execution can be constructed by postponing the transition of the monitor for a preaction until the configuration which calls this security relevant method is reached and by performing the transition of the monitor right after the return of the security method call if the update is for a postaction.

□

Theorem 4.15 (Correctness of Level I Annotations) *The level I annotated program* T *for policy* $\mathcal{P}$ *is valid, if and only if,* T *adheres to* $\mathcal{P}$*.*

Proof. The result follows in one direction from theorem 4.10, proposition B.7 and lemma B.6; the other direction follows from lemma B.6 and theorem 4.10. □

B.3.3 Proof of Theorem 4.18

Theorem 4.18 *The level II annotated program* T *with embedded state* $\overrightarrow{ms}$ *is valid if and only if for each execution* E *of* T*, the sequence* $w(E, \overrightarrow{ms})$ *is a method-local co-execution.*

Proof. (Sketch) The idea of the proof is to sample pre- and post-actions from E, immediately preceded and followed by a sample of the embedded state $\overrightarrow{ms}$. The sequence extracted in this way is almost a potential derivation, but in the case of a postaction followed, some time later, by a preaction, an intermediate automaton state may be missing. It is not clear, however, how to sample this state. Also, it is necessary to ensure that embedded state updates do not cross method boundaries. To this end, extracted sequences need to be completed by (a) missing intermediate automaton states, and (b) indicators of method boundary crossings at: method invocations that are not security relevant actions, return instructions, exceptional configurations with an unhandled exception, and at the first instruction of each method.

First, we note that the embedded state $\overrightarrow{ms}$ is equal to the ghost state $\overrightarrow{gs}$ at sampling points if and only if the synchronisation assertions added at level II hold. We show in the proof of theorem 4.15 that the ghost state and machine configurations constitute a co-execution if and only if the program annotated with level I annotations is valid. If the program annotated with level II annotations is valid then the sampling of the embedded stated as described above amounts to taking the co-execution of the ghost state and the program and "skipping" some ghost updates, which the embedded state does not follow (as the sampling of the embedded state is not done as frequently). Then $extract_{\mathrm{II}}$ applied to this sequence falls in the set stated in definition 4.17.

($\Leftarrow$) In this direction, we show the result by taking any execution E of a level II annotated program, which is valid. Since level II annotations include level I annotations, by theorem B.6 one can construct a co-execution of this program and the automaton $\mathcal{A}_{\mathcal{P}}$, in the sense of section 4.5, using the ghost state. By the placement of the synchronisation annotations, the value of the embedded state can be inferred at sampling points, using the value of the ghost state. Then, it is left to show that for the embedded state to be a monitor for the policy, it is sufficient that the embedded state is in synch with the ghost state at the points where level II annotations are asserted. For instance, the ghost state gets updated for a preaction, immediately before the action and by the validity of the level II annotations, at this point the embedded state is equal to the ghost state, hence if the embedded state has been a monitor until this point, this property will be preserved for the next action.

The proof is by induction on the length of the execution:

(*Base Case:*) The sequence produced for an execution C_0 depends on whether it is a sampling point or not. C_0 is not preceded by any configuration, and is not exceptional. Therefore, if $\mathrm{pc}(C_0)$ is an `invokevirtual` or a `return` C_0 is a sampling point and the sequence is $w(C_0, \overrightarrow{ms}) = (C_0, q)$ where $q = C_0(\overrightarrow{ms})$. Otherwise, $w(C_0, \overrightarrow{ms}) = (C_0, q_0)$. Let us carry out the case when this is a `return`. By the synchronisation annotation asserted by the *Ensures* clause of $\langle$`main`$\rangle$, $\overrightarrow{ms} = \overrightarrow{gs}$ at C_0. Since there are no ghost assignments associated with a `return` instruction, the ghost state is still the initial state of the automaton at C_0. The result of applying the extract function is then $extract_{\mathrm{II}}(w) = q_0\mathtt{brk}q_0$.

(*Induction Hypothesis:*) For all executions E_k of length $k \leq n$, if the level II annotation of T with embedded state $\overrightarrow{ms}$ is valid, then $w(E_k, \overrightarrow{ms})$ is a method-local co-execution.

