



**International Doctorate School in Information and
Communication Technologies**

Department of Information Engineering and Computer Science
(DISI) - University of Trento

**SECURITY-BY-CONTRACT USING AUTOMATA MODULO
THEORY**

Ida Sri Rejeki Siahaan

Advisor:

Prof. Fabio Massacci

University of Trento

February 2010

Abstract

Trust without control is a precarious solution to human nature. This belief has led to many ways for guaranteeing secure software such as statically analyzing programs to check that they comply to the intended specifications which results in software certification. One problem with this approach is that the current systems can only accept all or nothing without knowing what the software is doing. Another way to complement is by run-time monitoring such that programs are checked during execution that they comply to security policy defined by the systems. The problem with this approach is the significant overhead which may not be desirable for some applications.

This thesis describes a formalism, called Automata Modulo Theory, that allows us to have model of what programs do in more precise details thus giving semantics to certification. Automata Modulo Theory allows us to define very expressive policies with infinite cases while keeping the task of matching computationally tractable. This representation is suitable for formalizing systems with finitely many states but infinitely many transitions. Automata Modulo Theory consists of a formal model, two algorithms for matching the claims on the security behavior of a midlet (for short contract) with the desired security behavior of a platform (for short policy), and an algorithm for optimizing policy.

The prototype implementations of Automata Modulo Theory matching using language inclusion and simulation have been built, and the results from our experience with the prototype implementations are also evaluated in this thesis.

Keywords

language-based security, malicious code, security and privacy policies

Acknowledgments

Prof. Fabio Massacci, for guidance and advice on research and becoming researcher. N. Bielova, M. Dalla Torre, and S. Vogl for implementing the matching prototype. M. Roveri, S. Toneta, and A. Cimatti for the support in the usage of the NuSMV and MathSAT libraries. R. Sebastiani for his insightful comments while I was beginning my work.

Anonymous reviewers for the insightful comments on our papers that helped to improve the presentation in this thesis.

The Projects EU-FP6-IST-STREP-S3MS, EU-FP6-IP-SENSORIA, EU-FP7-IP-MASTER, and EU-FP7-FET-IP-SecureChange for partly supporting this research.

Finally, to my family and friends for their love and support.

Contents

1	Introduction	1
1.1	Objectives	2
1.1.1	Security Policies	2
1.1.2	Efficiency	2
1.2	The Contributions of the Thesis	3
1.3	Structure of the Thesis	4
1.4	List of Publications	5
2	Security by Contract	7
2.1	Security by Contract	7
2.2	From Security by Contract to Automata Modulo Theory	10
3	Related work	13
3.1	Language-based security	13
3.2	Mobile code security	14
3.3	Automata for security policies	16
3.4	Satisfiability Modulo Theories	16
4	Automata Modulo Theory	17
4.1	Introduction	17
4.2	Theory in Automaton Modulo Theory	18
4.3	Automaton Modulo Theory Preliminary	20
4.4	Operations in Automaton Modulo Theory	23
4.5	On-the-fly Language Inclusion Matching	31
5	On-the-fly Matching Prototype Implementation and Experiments	37
5.1	Introduction	37
5.2	The Architecture	38
5.3	Design Decisions	40

5.4	List of Abbreviations	42
5.5	Experiments on Desktop and on Device	42
6	Simulation	47
6.1	Introduction	47
6.2	Simulation in Automaton Modulo Theory	48
6.3	Simulation Matching	54
7	Simulation Matching Prototype Implementation and Experiments	61
7.1	Introduction	61
7.2	The Architecture	62
7.3	Design Decisions	63
7.4	Experiments on Desktop	64
8	IRM Optimization	67
8.1	Introduction	67
8.2	Security Models for Optimized IRM	68
8.2.1	Rewriter on Trusted part	69
8.2.2	Rewriter on Untrusted part	71
8.2.3	Optimizer and Rewriter on Untrusted part	72
8.3	A Search Procedure for IRM Optimization	73
9	Conclusions and FutureWork	81
9.1	Conclusions	81
9.2	Future Work	83
	Bibliography	85
A	On-the-fly Matching Prototype Class Diagram	93
B	Simulation Matching Prototype Class Diagram	95
C	On-the-fly Matching Prototype Experiments	97
D	Simulation Matching Prototype Experiments	99

List of Tables

- 4.1 Theories of Interest 19
- 5.1 Problems Suit 42
- 5.2 Running Problem Suit 10 Times 43
- 7.1 Running Problem Suit 10 Times 64
- C.1 Problems Suit 98
- C.2 Average Running Problem Suit 10 Times (s) 98
- D.1 Average Running Problem Suit 10 Times (s) 100

List of Figures

1.1	End Users' Distilled Security Requirements	2
2.1	Workflow in Security-by-Contract	8
2.2	Mobile Code Components with Security-by-Contract	9
2.3	Infinite Transitions Security Policies	11
4.1	<i>AMT</i> Examples	21
4.2	Boolean Abstraction	27
4.3	Automata Intersection	28
5.1	On-the-fly Implementation Architecture	39
5.2	Cumulative response time of matching algorithm on Desktop PC	44
5.3	Cumulative response time of matching on Device vs on Desktop PC	45
6.1	Symbolic vs Concrete Automaton	52
6.2	Normalization of an automaton	53
7.1	Simulation Implementation Architecture	62
7.2	Cumulative response time of matching algorithm on Desktop PC	65
8.1	Rewriter on Trusted part	70
8.2	Rewriter on Untrusted part	72
8.3	Optimizer and Rewriter on Untrusted part	73
8.4	Optimization alternatives	74
8.5	Inline Type Examples	75
8.6	Optimization Example	75
A.1	On-the-fly Class Diagram	94
B.1	Simulation Class Diagram	96

Chapter 1

Introduction

Currently security model has been based on trust. Either a program is *trusted* and it can do almost everything or *untrusted* and thus can do almost nothing.

Trusted program can be achieved by signing mechanism from trusted third parties. This approach leads to a vague “trust” because a signature on a piece of code only means that the application comes from the software factory of the signatory, but there is no clear definition of what guarantees it offers. It doesn't bind the software behavior.

Untrusted program can be dealt with the mechanism of permissions as in .NET [55] or Java [39]. Permissions are assigned to enable execution of potentially dangerous or costly functionality, such as starting various types of connections. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. Conditional permissions that allow and forbid use of the functionality depending on such factors as bandwidth or the previous actions of the application itself (e.g. access to sensitive files) are also out of reach.

The notion of Security-by-Contract (S×C) was proposed in [27, 13]. In S×C framework, a mobile code is augmented with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading the code. Thus, a digital signature does not just certify the origin of the code but also binds together the code with a contract with the main goal to provide a semantics for digital signatures on mobile code. This framework is a step in the transition from trusted code to trustworthy code.

This thesis provides a formal model and algorithms for matching contract with policy for realistic security scenarios.

USE of Costly functionalities Any invocation of paid services, such as sending text messages, using GPRS or wireless connections, must be controllable by the user.

NETwork connectivity Any external connections made by the application can be controlled.

PRIVate information management It is necessary to control what data is accessed by the application such as local files, PIM items or contacts from Contact List.

INTeraction with other applets This requirement makes necessary to control means of interprocess communication, in particular sockets and memory-mapped files.

Power consumption This requirement is two-fold: it makes necessary to control the invocation of power-consuming functionality, such as WiFi connections, and to control the battery level in course of running the application. This can be mapped into the NET and USE categories.

EXTended functionality If the device is equipped with some advanced functionality, such as camera or GPS receiver, its use is likely to be controlled by policies.

Figure 1.1: End Users' Distilled Security Requirements

1.1 Objectives

1.1.1 Security Policies

Contracts and policies may vary significantly but a number of analysis of security requirements for mobile and ubiquitous applications [47, 77, 88] have shown that they can be distilled in some categories (Figure 1.1). Figure 1.1 is taken from [62] by courtesy of K. Naliuka.

From Figure 1.1 the main requirements that our formalization needs to satisfy are:

- The security policies require both safety and liveness properties.
- The mechanism for defining a general security policies (that is not platform-specific).
- The mechanism for representing an infinite structure as a finite structure for dealing with a security policy such as only allows connections starting with “https://” that already generates an infinite automaton.

1.1.2 Efficiency

Our goal is to provide this midlet-contract vs platform-policy matching on-the-fly (during the actual download of the midlet), hence issues such as small memory footprint and effective computations play a key role. The tractability limit is the complexity of the satisfiability procedure for the background theories used to describe expressions. In the case of language inclusion approach, the complexity is $LIN - TIME^c$ and $NLOG -$

$SPACE^c$ -complete (Proposition 4.5.1) with an oracle to a decision procedure solver. And in the case of simulation approach, the complexity is $POL-TIME^c$ and $LIN-SPACE^c$ (Proposition 6.3.2). Finally, preliminary work in optimization of a policy with respect to a contract has complexity still in the complexity is $POL-TIME^c$ and $LIN-SPACE^c$ (Proposition 8.3.1).

Out of a number of requirements studies, most of the policies of interests can be captured by theories which only requires polynomial time decision procedures (we will discuss details in theory in Section 4.2).

1.2 The Contributions of the Thesis

The formal model used for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* (\mathcal{AMT}). \mathcal{AMT} generalizes the finite state automata of model-carrying code [74] and extends Büchi Automata (BA). It is suitable for formalizing systems with finitely many states but infinitely many transitions, by leveraging the power of satisfiability-modulo-theory (SMT for short) decision procedures. \mathcal{AMT} enables us to define very expressive and customizable policies as a model for *security-by-contract*, by capturing the infinite transition into finite transitions labeled as expressions in suitable theories.

The second contribution is a decision procedure (and its complexity characterization) for matching the mobile's policy and the midlet's security claims that realize the meta-level algorithm of security-by-contract [13]. We map the problem into classical automata theoretic construction such as product and emptiness test.

We have further customized the decision algorithm the security policy has a particular form. For instance, if one uses security automata à la Schneider those can be mapped to a particular form of \mathcal{AMT} (with all accepting states and an error absorbing state) for which particular optimizations are possible. In the original paper by Schneider security automata specify transitions as a function of the input symbols which can be the entire system state. Our \mathcal{AMT} differs from security automata in this respects: transitions are environmental parameters rather than system states. Writing policies in this way is closer to one's intuition.

This matching on-the-fly however requires to complement the policy of the mobile platform and if we assume a general non-deterministic automaton this complementation might lead to an exponential blow-up. A second problem is that in this way we need two representations of the policy: a direct representation of the policy as an automata that we can use for run-time monitor [81] and the complemented representation that we use for matching.

Thus, we further propose to use the notion of simulation for matching the security policy of the platform against the security claims of the midlet. Simulation is stronger than language inclusion (i.e. less midlets will obtain a green light) but they coincide for deterministic policies.

\mathcal{AMT} is a general model, thus it can be used not only for matching security policies but also in other enforcement mechanism for example Inlined Reference Monitor (IRM). IRM is a flexible mechanism to enforce the security of untrusted applications. However, one of the shortcomings of IRM is that it might introduce a significant overhead in otherwise perfectly secure application. Therefore, we propose six different framework models for IRM optimization with respect to components that are needed to be trusted or untrusted. We also describe an approach for IRM optimization using automata modulo theory. The key idea is that given a *policy* that represents the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the (trusted) claims on the security behavior of a application. The optimized policy is the one to be injected into the untrusted code.

1.3 Structure of the Thesis

This thesis book consists of the following chapters:

Chapter 2 briefly introduces our context namely a framework called *Security-by-Contract*.

Security-by-contract [27, 13] proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy on-the-fly, which provides semantics for digital signatures on mobile code.

Chapter 3 surveys related works in theoretical and practical realms including the state-of-the-art of the research pertaining to this thesis.

Chapter 4 introduces our main thesis \mathcal{AMT} and the corresponding automata operations in it. Furthermore, specific issues to be considered in \mathcal{AMT} are also discussed in this chapter. This work had been presented in [58, 57, 13].

Chapter 5 describes an approach for lifting finite state tools to \mathcal{AMT} implementation prototype. This work had been presented in [14, 15, 16].

Chapter 6 describes fair simulation for \mathcal{AMT} with specific issues to be considered in relation of concrete and symbolic \mathcal{AMT} simulation. This work had been presented in [59].

Chapter 7 describes an approach for for lifting finite state tools to \mathcal{AMT} simulation implementation prototype.

Chapter 8 describes a possible application of \mathcal{AMT} in Inlined Reference Monitor (IRM) optimization. This work had been presented in [60].

Chapter 9 presents future works and a concluding discussion.

1.4 List of Publications

The result of this thesis has been published in the following journals:

1. Security-by-contract on the .NET platform [26].
2. Matching in Security-by-Contract for Mobile Code [13].

The result of this thesis has been published in the following conferences or workshops:

1. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code [27].
2. Matching midlet's security claims with a platform security policy using automata modulo theory [58].
3. Matching Policies with Security Claims of Mobile Applications [16].
4. Simulating Midlet's Security Claims with Automata Modulo Theory [59].
5. Testing Decision Procedures for Security-by-Contract [14].
6. Security-By-Contract for the Future Internet [32].
7. Optimizing IRM with Automata Modulo Theory [60].

The result of this thesis has been published in the following refereed conferences or workshops without proceedings:

1. A Security-by-Contracts Architecture for Pervasive Services [57].
2. Security-By-Contract for the Future Internet ? [33].
3. Testing Decision Procedures for Security-by-Contract: Extended Abstract [15].

Other publications:

- Fast Signature Matching Using Extended Finite Automaton (XFA) [75].

Chapter 2

Security by Contract

This chapter briefly introduces a framework called Security-by-Contract as the first motivation for proposing Automata Modulo Theory and continues with positioning Automata Modulo Theory into this framework.

2.1 Security by Contract

Security-by-contract (S×C) [27, 13] has proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy on-the-fly, which provides semantics for digital signatures on mobile code. In an S×C framework [27, 13] a mobile code is augmented with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading the code.

At *development time* the mobile code developers are responsible for providing a description of the security behavior that their code finally provides. Such a code might also undergo a formal certification process by the developer's own company, the smart card provider, a mobile phone operator, or any other third party for which the application has been developed. By using suitable techniques such as static analysis, monitor in-lining, or general theorem proving, the code is certified to comply with the developer's contract. Subsequently, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a proof) and shipped for deployment as shown on Figure 2.2.

At *deployment time*, the target platform follows a workflow as depicted in Figure 2.1 [13]. This workflow is a modification of S×C workflow [13]) by adding optimization step. First, the correctness of the evidence of a code is checked. Such evidence can be a trusted signature [87] or a proof that the code satisfies the contract (one can use Proof-Carrying-Code (PCC) techniques to check it [63]).

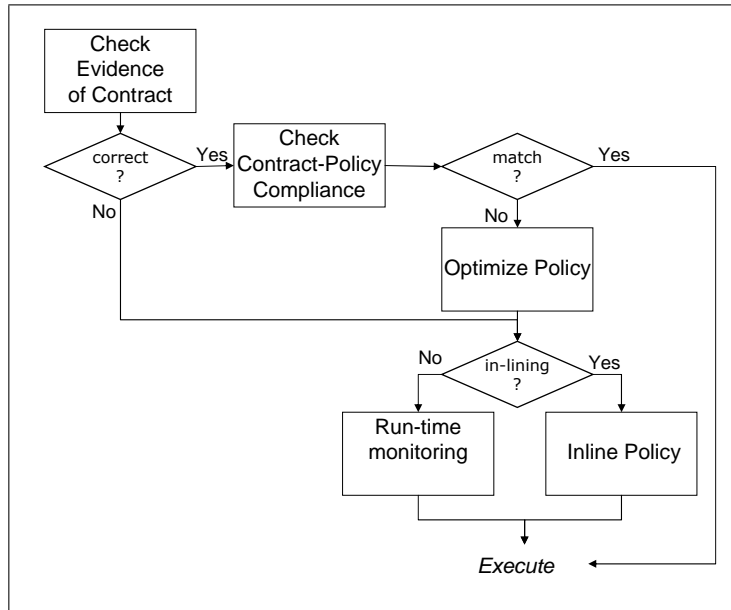


Figure 2.1: Workflow in Security-by-Contract

When there is evidence that a contract is trustworthy, a platform checks, that the claimed contract is compliant with the policy to enforce. If it is, then the application can be run without further ado. It is a significant saving from in-lining a security monitor. In case that at *run-time* we decide to still monitor the application, then we add a number of checks into the application so that any undesired behavior can be immediately stopped or corrected.

Matching succeeds, if and only if, by executing an application on the platform, every behavior of the application that satisfies its contract also satisfies the platform’s policy. If matching fails, but we still want to run the application, then we use either a security monitor in-lining, or run-time enforcement of the policy (by running the application in parallel with a reference monitor that intercepts all security relevant actions). However with a constrained device, where CPU cycles means also battery consumption, we need to minimize the run-time overheads as much as possible.

A *contract* is a formal specification of the behavior of an application for what concerns relevant security actions for example Virtual Machine API Calls, Web Messages. By signing the code the developer certifies that the code complies with the stated claims on its security-relevant behavior. A *policy* is a formal specification of the acceptable behavior of applications to be executed on a platform for what concerns relevant security actions. Thus, a digital signature does not just certify the origin of the code but also bind together the code with a contract with the main goal to provide a semantics for digital signatures on mobile code. Therefore, this framework is a step in the transition from trusted code to

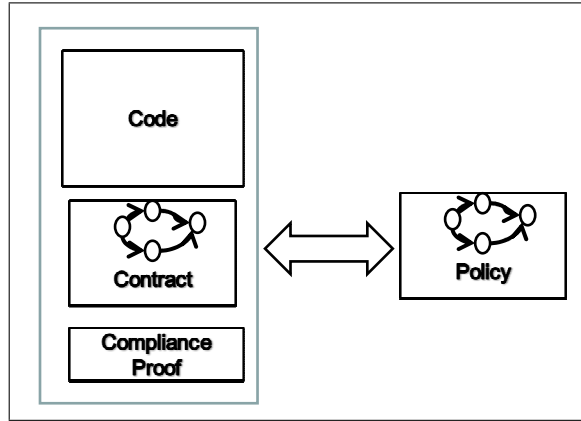


Figure 2.2: Mobile Code Components with Security-by-Contract

trustworthy code. Technically, a contract is a security automaton in the sense of Schneider [43], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton.

A *policy* covers a number of issues such as file access, network connectivity, access to critical resources, or secure storage. A single contract can be seen as a list of disjoint claims (for instance rules for connections). An example of a rule for sessions regarding A Personal Information Management (PIM) and connections is shown in Example 2.1.1, which can be one of the rules of a contract. Another example is a rule for method invocation of a Java object as shown in Example 2.1.2. This example can be one of the rules of a policy. Both examples describe safety properties, which are common properties that we want to verify.

Example 2.1.1 *PIM system on a phone has the ability to manage appointment books, contact directories, etc., in electronic form. A privacy conscious user may restrict network connectivity by stating a policy rule: “After PIM is opened no connections are allowed”. This contract permits executing the `javax.microedition.io.Connector.open()` method only if the `javax.microedition.pim.PIM.openPIMList()` method was never called before.*

Example 2.1.2 *The policy of an operator may only require that “After PIM was accessed only secure connections can be opened”. This policy permits executing the `javax.microedition.io.Connector.open(string url)` method only if the started connection is a secure one i.e. `url` starts with “https://”.*

We can have a slightly more sophisticated approach using Büchi automata [76] if we also want to cover liveness properties as shown in the following Example 2.1.3.

Example 2.1.3 *If the application should use all the permissions it requests then for each permission p at least one reachable invocation of a method permitted by p must exist in the code. For example if p is `io.Connector.http` then a call to method `Connector.open()` must exist in the code and the url argument must start with “http”. If p is `io.Connector.https` then a call to method `Connector.open()` must exist in the code and the url argument must start with “https” and so on for other constraints e.g. permission for sending SMS.*

2.2 From Security by Contract to Automata Modulo Theory

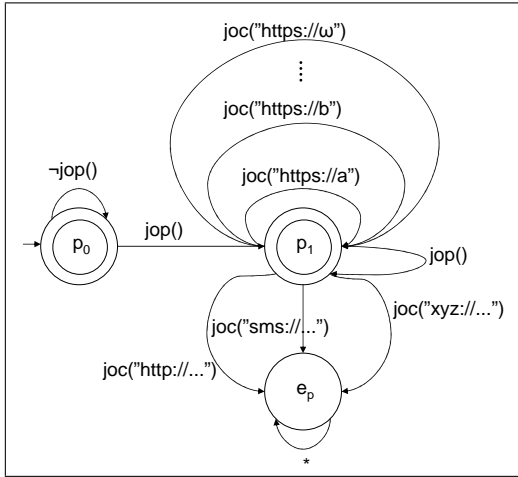
The security behaviors, provided by the contract and desired by the policy, can be represented as automata (shown on Figure 2.2), where transitions corresponds to invocation of APIs as suggested by Erlingsson [28, p.59] and Sekar et al. [74]. Thus, in this thesis we have mapped the operation of matching the midlet’s claim with platform policy into problems in automata theory.

The first mechanism we consider to represent matching is *language inclusion*, such that given two automata A^C and A^P representing respectively the formal specification of a contract and of a policy, we have a match when the execution traces of the midlet described by A^C are a subset of the acceptable traces for A^P . To check this property we can complement the automaton of the policy, thus obtaining the set of traces disallowed by the policy and check its intersection with the traces of the contract. If the intersection is not empty, any behavior in it corresponds to a security violation.

The second mechanism we consider is the notion of *simulation*, such that we have a match when every APIs invoked by A^C can also be invoked by A^P . In other words, every behavior of A^C is also a behavior of A^P . Simulation is a stronger notion than language inclusion as it requires that the policy allows the actions of the midlet’s contract in a “step-by-step” fashion, whereas language inclusion looks at an execution trace as a whole.

In the case that matching fails, but we still want to run the application, then we can optimize the supposed to be enforced security policy. The key idea is that given a *policy* that represents the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the (trusted) claims on the security behavior of an application. The optimized policy is the one to be injected into the untrusted code.

While this idea of representing the security-digest as an automaton is almost a decade old [74, 28], the practical realization has been hindered by a significant technical obstacle: we cannot use the naive encoding into automata for practical policies. Even the basic policies in Example 2.1.1 and Example 2.1.2 lead to automata with infinitely many transitions. For example an infinite automaton of Example 2.1.2 is shown on Figure 2.3a.



(a) An Infinite Automaton of Example 2.1.2

$joc(url) \doteq \text{javax.microedition.io.}$
 $\text{Connector.open}(url)$
 $jop() \doteq \text{javax.microedition.pim.}$
 $\text{PIM.openPIMList}(\dots)$

(b) Abbreviations for Java APIs

Figure 2.3: Infinite Transitions Security Policies

In order to overcome this technical obstacle we have proposed a new formalization for security policies using automata, called *Automata Modulo Theory*.

Chapter 3

Related work

Efforts have been made for enforcing security policies, for example by program rewriting, static analysis, or run-time monitoring or a hybrid approach. These works have also been applied by a variety of policy specification languages or frameworks. However, there is a tendency either to be system dependent (platform specific) or to be a general abstraction. Thus, in this chapter we survey closely related works and discuss similarities and differences between our work and the related efforts. Further in each subsequent chapter, we also survey related works specific to the given chapter.

3.1 Language-based security

The security problems arising when application developers and platform owners are not on the same (security) side are well known from the experience of Java web applications for the desktop. The confinement of Java applets [39] is a classical solution. Indeed, to deal with the untrusted code either .NET [55] or Java [39] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous or costly functionality, such as starting various types of connections. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used.

Conditional permissions that allow and forbid use of the functionality depending on such factors as bandwidth or the previous actions of the application itself (e.g. access to sensitive files) are also out of reach. The consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they come from a trusted source and then they can do almost everything. This approach is known as *code signing* that ensures the origin of the code by trust relationship.

In order to overcome the well-known limitation of the trusted signature or sandbox a

number of techniques have been proposed and implemented. For example *static analysis*, *execution monitoring*, or *program rewriting*. An effort to classify the security policies enforceable by each approach with respect to computation is detailed in [43]. In static analysis, untrusted program is checked for compliance to the security policy prior to running it, for example static type-checkers for type-safe languages. In execution monitors, the enforcement mechanisms operate alongside an untrusted program and prevent from violation of the given security policy. In program rewriting, untrusted programs are modified to satisfy the given security policy prior to their execution. The most promising one is the notion of Inlined Reference Monitor (IRM), where program-rewriters can also be seen as a generalization of execution monitoring. IRM works by embedding the untrusted program with the security policies.

IRM has been implemented in several systems, for example the PoET/PSLang toolkit [29], enforcing security policies whose transitions pattern-match event symbols using regular expressions, or Polymer [12] based on edit automata. The shortcoming of traditional IRM is the huge overhead resulting from inlining.

Even if current version of IRM can work on rich system such as today's smart phones, the overhead is still too much for the next frontier of web applications: Javacards. Indeed, the smartcard technology [61] evolved with larger memories, USB and TCP/IP support and the development of the Next-Generation (for short NG) Java Card platform [3, 4] with Servlet engine. This latter technology is a full fledged Java platform for embedded Web applications and opens new Web 2.0 opportunities such as NG Java Card Web 2.0 Applications. It can also serve as an alternative to personalized applications on remote servers so that personal data no longer needs to be transmitted to remote third-parties.

Phung et al. [69] proposed lightweight version of IRM for JavaScript that does not modify browser or original code i.e. it adds new code only in the header of the page. An alternative approach to IRM is by using reflection [85], where policies are implemented as meta-objects bounded to application objects at load time, such that the code becomes self-protecting.