(*Inductive Step:*) Assume that the level II annotation of T with embedded state $\overrightarrow{ms}$ is valid and consider the execution $E_{n+1} = C_0 \dots C_n$. The sequence $w(C_0 \dots C_n, \overrightarrow{ms})$ is built by extending $w(E_n, \overrightarrow{ms})$ with the pair (C_n, q). Notice that since $w(E_n, \overrightarrow{ms})$ is a method-local co-execution, the result of applying the $extract_{\mathrm{II}}$ function returns a sequence ending with some state q, except the case where C_{n-1} is an API method call that induces a preaction. If C_n is a sampling point, the state component q of this pair is $C_n(\overrightarrow{ms})$; the state component is the same as the state component of the last pair of $w(C_0 \dots C_{n-1}, \overrightarrow{ms})$, if C_n is not a sampling point. By lemma B.6, we know that $w(X_{E_n})$ is a closest updating co-execution and $w(X_{E_n}) \downarrow 1 = E_n$. Let the last element of X_{E_n} be $\epsilon, C_n, \sigma, \Sigma$ for some σ and Σ.

We consider the different cases for the pair C_{n-1}, C_n. Notice that for all cases except the last, C_n is a sampling point.

- C_n is an API method call and C_{n-1} is not a method call: By the definition of $extract_{\mathrm{II}}$,

$$extract_{\mathrm{II}}(w(E_n, \overrightarrow{ms})) = extract_{\mathrm{II}}(w(C_0 \dots C_{n-1}, \overrightarrow{ms}))\, C_n(\overrightarrow{ms})act^{\flat}(C_n)$$

By the validity assumption and the way level II annotations are inserted, $C_n(\overrightarrow{ms}) = \sigma(\overrightarrow{gs})$.

- *Unhandled*(C_n): `invokevirtual` instruction or if $n = 0$.
- C_n is not of the above: In this case, C_n is not a sampling point. Furthermore, by the definition of $extract_{\mathrm{II}}$: hence $extract_{\mathrm{II}}(w(E_n, \overrightarrow{ms})) = extract_{\mathrm{II}}(w(E_{n-1}, \overrightarrow{ms}))$ and the claim holds by the induction hypothesis.
- The other cases are similar.

($\Rightarrow$) The argument goes as follows: we take an arbitrary method-local co-execution and show that any execution that yields such a co-execution validates its assertions. For instance, as a base case, take q`brk`q. By definition of $extract_{\mathrm{II}}$, the sequence that yields this co-execution includes one and only one configuration which is an application method call. There are no other method calls (if this was the case the resulting co-execution would contain more automaton states.). Since the sampling begins with the initial automaton state q_0, q should be q_0. There are no s.r.a's and the ghost state is also the initial state throughout the execution, thus validating both the assertions on the ghost state being defined (if any) and the synchronisation assertion immediately before the method call. (Notice that there are no other assertions as there are no other states extracted and hence no other sampling points.) □

B.3.4 Proof of Theorem 4.22

Theorem 4.22 *Let* T *be a program, $\mathcal{P}$ a ConSpec policy, and I(*T*,$\mathcal{P}$) denote program* T *inlined for policy $\mathcal{P}$. The level III annotation of I(*T*,$\mathcal{P}$) is locally valid, and validity is efficiently checkable.*

Proof. (Sketch) We show that the verification conditions resulting from the level III annotation of I(T,$\mathcal{P}$) are valid and efficiently checkable. To simplify the presentation, we consider here post-actions only; the argument is easily adapted to pre-actions and exception actions.

Notice that for level III annotated programs, every instruction is annotated by a non-empty sequence of logical assertions alternated with ghost variable assignments, always starting and ending with a logical assertion. Notice also that $Ensures(\Gamma^*(M))$ and $Exsures(\Gamma^*(M))$ are all equal to the synchronization assertion $\overrightarrow{gs} = \overrightarrow{ms}$ for fully annotated programs. The first and last elements of the annotation sequence of $Requires(\Gamma^*(M))$ is also the synchronization assertion (except for $\langle$`main`$\rangle$, in which case $last(Requires(\Gamma^*(M)))$ is again $\overrightarrow{gs} = \overrightarrow{ms}$). Similarly, notice that for all instructions L, where L is not the label of an inlined instruction and is not a security relevant action, $last(A_M^{III}[L])$ is the synchronization assertion.

We assume that the return instruction is not the first instruction of an exception handler, the last element in its annotation sequence is the synchronisation annotation. We also assume that the inlined instructions do not raise exceptions.

Then, a level III annotation of I(T,$\mathcal{P}$) gives rise to a set of verification conditions described as follows.

First, there are three types of verification conditions arising from method compositionality, namely:

- $last(Requires(\Gamma^*(M))) \Rightarrow head(A_M^{III}[1])$,
- $last(A_M^{III}[R]) \Rightarrow Ensures(\Gamma^*(M))$,
- For all instructions L that is not a method call and that can raise an unhandled exception $last(A_M^{III}[L]) \Rightarrow Exsures(\Gamma^*(M))$

where R is the label of the return instruction in method M, and where *last* is a function on sequences returning the last element. The inlined instructions are assumed not to raise any exceptions, so no verification condition for exception raising is generated by these. Additionally, only inlined instructions and method calls change the embedded monitor state, hence the simple form of the verification conditions of the latter type. In the first two cases and in the last case when L is not the label of a method call, the antecedent and the consequent are (syntactically) equal to the synchronisation assertion. These verification conditions are therefore valid, and validity is efficiently checkable.

Second, every ghost variable assignment $\overrightarrow{g} := ce$ gives rise to a verification condition. If $\alpha \cdot (\overrightarrow{g} := ce) \cdot \alpha'$ is a subsequence of $A_M^{III}[L]$ for some L where α and α' are logical assertions, then $\alpha \Rightarrow \alpha'[ce/\overrightarrow{g}]$ is a verification condition. Due to the normalization performed in the construction of the level III annotation, α must contain a conjunct $\alpha'[ce/\overrightarrow{g}]$. Such verification conditions are therefore valid, validity being efficiently checkable.

Third, every non-method-call instruction $M[L]$ gives rise to a verification condition $last(A_M^{III}[L]) \Rightarrow wp(M[L])$. There are three cases to be considered: (a) if $M[L]$ is a non-inlined instruction with non-inlined successor instructions only, $last(A_M^{III}[L])$ is syntactically equal to $wp(M[L])$ by construction; (b) if $M[L]$ is a non-inlined instruction followed by an inlined instruction (in the case of post-actions only, the latter indicates the beginning of an inlined block serving to record the current values of the parameters and the object with which the following potentially security relevant instruction is called), then the synchronization assertion $\overrightarrow{gs} = \overrightarrow{ms}$ must appear as a conjunct in both $last(A_M^{III}[L])$ and $wp(M[L])$, and the only other conjuncts in the latter must be either of the shape $Defined^\sharp$ or s[i] = s[i]; (c) if $M[L]$ is an inlined instruction, $last(A_M^{III}[L])$ must contain a conjunct $wp(M[L])$ by construction. In all three cases, the verification condition is valid, validity being efficiently checkable; the only interesting case here is presented by $Defined^\sharp$, the consequent $\overrightarrow{gs} \neq \overrightarrow{\perp}$ of which is implied by $\overrightarrow{gs} = \overrightarrow{ms}$. Similarly, every non-method call instruction $M[L]$ that can raise an exception which is handled by the handler at label H gives rise to the verification condition $last(A_M^{III}[L]) \Rightarrow head(A_M^{III}[H])$. By the assumption that the inlined instructions do not raise an exception, this instruction can not be an inlined instruction. There are two cases to consider: (a) if

$M[H]$ is a non-inlined instruction, then both the antecedent and the consequent are the synchronisation annotation; (b) if the handler $M[H]$ is an inlined instruction (which is possible only if it is the first instruction a code inlined for a potentially preaction occuring in the original handler) this case becomes a subcase of the proof for pre-actions, handled similar to the final part of this proof.

Finally, every method-call instruction $M[L]$ calling some method M' gives rise to three types of verification conditions. If the method call is not potentially post-security relevant, these are:

- $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{Requires}(\Gamma^*(M'))$,
- $\mathit{Ensures}(\Gamma^*(M')) \Rightarrow \mathit{head}(A_M^{III}[L+1])$, and
- For all handler instructions H of L, $\mathit{Exsures}(\Gamma^*(M')) \Rightarrow \mathit{head}(A_M^{III}[H])$

In the first two formulae, the antecedent and the consequent are (syntactically) equal by construction, and hence valid. The last set of verification conditions are valid and efficiently checkable by the argument for the case when $M[L]$ is not a method-call presented above where $\mathit{last}(A_M^{III}[L])$ should be replaced by $\mathit{Exsures}(\Gamma^*(M'))$.