3.2 Mobile code security

Security model of mobile device operating systems is usually *system-centric* where applications statically checked for compliance of security policies at installation time. This approach has the limitation of the customize policies for example user-defined security policies. Some extensions to enable *application-centric* have been proposed. Sekar et al. [74] have proposed a seminal idea of Model Carrying Code (MCC). In MCC, a *model* of an application and the policy of the platform are also represented using Extended Finite

Automata (EFA)[80]. In loading time, when the language of the model automaton is included in the language of the security automaton then the application can be loaded.

EFA is based on finite state automata (FSA), thus it recognizes finite input. However, it differs from FSA on the alphabet, that instead of an atomic symbol EFA has introduced regular expressions over events(REEs). EFA has been implemented in a system for intrusion detection/prevention [80].

In spite of this expressiveness, MCC has limitations of concerning only safety properties and static expressions (it is not possible to add certain theories that may be needed to describe realistic policies). Furthermore, MCC has not fully developed issue of contract matching.

Later on, the Security-by-Contract (S×C) framework [27] was built upon the MCC seminal idea. In an S×C framework [27, 13] a mobile code is augmented with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading the code. If matching fails, then the application can be in-lined for the policies that can be statically checked. Desmet et al. have shown that an effective and comprehensive version of IRM can be deployed on mobile platforms [26] in an S×C framework. In other case we can still monitor the application at *run-time* as the last option.

Currently, Ongtang et al. [67] have proposed the Secure Application INTeraction (Saint) framework as extension to the Android ¹ security architecture. Saint enforces security policy at installation time by checking that an application requesting the permission P is permitted to be installed only if the policy for acquiring P is satisfied. Furthermore, Saint offers run-time enforcement among applications, where security policies depend on both the caller and callee applications.

In Saint, security policies are defined as conditions that consist of two sets namely the set of invariant conditions and the set of transient conditions. The system state of the phone at any given time is a truth assignment for Boolean variables for each condition. Thus the satisfiability of security policies is equals to satisfiability of conjunction of the caller's and callee's conditions. This simple logic approach limits Saint such that it concerns only static expressions, that is it is not possible to add certain theories that may be needed to describe realistic policies.

AMT solves the afore mentioned problems by allowing combination of theories of expressive policies and providing contract matching algorithms.

¹Android is an mobile phone platform developed by the Google-led Open Handset Alliance (OHA). <http://www.openhandsetalliance.com/>

3.3 Automata for security policies

A common formalization for representing security policy is using automata. *Security automata* is a seminal work in this area introduced by Schneider in [71]. In security automata, each transition is labeled with a computable predicate instead of an atomic symbol for infinite numbers of transitions in security policies. The class of security policies recognized by security automata has the ability to prevent system from violation. To extend this enforcement mechanism such that it also considers inserting or removing unwanted behavior, Bauer et al. [10] introduced *edit automata*.

Security automata is implemented for security monitors in several systems, for example the PoET/PSLang toolkit [29], that can enforce security policies whose transitions pattern-match event symbols using regular expressions. Edit automata is implemented in the Polymer system [11] to dynamically compose security automata. The Mobile system [44] implements a linear decision algorithm that verifies that annotated .NET bytecode binaries satisfy a class of policies that includes security automata and edit automata.

3.4 Satisfiability Modulo Theories

Automata Modulo Theory abstract infinite transitions into finite expression using formulas in Satisfiability Modulo Theories. The Satisfiability Modulo Theories (SMT) problem focuses on the satisfiability of quantifier-free first-order formulas modulo background theories (see survey on [73]). Some theories of interest are the theory of difference logic \mathcal{DL} , the theory \mathcal{EUF} of *Equality and Uninterpreted Functions*, the quantifier-free fragment of *Linear Arithmetic* over the rationals $\mathcal{LA}(\mathbb{Q})$ and that over the integers $\mathcal{LA}(\mathbb{Z})$. SMT is an active research area with many tools developed such as Z3[25], MathSAT[18], CVC[9], and UCLID[19]. \mathcal{AMT} uses the same notion of “theory” as in the SMT to accommodate expressive policies where each transition is labeled with a computable predicate in some theories for representing infinite numbers of transitions.

Chapter 4

Automata Modulo Theory

In this chapter we try to provide an answer to the following question: given expressive security policies, how can we model possibly infinite computations with finite ones? The key idea is to abstract infinite transitions into finite expression using formulas in Satisfiability Modulo Theories and using base structure as in Büchi Automata. This formalization is called Automata Modulo Theory.

4.1 Introduction

AMT enables matching a mobile's policy and a midlet's contract by mapping the problem into a variant of on-the-fly product and emptiness test from automata theory, without symbolic manipulation procedures of zones and regions nor finite representation of equivalence classes. The tractability limit is essentially the complexity of the satisfiability procedure for the theories, called as subroutines, where most practical policies require only polynomial time decision procedures [73](see summary in Table 4.1).

This chapter describes the theory of *Automata Modulo Theory* (*AMT*). We begin in Section 4.2 by introducing the concept of *theory* in *AMT*. Then, Section 4.3 defines the formalization of the automata including the concept of *tuple*, *run* and *word*. Section 4.3 continues with *operations* in *AMT*, namely intersection and complementation. Finally, Section 4.5 describes a decision procedure (and its complexity characterization) for matching the mobile's policy and the midlet's security claims that realize the meta-level algorithm of security-by-contract [13]. This algorithm for matching the contract with the security policy had been implemented and in the next chapter (Chapter 5), we will continue describing this prototype, its integration with decision solver based on MathSAT and NuSMV, and the results of our experiments on matching.

4.2 Theory in Automaton Modulo Theory

The notion of *theory* in \mathcal{AMT} is derived from the notion of *theory* in the Satisfiability Modulo Theories (SMT) problem. The SMT problem focuses on the satisfiability of quantifier-free first-order formulas modulo background theories [17]. Some theories of interest for example are the theory of equality and uninterpreted functions (\mathcal{EUF}), the quantifier-free fragment of linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$), and over the integers ($\mathcal{LA}(\mathbb{Z})$), and the corresponding subtheories of difference logic both over the rationals ($\mathcal{DL}(\mathbb{Q})$), and over the integers ($\mathcal{DL}(\mathbb{Z})$).

Example 4.2.1 *A security policy may set limits on resources that can be captured with constraints expressed in different theories*

1. *no communication allowed if the battery level falls below 30% ($\mathcal{LA}(\mathbb{Q})$ can be used);*
2. *no jpeg file can be downloaded with size more than 500KB while avi files can arrive up to 1MB ($\mathcal{LA}(\mathbb{Z})$ can be used here)*
3. *\mathcal{EUF} can be used when comparing a policy requiring $\mathit{protocol}(url) = \text{'https'}$ and $\mathit{port}(url) = \text{'8080'}$ with a contract claiming to use only connections where $\mathit{protocol}(url) = \text{'https'}$ or $\mathit{protocol}(url) = \text{'http'}$. We do not need to extract a protocol from the url. It is enough that we deal with protocol and port as uninterpreted functions and apply the theory of equality and uninterpreted functions \mathcal{EUF} .*

The previous examples show simple security policies each uses only one kind of theory. However, we are particularly interested in the combination of two or more theories to accommodate complex security policies.

Example 4.2.2 *A policy may allow only secure connections with limited size of downloads. To express this policy we combine \mathcal{EUF} for handling $\mathit{protocol}(url) = \text{'https'}$ and $\mathcal{LA}(\mathbb{Z})$ for handling downloading a file of at most 500KB.*

We use traditional first-order logic terminology [34] for defining a SMT theory. A signature Σ consists a set of function symbols \mathcal{F} and a set of predicate symbols \mathcal{P} with their arities, and a set of variables V . A 0-ary function symbol c is called a *constant* and 0-ary predicate symbol B is called a *Boolean atom*. A Σ -term is a variable in V or constructed from application of function symbols \mathcal{F} to Σ -terms. If t_1, \dots, t_n are Σ -terms and p is a predicate symbol then $p(t_1, \dots, t_n)$ is a Σ -atom. A Σ -literal is a Σ -atom or negation of Σ -atom. Σ -formula is defined over Σ -literals, the universal and the existential

Table 4.1: Theories of Interest

Theory	(Non)Convex	Decidability	Complexity
\mathcal{EUF}	convex	decidable	polynomial [5]
$\mathcal{LA}(\mathbb{Q})$	convex	decidable	polynomial [41]
$\mathcal{LA}(\mathbb{Z})$	non-convex	decidable	NP-Complete [68]
$\mathcal{DL}(\mathbb{Q})$	convex	decidable	polynomial [21]
$\mathcal{DL}(\mathbb{Z})$	non-convex	decidable	NP-Complete [54]

quantifiers \forall, \exists , and the boolean connectives \neg, \wedge . A Σ -formula is named quantifier-free when it contains no quantifier and sentence when it contains no free variables. A Σ -theory \mathcal{T} is a set of first-order sentences with signature Σ .

A Σ -structure \mathcal{M} is a model of Σ -theory \mathcal{T} if \mathcal{M} satisfies every sentences in \mathcal{T} . A Σ -structure \mathcal{M} consists of a set D of elements as *domain* and an interpretation \mathcal{I} as in first order logic. The interpretation of an n -ary function symbol is a mapping of each n -ary function symbol $f \in \Sigma$ to a total function $f^{\mathcal{M}} : D^n \rightarrow D$. The interpretation of a constant symbol is a mapping of each constant $c \in \Sigma$ to itself. The interpretation of an n -ary predicate symbol is a mapping of each n -ary predicate symbol $p \in \Sigma$ to a relation $p^{\mathcal{M}} \subseteq D^n$ and the interpretation of a Boolean atom is a mapping of each Boolean atom $B \in \Sigma$ to (\top, \perp) .

Let \mathcal{M} denote a Σ -structure, ϕ a formula, and \mathcal{T} a theory, all of signature Σ . We say that ϕ is satisfiable in \mathcal{M} (or ϕ is \mathcal{T} -satisfiable) if there exists some assignment α which assigns the set of variables to values in the domain such that $(\mathcal{M}, \alpha) \models \phi$.

A theory \mathcal{T} is convex [73] if all the conjunctions of literals are convex in theory \mathcal{T} . A conjunction of \mathcal{T} -literals in a theory \mathcal{T} is convex if for each disjunction $(\mathcal{M}, \alpha) \models \bigvee_{i=1}^n e_i$ if and only if $(\mathcal{M}, \alpha) \models e_i$ for some i , where e_i are equalities between variables occurring in (\mathcal{M}, α) .

The general definition above applies the full power of SMT. For practical purposes we make some additional restrictions.

First-order as base logic We use classical first-order logic based SMT. Extension to a higher-order logic is possible as proposed in [53], where they introduced *parametric theories*. In the sequel, we consider only quantifier-free Σ -formulas on theories \mathcal{T} where the \mathcal{T} -satisfiability of conjunctions of literals is decidable by a \mathcal{T} -solver [66].

Combination of theories is consistent Given a consistent theory \mathcal{T}_1 and a consistent theory \mathcal{T}_2 , we assume that the combination theory $\mathcal{T} := \mathcal{T}_1 \cup \mathcal{T}_2$ is also consistent and

there exists a \mathcal{T} -solver for the combined theory. We are interested in $\mathcal{T}_1 \cup \mathcal{T}_2$ -satisfiability of $\Sigma_1 \cup \Sigma_2$ -formulas that can be generalized to combine many possibly signature-disjoint theories $\mathcal{T}_1 \cup \dots \cup \mathcal{T}_n$.

The Nelson-Oppen (NO) combination procedure [64] is a seminal work in this area. NO combines decision procedures for first-order theories restricted to theories that are *stably-infinite* (informally the theory that has infinite models (see [64])) and that *have disjoint signatures* ($\Sigma_1 \cap \Sigma_2 = \emptyset$). Tinelli-Zarba's combination procedure [79] extends NO for combining an arbitrary theory which maybe stably infinite with a stably infinite theory that is also *shiny*. They also proposed a variant of the combination method for combining theories having only finite models with theories that are stably finite. Ghilardi's combination procedure [36] extends NO for combining theories that share signature with restriction that the theories are compatible with respect to a common sub theory in the shared signature.

Conjunctions of formulas Given theories \mathcal{T}_1 and \mathcal{T}_2 that can be combined $\mathcal{T} := \mathcal{T}_1 \cup \mathcal{T}_2$ and conjunctive normal form formula ϕ_1 (resp. ϕ_2) that is satisfiable in \mathcal{T}_1 (resp. \mathcal{T}_2) then $\phi_1 \wedge \phi_2$ is decidable in \mathcal{T} (not necessarily satisfiable). We do not impose restrictions as in Proposition 3.8. in [78], thus we do not have their result, i.e. $\phi_1 \wedge \phi_2$ is satisfiable in \mathcal{T} .

4.3 Automaton Modulo Theory Preliminary

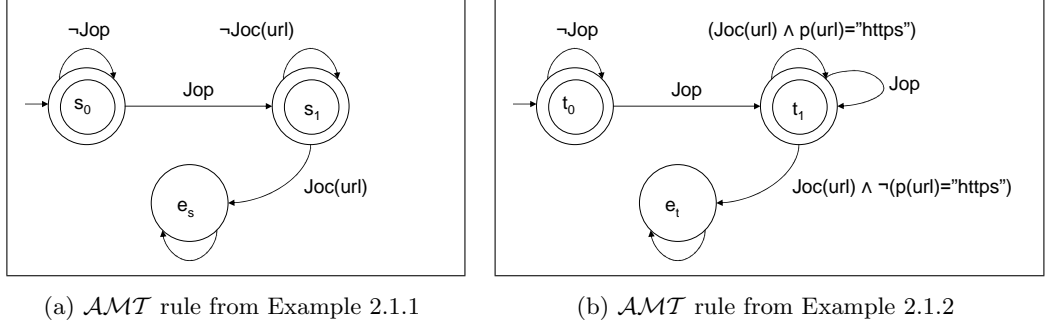
Having defined the *theory* in \mathcal{AMT} , in this section we continue by defining tuple, run, and word in \mathcal{AMT} .

An automaton in \mathcal{AMT} is defined as a tuple of a finite set of Σ -formulas in Σ -theory \mathcal{T} , a finite set of states, an initial state, a labeled transition relation, and a set of accepting states. Formally, it is given in Definition 4.3.1.

Definition 4.3.1 (Automaton Modulo Theory (\mathcal{AMT})) *An \mathcal{AMT} is a tuple $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$, where \mathcal{E} is a finite set of Σ -formulas in Σ -theory \mathcal{T} , S is a finite set of states, $\mathbf{s}_0 \in S$ is the initial state, $\Delta \subseteq S \times \mathcal{E} \times S$ is a labeled transition relation, and $F \subseteq S$ is a set of accepting states.*

Figure 4.1 shows two examples of \mathcal{AMT} using the signature for \mathcal{EUF} with a function symbol $p()$ representing the protocol type used for the opening of a *url*. As described in the cited examples the first automaton forbids the opening of plain http-connections as soon as the PIM is invoked while the second just restricts connections to be only https.

The transitions in these automata describe with an expression a potentially infinite set of transitions: the opening of all possible *urls* starting with https. The automaton



$$\begin{aligned}
Joc(url) &\doteq \text{Joc}(\text{joc}, \text{url}) \\
Jop &\doteq \text{Jop}(\text{jop}, x_1, \dots, x_n) \\
p(url) = type &\doteq \text{url.startsWith}(type) \\
joc &\doteq \text{javax.microedition.io.Connector.open} \\
jop &\doteq \text{javax.microedition.pim.PIM.openPIMList}
\end{aligned}$$

Joc, Jop are predicate symbols representing respectively $\text{joc}(\text{url}), \text{jop}(x_1, \dots, x_n)$ APIs.

(c) Abbreviations for expressions

Figure 4.1: \mathcal{AMT} Examples

modulo theory is therefore an abstraction for a concrete (but infinite) automaton. The *concrete automaton* corresponds to the behavior of the actual system in terms of API calls, value of resources and the likes.

From a formal perspective, the concrete model of an automaton modulo theory intuitively corresponds to the automaton where each symbolic transition labeled with an expression is replaced by the set of transitions corresponding to all satisfiable instantiations of the expression.

In order to characterize how an automaton captures the behavior of programs we need to define the notion of a trace. So, we start with the notion of a symbolic run which corresponds to the traditional notion of run in automata.

Definition 4.3.2 (\mathcal{AMT} symbolic run) Let $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ be an \mathcal{AMT} . A symbolic run of A is a sequence of states alternating with expressions $\sigma = \langle s_0 e_1 s_1 e_2 s_2 \dots \rangle$, such that:

1. $s_0 = \mathbf{s}_0$
2. $(s_i, e_{i+1}, s_{i+1}) \in \Delta$ and e_{i+1} is \mathcal{T} -satisfiable, that is there is some Σ -structure \mathcal{M} a model of Σ -theory \mathcal{T} and there exists some assignment α such that $(\mathcal{M}, \alpha) \models e_{i+1}$.

A finite symbolic run is denoted by $\langle s_0 e_1 s_1 e_2 s_2 \dots s_{n-1} e_n s_n \rangle$. An infinite symbolic run

is denoted by $\langle s_0 e_1 s_1 e_2 s_2 \dots \rangle$. A finite run is accepting if the last state goes through some accepting state, that is $s_n \in F$. An infinite run is accepting if the automaton goes through some accepting states infinitely often.

In order to capture the actual system invocations we introduce another type of run called *concrete run* which is defined over valuations that represent actual system traces. A valuation ν consists of interpretations and assignments.

Definition 4.3.3 (AMT concrete run) Let $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ be an AMT. A concrete run of A is a sequence of states alternating with a valuation $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$, such that:

1. $s_0 = \mathbf{s}_0$
2. there exists expressions $e_{i+1} \in \mathcal{E}$ such that $(s_i, e_{i+1}, s_{i+1}) \in \Delta$ and there is some Σ -structure \mathcal{M} a model of Σ -theory \mathcal{T} such that $(\mathcal{M}, \alpha_{i+1}) \models e_{i+1}$, where ν_{i+1} represents α_{i+1} and $\mathcal{I}(e_{i+1})$.

A finite concrete run is denoted by $\langle s_0 \nu_1 s_1 \nu_2 s_2 \dots s_{n-1} \nu_n s_n \rangle$. An infinite concrete run is denoted by $\langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$. A finite run is accepting if the last state goes through some accepting state, that is $s_n \in F$. An infinite run is accepting if the automaton goes through some accepting states infinitely often. The trace associated with $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$ is the sequence of valuations in the run. Thus a trace is accepting when the corresponding run is accepting.

We use definition of run as in [30] which is slightly different from the one we use in [58], where we use only states.

Example 4.3.1 An example of an accepting symbolic run of AMT rule from Example 2.1.2 shown in Figure 4.1b is

$p_0 \text{ Jop}(jop, file, permission) \ p_1 \text{ Joc}(joc, url) \wedge p(url) = \text{"https"} \ p_1 \text{ Jop}(jop, file, permission) \ p_1 \text{ Joc}(joc, url) \wedge p(url) = \text{"https"} \ \dots$

that corresponds with a non empty set of accepting concrete runs for example

$p_0(jop, PIM.CONTACT_LIST, PIM.READ_WRITE) \ p_1(joc, \text{"https://www.esse3.unitn.it/"})$

$p_1(jop, PIM.CONTACT_LIST, PIM.READ_ONLY) \ p_1(joc, \text{"https://online.unicreditbanca.it/login.htm"}) \ \dots$

Remark 4.3.1 A symbolic run defined in Definition 4.3.2 is interpreted by a non empty set of concrete runs in Definition 4.3.3. This is a nature of our application domain where security policies define AMT in symbolic level and the system to be enforced has concrete runs. In other domains where we need the converse, namely to define symbolic runs from concrete runs, then a symbolic run defined in Definition 4.3.2 can be considered as an abstraction of concrete runs by Definition 4.3.3.

The *alphabet* of \mathcal{AMT} is defined as a set of valuations \mathcal{V} that satisfy \mathcal{E} . A finite sequence of alphabet of A is called a *finite word* or *word* or *trace* denoted by $w = \langle \nu_1 \nu_2 \dots \nu_n \rangle$ and the length of w is denoted by $|w|$. An infinite sequence of alphabet of A is called an *infinite word* or *infinite trace* is denoted by $w = \langle \nu_1 \nu_2 \dots \rangle$. The set of infinite words recognized by an automaton A , denoted by $L_\omega(A)$, is the set of all accepting infinite traces in A . $L_\omega(A)$ is called the language accepted by A .

As we have noted already, the intuitive idea behind concrete runs is that they are sequences of models of the expressions of the abstract specification of the automaton modulo theory. In the practical setting, for example security policies over midlets, we want to capture sequences of API calls then this general theory can be actually narrowed.

Example 4.3.2 *A possible alternative is to use a predicate name corresponding to each API call (such as $joc(url, port)$, $jop()$, etc.) and then introduce a theory that specify that predicates are mutually exclusive.*

This formalization would correspond essentially to the guard-and-condition representation of Schneider’s security automata.

Example 4.3.3 *Another alternative is to use predicate $API(API\ symbol, parameters)$ with the first argument the API name itself as a constant symbol to identify different methods. For example $joc(url, port)$ is denoted as $Joc(joc, url, port)$ and $jop(x_1, \dots, x_n)$ is denoted as $Jop(jop, x_1, \dots, x_n)$ imposing each constant as unique, i.e. $joc \neq jop$.*

Both formalizations capture the same concrete behavior in terms of API calls. Our current implementation uses the second option as the unique name assumption was built-in the SMT solver implementation and therefore it could be used more efficiently.

The transition relation of A may have many possible transitions for each state and expression, i.e. A is potentially non-deterministic.

Definition 4.3.4 (Deterministic \mathcal{AMT}) $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ is a deterministic automaton modulo theory \mathcal{T} , if and only if, for every $s \in S$ and every $s_1, s_2 \in S$ and every $e_1, e_2 \in \mathcal{E}$, if $(s, e_1, s_1) \in \Delta$ and $(s, e_2, s_2) \in \Delta$, where $s_1 \neq s_2$ then the expression $(e_1 \wedge e_2)$ is unsatisfiable in the Σ -theory \mathcal{T} .

4.4 Operations in Automaton Modulo Theory

In order to define the test for language inclusion we introduce the operation of complement and intersection of \mathcal{AMT} operations at the concrete level, for example API calls, and then we give the notion of symbolic operations as in [45].

In this thesis we consider only the *complementation of deterministic \mathcal{AMT}* , for all security policies in our application domain are naturally deterministic because a platform owner should have a clear idea on what to allow or disallow.

Complementation of \mathcal{AMT} . \mathcal{AMT} automaton can be considered as a Büchi automaton where infinite transitions are represented as finite transitions. Therefore, for each deterministic \mathcal{AMT} automaton A there exists a (possibly nondeterministic) \mathcal{AMT} that accepts all the words which are not accepted by automaton A . The A^c can be constructed in a simple approach as in [82] as follows:

Definition 4.4.1 (\mathcal{AMT} Complementation) *Given a deterministic \mathcal{AMT} $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ the complement \mathcal{AMT} automaton $A^c = \langle S^c, \Sigma, \mathcal{T}, \mathcal{E}, \mathbf{s}_0^c, \Delta^c, F^c \rangle$ is:*

$$1. S^c = S \times \{0\} \cup (S - F) \times \{1\}, \quad \mathbf{s}_0^c = (\mathbf{s}_0, 0), \quad F^c = (S - F) \times \{1\},$$

2. and for every $s \in S$ and $e \in \mathcal{E}$

$$\begin{aligned} ((s, 0), e, s') \in \Delta^c, s' &= \begin{cases} \{(t, 0)\} & (s, e, t) \in \Delta \text{ and } t \in F \\ \{(t, 0), (t, 1)\} & (s, e, t) \in \Delta \text{ and } t \notin F \end{cases} \\ ((s, 1), e, s') \in \Delta^c, s' &= \{(t, 1)\} \text{ if } (s, e, t) \in \Delta \text{ and } t \notin F \end{aligned}$$

In order to apply complementation in Definition 4.4.1, the deterministic automata has to be completed, meaning the sum of the transitions labels covers all the set of formulas in \mathcal{E} . Return to our Example 2.1.1 shown in Figure 4.1a, the automaton is already a complete \mathcal{AMT} .

Proposition 4.4.1 *Let A be an \mathcal{AMT} over a set of valuations \mathcal{V} . Then a (possibly nondeterministic) \mathcal{AMT} A^c constructed by Definition 4.4.1 accepts all the concrete runs which are not accepted by A , that is A^c is a complement automaton such that $L_\omega(A^c) = \mathcal{V}^\omega - L_\omega(A)$.*

Proof.

Correctness.

“ \supseteq ” we take an arbitrary concrete run not accepted by A that corresponds to a word $w = \langle \nu_1 \nu_2 \nu_3 \dots \rangle$, meaning $w \in \mathcal{V}^\omega - L_\omega(A)$, so there is a unique concrete run $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$ of A . Hence, there is some k such that $\forall i > k, s_i \notin F$, meaning that $\sigma_C^c = \langle (s_0, 0) \nu_1 \dots (s_k, 0) \nu_{k+1} (s_{k+1}, 1) \dots \rangle$ is an accepting concrete run of A^c .