If $M[L]$ is calling some method M' which is potentially security relevant, the three types of verification conditions are

- $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{Requires}(\Gamma^*(M')) \wedge \phi$,
- $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \Rightarrow \mathit{head}(A_M^{III}[L+1])$, and
- For all instruction handler instructions H of L, $\mathit{Exsures}(\Gamma^*(M')) \wedge \phi \Rightarrow \mathit{head}(A_M^{III}[H])$

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{this} = r_{this})$. Notice that the invoked method does not change the local variables and the evaluation stack of the caller method (except for popping arguments from the stack and pushing its return value). Then a formula mentioning variables not changed by the invoked method (such as ϕ) can be added to both the pre-and postconditions of the invoked method [10].

The first of these conditions is again easy to show valid, since $\mathit{Requires}(\Gamma^*(M'))$ and all conjuncts in ϕ also appear as conjuncts in $\mathit{last}(A_M^{III}[L])$ by construction. The third set of verification conditions are similar to the last cases of the argument above, when $M[L]$ is calling a non-potentially security relevant action. The only really involved case in the whole proof is the second verification condition.

Let $\alpha_1, \ldots, \alpha_m$ be the guarded expressions $g_{\text{this}} : c'_i \wedge\ a_b\rho_i \rightarrow \overrightarrow{e_E}\rho_i$, $1 \leq i \leq m$, and α be $\neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \rightarrow \overrightarrow{gs}$, all induced by the policy for the instruction $M[L] = \texttt{invokevirtual}\ (c.m)$ as described in section 4.8.1 (cf. After Annotations). Then the second element of $A_M^{III}[L+1]$ must be a ghost assignment $\overrightarrow{gs} := ce$ where ce is the conditional expression $\alpha_1 \mid \cdots \mid \alpha_m \mid \alpha$. The block inlined immediately after the (potentially post-security relevant) instruction $M[L]$ has the important property that its weakest pre-condition with respect to the head

assertion of the first instruction following the block (which is the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$) is the logical assertion

$$\begin{array}{ll} & \bigwedge_{1 \leq i \leq m} r_{\text{this}} : c'_i \wedge \ a_b \rho'_i \ \rightarrow \ \overrightarrow{gs} = \overrightarrow{e_E} \rho'_i \\ \wedge & \neg (r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \ \rightarrow \ \overrightarrow{gs} = \overrightarrow{ms} \end{array}$$

where the substitution ρ'_i is defined as $[s[0]/x, r_0/x_0, \ldots r_{n-1}/x_{n-1}, r_{this}/\texttt{this}, \overrightarrow{ms}/\overrightarrow{gs}]$ if $r = (\tau\, x)$ and as $[r_0/x_0, \ldots r_{n-1}/x_{n-1}, r_{this}/\texttt{this}, \overrightarrow{ms}/\overrightarrow{gs}]$ if $r = \mathit{void}$. Therefore, $\mathit{head}(A_M^{III}[L+1])$ must be the logical assertion

$$\begin{array}{ll} & \phi \\ \wedge & \mathit{Defined}^\sharp \\ \wedge & \bigwedge_{1 \leq i \leq m} r_{\text{this}} : c'_i \wedge \ a_b \rho'_i \ \rightarrow \ ce = \overrightarrow{e_E} \rho'_i \\ \wedge & \neg (r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \ \rightarrow \ \ ce = \overrightarrow{ms} \end{array}$$

where ϕ is as explained above, and where ce is the tuple of conditional expressions $\overrightarrow{ce_i}$, obtained from ce by replacing each expression vector $\overrightarrow{e_E}$ occurring in ce with its i-th component. Now, validity of the verification condition $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \Rightarrow \mathit{head}(A_M^{III}[L+1])$ is established as follows. The first conjunct ϕ (actually a set of conjuncts) of $\mathit{head}(A_M^{III}[L+1])$ appears as a conjunct in $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$. The second conjunct $\mathit{Defined}^\sharp$ is implied by $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$ because $\mathit{Ensures}(\Gamma^*(M'))$ is $\overrightarrow{gs} = \overrightarrow{ms}$, which implies $\overrightarrow{gs} \neq \overrightarrow{\perp}$. Every conjunct $r_{\text{this}} : c'_i \wedge \ a_b \rho'_i \ \rightarrow \ ce = \overrightarrow{e_E} \rho'_i$ is valid under the equalities of $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$, since then every guard $r_{\text{this}} : c'_i \wedge \ a_b \rho'_i$ matches exactly the guard of α_i, and $\overrightarrow{e_E} \rho_i$ is equal to $\overrightarrow{e_E} \rho'_i$. Validity can thus be easily checked mechanically by simple equational reasoning and (syntactic) guard matching. Finally, validity of the conjunct $\neg (r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \ \rightarrow \ ce = \overrightarrow{ms}$ is established similarly.

When $M[L]$ can give rise to an exceptional postaction, the last set of verification conditions look slightly different. Notice that our inliner inserts a handler for each such potentially security relevant instruction that handles all types of exceptions. Let the label of the first instruction of this handler to be H for the instruction $M[L]$, then the three verification conditions are:

- $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{Requires}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L)$ and
- $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L) \Rightarrow \mathit{head}(A_M^{III}[L+1])$, and
- $\mathit{Exsures}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L) \Rightarrow \mathit{head}(A_M^{III}[H])$

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{this} = r_{this})$. The non-trivial case is then to show that the third verification condition is valid and efficiently checkable. This argument is similar to the argument made above for the non-exceptional case. □

Bibliography

[1] I. Aktug and D. Gurov. Towards state space exploration based verification of open systems. In *Proc. of the 4th International Workshop on Automated Verification of Infinite-State Systems (AVIS'05)*, To appear, 2005.

[2] I. Aktug and J. Linde. An inliner tool for mobile platforms. Available at `http://www.csc.kth.se/~irem/S3MS/Inliner/`

[3] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In F. Piessens and F. Massacci, editors, *Proc. of The First Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197-1 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, 2007.

[4] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *Proc. 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 380–397. Springer Verlag, 2004.

[5] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11 (1):147–167, 1989.

[6] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[7] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[8] H. R. Andersen. Partial model checking. In *Proc. of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95)*, page 398, Washington, DC, USA, 1995. IEEE Computer Society.

[9] F. Y. Bannwart and P. Müller. A logic for bytecode. Technical Report 469, ETH Zurich, 2004. Available at `http://www.sct.inf.ethz.ch/publications/`

[10] F. Y. Bannwart and P. Müller. A logic for bytecode. In *Proc. of Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'05)*, volume 141-1 of *ENTCS*, pages 255–273, 2005.

[11] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, TU München, 2007.

[12] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–314, 2005.

[13] L. Bauer, J. Ligatti, and D. Walker. Composing expressive run-time security policies. *ACM Transactions on Software Engineering and Methodology*, 2008. To appear, Available at `http://www.cse.usf.edu/~ligatti/papers/polymer-tosem.pdf`

[14] O. Bernholtz and O. Grumberg. Branching time temporal logic and amorphous tree automata. In *Proc. of the 4th International Conference on Concurrency Theory (CONCUR '93)*, volume 715, pages 262–277, London, UK, 1993. Springer Verlag-Verlag.

[15] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[16] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. of the 8th International Conference on Concurrency Theory (CONCUR '97)*, pages 135–150, London, UK, 1997. Springer Verlag-Verlag.

[17] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *ACM SIGPLAN Notices*, 38(1):62–73, 2003.

[18] G. Boudol and K. Larsen. Graphical versus logical specifications. *Theoretical Computer Science*, 106:3–20, 1992.

[19] J.C. Bradfield and C.P. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

[20] E. Bretagne, A. El Marouani, P. Girard, and J.-L. Lanet. Pacap purse and loyalty specification. Technical Report V 0.4, Gemplus, 2000.

[21] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.

[22] D. Bustan and O. Grumberg. Applicability of fair simulation. In *Proc. of 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 401–414. Springer Verlag-Verlag, 2002.

[23] S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, University of Edinburgh, 1993.

[24] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample–guided abstraction refinement. In *Proc. of the International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag-Verlag, 2000.

[25] E.M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (5):1512–1542, 1994.

[26] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proc. 9th IFIP Symp. Protocol Specification, Verification and Testing*, 1989.