“ \subseteq ” we take an arbitrary concrete run accepted by A^c that corresponds to a word $w = \langle \nu_1 \nu_2 \nu_3 \dots \rangle$, meaning that $w \in L_\omega(A^c)$, so there is a unique concrete run $\sigma_C^c = \langle (s_0, 0) \nu_1 \dots (s_k, 0) \nu_{k+1} (s_{k+1}, 1) \dots \rangle$ of A^c , corresponds to a concrete run $\sigma'_C = \langle s_0 \nu_1 \dots s_k \nu_{k+1} s_{k+1} \dots \rangle$ of A on w but this concrete run is rejecting.

Termination. This construction terminates because our states in S and formulas in \mathcal{E} are finite.

Complexity. The time and space complexity of the construction is linear. □

The construction in Definition 4.4.1 can be optimized if our security policy is a pure security automaton à la Schneider. The policy automaton for safety properties has all (but one) accepting states. The complementation will result in only one accepting state which is $(err, 1)$. However, the state can be collapsed with a non accepting state $(err, 0)$. Hence, no need to mark states with 0 and 1; and the only accepting state is (err) . Furthermore, the complementation transitions remain as the original transitions.

Intersection of AMT. \mathcal{AMT} automaton can be considered as a Büchi automaton where infinite transitions are represented as finite transitions. Therefore, for \mathcal{AMT} automata A^a, A^b , there is an \mathcal{AMT} A^{ab} that accepts all the words which are accepted by both A^a, A^b synchronously. The A^{ab} can be constructed in a simple approach as in [82] as follows:

Definition 4.4.2 (AMT Intersection) Let $\langle S^a, \Sigma^a, \mathcal{T}^a, \mathcal{E}^a, \Delta^a, \mathbf{s}_0^a, F^a \rangle$ and $\langle S^b, \Sigma^b, \mathcal{T}^b, \mathcal{E}^b, \Delta^b, \mathbf{s}_0^b, F^b \rangle$ be (non) deterministic \mathcal{AMT} , the \mathcal{AMT} intersection automaton $A^{ab} = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ is defined as follows:

1. $\mathcal{E} = \mathcal{E}^a \cup \mathcal{E}^b, \quad \mathcal{T} = \mathcal{T}^a \cup \mathcal{T}^b, \quad \Sigma = \Sigma^a \cup \Sigma^b,$
2. $S = S^a \times S^b \times \{1, 2\}, \quad \mathbf{s}_0 = \langle \mathbf{s}_0^a, \mathbf{s}_0^b, 1 \rangle, \quad F = F^a \times S^b \times \{1\},$
- 3.

$$\Delta = \left\{ \left\langle (s^a, s^b, x), e^a \wedge e^b, (t^a, t^b, y) \right\rangle \left| \begin{array}{l} (s^a, e^a, t^a) \in \Delta^a \text{ and} \\ (s^b, e^b, t^b) \in \Delta^b \text{ and} \\ \text{DecisionProcedure}(e^a \wedge e^b) = SAT \end{array} \right. \right\}$$

$$y = \begin{cases} 2 & \text{if } x = 1 \text{ and } s^a \in F^a \text{ or} \\ & \text{if } x = 2 \text{ and } s^b \notin F^b \\ 1 & \text{if } x = 1 \text{ and } s^a \notin F^a \text{ or} \\ & \text{if } x = 2 \text{ and } s^b \in F^b \end{cases}$$

Proposition 4.4.2 *Let A^a, A^b be \mathcal{AMT} over a set of valuations \mathcal{V} . Then an \mathcal{AMT} A^{ab} constructed by Definition 4.4.2 accepts all the concrete runs which are accepted by A^a, A^b , that is A^{ab} is an intersection automaton such that $L_\omega(A^{ab}) = L_\omega(A^a) \cap L_\omega(A^b)$.*

Proof.

Correctness.

“ \supseteq ” we take an arbitrary concrete run accepted by A^{ab} that corresponds to a word $w = \langle \nu_1 \nu_2 \nu_3 \dots \rangle$, where for all $i \geq 1$, ν_i satisfies $(e^a \wedge e^b)$, thus ν_i satisfies both e^a and e^b . Let the concrete run be $\langle (\mathbf{s}_0^a, \mathbf{s}_0^b, 1) \nu_1(s_1^a, s_1^b, x) \nu_2(s_2^a, s_2^b, x) \nu_3 \dots \rangle$ of A^{ab} . This concrete run corresponds to $\langle \mathbf{s}_0^a \nu_1 s_1^a \nu_2 s_2^a \nu_3 \dots \rangle$ of A^a , which is accepted by A^a because it goes infinitely often through $F^a \times S^b \times \{1\}$ thus it goes infinitely often through F^a . And $\langle \mathbf{s}_0^b \nu_1 s_1^b \nu_2 s_2^b \nu_3 \dots \rangle$ of A^b is also accepting because whenever the automaton goes through an accepting state of A^b , the marker changes to 1 again. Thus, the acceptance condition guarantees that the run of the automaton visits accepting states of A^b infinitely often.

“ \subseteq ” we take an arbitrary concrete run $\langle \mathbf{s}_0^a \nu_1 s_1^a \nu_2 s_2^a \nu_3 \dots \rangle$ accepted by A^a , where for all $i \geq 1$, ν_i satisfies e^a . And an arbitrary concrete run $\langle \mathbf{s}_0^b \nu_1 s_1^b \nu_2 s_2^b \nu_3 \dots \rangle$ accepted by A^b , where for all $i \geq 1$, ν_i satisfies e^b . Both runs correspond to a word $w = \langle \nu_1 \nu_2 \nu_3 \dots \rangle$. So, there is a concrete run $\langle (\mathbf{s}_0^a, \mathbf{s}_0^b, 1) \nu_1(s_1^a, s_1^b, x) \nu_2(s_2^a, s_2^b, x) \nu_3 \dots \rangle$ of A^{ab} on w , where for all $i \geq 1$, ν_i satisfies $(e^a \wedge e^b)$ and whenever the automaton goes through an accepting state, the marker changes. Thus, the acceptance condition guarantees that the run of the automaton visits accepting states infinitely often, since a run accepts if and only if it goes infinitely often through $F^a \times S^b \times \{1\}$.

Termination. This construction terminates because our states in S and formulas in \mathcal{E} are finite.

Complexity. The construction uses an oracle to an SMT solver to solve $DecisionProcedure(e^a \wedge e^b) = SAT$, where the theory \mathcal{T} is decidable in the complexity class \mathcal{C} . Hence, the time and space complexity of the construction is $O(|S^a| \cdot |S^b| \cdot |\Delta_{\mathcal{T}}^a| \cdot |\Delta_{\mathcal{T}}^b|)^{\mathcal{C}}$. \square

Intersection of automata illustrates another subtle difference with lazy satisfiability approach (based on boolean abstraction in SMT). For example, in Figure 4.2a, classically

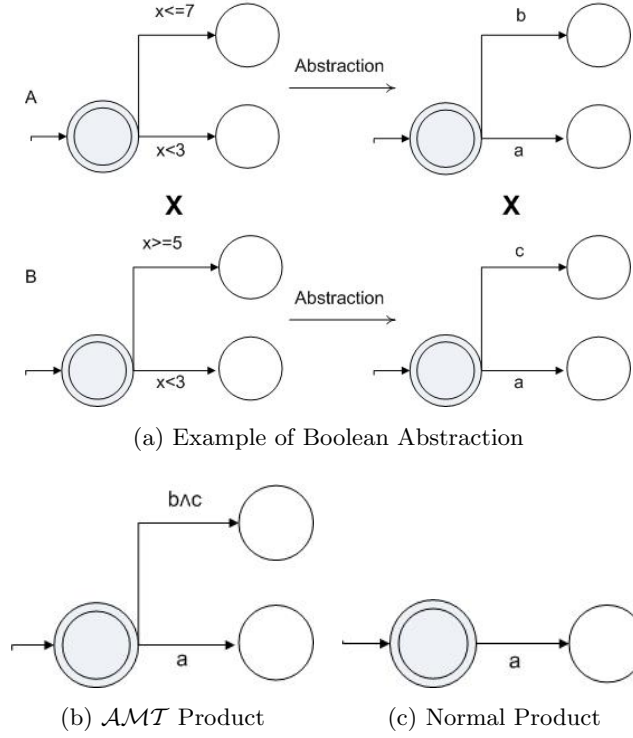


Figure 4.2: Boolean Abstraction

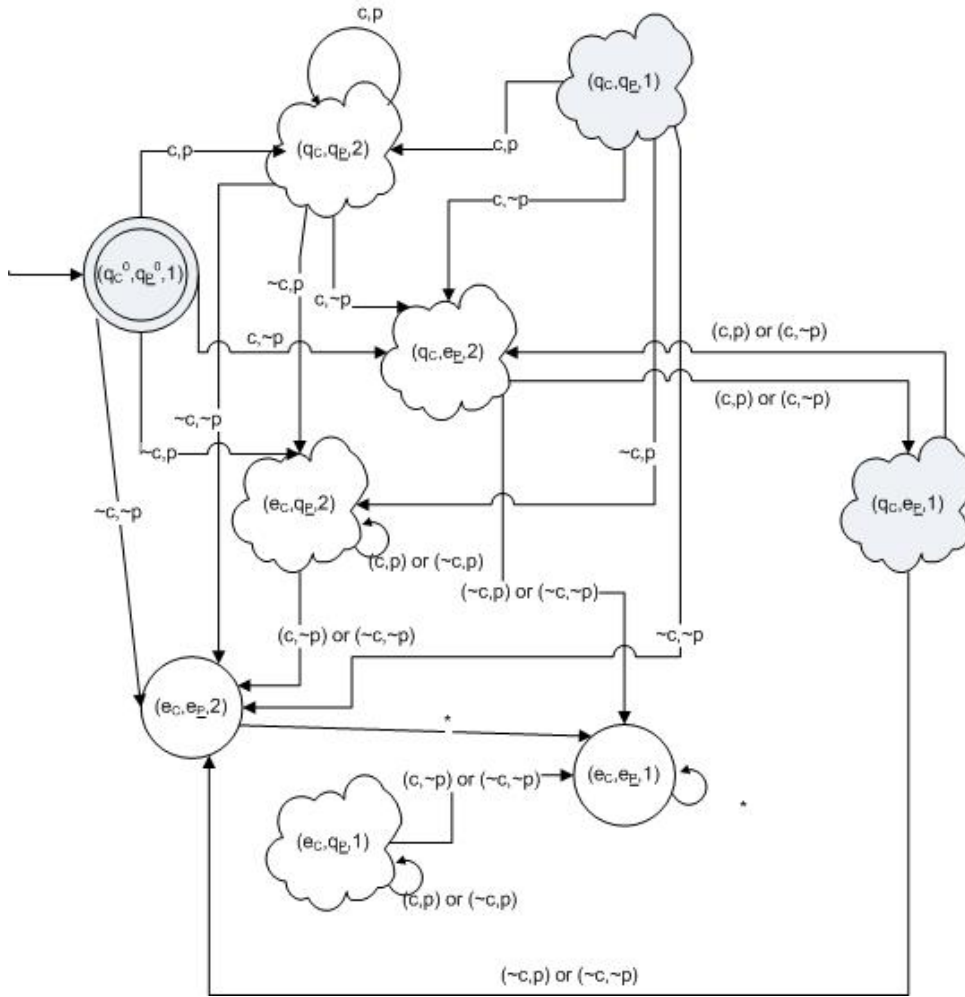
we have the result of automata intersection as in Figure 4.2c, where we only have reduced the possible results. However, in \mathcal{AMT} , we can have more transitions, as shown in Figure 4.2b.

Definition 4.4.2 is a general construction, as depicted on Figure 4.3a (see abbreviations on Fig. 4.3c). However, when we consider our domain of application, namely matching a mobile's policy and a midlet's contract, then the fact that we intersect a contract automaton with a special property (i.e. it has only one non accepting state (namely the error state)) and a complement of policy automaton which has also a special property (i.e. it has only one accepting state that is the error state), enable us to optimize the intersection such that we only consider correct contract transitions (shown in Figure 4.3b).

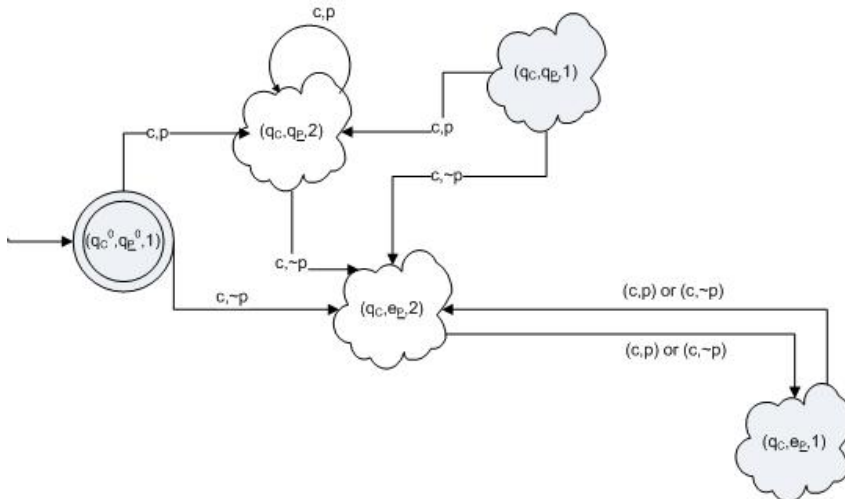
Emptiness problem of \mathcal{AMT} . An \mathcal{AMT} automaton A is not empty when there exists some words accepted by A , meaning $L_\omega(A) \neq \emptyset$ if and only if there exists some accepting concrete run as defined in Definition 4.3.3.

Proposition 4.4.3 *Let the theory \mathcal{T} be decidable with an oracle for the SMT problem in the complexity class \mathcal{C} then:*

1. *The non-emptiness problem for \mathcal{AMT} is decidable in $LIN - TIME^{\mathcal{C}}$.*



(a) Automata Intersection without Optimization



(b) Automata Intersection with Optimization

$c \doteq$ valid contract transition
 $\neg c \doteq$ invalid contract transition
 $\neg p \doteq$ invalid policy transition
 shaded areas are accepting states

(c) Abbreviations

Figure 4.3: Automata Intersection

2. The non-emptiness problem for \mathcal{AMT} is $NLOG - SPACE^C$ -complete.

Proof. we prove Proposition 4.4.3 by showing that $L_\omega(A) \neq \emptyset$ if and only if there exists some accepting state which is connected to the initial state and also connected to itself as in [82]. Let $A = \langle S, \Sigma, T, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$.

Correctness.

“ \supseteq ” we assume that $L_\omega(A) \neq \emptyset$, meaning there exists an arbitrary concrete run $\sigma_C = \langle s_0\nu_1s_1\nu_2s_2 \dots \rangle$ accepted by A that corresponds to a word $w = \langle \nu_1\nu_2\nu_3 \dots \rangle$. By Definition 4.3.3 $\forall i \geq 0$ state s_i is directly connected to state s_{i+1} . Thus, when $i < k$ then s_i is connected to s_k . Furthermore, there exists some accepting state which is visited infinitely often, meaning that there is some $s_t \in F$ and there are i, k where $0 < i < k$ such that $s_t = s_i = s_k$. Hence, s_t is connected to the initial state s_0 and also connected to itself.

“ \subseteq ” we assume that there exists some accepting state $s_t \in F$ which is connected to the initial state and also connected to itself. So, there is a sequence of states $\langle s_{s_0}s_{s_1}s_{s_2} \dots s_{s_k} \rangle$ from the initial state to $s_{s_k} = s_t$ that corresponds to a word $\langle \nu_{s_1}\nu_{s_2}\nu_{s_3} \dots \nu_{s_k} \rangle$ and $\forall i \geq 0$ state s_{s_i} is directly connected to state $s_{s_{i+1}}$. Furthermore, there are also sequences of states $\langle s_{t_0}s_{t_1}s_{t_2} \dots s_{t_l} \rangle$ from $s_{t_0} = s_t$ to $s_{t_l} = s_t$ that corresponds to a word $\langle \nu_{t_1}\nu_{t_2}\nu_{t_3} \dots \nu_{t_l} \rangle$ and $\forall i \geq 0$ state s_{t_i} is directly connected to state $s_{t_{i+1}}$. Thus $\langle \nu_{s_1}\nu_{s_2}\nu_{s_3} \dots \nu_{s_k} \rangle \langle \nu_{t_1}\nu_{t_2}\nu_{t_3} \dots \nu_{t_l} \rangle^\omega$ is accepted by A and $L_\omega(A) \neq \emptyset$.

Complexity. The emptiness problem of \mathcal{AMT} can be reduced to graph reachability. A combination of an algorithm based on Nested DFS [72] with a decision procedure for SMT can solve this problem. The algorithm takes as input the an \mathcal{AMT} automaton A and starts a depth first search procedure **check_safety** (\mathbf{s}_0) (Algorithm 1) over the initial state \mathbf{s}_0 . When an accepting state in \mathcal{AMT} is reached, we start a new depth first search (Algorithm 2) from the candidate state to determine whether it is in a cycle, in other words if it is reachable from itself. If it is, then we report that the automaton is non-empty.

When a state is first generated, it is marked as **unchecked**. During an unfinished search in Algorithm 1, a state is marked as **in_current_path**. When a state has finished its Algorithm 1 and not yet processed in Algorithm 2, then it is marked as **safety_checked**. Finally, a state is marked **availability_checked** when it has been processed by both Algorithm 1 and Algorithm 2.

This algorithm can be solved in linear time on the size of A 's states. In addition an oracle to an SMT solver is used to solve $DecisionProcedure(e) = SAT$. Hence, its complexity is $LIN - TIME^C$.

Algorithm 1 `check_safety(s)` Procedure

Input: state s ;

```
1:  $map(s) := in\_current\_path$ ;  
2: for all  $((s, e, t) \in \Delta)$  do  
3:   if  $(DecisionProcedure(e) = SAT)$  then  
4:     if  $(map(t) = in\_current\_path \wedge ((s \in F) \vee (t \in F)))$  then  
5:       report non-empty;  
6:     else if  $(map(t) = unchecked)$  then  
7:       check_safety( $t$ );  
8:   if  $(s \in F)$  then  
9:     check_availability( $s$ );  
10:   $map(s) := availability\_checked$ ;  
11: else  
12:   $map(s) := safety\_checked$ ;
```

Algorithm 2 `check_availability(s)` Procedure

Input: state s ;

```
1: for all  $((s, e, t) \in \Delta)$  do  
2:   if  $(DecisionProcedure(e) = SAT)$  then  
3:     if  $(map(t) = in\_current\_path)$  then  
4:       report non-empty;  
5:     else if  $(map(t) = safety\_checked)$  then  
6:        $map(t) := availability\_checked$   
7:     check_availability( $t$ );
```

The algorithm needs only a logarithmic memory, since at each step it needs to remember fewer states than the number of its total states and there are only two bits added to each state for the marker. Also, an SMT solver is used to solve $DecisionProcedure(e) = SAT$ and Jones [51] showed that graph reachability problem is $NLOG - SPACE$ -hard. Hence, the emptiness problem of \mathcal{AMT} is $NLOG - SPACE^c$ -complete. \square

Language inclusion problem of \mathcal{AMT} . Language of an \mathcal{AMT} automaton A^a is subsumed by the language of an \mathcal{AMT} automaton A^b when for all the words $w = \langle \nu_1 \nu_2 \dots \rangle$ (as defined in Definition 4.3.3) accepted by A^a , w is also accepted by A^b .

Proposition 4.4.4 *Let A^a, A^b be \mathcal{AMT} over a set of valuations \mathcal{V} . Then $\mathcal{L}_{A^a} \subseteq \mathcal{L}_{A^b}$ such that A^b accepts all the concrete runs which are accepted by A^a is decidable.*

Proof. we prove Proposition 4.4.4 by showing that $\mathcal{L}_{A^a} \subseteq \mathcal{L}_{A^b}$ if and only if the language of $A^a \times \overline{A^b}$ is empty that is:

$$\mathcal{L}_{A^a} \subseteq \mathcal{L}_{A^b} \Leftrightarrow \mathcal{L}_{A^a} \cap \overline{\mathcal{L}_{A^b}} = \emptyset \Leftrightarrow \mathcal{L}_{A^a} \cap \mathcal{L}_{\overline{A^b}} = \emptyset \Leftrightarrow \mathcal{L}_{A^a \times \overline{A^b}} = \emptyset.$$

Correctness.

“ \supseteq ” we assume that there exists some concrete run which is accepted by A^a but not by A^b . Thus, $\mathcal{L}_{A^a \times \overline{A^b}}$ is not empty, which is a contradiction.

“ \subseteq ” we assume that $\mathcal{L}_{A^a \times \overline{A^b}}$ is not empty, meaning there exists some concrete runs accepted by $A^a \times \overline{A^b}$. Thus, this run is accepted by both A^a and $\overline{A^b}$. Because $L_\omega(\overline{A^b}) = \mathcal{V}^\omega - L_\omega(A^b)$, thus there exists some concrete run which is accepted by A^a but not by A^b , which is a contradiction.

Complexity. Language inclusion problem of \mathcal{AMT} is decidable follows from Proposition 4.4.1, Proposition 4.4.2, and Proposition 4.4.3 and derived the complexity from the afore mentioned propositions. \square

The language inclusion problem of \mathcal{AMT} (Proposition 4.4.4) is defined over concrete runs, thus in \mathcal{AMT} symbolic language inclusion coincides with concrete language inclusion.

4.5 On-the-fly Language Inclusion Matching

In order to do matching between a contract with a security policy, our algorithm takes as input two automata A^C and A^P representing respectively the formal specification of a contract and of a policy. A match is obtained when the language accepted by A^C (the execution traces of the midlet) is a subset of the language accepted by A^P (the

acceptable traces for the policy). The matching problem can be reduced to an emptiness test: $\mathcal{L}_{A^C} \subseteq \mathcal{L}_{A^P} \Leftrightarrow \mathcal{L}_{A^C} \cap \overline{\mathcal{L}_{A^P}} = \emptyset \Leftrightarrow \mathcal{L}_{A^C} \cap \mathcal{L}_{\overline{A^P}} = \emptyset \Leftrightarrow \mathcal{L}_{A^C \times \overline{A^P}} = \emptyset$. In other words, there is no behavior of A^C which is disallowed by A^P . If the intersection is not empty, then any behavior in it corresponds to a counterexample.

Constructing the product automaton explicitly is not practical for mobile devices. First, this can lead into an automaton too large for the mobile limited memory footprint. Second, to construct a product automata we need software libraries for the explicit manipulation and optimizations of symbolic states, which are computationally heavy and not available on mobile phones. Furthermore, we can exploit the explicit structure of the contract-policy as a number of separate requirements. Hence, we use on-the-fly emptiness test (constructing product automaton while searching the automata). The on-the-fly emptiness test can be lifted from the traditional algorithm by a technique from Coucubertis et al. [23] while modification of this algorithm from Holzmann et al's [48] is considered as state-of-the-art (used in Spin [49]). Gastin et al [35] proposed two modifications to [23] for finding faster and minimal counterexample.

Remark 4.5.1 *Our algorithm is tailored particularly for contract-policy matching, as such, it exploits a special property of \mathcal{AMT} representing security policies, namely each automaton has only one non accepting state (the error state). The algorithm can be generalized by removing all specialized tests, for example on line 8 from Algorithm 3 $\dots \wedge s^{\overline{P}} = \text{err}^{\overline{P}} \wedge \dots$ can be replaced by accepting states from $\overline{A^P}$, and reporting only availability violation (corresponding to a non-empty automaton). This generic algorithm corresponds to on-the-fly algorithm for model checking of BA.*

We are now in the position to state our contract-policy matching's result using language inclusion:

Proposition 4.5.1 *Let the theory \mathcal{T} be decidable with an oracle for the SMT problem in the complexity class \mathcal{C} then:*

1. *The contract-policy matching problem for \mathcal{AMT} using language inclusion is decidable in $LIN - TIME^{\mathcal{C}}$.*
2. *The contract-policy matching problem for \mathcal{AMT} using language inclusion is decidable in $NLOG - SPACE^{\mathcal{C}}$ -complete.*

Proof. We prove Proposition 4.5.1 by showing that $\mathcal{L}_{A^C \times \overline{A^P}} = \emptyset$ if and only if there exists no accepting state of $A^C \times \overline{A^P}$ which is connected to the initial state of $A^C \times \overline{A^P}$ and also connected to itself where $A^C \times \overline{A^P} = A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$. Let A accept all the concrete runs which are accepted by A^C and $\overline{A^P}$, that is A is an intersection automaton

Algorithm 3 $\text{check_safety}(s^c, s^{\bar{p}}, x)$ Procedure

Input: state s^c , state $s^{\bar{p}}$, marker x ;

```
1:  $\text{map}(s^c, s^{\bar{p}}, x) := \text{in\_current\_path}$ ;  
2: for all  $((s^c, e^c, t^c) \in \Delta^c)$  do  
3:   for all  $((s^{\bar{p}}, e^{\bar{p}}, t^{\bar{p}}) \in \Delta^{\bar{p}})$  do  
4:     if  $(\text{DecisionProcedure}(e^c \wedge e^{\bar{p}}) = \text{SAT})$  then  
5:        $y := \text{condition}(s^c, s^{\bar{p}}, x, S^c, S^{\bar{p}})$   
6:       if  $(\text{map}(t^c, t^{\bar{p}}, y) = \text{in\_current\_path} \wedge ((s^c \in S^c \wedge s^{\bar{p}} = \text{err}^{\bar{P}} \wedge x = 1) \vee (t^c \in S^c \wedge t^{\bar{p}} = \text{err}^{\bar{P}} \wedge y = 1)))$  then  
7:         report policy violation;  
8:       else if  $(\text{map}(t^c, t^{\bar{p}}, y) = \text{in\_current\_path} \wedge ((s^c \in S^c \wedge s^{\bar{p}} \in (S^{\bar{p}} \setminus \{\text{err}^{\bar{P}}\}) \wedge x = 1) \vee (t^c \in S^c \wedge t^{\bar{p}} \in (S^{\bar{p}} \setminus \{\text{err}^{\bar{P}}\}) \wedge y = 1)))$  then  
9:         report availability violation;  
10:      else if  $(\text{map}(t^c, t^{\bar{p}}, y) = \text{unchecked})$  then  
11:         $\text{check\_safety}(t^c, t^{\bar{p}}, y)$ ;  
12: if  $(s^c \in S^c \wedge s^{\bar{p}} \in S^{\bar{p}} \wedge x = 1)$  then  
13:    $\text{check\_availability}(s^c, s^{\bar{p}}, x)$ ;  
14:    $\text{map}(s^c, s^{\bar{p}}, x) := \text{availability\_checked}$ ;  
15: else  
16:    $\text{map}(s^c, s^{\bar{p}}, x) := \text{safety\_checked}$ ;
```

Algorithm 4 $\text{check_availability}(s^c, s^{\bar{p}}, x)$ Procedure

Input: state s^c , state $s^{\bar{p}}$, marker x ;

```
1: for all  $((s^c, e^c, t^c) \in \Delta^c)$  do  
2:   for all  $((s^{\bar{p}}, e^{\bar{p}}, t^{\bar{p}}) \in \Delta^{\bar{p}})$  do  
3:     if  $(\text{DecisionProcedure}(e^c \wedge e^{\bar{p}}) = \text{SAT})$  then  
4:        $y := \text{condition}(s^c, s^{\bar{p}}, x, S^c, S^{\bar{p}})$   
5:       if  $(\text{map}(t^c, t^{\bar{p}}, y) = \text{in\_current\_path})$  then  
6:         if  $(t^{\bar{p}} = \text{err}^{\bar{P}})$  then  
7:           report policy violation;  
8:         else  
9:           report availability violation;  
10:      else if  $(\text{map}(t^c, t^{\bar{p}}, y) = \text{safety\_checked})$  then  
11:         $\text{map}(t^c, t^{\bar{p}}, y) := \text{availability\_checked}$   
12:         $\text{check\_availability}(t^c, t^{\bar{p}}, y)$ ;
```

such that $L_\omega(A) = L_\omega(A^C) \cap L_\omega(\overline{A^P})$.