[27] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Proc. of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 24–37, New York, NY, USA, 1990. Springer Verlag-Verlag New York, Inc.

[28] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

[29] P. Cousot. Verification by abstract interpretation. In N. Dershowitz, editor, *International Symposium on Verification – Theory & Practice - Honoring Zohar Manna's 64th Birthday*, number 2772 in Lecture Notes in Computer Science, pages 243–268. Springer Verlag, 2003.

[30] M. Dam. Fixed points of Büchi automata. In *Proc. of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '92)*, volume 652 of *Lecture Notes in Computer Science*, pages 39–50, 1992.

[31] M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140(2):95–114, 1998.

[32] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In H. Langmaack, A. Pnueli, and W.-P. de Roever, editors, *Compositionality: the Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 150–185. Springer Verlag-Verlag, 1998.

[33] M. Dam and D. Gurov. Compositional verification of CCS processes. In *Proc. of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI '99)*, pages 247–256. Springer Verlag-Verlag, 2000.

[34] D. Dams and K.S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 335–344, Los Alamitos, CA, 2004. IEEE Computer Society Press.

[35] D. Dams and K.S. Namjoshi. Automata as abstractions. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385, pages 216–232. Springer Verlag, 2005.

[36] F. Diotalevi. Contract enforcement with AOP. Available at `http://www-128.ibm.com/developerworks/library/j-ceaop/`

[37] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science, vol. B, Formal Models and Semantics*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[38] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Proc. of the 32nd Annual Symposium on Foundations of Computer Science*, IEEE, pages 368–377. Computer Society Press, 1991.

[39] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Dep. of Computer Science, Cornell University, 2004.

[40] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the IEEE Symposium on Security and Privacy*, page 246. IEEE Computer Society, 2000.

[41] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the Workshop on New Security Paradigms (NSPW '99)*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[42] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV '00)*, number 1855 in Lecture Notes in Computer Science, pages 232–247. Springer Verlag, 2000.

[43] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503, Vienna, Austria, 2006.

[44] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software (TACS '01)*, number 2215 in Lecture Notes in Computer Science, pages 316–339. Springer Verlag, 2001.

[45] J. Esparza and A. Podelski. Efficient algorithms for pre and post on interprocedural parallel flow graphs. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 1–11, New York, NY, USA, 2000. ACM.

[46] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[47] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[48] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1196–1250, 1999.

[49] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction–based model checking using modal transition systems. In *Proc. of the 12th International Conference on Concurrency Theory (CONCUR '01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, 2001.

[50] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition*. The Java Series. 2005.

[51] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3): 843–871, 1994.

[52] O. Grumberg and S. Shoham. Monotonic abstraction-refinement for CTL. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 546–560. Springer Verlag-Verlag, 2004.

[53] D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7): 840–868, 2008.

[54] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 7–16, June 2006.

[55] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

[56] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and models of concurrent systems*, volume 13, pages 477–498. Springer Verlag, New York, NY, USA, 1985.

[57] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6-2: 158–173, 2004.

[58] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[59] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pages 342–356, London, UK, 2002. Springer Verlag-Verlag.

[60] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5): 279–295, 1997.

[61] M. Huisman and D. Gurov. Composing modal properties of programs with procedures. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2007)*, 2008. To appear.

[62] M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, FASE'04*, number 2984 in Lecture Notes in Computer Science, pages 84–98. Springer Verlag, 2004.

[63] M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proc. of the 10th European Symposium on Programming Languages and Systems (ESOP '01)*, volume 2028, pages 155–169, London, UK, 2001. Springer Verlag-Verlag.

[64] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '98)*, pages 67–74, New York, NY, USA, 1998. ACM Press.

[65] R. Kaivola. On modal mu-calculus and Büchi tree automata. *Information Processing Letters*, 54(1):17–22, 1995.

[66] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proc. of the 5th International Conference on Computer Aided Verification (CAV '93)*, pages 97–109, London, UK, 1993. Springer Verlag-Verlag.

[67] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer Verlag-Verlag, Berlin, Heidelberg, and New York, 1997.

[68] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. The Moped Tool. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`

[69] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proc. of the 1st International Workshop on Run-time Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, July 2001.

[70] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[71] O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):87–128, 2000.