Correctness.

The proof is similar to Proof 4.4, however we consider the product of two automata.

“ \supseteq ” we assume that $L_\omega(A) \neq \emptyset$, meaning there exists an arbitrary concrete run $\sigma_C = \langle s_0\nu_1s_1\nu_2s_2\dots \rangle$ accepted by A that corresponds to a word $w = \langle \nu_1\nu_2\nu_3\dots \rangle$ where for all $i \geq 1$, ν_i satisfies $(e^c \wedge e^{\overline{p}})$, thus ν_i also satisfies e^c and $e^{\overline{p}}$. By Definition 4.3.3 $\forall i \geq 0$ state s_i is directly connected to state s_{i+1} . Thus, when $i < k$ then s_i is connected to s_k . Furthermore, there exists some accepting state which is visited infinitely often, meaning that there is some $s_t \in F$ and there are i, k where $0 < i < k$ such that $s_t = s_i = s_k$. Hence, s_t is connected to the initial state s_0 and also connected to itself.

“ \subseteq ” we assume that there exists some accepting state $s_t \in F$ which is connected to the initial state and also connected to itself. So, there is a sequence of states $\langle s_{s_0}s_{s_1}s_{s_2}\dots s_{s_k} \rangle$ from the initial state to $s_{s_k} = s_t$ that corresponds to a word $\langle \nu_{s_1}\nu_{s_2}\nu_{s_3}\dots \nu_{s_k} \rangle$, where for all $i \geq 1$, ν_{s_i} satisfies $(e^c \wedge e^{\overline{p}})$, thus ν_{s_i} also satisfies e^c and $e^{\overline{p}}$, and $\forall i \geq 0$ state s_{s_i} is directly connected to state $s_{s_{i+1}}$. Furthermore, there are also sequences of states $\langle s_{t_0}s_{t_1}s_{t_2}\dots s_{t_l} \rangle$ from $s_{t_0} = s_t$ to $s_{t_l} = s_t$ that corresponds to a word $\langle \nu_{t_1}\nu_{t_2}\nu_{t_3}\dots \nu_{t_l} \rangle$, where for all $i \geq 1$, ν_{t_i} satisfies $(e^c \wedge e^{\overline{p}})$, thus ν_{t_i} also satisfies e^c and $e^{\overline{p}}$, and $\forall i \geq 0$ state s_{t_i} is directly connected to state $s_{t_{i+1}}$. Thus $\langle \nu_{s_1}\nu_{s_2}\nu_{s_3}\dots \nu_{s_k} \rangle \langle \nu_{t_1}\nu_{t_2}\nu_{t_3}\dots \nu_{t_l} \rangle^\omega$ is accepted by A and $L_\omega(A) \neq \emptyset$.

Complexity. The matching between a contract with a security policy problem can be reduced to an emptiness test of the product automaton of between a contract with a complement of security policy. A combination of an algorithm based on Nested DFS [72] with a decision procedure for SMT can solve this problem. The algorithm takes as input the midlet’s claim and the mobile platform’s policy and starts a depth first search procedure **check_safety** ($\mathbf{s}_0^C, \mathbf{s}_0^{\overline{P}}, 1$) (Algorithm 3) over the initial state $(\mathbf{s}_0^C, \mathbf{s}_0^{\overline{P}}, 1)$. When an accepting state in \mathcal{AMT} is reached, we have two cases. First, when the state contains an error state of complemented policy ($err^{\overline{P}}$), then we report a security policy violation without further ado.¹ Second, the state does not contain an error state of complemented policy ($S^{\overline{P}} \setminus \{err^{\overline{P}}\}$). Then, we start a new depth first search (Algorithm 4) from the candidate state to determine whether it is in a cycle, in other words if it is reachable from itself. If it is, then we report an availability violation.

We use the same marking as in \mathcal{AMT} emptiness check, namely when a state is first

¹The Error state is a convenient mathematical tool, but the trust assumption of the matching algorithm is that the code obeys the contract and therefore, it should never reach the error state where any action is permitted.

generated, it is marked as `unchecked`. During an unfinished search in Algorithm 3, a state is marked as `in_current_path`. When a state has finished its Algorithm 3 and not yet processed in Algorithm 4, then it is marked as `safety_checked`. Finally, a state is marked `availability_checked` when it has been processed by both Algorithm 3 and Algorithm 4. We also apply function `condition(s, t, x, F1, F2)` that implements marker signing of y given x and current states from the Definition 4.4.2 of \mathcal{AMT} intersection.

This algorithm can be solved in linear time on the size of the number of the states of the product. In addition an oracle to an SMT solver is used to solve $DecisionProcedure(e^c \wedge e^{\bar{p}}) = SAT$. Hence, its complexity is $LIN - TIME^C$.

The algorithm needs only a logarithmic memory, since at each step it needs to remember fewer states than the number of the total product states and there are only two bits added to each state for the marker. Also, an SMT solver is used to solve $DecisionProcedure(e^c \wedge e^{\bar{p}}) = SAT$ and as in non-emptiness of \mathcal{AMT} we have $NLOG - SPACE$ -hardness follows from Jones [51] who showed that graph reachability problem is $NLOG - SPACE$ -hard. Hence, the contract-policy matching problem of \mathcal{AMT} is $NLOG - SPACE^C$ -complete. \square

As we have shown, matching between a contract with a security policy problem can be reduced to an emptiness test of the product automaton of a contract with a complement of security policy: $\mathcal{L}_{AC} \subseteq \mathcal{L}_{AP} \Leftrightarrow \mathcal{L}_{AC \times \overline{AP}} = \emptyset$. Furthermore, the set of infinite words recognized by an automaton A , denoted by $L_\omega(A)$, is the set of all accepting infinite traces in A ($w = \langle \nu_1 \nu_2 \dots \rangle$). Because the language of an automaton A is defined in concrete level, thus the symbolic language coincides with the concrete language. Therefore, contract-policy matching using language inclusion in symbolic and concrete notion coincides.

Chapter 5

On-the-fly Matching Prototype Implementation and Experiments

In this chapter, we try to provide an answer to the following question: how can we implement matching and what is the best configuration of integrating automata-based inclusion algorithm with decision procedure? To address this issue we give possible design decisions and run experiment both on desktop and mobile device. We continue with detailing the running-time on the mobile platform for one design decision only to give the reader a feeling for how the matching algorithm with integrated decision procedure can run in real application.

5.1 Introduction

This chapter describes the prototype implementation of contract-policy matching in \mathcal{AMT} , its integration with decision solver based on MathSAT and NuSMV, and the results of our experiments on matching.

We begin in Section 5.2 by discussing the overall implementation architecture and the integration issues with the procedure solver NuSMV [22] integrated with its MathSAT libraries [18]. Since our goal is to provide this midlet-contract vs platform-policy matching on-the-fly (during the actual download of the midlet) issues like small memory footprint, and effective computations play a key role.

Section 5.3 continues with implementation of language inclusion as emptiness test using an on-the-fly procedure with oracle calls to the decision procedures available in NuSMV. Therefore our design decision \mathcal{AMT} makes reasoning about infinite transitions systems with finite states possible without symbolic manipulation procedures of zones and regions

or finite representation by equivalence classes whose memory intensive characteristic is not suitable for our application.

Our prototype was first implemented in Java and was run on a Desktop PC with operating system Linux. Then, it had also been ported to .NET for actual detailed profiling, namely for HTC P3600 (3G PDA phone) with ROM 128MB, RAM 64MB, Samsung®SC32442A processor 400MHz and operating system Microsoft®Windows Mobile®5.0 with Direct Push technology.

Finally, Section 5.5 presents a detailed performance analysis of the integration design alternatives regarding the construction of expressions, the initialization of solver, and the caching of temporary results by considering both running time and internal metrics of various available options.

5.2 The Architecture

In this section we describe the conceptual architecture of the prototype that implements the overall matching algorithm and supports integration with a decision procedure solver NuSMV [22] integrated with its MathSAT libraries [18]. We provide an overview of how the prototype is implemented to show the possible options for integration with the solver. The contract-matching prototype takes as input a contract and a policy and checks whether or not the contract matches the policy. The prototype architecture is depicted in Figure 5.1. Detailed class diagram is available on Appendix A.

Our first observation is that the policy has to be deployed on the device and it is unlikely to change frequently. The second observation is that, even if applications (and related contracts) will change frequently and dynamically, the binding between an application and its contract will considerable be static. If a digital signature or a proof carrying code is used, the contract has to be shipped with the application. In the case of Java application, this contract must be essentially included in the JAR file that represents the application and must be directly accessible to the virtual machine that is responsible for the matching and the enforcement of the security policy (see [81] for details).

The prototype consists of two parts, namely on-device and off-device implementations. During off-device part execution, the contract and policy are transformed into a suitable internal representation for the on-the-fly algorithm. The policy automaton is also complemented at this step of the execution. In on-device part of the prototype the main on-the-fly algorithm runs on the contract and policy input and make calls to the decision procedure during its execution.

Initially, we implemented our prototype in Java platform and subsequently the architecture remained the same for the .NET platform. Thus, we are only describing our

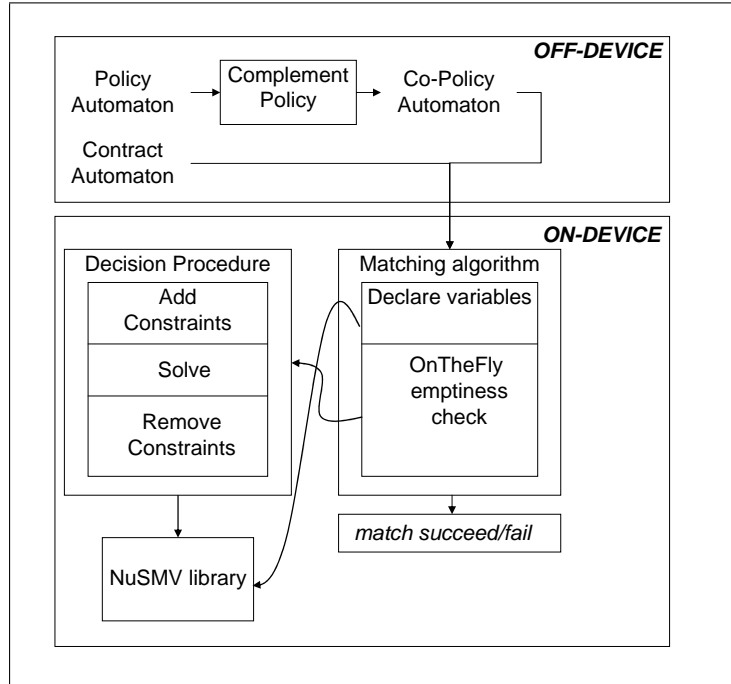


Figure 5.1: On-the-fly Implementation Architecture

architecture in Java platform. The initial algorithm transforms a contract (resp. a policy) into a Java class, `ContractAutomaton.java` (resp. `PolicyAutomaton.java`) that can be directly manipulated by the actual algorithm responsible for the on-the-fly policy matching (i.e. emptiness test). If the policy option is specified then the parser also performs the complementation of the policy. Management of the variables declaration is discussed later in Section 5.3.

Since a contract-policy matching algorithm should frequently call the decision procedure during its execution, we need a design decision for an internal representation of \mathcal{AMT} . We discuss this particular form of \mathcal{AMT} in details. First, we associate a number of variables to every edge, where *method* is an API call that the policy is supposed to rule, *cond* - a guarded command which must be true in order for the method to be executed, for instance a *cond* specifies that the url must start with the string “https”.

For further representation simplification, we follow the semantics for security automata proposed in [8] so that we have a prioritized execution among guards: we go to the next guard only if the guards before it have all failed. Such information is represented in *otherConds* - the other guarded commands that failed before reaching the current guard *otherMethods* - an expression consists of all other methods that are not supposed to rule at the current moment.

Once contract and policy automata are made available to the main system, we can

run the on-the-fly procedure which has been also implemented in Java using only MIDP libraries to guarantee portability (and we have similarly developed a .NET mobile implementation in C#).

The next stage is a non-trivial point because we need to interact with a decision procedure for solving \mathcal{AMT} 's expressions which are defined in complex theories for example boolean expressions and mathematical expressions. We use the solver as a black box (an oracle) for the general algorithm that gives the answer whether the problem is satisfiable or not. We have further decided to interface with the solver without using its internal data structure but rather to interact with the decision procedure by using strings. While this creates a bit of overhead for parsing, it makes it significantly easier to replace the solver as needed.

5.3 Design Decisions

Different design decisions are made in order to *decide the best configuration of integrating automata-based inclusion algorithm with decision procedure* as the problem is not trivial. Every option of the configuration proposed below has different memory impact and this information and results of such analysis is very important because of the resource constraints of mobile device. This restriction is not commonly studied in classical decision procedure integration papers because the problem of resources is not critical.

In integrating matching algorithm with the theory solver we faced a number of design options:

One.vs.Many Solver in object oriented languages is by itself an object. We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure.

MUTEX_SOLVER if an edge in the automaton correspond to a call to a method it is obviously incompatible with another edge calling a different method. Such constraints could be directly incorporated into the algorithm without the need to represent them as boolean mutual exclusion constraints on the boolean variables representing method invocations. In this case all the method names are declared as mutex constants at the moment of declaring all variables, then the expression sent to the solver has the following structure: $method = name \wedge cond \wedge otherConds$. Hence, if the method names of two edges are not the same then the DecisionProcedure returns false.

MUTEX_MC allows the on-the-fly algorithm to check whether method names are the same. The DecisionProcedure is called with parameters: $cond \wedge otherConds$ only if this

check is passed.

PRIORITY_MC the semantics for security policy is that guards are evaluated using *priority* or hence we can optimize the expressions sent to the decision procedure as lemmas. Using the lemma, the Expression sent to the DecisionProcedure is minimized and it has only *cond*.

CACHING_MC Since many edges will be traversed again and again we could save time by caching the results of the matching. The solver itself has a caching mechanism that could be equally used (**CACHING_SOLVER**).

While we assumed that all decision could be just taken after considering preliminary experimental results it turned out that at least for the `One_vs_Many` decision this was not possible. The cause is the management of garbage collection both by the Java virtual machine and by the libraries of MathSAT/NuSMV which requires only one instance of solver exists at time in order to interact correctly with the NuSMV library. This leads to use a static invocation for the solver and set significant constraints on the interaction.

For example, before starting to visit all constraints to the library, all variables used in expressions must be declared. The NuSMV library has to invoke *DeclareNewBooleanVar*, *DeclareNewWordVar*, *DeclareNewStringVar* methods for declaration of boolean, integer and string variables respectively. Only after declaring all the variables from contract and policy expressions, the on-the-fly algorithm can actually start invoking the decision procedure in its visit. A consequence of this rule is that with this implementation we cannot insert edges that introduce new variables because the solver can be called only after declaring all the variables and adding all the needed constraints.

Therefore, during the visit of the algorithm we must at first upload constraints to the solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*.

The rest design alternatives can be implemented and tested thus giving way to the six alternative configurations (see Fig. 5.2d) of the interactions between the solver and the on-the-fly emptiness check algorithm.

Table 5.1: Problems Suit

Problem	Contract	Policy	SC	TC	SP	TP
P1	size_100_512_contract.pol	size_10_1024_policy.pol	2	4	2	4
P2	maxKB512_contract.pol	maxKB1024_policy.pol	2	4	2	4
P3	noPushRegistry_contract.pol	oneConnRegistry_policy.pol	2	3	3	9
P4	notCreateRS_contract.pol	notCreateSharedRS_policy.pol	2	4	2	4
P5	pimNoConn_contract.pol	pimSecConn_policy.pol	3	7	3	9
P6	2hard_contract.pol	2hard_policy.pol	3	7	3	7
P7	http_contract.pol	https_policy.pol	3	7	3	7
P8	3hard_contract.pol	3hard_policy.pol	3	7	3	7
P100	noSMS_contract.pol	100SMS_policy.pol	2	4	102	304

5.4 List of Abbreviations

In this thesis we use the following abbreviations:

SC: Number of States of Contract

TC: Number of Transitions of Contract

SP: Number of States of Policy

TP: Number of Transitions of Policy

SG: Number of States of generated Policy/Contract

ART: Average Runtime for 10 runs

5.5 Experiments on Desktop and on Device

To understand the best option we collected data on resources used, namely number of visited states, number of visited transitions, running time for each problem in each design alternative, and the number of solved problems against time. For sake of example we list in Table 5.1 some sample possible combinations of policy-contract (mis)matching pairs. For instance, the contract `pimNoConn_contract.pol` represents Example 2.1.1 and the policy `pimSecConn_policy.pol` corresponds to Example 2.1.2.

With the exception of the pathological problem P100, which has been designed that way, most problems have few states and transitions and, as we shall see in the next table (Table 5.2 showing performance of ten times run for each problem set and each design alternative), they also require little time for being assessed.

Notice that the number of states and transitions in the AMT for each contract and policy in Table 5.1 is a number of reachable states and transitions. During the running of matching algorithm there may be the case when the algorithm stops working (producing "do not match" answer) without reaching all the states of contract and/or policy. And this case is explicitly shown in P6, P7 and P8 examples in Table 5.2. That is why we

Table 5.2: Running Problem Suit 10 Times

MUTEX_MC ONE_INSTANCE CACHING_SOLVER									
Problem	Desktop				Mobile				Result
	ART (s)	CRT (s)	SV	TV	ART (s)	CRT (s)	SV	TV	
P1	2.4	2.4	2	6	4.3	4.3	2	6	Match
P2	2.4	4.8	2	6	4.1	8.4	2	6	Match
P3	2.4	7.2	3	11	3.9	12.3	3	11	Match
P4	2.4	9.6	2	6	4.0	16.3	2	6	Match
P5	4.7	14.3	3	11	4.1	20.4	3	11	Match
P6	2.9	2.9	4	4	3.8	3.8	3	6	Not Match
P7	2.8	5.7	5	7	3.8	7.6	2	4	Not Match
P8	2.9	8.6	5	7	3.8	11.4	3	6	Not Match
P100	9.3	9.3	102	307	11.3	11.3	102	307	Match

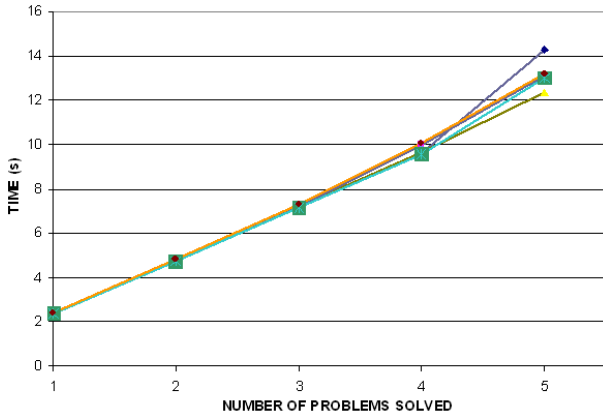
only present here the number of reachable states in Table 5.1 and number of visited states during on-the-fly running in Table 5.2.

We run our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Linux version 2.6.20-16-generic, Kubuntu 7.04 (Feisty Fawn). Currently, we are also porting the application to the mobile for actual detailed profiling, namely HTC P3600 (3G PDA phone) with ROM 128MB, RAM 64MB, Samsung®SC32442A processor 400MHz and operating system Microsoft®Windows Mobile®5.0 with Direct Push technology.

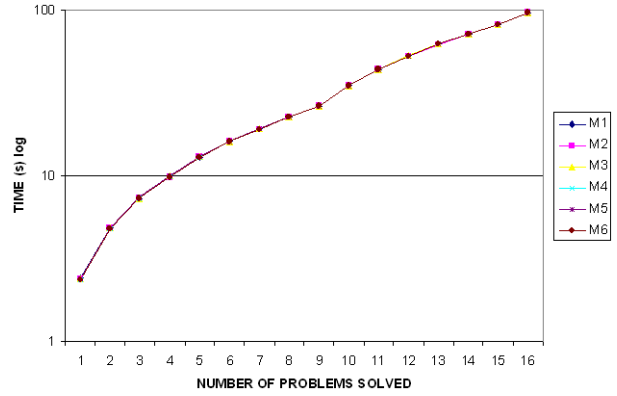
For the sake of example we present the result obtained for alternative with MUTEX_MC ONE_INSTANCE CACHING_SOLVER in Table 5.2. The results for all design alternatives are mapped into diagram shown in Figure 5.2a for matching problems and Figure 5.2c for not matching problems. Notice that we only provide the cumulative running time that is necessary to solve all problems. This is important because our goal is to match (or not match) all rules in a contract with all corresponding rules in a policy. Thus, the value of the single problem is not important except for some cases where the average output might be significantly off due to some off scale rule.

We singled out P100 as a challenging artificial problem because it has a large number of states compared to the others: essentially this happened because we draw an automaton modulo theory with 100 states and which traverse from one state to another by adding 1 to the number of SMS sent.

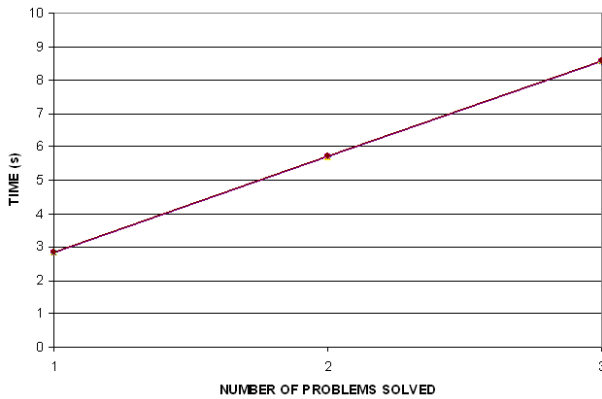
In this case there is a difference between M1 and M5, namely 9.259 s and 9.117 s resp., that is M5 is better around 1.5% than M1. In order to study this in more details, we generated more unreal problem sets: as P100 with combination of sent SMS none, 1, 10,



(a) Match succeeds for real policies



(b) Matches among synthetic contracts and policies



(c) Match fails for real policies

- M1: MUTEX_MC ONE_INSTANCE CACHING_SOLVER
M2: MUTEX_SOLVER ONE_INSTANCE CACHING_SOLVER
M3: PRIORITY_MC ONE_INSTANCE CACHING_SOLVER
M4: MUTEX_MC ONE_INSTANCE CACHING_MC
M5: MUTEX_SOLVER ONE_INSTANCE CACHING_MC
M6: PRIORITY_MC ONE_INSTANCE CACHING_MC

(d) Abbreviations for Configurations

Figure 5.2: Cumulative response time of matching algorithm on Desktop PC

and 100 for both contract and policy. The data of the experiment is given on Appendix C. The generated cases cumulative running time of implementation is proportional to the number of problems solved (see Figure 5.2b). In this case the difference among M1 until M8 is negligible as can be seen from Figure 5.2b that the results construct almost a line.

All methods seem to perform equally well because the problems are not stressful enough for the different configurations. This is actually a promising result for the deployment to the resource constrained in mobile device domain. Therefore, we have implemented the same algorithm for the mobile platform HTC P3600 (3G PDA phone). We run the problem suit of P1-P8 and P100 with MUTEX_MC ONE_INSTANCE CACHING_SOLVER configuration.