[72] O. Kupferman and M. Y. Vardi. Robust satisfaction. In *Proc. of the 10th International Conference on Concurrency Theory (CONCUR '99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 383–398, London, UK, 1999. Springer Verlag-Verlag.

[73] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems*, 22(1):87–128, 2000.

[74] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of ACM*, 47(2):312–360, 2000.

[75] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[76] L. Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80)*, pages 174–185, New York, NY, USA, 1980. ACM.

[77] K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 232–246. Springer Verlag-Verlag, 1989.

[78] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[79] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[80] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proc. of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, pages 355–373, Sep 2005.

[81] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition.* Sun Microsystems, Inc., 1999.

[82] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, 2005.

[83] F. Martinelli. Analysis of security protocols as open systems. *Theoretical Compututer Science*, 290(1):1057–1106, 2003.

[84] F. Massacci and K. Naliuka. Multi-session security monitoring for mobile code. Technical Report DIT-06-067, UNITN, 2006. Available at `http://eprints.biblio.unitn.it/archive/00001091/`

[85] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proc. of The 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, October 2007.

[86] J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, Jan 1996.

[87] M. Méndez, J. Navas, and M.V. Hermenegildo. An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In M. Huisman and F. Spoto, editors, *Bytecode 2007*, pages 51–66, 2007.

[88] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. *Computation Theory*, 208:pages 157–168, 1984.

[89] D. E. Müller and P. E. Schupp. Alternating automata on infinite objects, determinacy and rabin's theorem. In *Automata on Infinite Words, Ecole de Printemps d'Informatique Théorique,*, volume 192, pages 100–107, London, UK, 1985. Springer Verlag-Verlag.

[90] G. C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[91] J. Obdržáek. Model checking Java using pushdown systems. In *Proceedings of FTfJP'02*, Malaga, June 2002. Available as Technical Report NIII-R0204, Computing Science Department, University of Nijmegen.

[92] C. E. Ortiz. An introduction to Java Card technology. `http://developers.sun.com/mobility/javacard/articles/javacard1/`, 2003.

[93] C. E. Ortiz. A survey of Java ME today. `http://developers.sun.com/mobility/getstart/articles/survey/`, 2007.

[94] AspectJ Project Home Page. *http://eclipse.org/aspectj*. URL `http://eclipse.org/aspectj`.

[95] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*. Springer Verlag, 1984.

[96] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential java. In S. D. Swierstra, editor, *Proc. of the 8th European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, March 1999.

[97] D. Polansky. Implementation of the model checker for pushdown systems and alternation-free mu-calculus. Master's thesis, FI MU Brno, 2000.

[98] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, Lecture Notes in Computer Science. Springer Verlag, 2005.

[99] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416–430, 2000. ISSN 0164-0925.

[100] T. Rezk. *Verification of Confidentiality Policies for Mobile Code*. PhD thesis, INRIA Sophia Antipolis and University of Nice Sophia Antipolis, November 2006.

[101] J.H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[102] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[103] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 15–28, New York, NY, USA, 2003. ACM.

[104] C. Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer Verlag-Verlag, 2001.

[105] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 541–545, Edinburgh, UK, 2005. Springer Verlag. Tool paper.

[106] P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[107] W. Thomas. *Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics*, chapter Automata on Infinite Objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[108] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.

[109] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999. Available at
`http://www.sable.mcgill.ca/soot/`

[110] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Proc. of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, Lecture Notes in Computer Science. Springer Verlag, 2008. to appear.

[111] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of the First Symposium on Logic in Computer Science*, pages 332–344, 1986.

[112] M. Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania, 2000. Supervisor-Sampath Kannan and Supervisor-Insup Lee.

[113] S. Winwood, G. Klein, and M. M. T. Chakravarty. On the automated synthesis of proof-carrying temporal reference monitors. In G. Puebla, editor,

Logic-Based Program Synthesis and Transformation, 16th International Symposium (LOPSTR'06), volume 4407 of *Lecture Notes in Computer Science*, pages 111–126. Springer Verlag, 2006.

[114] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths (extended abstract). In *Proc. of the 24th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE, 1983.

[115] A. Zobel, C. Simoni, D. Piazza, X. Nunez, and D. Rodriguez. Business case and security requirements. Public Deliverable D5.1.1, S3MS, http://s3ms.org, October 2006.