Table 5.2 shows the results on device, where the runtime of every single problem running is longer than on Desktop PC. This result is obviously due to higher performance of desktop platform. However, the cumulative time of solved problems is still manageable for the mobile user to obtain. The algorithm's runtime will be longer for the problems that match (the algorithm has to run over all states until the cycle is found) than for

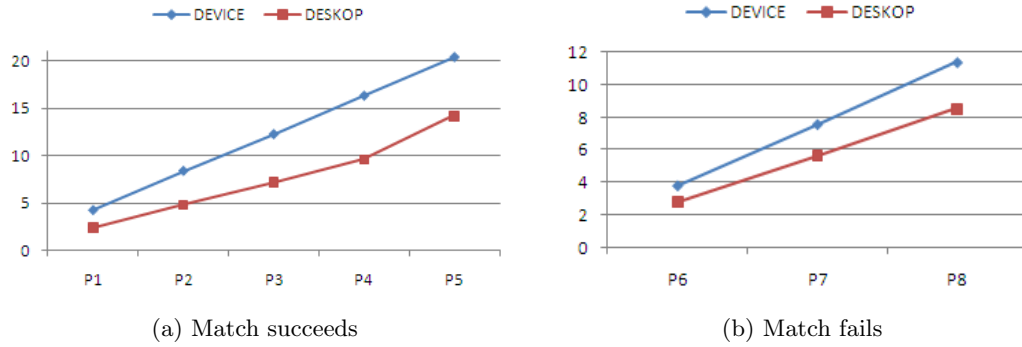


Figure 5.3: Cumulative response time of matching on Device vs on Desktop PC

the problems that do not match (the algorithm stops working as soon as counterexample is found). Note also that the number of visited states and transitions for the matched problems are the same exactly because of the search all over the states; otherwise the counterexample can be found in a different time and it does not depend on the run. Cumulative time of problems is presented in Fig. 5.3a for matching and Figure 5.3b for not matching.

Our current implementation uses `PRIORITY_MC ONE.INSTANCE CACHING_MC` configuration. `PRIORITY_MC` is preferred because of the nature of rules in policies which is *priority or*, also because `MUTEX_SOLVER` does not allow empty methods such as $\neg m_i \wedge \neg m_j$ which is possible in the matching algorithm. `ONE.INSTANCE` is chosen because of garbage collection problem. `CACHING_MC` is desired in order to save calls to solver for the already solved rules.

Chapter 6

Simulation

In this chapter we revisit the same question as in Chapter 4 namely: given expressive security policies, how can we model possibly infinite computations with finite ones? To address this issue we propose Automata Modulo Theory simulation. The key idea is to use fair simulation which is computed using parity game based on small progress measures.

6.1 Introduction

On the previous chapters we have seen on-the-fly matching using language inclusion and this approach requires complementation of the policy of the mobile platform. However, matching using language inclusion as in presented in Chapter 4 has a limitation in the structure of the policy automaton, i.e. only deterministic automaton. The constraint arises from the \mathcal{AMT} complementation, whereas BA complementation, the non-deterministic complementation is complex and exponentially blow-up in the state space [20]. Safra in [70], gives a better lower bound ($2^{O(n \log n)}$) for nondeterministic BA complementation, however it is still exponential (see [83]). This limitation does not evolve in matching using simulation as presented in this chapter, because using simulation approach we can also deal with nondeterministic automata.

The notion of *simulation* in \mathcal{AMT} is both *fair* and *symbolic*. The fairness in \mathcal{AMT} is similar to *fair simulation* in Büchi automata as in [46]. A system fairly simulates another system if and only if in the simulation game, there is a strategy that matches each fair computation of the simulated system with a fair computation of the simulating system. Efficient algorithms for computing a variety of simulation relations on the state space of a Büchi automaton were proposed in [30] using parity game framework, that is based on small progress measures [52]. Another algorithm based on the notion of fair simulation was presented in [40]. The *symbolism* in \mathcal{AMT} is similar to the theory of

symbolic bi-simulation for the π -calculus [45]. This symbolic representation can express the operational semantics of many value-passing processes in terms of finite symbolic transition graphs despite the infinite underlying labeled transitions graph.

This chapter chapter consitutes the theory of simulation in \mathcal{AMT} . We begin in Section 6.2 by introducing the concept of simulation at the concrete level, among valuations i.e. API calls, and the notion of symbolic simulation as in [45]. Then, Section 6.3 describes a decision procedure (and its complexity characterization) for matching the mobile's policy and the midlet's security claims using simulation.

6.2 Simulation in Automaton Modulo Theory

In the sequel we will use s to denote states of the application's contract and t to denote state of the platform's policy.

Definition 6.2.1 (Concrete Fair Compliance Game) *Let A^c and A^p be \mathcal{AMT} with initial states s_0 and t_0 respectively. A Concrete Fair Compliance Game $G_{A^c, A^p}^C(s_0, t_0)$ is played by two players, **Contract** and **Policy**, in rounds.*

1. In the first round **Contract** is on the initial state $s_0 \in S^c$ and **Policy** is on the initial state $t_0 \in S^p$.
2. **Contract** chooses a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ with a valuation ν_i represents α_i and $\mathcal{I}(e_i)$ such that $(\mathcal{M}, \alpha_i) \models e_i^c$ and moves to s_{i+1} .
3. **Policy** responds by a transition $\langle t_i, e_i^p, t_{i+1} \rangle \in \Delta_{\mathcal{T}}^p$ such that $(\mathcal{M}, \alpha_i) \models e_i^p$ and moves to t_{i+1} .

The winner of the game is determined by the following rules:

- If the **Contract** cannot move then **Policy** wins.
- If the **Policy** cannot move then **Contract** wins.
- Otherwise there are two infinite concrete runs $\vec{s} = \langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$ and $\vec{t} = \langle t_0 \nu_1 t_1 \nu_2 t_2 \dots \rangle$ respectively of A^c and A^p . If $\vec{s} = \langle s_0 \nu_1 s_1 \nu_2 s_2 \dots \rangle$ is an accepting concrete run for A^c and $\vec{t} = \langle t_0 \nu_1 t_1 \nu_2 t_2 \dots \rangle$ is not an accepting concrete run for A^p then **Contract** wins. In other cases, **Policy** wins.

Intuitively in the compliance game, the **Contract** tries to make a concrete move and the **Policy** follows accordingly to show that the **Contract** move is allowed. If the **Policy** cannot move then **Contract** is not compliant, meaning there is a move that the **Policy** can not do, that is that particular action is a violation.

Example 6.2.1 *In a game between the Contract from Figure 4.1a and the Policy from Figure 4.1b, the Contract can choose to invoke the url `http://www.google.com` and the Policy can respond by selecting the appropriate expression which is satisfied by that valuation.*

A more complex situation occurs in the infinite case where infinite runs correspond to liveness properties, i.e. something good will eventually happen. An example of this property is shown in Example 2.1.3. In this case, the Contract only wins (i.e. it breaks the Policy) when according to its view of the world there are infinitely many good things but not for the Policy which after some initial good things is trapped in an endless sequence of unsatisfactory states.

Example 6.2.2 *In a game between the Contract and Policy from Ex.2.1.3, the Contract can choose to invoke the url `https://sourceforge.net` in a certain step after in some previous steps it invokes permission `io.Connector.https`. The Policy can respond by selecting the appropriate expression which is also satisfied by the same assignment, which is possible in the game if Policy has requested permission `io.Connector.https` in some previous steps.*

The notion of *concrete strategy* for Policy in game $G_{Ac, Ap}^C(s_0, t_0)$ is a partial function that determines the next move of Policy given the history of the concrete game up to a certain point.

Definition 6.2.2 (Concrete Strategy) *A partial function $f : S^c \times (S^p \times \nu \times S^c)^* \rightarrow S^p$ is a concrete strategy if for any sequence $\langle s_0 \nu_1 s_1 \nu_2 \dots s_i \nu_i s_{i+1} \rangle$ which is a valid concrete run for A^c*

- $f(s_0) = t_0$
- $f(\langle s_0 t_0 \nu_1 s_1 \dots s_i t_i \nu_{i+1} s_{i+1} \rangle) = t_{i+1}$ such that $\langle t_i, e_i^p, t_{i+1} \rangle \in \Delta_T^p$ and $(\mathcal{M}, \alpha_i) \models e_i^p$, where ν_i represents α_i and $\mathcal{I}(e_i)$.

A concrete strategy f of a game is a *Policy winning strategy* if and only if whenever a Policy selects the moves of game as in Definition 6.2.1 according to f then Policy wins.

Definition 6.2.3 (AMT Concrete Fair Simulation Relation) *An automaton A^p concretely fair simulates an automaton A^c if and only if there is a concrete winning strategy for A^p we denote as $A^c \sqsubseteq A^p$. We also say that A^c complies with A^p .*

We have now the machinery to generalize the notion of simulation to symbolic level, among expressions.

Definition 6.2.4 (AMT Fair Compliance Game) A Fair Compliance Game $G_{A^c, A^p}(s_0, t_0)$ is played by two players, **Contract** and **Policy**, in rounds.

1. In the first round **Contract** is on the initial state $s_0 \in S^c$ and **Policy** is on the initial state $t_0 \in S^p$.
2. **Contract** chooses a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_T^c$ such that e_i^c is satisfiable and moves to s_{i+1} .
3. **Policy** responds by a transition $\Delta_T^p(t_i, e_i^p, t_{i+1})$ such that $(e_i^c \rightarrow e_i^p)$ is valid and moves to t_{i+1} .

The winner of the game is determined by the rules as in Definition 6.2.1 with the difference in run where we define run over expressions instead of assignments.

The intuition is similar to concrete game: **Contract** tries to make a symbolic move and the **Policy** follows suit in order to show that the **Contract** move is allowed. If the **Policy** cannot move this means that the **Contract** may not be compliant because there is a symbolic move that the **Policy** could not do. However, as we shall see this might not imply that at the concrete level the **Contract** is really non-compliant.

Definition 6.2.5 (Strategy) A partial function $f : S^c \times (S^p \times \mathcal{E} \times S^c)^* \rightarrow S^p$ is a symbolic strategy if and only if for any sequence $\langle s_0 e_0^c s_1 e_1^c \dots s_i e_i^c s_{i+1} \rangle$ which is a valid symbolic run for A^c

- $f(s_0) = t_0$
- $f(\langle s_0 t_0 e_0^c s_1 t_1 e_1^c \dots s_i t_i e_i^c s_{i+1} \rangle) = t_{i+1}$ such that $\Delta_T^p(t_i, e_i^p, t_{i+1})$ and $(e_i^c \rightarrow e_i^p)$ is valid.

A strategy f of the game is a *Policy winning strategy* if and only if whenever a **Policy** select the moves of game as in Definition 6.2.4 according to f then **Policy** wins.

Definition 6.2.6 (AMT Fair Simulation Relation) An automaton A^p fair simulates an automaton A^c if and only if there is a winning strategy for A^p we denote as $A^c \leq A^p$. We also say that A^c complies with A^p .

Proposition 6.2.1 If $A^c \leq A^p$ is an AMT fair simulation relation then $A^c \sqsubseteq A^p$ is a concrete fair simulation relation.

Proof.

Assume that $A^c \leq A^p$ is an AMT fair simulation relation. By Definition 6.2.6 there is a winning strategy for A^p , such that whenever a **Policy** select the moves of game defined

in Definition 6.2.4 according to strategy f then **Policy** wins the game. We construct a concrete strategy f' from f .

By Definition 6.2.4 there are two cases where **Policy** wins the game:

- Finite game: If the **Contract** cannot move then **Policy** wins.
Contract moves by choosing a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ such that e_i^c is satisfiable.
Contract cannot move means that there exists no valuations and by Definition 6.2.1 in concrete game **Contract** cannot move either.
- Infinite game: There are infinitely many j such that $t_j \in F^p$ or there are only finitely many i such that $s_i \in F^c$.

The compliance game has infinitely many j such that $t_j \in F^p$ when **Policy** is able to respond infinitely often by a transition $\Delta_{\mathcal{T}}^p(t_j, e_j^p, t_{j+1})$ where $(e_j^c \rightarrow e_j^p)$ is valid, meaning for all α_j , $(\mathcal{M}, \alpha_j) \models (e_j^c \rightarrow e_j^p)$. And by Definition 6.2.1 with $(\mathcal{M}, \alpha_j) \models e_j^p$, **Policy** can respond by a transition $\langle t_j, e_j^p, t_{j+1} \rangle \in \Delta_{\mathcal{T}}^p$.

Finitely many i occurs when there is some k such that $\forall i > k, s_i \notin F^c$, meaning **Contract** moves by choosing a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ such that e_i^c is satisfiable, i.e. there exist α_i where $(\mathcal{M}, \alpha_i) \models e_i^c$ and by Definition 6.2.1 **Contract** can also move in concrete game.

It is clear that the constructed concrete strategy f' is a winning strategy for A^p in concrete compliance game, hence by Definition 6.2.3 $A^c \sqsubseteq A^p$. \square

In contrast to the language inclusion approach discussed in Section 4.4, where symbolic language inclusion coincides with concrete language inclusion, and also the simulation notions of [45], the converse of Proposition 6.2.1 does not hold in general.

Proposition 6.2.2 *AMT fair simulation is stronger than AMT language inclusion.*

Proof. The pair of automata in Figure 6.1b and Figure 6.1a is a simple counter example. We can see that both automata coincide with the same concrete automaton as in Figure 6.1c. Thus in concrete level the same automaton having not just simulation but also bi-simulation to itself. However, the symbolic **AMT** on Figure 6.1a cannot simulate the symbolic **AMT** on Figure 6.1b. For example if we have policy represented as Figure 6.1b and contract represented as Figure 6.1a, where both automata accept the same language but according to simulation $VALID(e^2 \rightarrow e^{11})$ does not hold nor $VALID(e^2 \rightarrow e^{12})$, thus we do not have simulation (see abbreviation in Figure 6.1d). \square

In order to show that **AMT** simulation coincides with concrete simulation we must impose some additional syntactic constraints on the automaton.

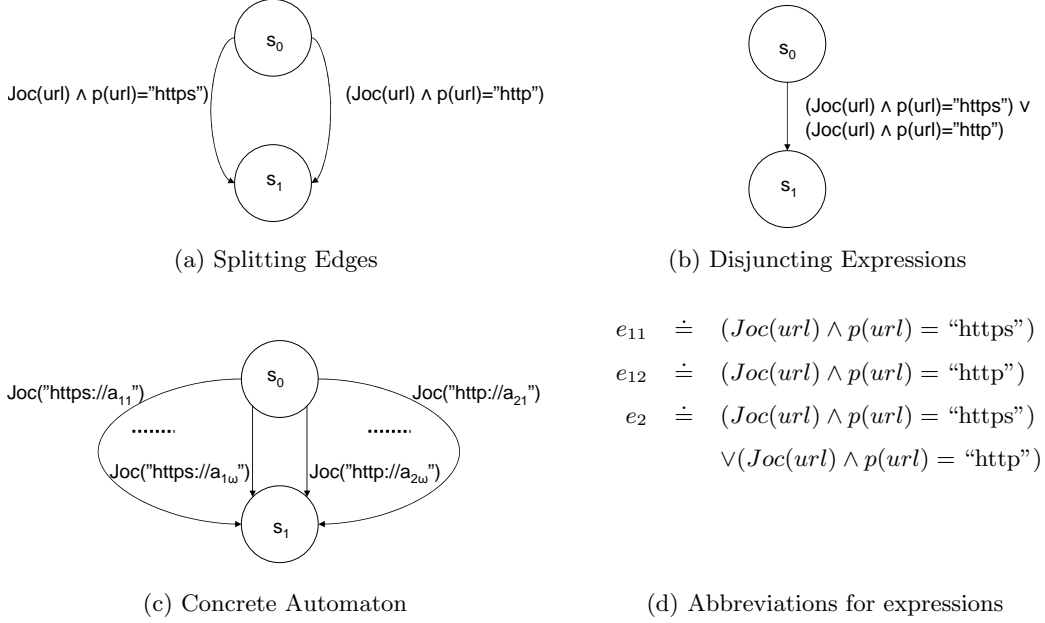


Figure 6.1: Symbolic vs Concrete Automaton

Definition 6.2.7 (Normalized \mathcal{AMT}) $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ is a normalized automaton modulo theory \mathcal{T} if and only if for every $s, s_1 \in S$ there is at most one expression $e_1 \in \mathcal{E}$ such that $s_1 \in \Delta_{\mathcal{T}}(s, e_1)$.

For example Figure 6.1a is a normalized automaton while Figure 6.1b is not normalized.

Lemma 6.2.1 It is possible to normalize an \mathcal{AMT} automaton $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ when theory \mathcal{T} is convex and closed under disjunction.

Proof. A theory \mathcal{T} is convex [73] if all the conjunctions of literals are convex in theory \mathcal{T} . A conjunction of \mathcal{T} -literals in a theory \mathcal{T} is convex if for each disjunction $(\mathcal{M}, \alpha) \models \bigvee_{i=1}^n e_i$ if and only if $(\mathcal{M}, \alpha) \models e_i$ for some i , where e_i are equalities between variables occurring in (\mathcal{M}, α) . If a theory \mathcal{T} is convex then we can normalize an automaton by considering the disjunction of all expressions going to the same state.

A theory \mathcal{T} is called closed under disjunction if disjunctions of \mathcal{T} -formulas $\bigvee_{i=1}^n e_i$, where e_i are \mathcal{T} -formulas, is also a \mathcal{T} -formula. For most theories this closure holds. An example where the closure does not hold is when a \mathcal{T} consists of only Horn-formulas that allows at most one positive literal. Suppose we have two Horn-formulas e_1 and e_2 , where $e_1 \doteq p_1 \wedge p_2 \rightarrow p$ and $e_2 \doteq q_1 \wedge q_2 \rightarrow q$, then $e_1 \vee e_2 \doteq p_1 \wedge p_2 \wedge q_1 \wedge q_2 \rightarrow p \vee q$ which is not a Horn-formula. \square

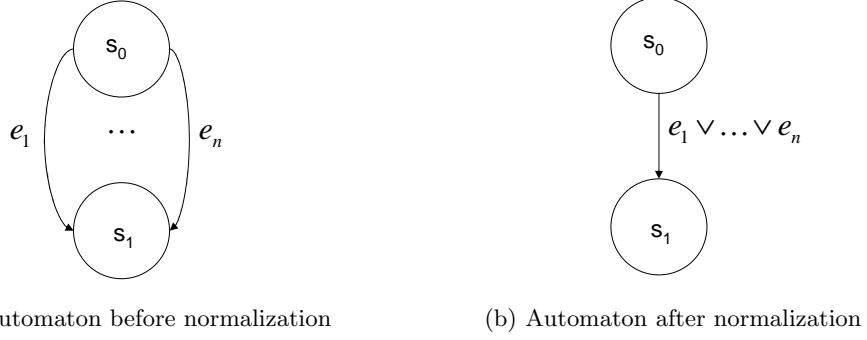


Figure 6.2: Normalization of an automaton

Lemma 6.2.2 *Normalization preserves the determinism of an \mathcal{AMT} .*

Proof. By Definition 4.3.4 $A = \langle S, \Sigma, \mathcal{T}, \mathcal{E}, \Delta, \mathbf{s}_0, F \rangle$ is a *deterministic automaton* modulo theory \mathcal{T} , if and only if, for every $s \in S$ and every $s_1, s_2 \in S$ and every $e_1, e_2 \in \mathcal{E}$, if $(s, e_1, s_1) \in \Delta$ and $(s, e_2, s_2) \in \Delta$, where $s_1 \neq s_2$ then the expression $(e_1 \wedge e_2)$ is unsatisfiable in the Σ -theory \mathcal{T} .

Let $(s, e_{1j}, s_1) \in \Delta$ where $j \in \{1, \dots, m\}$, and let $(s, e_{2k}, s_2) \in \Delta$ where $k \in \{1, \dots, n\}$, and $s_1 \neq s_2$. Thus, each expression $(e_{1j} \wedge e_{2k})$ is unsatisfiable in the Σ -theory \mathcal{T} . By normalization we have $(\bigvee_{j=1}^m e_{1j})$ and $(\bigvee_{k=1}^n e_{2k})$, where $(\bigvee_{j=1}^m e_{1j}) \wedge (\bigvee_{k=1}^n e_{2k}) \Leftrightarrow \forall j \in \{1, \dots, m\}, \forall k \in \{1, \dots, n\}, (e_{1j} \wedge e_{2k})$. If each expression $(e_{1j} \wedge e_{2k})$ is unsatisfiable then $(\bigvee_{j=1}^m e_{1j}) \wedge (\bigvee_{k=1}^n e_{2k})$ is also unsatisfiable when the theory \mathcal{T} is convex. Thus, normalization preserves the determinism of an \mathcal{AMT} . \square

Proposition 6.2.3 *For normalized \mathcal{AMT} if $A^c \sqsubseteq A^p$ is a concrete fair simulation relation then $A^c \leq A^p$ is an \mathcal{AMT} fair simulation relation.*

Proof.

Assume that $A^c \sqsubseteq A^p$ is a concrete fair simulation relation. By Definition 6.2.3 there is a winning strategy for A^p , such that whenever a **Policy** select the moves of game defined in Definition 6.2.1 according to strategy f then **Policy** wins the game. We construct a concrete strategy f' from f .

By Definition 6.2.1 there are two cases where **Policy** wins the game:

- **Finite game:** If the **Contract** cannot move then **Policy** wins.
Contract moves by choosing a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ with a valuation ν_i represents α_i and $\mathcal{I}(e_i)$ such that $(\mathcal{M}, \alpha_i) \models e_i^c$, meaning e_i^c is satisfiable. **Contract** cannot move means that there exists no valuations and by Definition 6.2.4 in compliance game **Contract** cannot move either.

- Infinite game: There are infinitely many j such that $t_j \in F^p$ or there are only finitely many i such that $s_i \in F^c$.

The concrete compliance game has infinitely many j such that $t_j \in F^p$ when **Policy** is able to respond infinitely often by a transition $\Delta_{\mathcal{T}}^p(t_j, e_j^p, t_{j+1})$ where for all valuations ν_j represents α_j and $\mathcal{I}(e_j)$ such that $(\mathcal{M}, \alpha_j) \models (e_j^c \rightarrow e_j^p)$, meaning $(e_j^c \rightarrow e_j^p)$ is valid. And by Definition 6.2.4 **Policy** can respond by a transition $\langle t_j, e_j^p, t_{j+1} \rangle \in \Delta_{\mathcal{T}}^p$ with a valuation ν_j represents α_j and $\mathcal{I}(e_j)$ such that $(\mathcal{M}, \alpha_j) \models e_j^p$.

Finitely many i occurs when there is some k such that $\forall i > k, s_i \notin F^c$, meaning **Contract** moves by choosing a transition $\langle s_i, e_i^c, s_{i+1} \rangle \in \Delta_{\mathcal{T}}^c$ with a valuation ν_i represents α_i and $\mathcal{I}(e_i)$ such that $(\mathcal{M}, \alpha_i) \models e_i^c$ and by Definition 6.2.4 **Contract** can also move in concrete game.

It is clear that the constructed strategy f' is a winning strategy for A^p in compliance game, hence by Definition 6.2.6 $A^c \leq A^p$. \square

If automata are in normalized form then we have the following theorem from [59]:

Theorem 6.2.1 *For normalized AMT $A^c \leq A^p$ is an AMT fair simulation if and only if $A^c \sqsubseteq A^p$ is a concrete fair simulation.*

Proof.

“ \supseteq ” By Proposition 6.2.1.

“ \subseteq ” If a normalization that preserves automata determinism (Lemma 6.2.2) is possible (Lemma 6.2.1), then By Proposition 6.2.3. \square

6.3 Simulation Matching

In this section we describe a different algorithm for matching from Section 4.5 that uses the concepts of language inclusion. Here we use fair simulation for matching and adapts the Jurziński’s algorithm on parity games [52]. The simulation algorithm Algorithm 5 takes as input two automata A^C and A^P representing respectively the formal specification of a contract and of a policy. A match is obtained when every security-relevant action invoked by A^C can also be invoked by A^P . In other words, every behavior of A^C is also a behavior of A^P .

At the first step (line 1) a compliance game graph $G = \langle V_1, V_0, E, l \rangle$ is constructed out of automata A^C and A^P . A compliance game graph can be formally defined as follows:

Definition 6.3.1 (Compliance Graph) *Given $\langle S^c, \Sigma^c, \mathcal{T}^c, \mathcal{E}^c, \Delta^c, \mathbf{s}_0^c, F^c \rangle$ and $\langle S^p, \Sigma^p, \mathcal{T}^p, \mathcal{E}^p, \Delta^p, \mathbf{s}_0^p, F^p \rangle$, construct a $\langle V_1, V_0, E, l \rangle$ as follows:*

- $V_1 = \{v_{(s^c, s^p)} \mid s^c \in S^c, s^p \in S^p\}$

Algorithm 5 Simulation Algorithm

Input: two \mathcal{AMT} automata A^C and A^P

- 1: Construct compliance game graph $G = \langle V_1, V_0, E, l \rangle$
 - 2: **for all** $v \in V$ **do**
 - 3: $\mu(v) := \mu_{\text{new}}(v) := 0$
 - 4: **repeat**
 - 5: $\mu := \mu_{\text{new}}$
 - 6: **for all** $v \in V_0$ **do**
 - 7: $\mu_{\text{new}}(v) := \begin{cases} \infty & \text{if } \{\mu(w)|(v, w)\} = \emptyset \\ \min \{\mu(w)|(v, w)\} & \text{otherwise} \end{cases}$
 - 8: **for all** $v \in V_1$ **do**
 - 9: $max_v := \max \{\mu(w)|(v, w) \in E\}$
 - 10: $\mu_{\text{new}}(v) := \begin{cases} \infty & \text{if } max_v = \infty \\ 0 & \text{if } l(v) = 0 \\ max_v + 1 & \text{if } l(v) = 1 \\ max_v & \text{if } l(v) = 2 \end{cases}$
 - 11: **until** $\mu = \mu_{\text{new}}$
 - 12: **if** $\mu(v_{(s_0^c, s_0^p)}) < \infty$ **then**
 - 13: Simulation exists
-

- $V_0 = \{v_{(s^c, s^p, e^c)} \mid s^c \in S^c, s^p \in S^p, \exists r^c. (r^c, e^c, s^c) \in \Delta^c\}$
- $E = \{(v_{(s^c, s^p, e^c)}, v_{(s^c, t^p)}) \mid (s^p, e^p, t^p) \in \Delta^p \wedge \text{VALID}(e^c \rightarrow e^p)\} \cup \{(v_{(s^c, s^p)}, v_{(t^c, s^p, e^c)}) \mid (s^c, e^c, t^c) \in \Delta^c\}$

$$l(v) = \begin{cases} 0 & \text{if } v = v_{(s^c, s^p)} \text{ and } s^p \in F^p \\ 1 & \text{if } v = v_{(s^c, s^p)} \text{ and } s^c \in F^c \text{ and } s^p \notin F^p \\ 2 & \text{otherwise} \end{cases}$$

A compliance graph G is the tuple $\langle V_1, V_0, E, l \rangle$

Intuitively the compliance level $l(v)$ is 0 when the simulating automaton accepts, 1 when the simulated automaton accepts (but the simulating automaton has not accepted yet) and 2 when neither of them accepts. V_1 consists of $v_{(s^c, s^p)}$ where A^C is on s^c and A^P is on s^p and it is **Contract** turn to move. V_0 consists of $v_{(s^c, s^p, e^c)}$ where A^C is on s^c and A^P is on s^p , **Contract** just made a move e^c and it is **Policy** turn to move such that $\text{VALID}(e^c \rightarrow e^p)$ by querying to an oracle for the SMT solver.

Lemma 6.3.1 *Let $A^C = \langle S^C, \Sigma^C, \mathcal{T}^C, \mathcal{E}^C, \Delta^C, \mathbf{s}_0^C, F^C \rangle$ and $A^P = \langle S^P, \Sigma^P, \mathcal{T}^P, \mathcal{E}^P, \Delta^P, \mathbf{s}_0^P, F^P \rangle$ be **AMT** automata and let the theory $\mathcal{T} = \mathcal{T}^C \cup \mathcal{T}^P$ be decidable with an oracle for the SMT problem in the complexity class \mathcal{C}*

1. $|G = \langle V_1, V_0, E, l \rangle|$ constructed out of automata A^C and A^P by Definition 6.3.1 is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)^{\mathcal{C}}$
2. $|l^{-1}(1)|$ defined as in Definition 6.3.1 is in $O(|S^c| \cdot |S^p|)$

Proof. We prove part 1 by computing the vertices and edges of $\langle V_1, V_0, E, l \rangle$

- $|V_1|$ is in $O(|S^c| \cdot |S^p|)$
- $|V_0|$ is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)$
- $|E|$ is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)^{\mathcal{C}}$ because an edge exists from a node in V_0 to a node in V_1 when $\text{VALID}(e^c \rightarrow e^p)$ that needs a call to oracle for the SMT solver.

Thus, we can conclude that $|G = \langle V_1, V_0, E, l \rangle|$ is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)^{\mathcal{C}}$

For part 2 vertices with $l = (1)$ are contained in V_1 , thus $|l^{-1}(1)|$ is in $O(|S^c| \cdot |S^p|)$ \square

A compliance game $P(G, v_0)$ on G starting at $v_0 \in V$ is played by two players **Policy** (for A^P) and **Contract** (for A^C). The game starts by placing pebble on v_0 . At round i with pebble on v_i , $v_i \in V_0(V_1)$, **Policy** (**Contract** resp.) plays and moves the pebble to v_{i+1} such that $(v_i, v_{i+1}) \in E$. The player who cannot move loses. For infinite play

$\pi = v_0v_1v_2\dots$, the winner defined as the minimum compliance level that occurs infinitely often, namely if the minimum compliance level is 0 or 2 then **Policy** wins, otherwise **Contract** wins.

Next, we define a *compliance measure* $\mu : V \rightarrow \{x | x \leq |l^{-1}(1)|\} \cup \{\infty\}$. μ ranges from 0 to $|l^{-1}(1)|$ because at $l(v)=1$ the simulated automaton (contract) accepts but the simulating automaton (policy) has not accepted yet. Thus, progressing the measure has the analogy of computing the pre-fixed point where the **Contract** remains winning and ∞ shows that the μ keeps progressing beyond this limit, meaning **Contract** wins the game. If $l(v) = 1$, then $\mu(v) > \mu(w)$, where $|l^{-1}(1)| + 1 = \infty$. If $l(v) = 2$ or $l(v) = 0$, then $\mu(v) \geq \mu(w)$.

The compliance measure for each node is the number of potential bad nodes, namely nodes where the contract accepts but the policy does not, that it can reach. Thus, $\mu(v) = \infty$ means that there is an infinite path where policy cannot return to compliance level 0. We slightly modify the Jurdziński progress measure [52] to *compliance measure* where instead of a pair $(0, x)$ we only use x . This is due to our observation of our domain where we only have three priorities, namely $l(v) \in 0, 1, 2$ thus for ordering $(0, x) \geq_{l(v)} (0, x')$ the first component will not effect the ordering.

Jurdziński's algorithm on parity games [52] defines that **Policy** has a winning strategy from precisely the vertices v where after its lifting algorithm halts has $\mu(v) < \infty$. However, in contract-policy matching we are interested when there is a winning strategy from the initial vertex $v_{(s_0^c, s_0^p)}$, depicted in Algorithm 5 as $\mu(v_{(s_0^c, s_0^p)}) < \infty$.

Proposition 6.3.1 *Let G be a parity game constructed from two \mathcal{AMT} automata A^C and A^P constructed as in Definition 6.3.1. **Policy** has a winning strategy from the initial vertex $v_{(s_0^c, s_0^p)}$ when Algorithm 5 halts with $\mu(v_{(s_0^c, s_0^p)}) < \infty$.*

Proof. Correctness.

The correctness derived from Jurdziński's algorithm on parity games [52]. Jurdziński defined a parity game between two players where an even player (in our case **Policy**) wins when the lowest priority occurring infinitely often in the play is even (in our case **Policy** can return to compliance level 0 infinitely often). He proposed computing the game using progress measure which is defined as $M_G = [1] \times [n_1 + 1] [1] \times [n_3 + 1] \times \dots \times [1] \times [n_{d-1} + 1]$, where d is the maximum priority in the game. In our setting, we slightly modify the Jurdziński progress measure [52] to *compliance measure* where instead of a pair $(0, x)$ we only use x . As we have mentioned afore, this is due to our observation of our domain where we only have 3 priorities, namely $l(v) \in 0, 1, 2$ thus for ordering $(0, x) \geq_{l(v)} (0, x')$ the first component will not effect the ordering.

Jurdziński reasoned that each vertex can only be lifted $|M_G|$ times. This lifting pro-

cedure is implemented in Algorithm 5 presented as a loop where compliance measure progressing until reaching a pre-fixed point ($\mu = \mu_{\text{new}}$). He also defined that Even has a winning strategy from precisely the vertices v where after its lifting algorithm halts has $\mu(v) < \infty$. However, in contract-policy matching we are interested when there is a winning strategy from the initial vertex $v_{(s_0^c, s_0^p)}$. Thus, in Algorithm 5 Policy wins when $\mu(v_{(s_0^c, s_0^p)}) < \infty$.

Termination. This parity game terminates because each vertex can only be lifted $|M_G|$ times.

Complexity. Lifting procedure in Jurdziński [52] has time complexity $O\left(\sum_{v \in V} d \cdot od(v) \cdot |M_G|\right) = O(d \cdot m \cdot |M_G|)$ where d is the maximum priority in the game, m the number of edges, $od(v)$ the degree outgoing edges from v , and V is the set of vertices in the game graph. He reasoned that for every vertex v with outgoing edges degree $od(v)$ and the tuple of progress measure has the length of maximum priority d can only be lifted $|M_G|$ times:

$$|M_G| = \prod_{i=1}^{\lfloor d/2 \rfloor} (n_{2i-1} + 1) \leq \left(\frac{n}{\lfloor d/2 \rfloor}\right)^{\lfloor d/2 \rfloor}, \text{ where } d \text{ is the maximum priority in the game.}$$

In our setting, d equals to two, because our compliance measure is in $\{0, 1, 2\}$. Thus, $|M_G| = [n_1 + 1] = |l^{-1}(1)| + 1 \leq |V_1|$ and from Lemma 6.3.1 $|V_1| = O(|S^c| \cdot |S^p|)$. In addition, the number of edges $|E|$ is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)^c$ (from Lemma 6.3.1). Thus, the time complexity of Algorithm 5 is $O(2 \cdot |E| \cdot |M_G|)$

Lifting procedure in Jurdziński [52] has space complexity $O(dn)$ where d is the maximum priority in the game and n the number of vertices in the game graph. He reasoned that every vertex v in the game graph only needs to keep the compliance measure, which is a d -tuple of integers. In our setting, d equals to two because our compliance measure is in $\{0, 1, 2\}$, however our compliance measure only use an integer x instead of a 2-tuple $(0, x)$. As we have mentioned afore, this is due to our observation of our domain where we only have 3 priorities, namely $l(v) \in 0, 1, 2$ thus for ordering $(0, x) \geq_{l(v)} (0, x')$ the first component will not effect the ordering. In addition, from Lemma 6.3.1 $|V_1| = O(|S^c| \cdot |S^p|)$ and $|V_0|$ is in $O(|S^c| \cdot |S^p| \cdot |\Delta_{\mathcal{T}}^c|)$ where the total number of vertices equals to $V = |V_1| + |V_0|$. Thus, the space complexity of Algorithm 5 is $O(|V|)$. \square

We are now in the position to state our contract-policy matching's result using fair simulation:

Proposition 6.3.2 *Let the theory \mathcal{T} be decidable with an oracle for the SMT problem in the complexity class \mathcal{C} then:*

1. *The contract-policy matching problem for \mathcal{AMT} using fair simulation is decidable in time $O(2 \cdot |E| \cdot |M_G|)$.*
2. *The contract-policy matching problem for \mathcal{AMT} using fair simulation is decidable in space $O(|V|)$.*

Proof. The matching between a contract with a security policy problem can be reduced to a fair simulation between a contract with a security policy. A combination of an algorithm based on Jurdziński's algorithm on parity games [52] with a decision procedure for SMT given in Algorithm 5 can solve this problem in time $O(2 \cdot |E| \cdot |M_G|)$ and in space $O(|V|)$. The algorithm takes as input the midlet's claim and the mobile platform's policy and constructs compliance game graph $G = \langle V_1, V_0, E, l \rangle$. The correctness and complexity follow from Proposition 6.3.1. \square

Chapter 7

Simulation Matching Prototype Implementation and Experiments

In this chapter, we try to provide an answer to the following question: how can we implement matching using simulation and what is the best configuration of integrating automata-based inclusion algorithm with decision procedure? To address this issue we give possible design decisions and run experiment on desktop as in Chapter 5. We continue with detailing the running-time on the mobile platform for one design decision only to give the reader a feeling how the matching algorithm with integrated decision procedure can run in real application.

7.1 Introduction

This chapter describes the prototype implementation of contract-policy matching in *AMT* using simulation, its integration with decision solver based on MathSAT and NuSMV, and the results of our experiments on matching.

We begin in Section 7.2 by discussing the overall implementation architecture and the integration issues with the procedure solver NuSMV [22] integrated with its MathSAT libraries [18].

Section 7.3 continues with implementation of simulation as parity game with oracle calls to the decision procedures available in NuSMV. Our prototype was implemented in .NET and was run on a Desktop PC with operating system Microsoft Windows XP Professional Version 2002 Service Pack 3.

Finally, Section 7.4 presents a detailed performance analysis of the integration design alternatives regarding the construction of expressions, the initialization of solver, and the

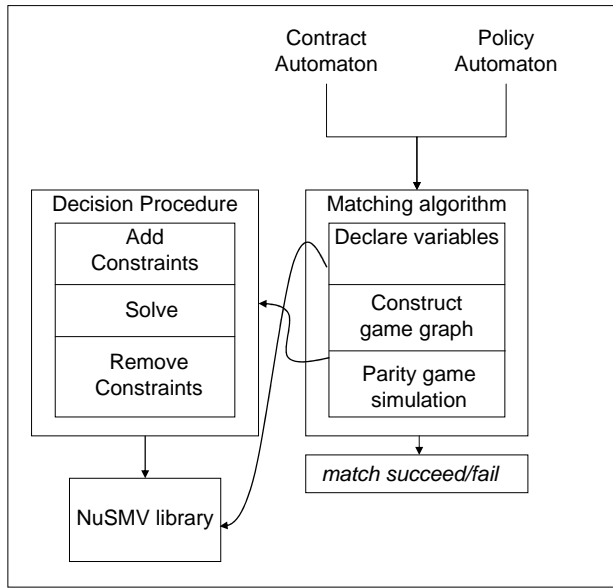


Figure 7.1: Simulation Implementation Architecture

caching of temporary results by considering running time of various available options.

7.2 The Architecture

In this section we describe the conceptual architecture of the prototype that implements the overall matching algorithm and supports integration with state of the art decision procedure solver NuSMV [22] integrated with its MathSAT libraries [18]. The main aim is to provide a concrete overview of how the prototype is implemented so that one can easily understand the possible options for integration with the solver. The contract-matching prototype takes as input a contract and a policy both specified in ConSpec and checks whether or not the contract matches the policy. The source code itself is thoroughly documented and should therefore be easy to understand. In addition, the following class diagram should provide the reader with a good overview over the Simulation Algorithm namespace and its classes as shown in Figure 7.1. Detailed class diagram is available on Appendix B.

The prototype had been implemented as a Desktop version by extending the prototype from Chapter 5. The prototype consists of only one part which is off-device implementations. At the first step of matching, a compliance game graph $G = \langle V_1, V_0, E, l \rangle$ is constructed out of automata A^C and A^P . The main parity game algorithm runs on the constructed game graph and makes calls to the decision procedure during its execution. The different step from the on-the-fly implementation is that the policy automata need not be complemented. The rest of integration issues with decision solver based on MathSAT and NuSMV follows from on-the-fly matching implementation, for example we use the solver as a black box (an oracle) for the general algorithm that gives the answer whether

the problem is satisfiable or not.

7.3 Design Decisions

As in on-the-fly matching implementation, different design decisions are made in order to *decide the best configuration of integrating automata-based inclusion algorithm with decision procedure* as the problem is not trivial. Every option of the configuration proposed below has different memory impact and this information and results of such analysis is very important because of the resource constraints of mobile device. In integrating matching algorithm with the theory solver we faced a number of design options:

One_vs.Many Solver in object oriented languages is by itself an object. We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure.

ALL_INSTANCES The expression sent to the solver has the following structure: $method \wedge otherMethods \wedge cond \wedge otherConds$.

CACHING_MC Since many edges will be traversed again and again we could save time by caching the results of the matching. The solver itself has a caching mechanism that could be equally used (**CACHING_SOLVER**).

Unlike in on-the-fly matching implementation, we do not have **MUTEX_SOLVER**, **MUTEX_MC**, and **PRIORITY_MC** options instead we introduce **ALL_INSTANCES** which is suitable for representation of only policy automaton and not the complementation of policy automaton.

As in on-the-fly matching implementation, the **One_vs.Many** option was not possible which requires only one instance of solver exists at time in order to interact correctly with the NuSMV library. This leads to use a static invocation for the solver and set significant constraints on the interaction. For example, before starting to visit all constraints to the library, all variables used in expressions must be declared. The NuSMV library has to invoke *DeclareNewBooleanVar*, *DeclareNewWordVar*, *DeclareNewStringVar* methods for declaration of boolean, integer and string variables respectively. Only after declaring all the variables from contract and policy expressions, the simulation algorithm can actually start invoking the decision procedure in its visit. A consequence of this rule is that with this implementation we cannot insert edges that introduce new variables because the solver can be called only after declaring all the variables and adding all the needed constraints. Therefore, during the visit of the algorithm we must at first upload constraints to the

Table 7.1: Running Problem Suit 10 Times

ALL_INSTANCES ONE_INSTANCE CACHING_MC			
Problem	ART (s)	CRT (s)	Result
P1	2.014	2.014	Match
P2	1.934	3.948	Match
P3	1.886	5.834	Match
P4	1.886	7.72	Match
P6	1.998	1.998	Not Match
P7	2.06	4.058	Not Match
P8	1.998	6.056	Not Match
P100	5.528	5.528	Match

solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*.

Therefore, during the visit of the algorithm we must at first upload constraints to the solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*.

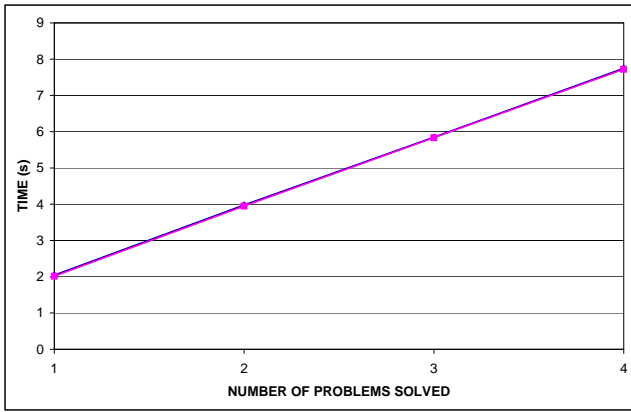
7.4 Experiments on Desktop

To understand the best option we collected data on running time for each problem in each design alternative and the number of solved problems against time. From (Section 7.3) the design alternatives can be implemented and tested in two alternative configurations and we use the same problem suit as in Table 5.1 for possible combinations of policy-contract (mis)matching pairs.

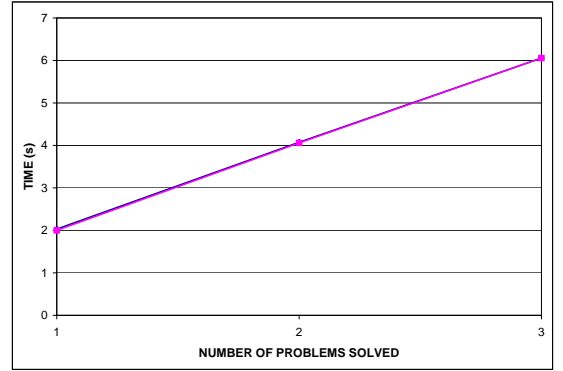
We run our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Microsoft Windows XP Professional Version 2002 Service Pack 3. The result is shown in Table 7.1.

For the sake of example we present the result obtained for alternative with ALL_INSTANCES ONE_INSTANCE CACHING_MC in Table 7.1. The results for all design alternatives are mapped into diagram shown in Figure 7.2a for matching problems and Figure 7.2b for not matching problems. Notice that we only provide the cumulative running time that is necessary to solve all problems as for on-the-fly implementation experiments. This is important because our goal is to match (or not match) all rules in a contract with all corresponding rules in a policy. Thus, the value of the single problem is not important except for some cases where the average output might be significantly off due to some off scale rule.

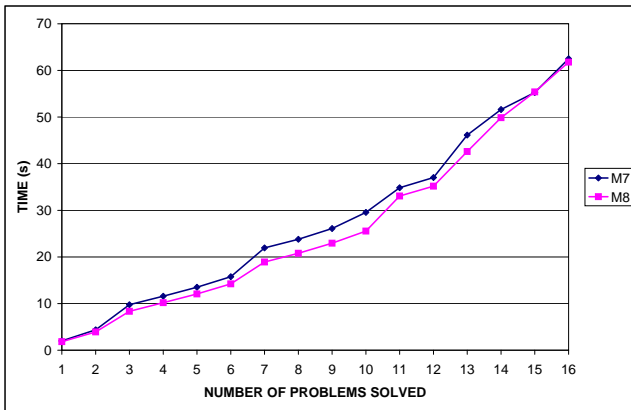
We singled out P100 as a challenging artificial problem because it has a large number of states compared to the others: essentially this happened because we draw an automaton



(a) Match succeeds for real policies



(b) Match fails for real policies



(c) Matches among synthetic contracts and policies

M7: ALL_INSTANCES ONE_INSTANCE CACHING_SOLVER
M8: ALL_INSTANCES ONE_INSTANCE CACHING_MC

(d) Abbreviations for Configurations

Figure 7.2: Cumulative response time of matching algorithm on Desktop PC

modulo theory with 100 states and which traverse from one state to another by adding 1 to the number of SMS sent.

In this case there is a difference between M7 and M8, namely 5.387 s and 4.434 s resp., that is M8 is better around 21.5% than M7. In order to study this anomaly in more details, we generated more unreal problem sets: as P100 with combination of sent SMS none, 1, 10, and 100 for both contract and policy. The data of the experiment is given on Appendix D. The generated cases cumulative running time of implementation is proportional to the number of problems solved (see Figure 7.2c). In this case the difference between M7 and M8 is only around 9.8% still with M8 better than M7. This result conforms to our intuition because M8 uses fewer calls to solver due to its caching and thus save computations.

All methods seem to perform equally well because the problems are not stressful enough for the different configurations. This is actually a promising result for the deployment to

the resource constrained in mobile device domain. However, we have not yet implemented the same algorithm for a mobile platform.

In this chapter, we have given possible design decisions and run experiment on PC for \mathcal{AMT} simulation. Furthermore, we have detailed the time of the running on the mobile platform for one design decision only to give the reader a feeling how the matching algorithm with integrated decision procedure can run in real life and that it will take a reasonable time. Our current implementation uses `ALL.INSTANCE` `CACHING.MC` configuration. `ALL.INSTANCE` is preferred because of the nature of rules in policies when an automaton is not complemented. `ONE.INSTANCE` is chosen because of garbage collection problem. `CACHING.MC` is desired in order to save calls to solver for the already solved rules.

Chapter 8

IRM Optimization

In this chapter, we try to provide an answer to the following question: given an untrusted code and a policy that a platform specifies to be inlined, how can we obtain an optimized Inlined Reference Monitor ? To address this issue, we propose six different framework models for optimization with respect to components that are needed to be trusted or untrusted. We also describe an approach for optimization based on automata theory. The key idea is that given a policy that represent the desired security behavior of a platform to be inlined, we compute an optimized policy with respect to the claims on the security behavior of a application that we inject to the untrusted code.

8.1 Introduction

AMT is a general model, thus it can be used not only for matching security policies but also in other enforcement mechanism for example Inlined Reference Monitor (IRM). IRM is a flexible mechanism to enforce the security of untrusted applications. Even if current version of IRM can work on rich system such as today's smart phones, the overhead is still too much for the next frontier of web applications: Java cards. Indeed, the smart card technology [61] evolved with larger memories, USB and TCP/IP support and the development of the Next-Generation (for short NG) Java Card platform [3, 4] with Servlet engine. This latter technology is a full fledged Java platform for embedded Web applications and opens new Web 2.0 opportunities such as NG Java Card Web 2.0 Applications. It can also serve as an alternative to personalized applications on remote servers so that personal data no longer needs to be transmitted to remote third-parties.

Thus, optimizing redundant monitoring without compromising security is needed. The key idea is that given a *policy* that represent the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the claims on the security

behavior of a application (for short *contract*). Then, we use this *optimized policy* to inject the untrusted code. In the first work [86] proposed IRM optimization for a constrained history-based access control policy such as Chinese Wall policies using compiler optimization approach. Unfortunately, this approach is severely limited by the expressivity of the language: it only consider propositional conditions on policies. As a result even a simple policy such as "Only allows connections to urls starting with http" cannot be optimized. An earlier work [50, 29] suggested to apply static program analysis as in compiler optimization to tame the overhead of code instrumentation.

Three issues arise from the problem of IRM optimization. One issue regards the questions of "*How can we formalize the notion of optimization ?*". Our work attempts to give a preliminary formalization using the concept of *AMT*. The second question is "*What is the optimal policy to be enforced with respect to the claimed applications policy?*". In this issue, we are not interested in finding a unique minimum policy, given only the policy itself, instead we are interested in finding the optimal policy guided by claimed applications policy. The last issue is "*Is this optimal policy computable with an efficient algorithm ?*"

This chapter attempts to answer the afore mentioned questions specifically by using *AMT* formalization. We begin in Section 8.2 by identifying the different trust models for IRM optimization, i.e. the relative position of the optimizer and the inliner with respect to the trust border. We continue by optimization algorithm in Section 8.3 using simulation from Chapter 6 as the basic block.

8.2 Security Models for Optimized IRM

In this section, we introduce our IRM trust models. Figure 2.1 from Chapter 2 illustrates our general optimization workflow model. As we have already mentioned, this model is a modification of original S×C workflow of [13]) by adding optimization step. First, a code is analyzed in order to extract contract out of it. This can be done by trusted or untrusted parties. If done by untrusted parties, then the claimed contract needs to be verified whether it complies to the code. If it complies, then we simulate the contract with the policy to verify if the policy is already enforced by the contract. On failure of simulation, we optimize policy by discharging behaviors which are already enforced by a contract and we inject this optimized policy to the code. The overall model consists of the following components:

ContractExtractor and ClaimChecker The former extract policies from code based on control flow graphs and possibly annotation existing on the code [38]. The latter is the basic component of Proof-Carrying Code [63], where the untrusted code supplier

must provide with the code a safety proof that attests to the code’s safety properties. In mobile system domain [44] implements a linear decision algorithm verifying that annotated .NET binaries satisfy a class of policies using security and edit automata.

SimulationChecker uses fair simulation for *AMT* [59]. This key idea is based on symbolic simulation [30, 52]. A system fairly simulates another system if and only if in the simulation game, there is a strategy that matches with each fair computation of the simulated system a fair computation of the simulating system. We can use this techniques to decide if the update is acceptable by different notion of simulation.

Rewriter We use rewriter instead of inliner because it is not necessary to actually inline the entire security automaton. Some example of works on rewriter are Naccio [31], PoET/Pslang [28], and Polymer [56]. These approaches compile policy language into plain Java and then into Java bytecode monitor which is injected into ordinary Java bytecode by inserting calls in all the necessary places. Other rewriter uses reflection [85] where policies are implemented as meta-objects bounded to application objects at load time through bytecode rewriting. This approach is implemented using Kava which provides a non-bypassable meta level. An alternative approach to rewriter is an inliner, for example [26] that only inlines hooks to the monitor with the monitor itself runs in a separate thread.

Optimizer can be performed by compiler optimization approach as in [86] or by our approach described in (§8.3).

The IRM approach is facilitated by the trend toward using higher-level languages, especially type safe languages, for software development. Not only do those languages define application abstractions on which policies can be enforced, but they also provide strong guarantees that can be used to ensure a secured application cannot compromise its IRM. By leveraging these guarantees, an IRM security policy can provide a single cohesive description of both the intent and the means by which a policy is enforced. This potentially allows the IRM approach to give greater assurance, since enforcement now relies on a trustworthy component of moderate size whose full specification can be studied in isolation.

The main consideration for our models is the trade off between moving more processes out of trusted part and the complexity of the whole process (inspired by model in [42]).

8.2.1 Rewriter on Trusted part

Model 1. In the simplest model (Figure 8.1a), the untrusted part consists of only **Code**. First, the application’s contract (**Contract**) is extracted by **CONTRACTEXTRACTOR** on

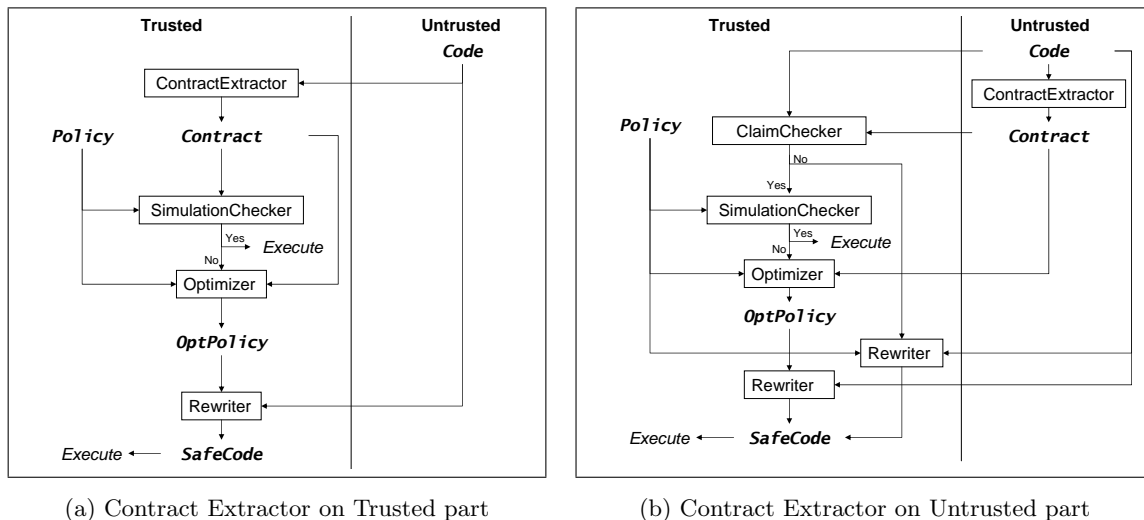


Figure 8.1: Rewriter on Trusted part

the trusted part. Then, a compliance of **Contract** to **Policy** is checked by **SIMULATIONCHECKER**. If the simulation succeed, then **Code** can be executed without further ado. Otherwise, an an **OPTIMIZER** is used to optimize **Policy** which gives result to **OptPolicy**. Finally, the **OptPolicy** is injected by **REWRITER** which gives result an **SafeCode** that is ready to be executed.

CONTRACTEXTRACTOR extracts *security relevant* behaviors. Depending on **Contract** representation, this extraction can be data flow analysis [6], control flow analysis [65], abstract interpretation [24], model extraction [74], or contract extraction as in **S×C**[27, 13]. The feasibility depends on the available resources and environment. For example, in mobile system when downloading an untrusted application. The memory is limited, i.e. it is not desirable to not be able receiving calls while downloading an app. The time is also limited, because usually human expects a response in two second for asking a system to do a certain work [1, 2], meaning the whole work-flow in Figure 8.1a. And contract extraction is only a fraction of it. Thus, mechanisms as model extraction or contract extraction in **S×C** is suitable for this domain. However, for system with sufficient resources, for example off-line system testing before certification that allows hours of verification time, then data flow analysis, control flow analysis, or abstract interpretation can be applied with higher degree of confidence.

In this model, we restrict compliance check of **Contract** to **Policy** by “simulation”. However, again it depends on **Contract** and **Policy** representation on how the “simulation” is defined. In case both represented in **AMT**, then “simulation” is defined as fair simulation and Algorithm 5 can be applied.

The same reasoning applies to **OPTIMIZER**. This process also depends on how **Contract**

and `Policy` represented for example “optimization” can be performed by compiler optimization approach as in [86]. In case both represented in \mathcal{AMT} , then “optimization” is defined in (§8.3).

The `REWRITER` process also depends on how `OptPolicy` represented. Some example of works on rewriter are Naccio [31], PoET/Pslang [28], and Polymer [56], [85]. In case `OptPolicy` represented in \mathcal{AMT} , then hook-inliner approach as in [26], that only inlines hooks to the monitor with the monitor itself runs in a separate thread, can be applied.

Model 2. In the second model (Figure 8.1b), `CONTRACTEXTRACTOR` is positioned on the untrusted part. The modification from Model 1 lies on an extra step where `Contract` must be verified against `Code`. The cost of this extra step depends on the trade off between extracting `Contract` and validating `Contract` against `Code`. If `Contract` does not comply to `Code`, then optimization is not possible, thus `Policy` is directly inlined into `Code`. In the case that `Contract` complies, `SIMULATIONCHECKER` is applied and the flow goes through as of Model 1.

Model 2 gives an advantage when the untrusted part has large resources for example high computing ability. The idea is similar to *Proof-Carrying Code*[63] where the producer provides a proof carried by an application. Some works have been developed along this line, signature verifier in [37], or weakest precondition based annotation checker [7] specified with ConSpec language[8].

8.2.2 Rewriter on Untrusted part

Model 3. The third model (Figure 8.2a) is similar to Model 1 but `REWRITER` is moved to untrusted part. The modification from Model 1 lies on an extra step where `OptPolicy` must be verified against `Code`. This step is needed because the `REWRITER` is on untrusted part thus there exists uncertainty `REWRITER` really injecting `OptPolicy`. The cost of this extra step depends on the trade off between injecting `Code` with `OptPolicy` and verifying `OptPolicy` against `Code`.

Positioning `REWRITER` in untrusted part on Model 3 (and later on Model 4) is similar to the approach of Hamlen’s certified IRM [44] where they use concept of type-safety in the `SafeCode`.

Model 3 gives an advantage when the system is distributed and untrusted code producers may involve in making optimization effective. For example in Yan and Fong’s work in[86], where IRM optimization framework can be distributed with an untrusted code producer involves in optimization.

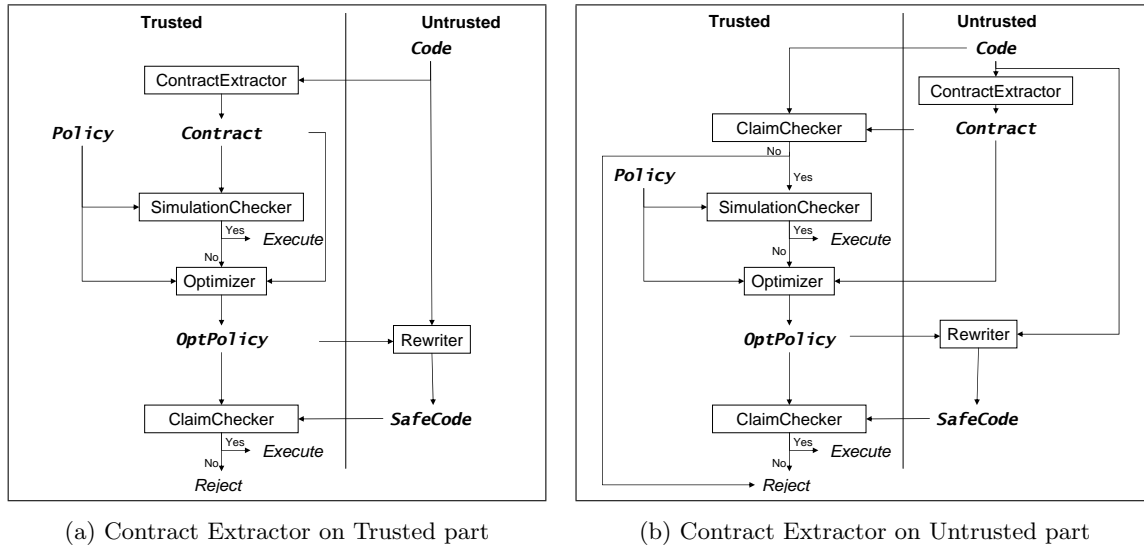


Figure 8.2: Rewriter on Untrusted part

Model 4. The fourth model (Figure 8.2b) is similar to Model 2 by positioning REWRITER on untrusted part. The modification from Model 2 lies on an extra step where *OptPolicy* must be verified against *Code*. The reasoning is the same as in Model 3.

Model 4 derives the similar advantage as Model 2 when the untrusted part has large resources for example high computing ability and from Model 3 when the system is ditributed and untrusted code producers may involve in making optimization effective. However, Model 4 adds cost of extra steps both for verifying *Contract* against *Code* and *OptPolicy* against *Code*.

8.2.3 Optimizer and Rewriter on Untrusted part

Model 5. The fifth model (Figure 8.3a) is similar to Model 3 (thus also to Model 1). In Model 5, not only REWRITER resides on untrusted part but also OPTIMIZER. The modification from Model 3 lies on optimization process being done on untrusted part. However, this does not add any extra step to the work-flow because verification of *OptPolicy* against *Code* is adequate.

Model 5 derives the similar advantage as Model 3 when the system is ditributed and untrusted code producers may involve in making optimization effective. Another advantage of this models is when the untrusted part has large resources for example high computing ability for optimization without adding any extra step to the work-flow in Model 3. Thus, compared to Model 3, Model 5 is better.

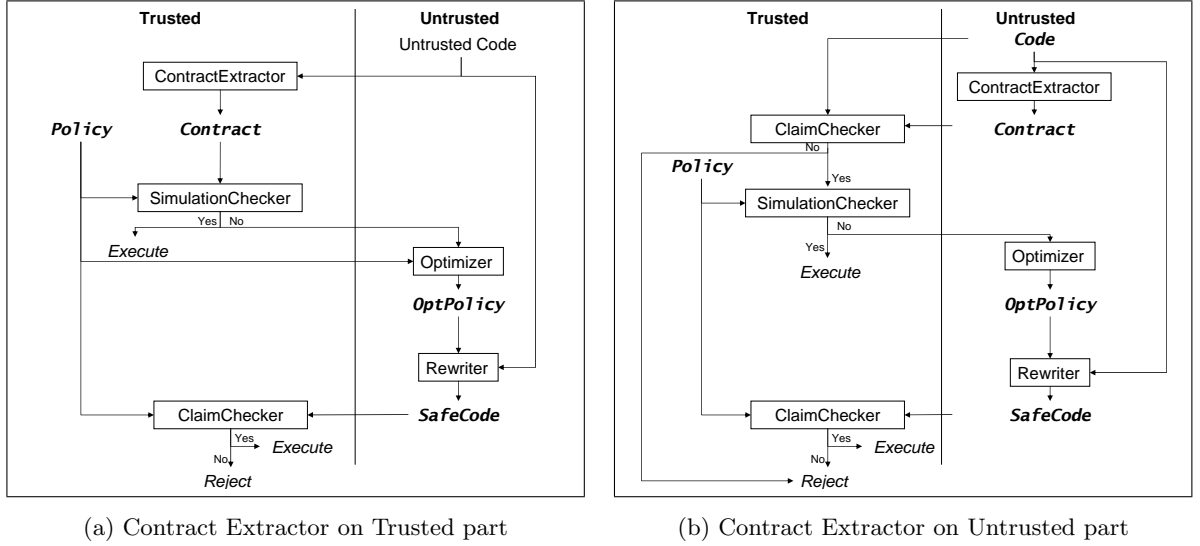


Figure 8.3: Optimizer and Rewriter on Untrusted part

Model 6. The sixth model (Figure 8.3b) has most of the components out of the trusted domain. In Model 6, after running the `CONTRACTEXTRACTOR`, `Contract` is checked against `Code` using the `CLAIMCHECKER`. If the `Contract` does not comply to `Code`, then `Code` is rejected. However, rejection might be too restrictive, thus another option is to deploy directly the `Policy` object in charge on monitoring in `Code` by using the `REWRITER` which gives result an `SafeCode`.

Model 6 derives the similar advantages and disadvantages as Model 4. However, Model 6 also has another advantage as in Model 5, namely when the untrusted part has large resources for optimization without adding any extra step to the work-flow in Model 4. Thus, compared to Model 4, Model 6 is better.

Overall the six models, the main constraint in feasibility of enforcement mechanisms to be applied is the available resources and environment where the models are to be applied.

8.3 A Search Procedure for IRM Optimization

The first issue to be solved in the problem of IRM optimization is regarding formalization of the notion of optimization. Our work attempts to give a preliminary formalization using the concept of \mathcal{AMT} . In \mathcal{AMT} the problem of searching an optimized policy can be stated intuitively as follows: given two automata A^C and A^P representing respectively the formal specification of a contract and of a policy, we have an efficient IRM A^O derived from A^P with respect to A^C when:

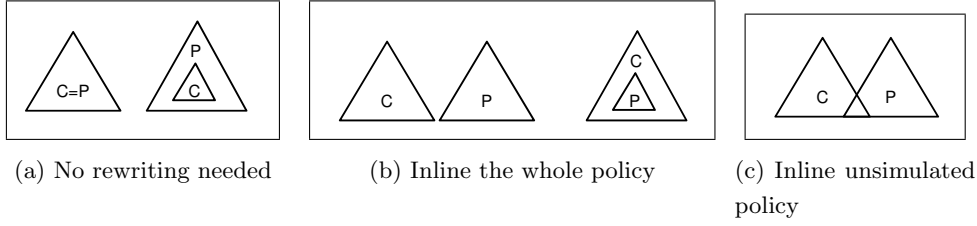


Figure 8.4: Optimization alternatives

1. every security-relevant event invoked by the intersection of A^O and A^C can also be invoked by A^P , and
2. A^O has smaller or equal number of transitions or states compared to A^P .

Intuitively, there are three possible cases in inlining a policy.

no-inline In this case no policy is needed to be inlined because contract complies to policy. There two cases, the first when contract is equal to policy and the second case is when contract subsumed by policy (Figure 8.4a).

inline-all In this case all policy needs to be inlined because contract does not comply to policy at all. There two cases, the first when contract is completely differs from policy and the second case is when policy is subsumed by contract (Figure 8.4b).

inline-partial In this case some policy needs to be inlined because contract complies partially to policy. Optimization is intended to be applied to this case (Figure 8.4c).

To illustrate possible cases in inlining a policy, we give a simple example with a simple alphabet $\{a, b, c, d, e, f, g, m, n\}$ that represent security relevant behaviors. *no-inline* case is shown in the first two rows in Figure 8.6 and *inline-all* is shown in the last two rows in Figure 8.5.

For example we have a rule for a contract and a rule for a policy, then the optimized rule of the policy is represented as in Figure 8.6.

The second issue is finding the optimal policy to be enforced with respect to the claimed applications policy. Thus, we are not interested in finding a unique minimum policy, given only the policy itself, instead we are interested in finding the optimal policy guided by claimed applications policy. This is can be solved by Algorithm 6 which is a modification of Algorithm 5. The idea is, during simulation game, we search for states which are simulated from the policy initial state and set the outgoing transition as true, meaning allowed all actions because they are already guaranteed by the contract shown in Algorithm 6.

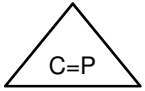
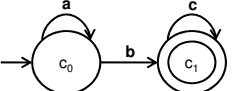

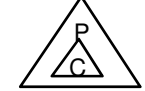
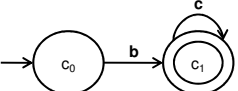

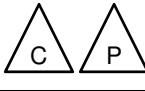
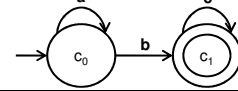
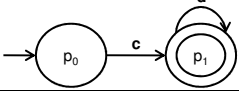
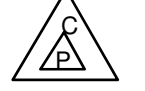
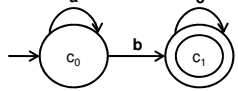
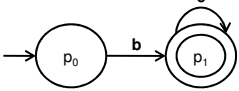
Inline-type	Contract	Policy
		
		
		
		

Figure 8.5: Inline Type Examples


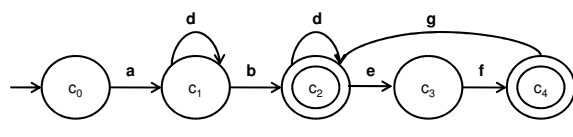
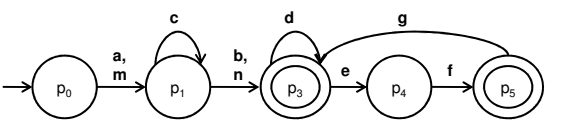
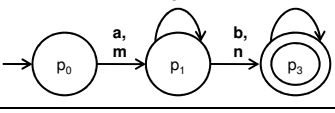
Inline-type	
Contract	
Policy	
Optimized Policy	

Figure 8.6: Optimization Example

Algorithm 6 Optimization Algorithm

Input: two \mathcal{AMT} automata A^C and A^P

```
1: Construct compliance game graph  $G = \langle V_1, V_0, E, l \rangle$ 
2: for all  $v \in V$  do
3:    $\mu(v) := \mu_{\text{new}}(v) := 0$ 
4: repeat
5:    $\mu := \mu_{\text{new}}$ 
6:   for all  $v \in V_0$  do
7:      $\mu_{\text{new}}(v) := \begin{cases} \infty & \text{if } \{\mu(w)|(v, w)\} = \emptyset \\ \min \{\mu(w)|(v, w)\} & \text{otherwise} \end{cases}$ 
8:   for all  $v \in V_1$  do
9:      $max_v := \max \{\mu(w)|(v, w) \in E\}$ 
10:     $\mu_{\text{new}}(v) := \begin{cases} \infty & \text{if } max_v = \infty \\ 0 & \text{if } l(v) = 0 \\ max_v + 1 & \text{if } l(v) = 1 \\ max_v & \text{if } l(v) = 2 \end{cases}$ 
11: until  $\mu = \mu_{\text{new}}$ 
12: if  $\mu(v_{(s_0^c, s_0^p)}) < \infty$  then
13:   Do Nothing //no-inline, because simulation exists
14: else
15:   Add  $v_0$  to ToBeInlined
16:   for all  $v \in (V_1 - \{v_0\})$  do
17:     if Reachable( $v$ ) then
18:       if ( $\mu(v) = \infty$ ) then
19:         Add  $v$  to ToBeInlined
20:       else
21:         Add  $v$  to SimulatedToBeInlined
22:   if SimulatedToBeInlined =  $\emptyset$  then
23:     Inline  $A^P$  //inline-all
24:   else
25:     for all  $v = v_{(s^c, s^p)} \in \textit{ToBeInlined}$  do
26:       Add  $s^p$ , in-coming transitions of  $s^p$ , and out-going transitions of  $s^p$  into  $A^O$ 
27:       Compute shortest distance from initial state to all the states called it TmpDistance
28:       for all  $v = v_{(s^c, s^p)} \in \textit{SimulatedToBeInlined}$  do
29:          $tmpElement.state := s^p$ ;  $tmpElement.stat := \textit{Live}$ ;  $tmpElement.dist := \textit{TmpDistance}[s^p]$ 
30:         Add  $tmpElement.dist$  to TempStatus
31:       OrderedSimulatedToBeInlined := Order(SimulatedToBeInlined)
32:       for all  $tmp \in \textit{OrderedSimulatedToBeInlined}$  do
33:         if TempStatus[ $tmp.state$ ].stat  $\neq$  Kill then
34:           for all  $tmpNextState \in tmp.succ$  do
35:             Set TempStatus[ $tmpNextState$ ].stat = Kill
36:           Add  $tmp.state$ , in-coming transitions of  $tmp.state$ , and out-going transition of  $tmp.state$ 
           as * into  $A^O$ 
37:       Inline  $A^O$  //inline-partial
```

At the first step (line 1) a compliance game graph $G = \langle V_1, V_0, E, l \rangle$ is constructed out of automata A^C and A^P . After finishing lifting compliance measure possible cases in inlining a policy. We analyze the three possible cases in inlining a policy. First case contract complies to policy either when contract is equal to policy or when contract subsumed by policy and *no-inline* is needed (line 13). It occurs when simulation exists.

If there exists no simulation, then we collect vertices which are not simulated into set *ToBeInlined* and vertices which are simulated into set *SimulatedToBeInlined* and ensure that they are reachable from initial vertex $(v_{(s_0^C, s_0^P)})$ in line 17. At least one element is in *ToBeInlined* because there exists no simulation, hence $\mu(v_{(s_0^C, s_0^P)}) < \infty$ and initial vertex is added into set *ToBeInlined*.

Second case occurs when contract does not comply to policy at all and *inline-all* is needed (line 23). It occurs when contract is completely differs from policy or when policy is subsumed by contract (*SimulatedToBeInlined* = \emptyset).

Third case occurs when some policy needs to be inlined because contract complies partially to policy (*inline-partial*). Optimization occurs in this case. First, all the states and transitions which are not simulated, i.e. *ToBeInlined*, are inlined (line 25). Next, to add simulated state with such that it is the nearest to the initial state, we introduce some data structure. As an indexed table of tuple $\langle \text{state}, (\text{stat}, \text{dist}) \rangle_i$ called *TempStatus* to hold the temporary status of states, where *state* is the key of the table, *stat* is the status of that state with Live meaning still in process, and Kill meaning cannot be added to optimized automaton, and *dist* is the shortest path from the initial state to that state. *dist* can be computed using Dijkstra's algorithm with initial state assigned 0 distance. First, all states are set to *Live* meaning it is still in process. Next, for each s^P in *SimulatedToBeInlined* we make list of tuple $\langle \text{state}, \text{succ} \rangle_i$ called *OrderedSimulatedToBeInlined* where *succ* consists reachable states from s^P with longer distance to initial state. At each step, an element from *OrderedSimulatedToBeInlined* is analyzed for possibility to insert. If the state's status, say s^P is not Kill, then we mark all reachable states from s^P as KillThen. Then add s^P , in-coming transitions of s^P , and set out-going transition of s^P as *, meaning allowed all actions.

We are now in the position to state our optimization result using fair simulation:

Proposition 8.3.1 *Let the theory \mathcal{T} be decidable with an oracle for the SMT problem in the complexity class \mathcal{C} then:*

1. *The policy optimization problem for \mathcal{AMT} using fair simulation is decidable in time $O(2 \cdot |E| \cdot |M_G| + |S^p|^2 + |S^p| \cdot |\Delta_{\mathcal{T}}^p|)$.*
2. *The policy optimization problem for \mathcal{AMT} using fair simulation is decidable in space $O(|V| + |S^p| + \text{LOG}(|S^p| \cdot |\Delta_{\mathcal{T}}^p|))$.*

Proof. The policy optimization problem for \mathcal{AMT} using fair simulation can be reduced to a fair simulation between a contract with a security policy with adding processes for constructing the optimized policy. Algorithm 6 which is a modification of Algorithm 5 can solve this problem. The algorithm takes as input the midlet’s claim and the mobile platform’s policy and constructs compliance game graph $G = \langle V_1, V_0, E, l \rangle$.

Correctness.

As in proof of Proposition 6.3.1, the correctness derived from Jurdziński’s algorithm on parity games [52]. Jurdziński defined a parity game between two players and defining and even player (in our case **Policy**) wins when the lowest priority occurring infinitely often in the play is even (in our case **Policy** can return to compliance level 0 infinitely often). He proposed computing the game using progress measure which is defined as $M_G = [1] \times [n_1 + 1] [1] \times [n_3 + 1] \times \dots \times [1] \times [n_{d-1} + 1]$, where d is the maximum priority in the game. In our setting, we slightly modify the Jurdziński progress measure [52] to *compliance measure* where instead of a pair $(0, x)$ we only use x . As we have mentioned afore, this is due to our observation of our domain where we only have 3 priorities, namely $l(v) \in 0, 1, 2$ thus for ordering $(0, x) \geq_{l(v)} (0, x')$ the first component will not effect the ordering.

Jurdziński reasoned that each vertex can only be lifted $|M_G|$ times. This lifting procedure is implemented in Algorithm 5 presented as a loop where compliance measure progressing until reaching a pre-fixed point ($\mu = \mu_{\text{new}}$). He also defined that Even has a winning strategy from precisely the vertices v where after its lifting algorithm halts has $\mu(v) < \infty$. However, in contract-policy matching we are interested when there is a winning strategy from the initial vertex $v_{(s_0^c, s_0^p)}$. Thus, in Algorithm 6 **Policy** wins when $\mu(v_{(s_0^c, s_0^p)}) < \infty$.

If there exists simulation *no-inline* is needed (line 13). If there exists no simulation, then we collect vertices which are not simulated into set *ToBeInlined* and vertices which are simulated into set *SimulatedToBeInlined* and ensure that they are reachable from initial vertex $(v_{(s_0^c, s_0^p)})$ in line 17. This solves, the second case when contract does not comply to policy at all and *inline-all* is needed (line 23). Third case occurs when some policy needs to be inlined because contract complies partially to policy (*inline-partial*). Optimization occurs in this case. First, all the states and transitions which are not simulated, i.e. *ToBeInlined*, are inlined (line 25).

Next, we construct *TempStatus* to hold the temporary status of states then all states are set to *Live* meaning it is still in process. For each s^P in *SimulatedToBeInlined*, *OrderedSimulatedToBeInlined* is computed where *succ* consists reachable states from s^P with longer distance to initial state. At each step, an element from *OrderedSimulatedToBeInlined* is analyzed for possibility to insert. If the state’s status, say s^P is not Kill, then we mark

all reachable states from s^P as KillThen. Then add s^P , in-coming transitions of s^P , and set out-going transition of s^P as *, meaning allowed all actions. Algorithm 6 halts either in *no-inline*, or *inline-all*, or *inline-partial* has been processed.

Termination. This optimization using Algorithm 6 terminates because the parity game terminates and each step in subprocedure if there exists no simulation (begins from line 14) halts.

Complexity. As in proof of Proposition 6.3.1, the time complexity analysis follows as in lifting procedure in Jurdziński [52]. Thus, the time complexity of simulation part of Algorithm 6 is as in Algorithm 5, i.e. $O(2 \cdot |E| \cdot |M_G|)$. Adding states from *ToBeInlined* into A^O has time complexity of $O(|V_1|)$ by Lemma 6.3.1, it is in $O(|S^c| \cdot |S^p|)$. While adding transitions from *ToBeInlined* into A^O has time complexity of $O(|\Delta_{\mathcal{T}}^p|)$. Computing distance in *TempStatus* using Dijkstra's like algorithm has time complexity of $O(|S^p|^2)$ if we do not consider a smart implementation of it. Computing reachable states in *OrderedSimulatedToBeInlined* has time complexity of $O(|S^p| \cdot |\Delta_{\mathcal{T}}^p|)$. At each step, an element from *OrderedSimulatedToBeInlined* is analyzed for possibility to insert. If the state's status, say s^P is not Kill, then we mark all reachable states from s^P as KillThen. Then add s^P , in-coming transitions of s^P , and set out-going transition of s^P as *. This step has time complexity of $O(|S^p| \cdot |\Delta_{\mathcal{T}}^p|)$. Thus, the time complexity of Algorithm 6 is $O(2 \cdot |E| \cdot |M_G| + |S^p|^2 + |S^p| \cdot |\Delta_{\mathcal{T}}^p|)$.

As in time complexity analysis, the space complexity follows as in lifting procedure in Jurdziński [52] in proof of Proposition 6.3.1. Thus, the space complexity of simulation part of Algorithm 6 is as in Algorithm 5, i.e. $O(|V|)$, where the total number of vertices equals to $V = |V_1| + |V_0|$. Adding states and transitions from *ToBeInlined* into A^O has space complexity of $O(|V_1|)$ by Lemma 6.3.1 is in $O(|S^c| \cdot |S^p|)$.

Computing distance in *TempStatus* using Dijkstra's like algorithm has space complexity of $O(|S^p|)$. Computing reachable states in *OrderedSimulatedToBeInlined* has space complexity of $O(\text{LOG}(|S^p| \cdot |\Delta_{\mathcal{T}}^p|))$. At each step, an element from *OrderedSimulatedToBeInlined* is analyzed for possibility to insert. If the state's status, say s^P is not Kill, then we mark all reachable states from s^P as KillThen. Then add s^P , in-coming transitions of s^P , and set out-going transition of s^P as *. This step has space complexity of $O(\text{LOG}(|S^p| \cdot |\Delta_{\mathcal{T}}^p|))$. Thus, the space complexity of Algorithm 6 is $O(|V| + |S^p| + \text{LOG}(|S^p| \cdot |\Delta_{\mathcal{T}}^p|))$. \square

The third issue we can compute such an optimal policy with an efficient algorithm. From Proposition 8.3.1 the time complexity is $O(2 \cdot |E| \cdot |M_G| + |S^p|^2 + |S^p| \cdot |\Delta_{\mathcal{T}}^p|)$ and space complexity is $O(|V| + |S^p| + \text{LOG}(|S^p| \cdot |\Delta_{\mathcal{T}}^p|))$. Our current result has not yet satisfied it.

Chapter 9

Conclusions and Future Work

This thesis provides a formal model called Automata Modulo Theory (\mathcal{AMT}) that shows the possibility to define both safety and liveness policies in a general way, and to perform matching of those policies efficiently with the tractability limit in the complexity of the satisfiability procedure for the theories incorporated with the proof of correctness and completeness of our matching algorithms.

9.1 Conclusions

The security policies require both safety and liveness properties. \mathcal{AMT} extends Büchi Automata (BA) with edges labeled by expressions in a decidable theory. \mathcal{AMT} is apt to accept both finite and infinite input with acceptance condition as in BA. This feature enables \mathcal{AMT} to express both safety and liveness properties including renewal properties which are not common but exist in real security policies.

The mechanism for defining a general security policies (that is not platform-specific). \mathcal{AMT} has edges labeled by expressions in a decidable theory. The theory can be a combination of theories by taking into account its complexity. Due to the tractability limit of \mathcal{AMT} which is essentially the complexity of the satisfiability procedure for the theories, called as subroutines. In our case, we have applied a signature of API theory where the names from Java VM are used for notation e.g. *javax.microedition*.

The mechanism for representing an infinite structure as a finite structure. To capture realistic scenarios with potentially infinite transitions (e.g. “only connections to urls starting with https”) \mathcal{AMT} abstracts away these transitions as an expression in a decidable theory. Thus transforming an infinite system into a finite one. However, there is still

an open problem in finding a suitable approximation of a finite system given an infinite one.

Efficiency. Our goal is to provide this midlet-contract vs platform-policy matching on-the-fly (during the actual download of the midlet), hence issues such as small memory footprint and effective computations play a key role. The tractability limit is the complexity of the satisfiability procedure for the background theories used to describe expressions with the oracle in the complexity class \mathcal{C} , then the following results can be drawn:

- The contract-policy matching problem for \mathcal{AMT} using language inclusion is decidable in $LIN - TIME^{\mathcal{C}}$ and in $NLOG - SPACE^{\mathcal{C}}$ -complete (Proposition 4.5.1).
- The contract-policy matching problem for \mathcal{AMT} using fair simulation defined in Algorithm 5 is decidable in time $O(2 \cdot |E| \cdot |M_G|)$ and in space $O(|V|)$ (Proposition 6.3.2).
- The policy optimization problem for \mathcal{AMT} using fair simulation is decidable in time $O(2 \cdot |E| \cdot |M_G| + |S^p|^2 + |S^p| \cdot |\Delta_T^p|)$ and in space $O(|V| + |S^p| + LOG(|S^p| \cdot |\Delta_T^p|))$ (Proposition 8.3.1).

The feasibility of our approach was shown by developing a prototype on Linux operating system which has also been ported to the mobile for actual detailed profiling, namely HTC P3600 (3G PDA phone) and on Microsoft®Windows Mobile®5.0 operating system. The following conclusions can be drawn on the feasibility and efficiency of the system based on our experimental results:

- \mathcal{AMT} makes it possible to match the mobile's policy and the midlet's contract by mapping the problem into a variant of the on-the-fly product and emptiness test from automata theory, without symbolic manipulation procedures of zones and regions nor finite representation of equivalence classes. The tractability limit is essentially the complexity of the satisfiability procedure for the theories, called as subroutines, where most practical policies require only polynomial time decision procedures [58].
- This matching using language inclusion however has a limitation in the structure of the policy automaton (only deterministic automaton). The constraint arises from the \mathcal{AMT} complementation. As BA complementation, the non-deterministic complementation is complicated and demonstrates exponential blow-up in the state space [20]. Safra in [70] gives a better lower bound ($2^{O(n \log n)}$) for nondeterministic BA complementation, however it is still exponential(see [83]).

- The determinism constraint complies to our domain of interest because the security policies in our application domain are naturally deterministic, as the platform owner should have a clear idea on what to allow or disallow. Furthermore, to cope with non-deterministic \mathcal{AMT} , we can use the approach as in [59].

9.2 Future Work

An approach to address scalability (if our smart-phone must cope with the web applications of its internal web server) is to give up soundness of the matching and use algorithms for simulation and testing. A challenge to be addressed is how to measure the coverage of approximate matching. Which value should give a reasonable assurance about security? Should it be an absolute value? Should it be in proportion of the number of possible executions? In proportion to the likely executions? An interesting approach could be to recall to life a neglected section on model checking by Courcoubetis et al [23] in which they traded off a better performance of the algorithm in change for the possibility of erring with a small probability.

A second approach is to use the contract as a model of the application in order to generate security tests by applying techniques from Model Based Testing [84]. Losing soundness is a major disadvantage: an application may pass all the generated tests and still turn out to violate the contract once fielded. However, the advantages are also important: no annotations on the application source code are needed, and the tests generated from the contract can be easily injected in the standard platform testing phase, thus making this approach very practical. A challenge to be addressed here is how to measure the coverage of such security tests. When are there enough tests to give a reasonable assurance about security?

A known problem with security automata and infinity yet to be addressed is the encoding of policies such as “we must allow certain strings that we have seen in the past”. If the set of strings is unbounded, then it is difficult (if not impossible) to encode it with finite states.

Another interesting problem for future work is a scenario when the claimed security contract is missing (as is the case for current MIDP applications). In that case, based on the platform security policy, the “claimed” security contract could be inferred by static analysis as an approximation automaton. If such an approximation is matched, then monitoring the code becomes unnecessary. The feasibility of this approach depends on the cost of inferring approximation automata on-the-fly.

Bibliography

- [1] *Response time in man-computer conversational transactions*. ACM Press, 1968.
- [2] *The information visualizer, an information workspace*. ACM Press, 1991.
- [3] Card specification version 2.2. Technical report, GlobalPlatform, March 2006. Report available at www.globalplatform.org.
- [4] Confidential card content management card specification v 2.2 - amendment a. Public Release GPC_SPE_007, GlobalPlatform, October 2007. Report available at www.globalplatform.org.
- [5] W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland Pub. Co., 1954.
- [6] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.
- [7] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. *J. of Logic and Algebraic Programming*, 2009.
- [8] I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. In *Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, Dresden, Germany, 2007.
- [9] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.
- [10] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Found. of Comp. Security*, 2002.
- [11] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proc. of the ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Implementation*, pages 305–314. ACM Press, 2005.

- [12] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. of Inform. Sec.*, 4(1-2):2–16, 2005.
- [13] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *J. of Logic and Algebraic Programming*, 78:340–358, May-June 2009.
- [14] N. Bielova, F. Massacci, and I. Siahaan. Testing decision procedures for security-by-contract. In *Joint Workshop on Found. of Comp. Sec., Automated Reasoning for Sec. Protocol Analysis and Issues in the Theory of Sec. (FCS-ARSPA-WITS'08)*, 2008.
- [15] N. Bielova, F. Massacci, and I. Siahaan. Testing decision procedures for security-by-contract: Extended abstract. IEEE Symp. on Logic in Comp. Scie. and Comp. Sec. Found. Workshop (LICS-CSF'08) short talk in joint session, June 2008.
- [16] N. Bielova, M. Dalla Torre, N. Dragoni, and I. Siahaan. Matching policies with security claims of mobile applications. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security (ARES'08)*. IEEE Press, 2008.
- [17] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P.v. Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In K. Etessami and S.K. Rajamani, editors, *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 335–349. Springer-Verlag, 2005.
- [18] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P.v. Rossum, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. of Autom. Reas.*, 35(1):265–293, 2005.
- [19] R. E. Bryant, S.K. Lahiri, , and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, LNCS, pages 78–92. Springer-Verlag, 2002.
- [20] J.R. Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel et al., editor, *Int. Cong. on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
- [21] B.V. Cherkassky and A.V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

- [22] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, LNCS, pages 359–364. Springer-Verlag, 2002.
- [23] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in Syst. Design*, 1(2-3):275–288, 1992.
- [24] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, 2(4):511–547, 1992.
- [25] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of the 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340. Springer-Verlag, 2008.
- [26] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Tech. Rep.*, 13(1):25 – 32, 2008.
- [27] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of the 4th European PKI Workshop Theory and Practice (EUROPKI'07)*, page 297. Springer-Verlag, 2007.
- [28] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2004.
- [29] U. Erlingsson and F.B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the 2000 IEEE Symp. on Security and Privacy*, pages 246–255, 2000.
- [30] K. Etessami, T. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM J. on Comp.*, 34(5):1159–1175, 2005.
- [31] D. Evans. *Policy-Directed Code Safety*. PhD thesis, MIT, 1999.
- [32] F. Piessens F. Massacci and I. Siahaan. Security-by-contract for the future internet. In *Proc. of the 1st Future Internet Symposium (FIS 2008)*, LNCS, pages 29–43. Springer-Verlag, 2008.
- [33] F. Piessens F. Massacci and I. Siahaan. Security-By-Contract for the Future Internet ? In *2nd Workshop on Formal Languages and Analysis of Contract-Oriented Softw. (FLACOS '08)*, 2008.

- [34] M. Fitting. *First-order logic and automated theorem proving*. Springer-Verlag, 1996.
- [35] P. Gastin, B. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *Proc. of the 11th Int. SPIN Workshop*, volume 2989 of *LNCS*, pages 92–108. Springer-Verlag, 2004.
- [36] S. Ghilardi. Model-theoretic methods in combined constraint satisfiability. *J. of Autom. Reas.*, 33(3):221–249, 2004.
- [37] D. Ghindici, G. Grimaud, and I. Simplot-Ryl. An information flow verifier for small embedded systems. In D. Sauveron et al., editor, *Proc. Workshop in Information Security Theory and Practices: Smart Cards, Mobile and Ubiquitous Computing Systems (WISTP'07)*, volume 4462 of *LNCS*, pages 189–201. Springer-Verlag, May 2007.
- [38] D. Ghindici, I. Simplot-Ryl, and J.-M. Talbot. A sound analysis for secure information flow using abstract memory graphs. In *The 3rd Int. Conf. on Fundamentals of Sw. Eng. (FSEN'09)*, 2009.
- [39] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*. Addison-Wesley Professional, 2003.
- [40] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, pages 610–624. Springer-Verlag, 2002.
- [41] L.G. Hacijan. A polynomial algorithm in linear programming. In *Dokl. Akad. Nauk SSSR*, volume 244, pages 1093–1096, 1979.
- [42] K. Hamlen. *Security policy enforcement by automated program-rewriting*. PhD thesis, Cornell University, 2006.
- [43] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.
- [44] K.W. Hamlen, G. Morrisett, and F.B. Schneider. Certified in-lined reference monitoring on .net. In *Proc. of the 2006 workshop on Prog. Lang. and analysis for security*, pages 7–16. ACM Press, 2006.
- [45] M. Hennessy and H. Lin. Symbolic bisimulations. In *MFPS'92: Selected papers of the meeting on Math. Foundations of Programming Semantics*, pages 353–389. Elsevier Sci. Publishers B. V., 1995.
- [46] T.A. Henzinger, O. Kupferman, and S.K. Rajamani. Fair simulation. In *Proc. of the 8th Int. Conf. on Concurrency Theory*, pages 273–287. ACM Press, 1997.

- [47] M. Hilty, A. Pretschner, C. Schaefer, and T. Walter. Usage control requirements in mobile and ubiquitous computing applications. In *Proc. of the Int. Conf. on Sys. and Net. Comm. (ICSNC 2006)*, pages 27–27. IEEE Press, 2006.
- [48] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of the 2nd Int. SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.
- [49] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.
- [50] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. *ACM SIGPLAN Notices*, 33(7):67–74, 1998.
- [51] N.D. Jones. Space-bounded reducibility among combinatorial problems. *J. of Comp. and Syst. Sci.*, 11(1):68–85, 1975.
- [52] M. Jurdzinski. Small progress measures for solving parity games. In *STACS '00: Proc. of the 17th Annual ACM Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer-Verlag, 2000.
- [53] S. Krstic, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *Proc. of the 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424, page 602. Springer-Verlag, 2007.
- [54] S.K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI constraints. In *Proc. of the 5th Int. Workshop on Frontiers of Combining Systems (FroCoS'05)*, volume 3717. Springer-Verlag, 2005.
- [55] B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
- [56] J.A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, 2006.
- [57] F. Massacci, N. Dragoni, and I. Siahaan. A Security-by-Contracts Architecture for Pervasive Services. In *1st Workshop on Formal Languages and Analysis of Contract-Oriented Softw. (FLACOS '07)*, 2007.
- [58] 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)*, 2007.

- [59] F. Massacci and I. Siahaan. Simulating midlet's security claims with automata modulo theory. In *Proc. of the 2008 workshop on Prog. Lang. and analysis for security*, pages 1–9, 2008.
- [60] F. Massacci and I. Siahaan. Optimizing IRM with Automata Modulo Theory. In *Proc. of the 5th Int. Workshop on Security and Trust Management (STM 2009)*, 2009.
- [61] K.E. Mayes and K. Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer-Verlag, 2008.
- [62] K. Naliuka. *Security Run-Time Monitoring for Mobile Devices*. PhD thesis, University of Trento, 2008.
- [63] G.C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 106–119. ACM Press, 1997.
- [64] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.
- [65] F. Nielson and H.R. Nielson. Flow logic for imperative objects. In *Proc. of the 23rd Int. Symp. on Math. Foundations of Comp. Scie.*, pages 220–228. Springer-Verlag, 1998.
- [66] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM*, 53(6):937–977, 2006.
- [67] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proc. of the 25th Annual Comp. Sec. Applications Conf. (ACSAC'09)*, 2009.
- [68] C.H. Papadimitriou. On the complexity of integer programming. *J. of the ACM*, 28(4):765–768, 1981.
- [69] P.H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *Proc. of the 4th ACM Symposium on Information Comp. and Comm. Sec. (ASIACCS 2009)*, pages 10–12, 2009.
- [70] S. Safra. On the Complexity of omega-Automata. In *IEEE Symp. on Found. Comp. Science (FOCS'88)*, pages 319–327, White Plains, New York, USA, 1988. IEEE Press.

- [71] F.B. Schneider. Enforceable security policies. *ACM Trans. on Inf. and Syst. Security*, 3(1):30–50, 2000.
- [72] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. Technical Report 2004/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, November 2004.
- [73] R. Sebastiani. Lazy satisfiability modulo theories. *J. on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.
- [74] R. Sekar, V.N. Venkatakrisnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symp. on Operating Syst. Princ.*, pages 15–28. ACM Press, 2003.
- [75] R. Smith, C. Estan, S. Jha, and I. Siahaan. Fast signature matching using extended finite automaton (xfa). In *Proc. of the 4th Int. Conf. on Inform. Syst. Sec. (ICISS 2008)*, pages 158–172, 2008.
- [76] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.
- [77] MOBIUS Project Team. Framework- and application-specific security requirements. Public Deliverable D1.2, Mobility, Ubiquity and Security - MOBIUS, 2006. Report available at <http://mobius.inria.fr>.
- [78] C. Tinelli and M.T. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. *Proc. of the 1st Int. Workshop on Frontiers of Combining Systems (FroCoS'96)*, 3:103–120, 1996.
- [79] C. Tinelli and C.G. Zarba. Combining nonstably infinite theories. *J. of Autom. Reas.*, 34(3):209–238, 2005.
- [80] P. Uppuluri. *Intrusion Detection/Prevention Using Behavior Specifications*. PhD thesis, Stony Brook University, 2003.
- [81] D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci. A flexible security architecture to support third-party applications on mobile devices. In *Proc. of the 1st ACM Comp. Sec. Arch. Workshop*, 2007.
- [82] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of the 8th Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, LNCS, pages 238–266. Springer-Verlag, 1996.

- [83] M.Y. Vardi. Büchi complementation a 40-year saga, March 2006.
- [84] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. of the 10th Eur. Softw. Eng. Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, pages 273–282. ACM Press, 2005.
- [85] I. Welch and R.J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *J. of Comp. Sec.*, 10(4):399–432, 2002.
- [86] F. Yan and P. W. L. Fong. Efficient IRM Enforcement of History-Based Access Control Policies. In *Proc. of the 4th ACM Symposium on Information Comp. and Comm. Sec. (ASIACCS 2009)*, pages 35–46, 2009.
- [87] B.S. Yee. A sanctuary for mobile agents. In J. Vitek and C.D. Jensen, editors, *Secure Internet Programming*, pages 261–273. Springer-Verlag, 1999.
- [88] A. Zobel, C. Simoni, D. Piazza, X. Nunez, and D. Rodriguez. Business case and security requirements. Public Deliverable D5.1.1, EU Project S3MS, 2006. Report available at www.s3ms.org.

Appendix A

On-the-fly Matching Prototype Class Diagram

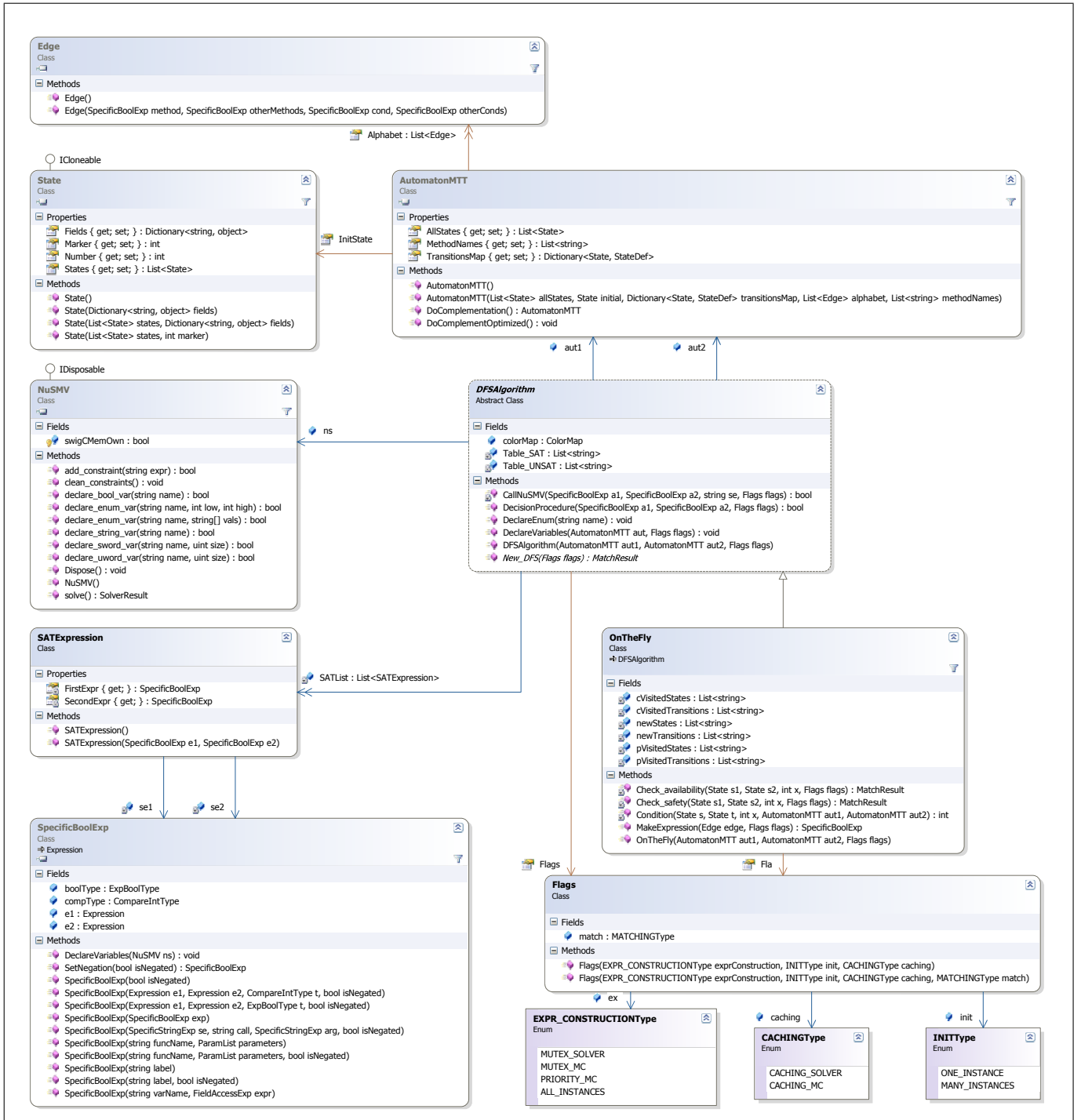


Figure A.1: On-the-fly Class Diagram

Appendix B

Simulation Matching Prototype Class Diagram

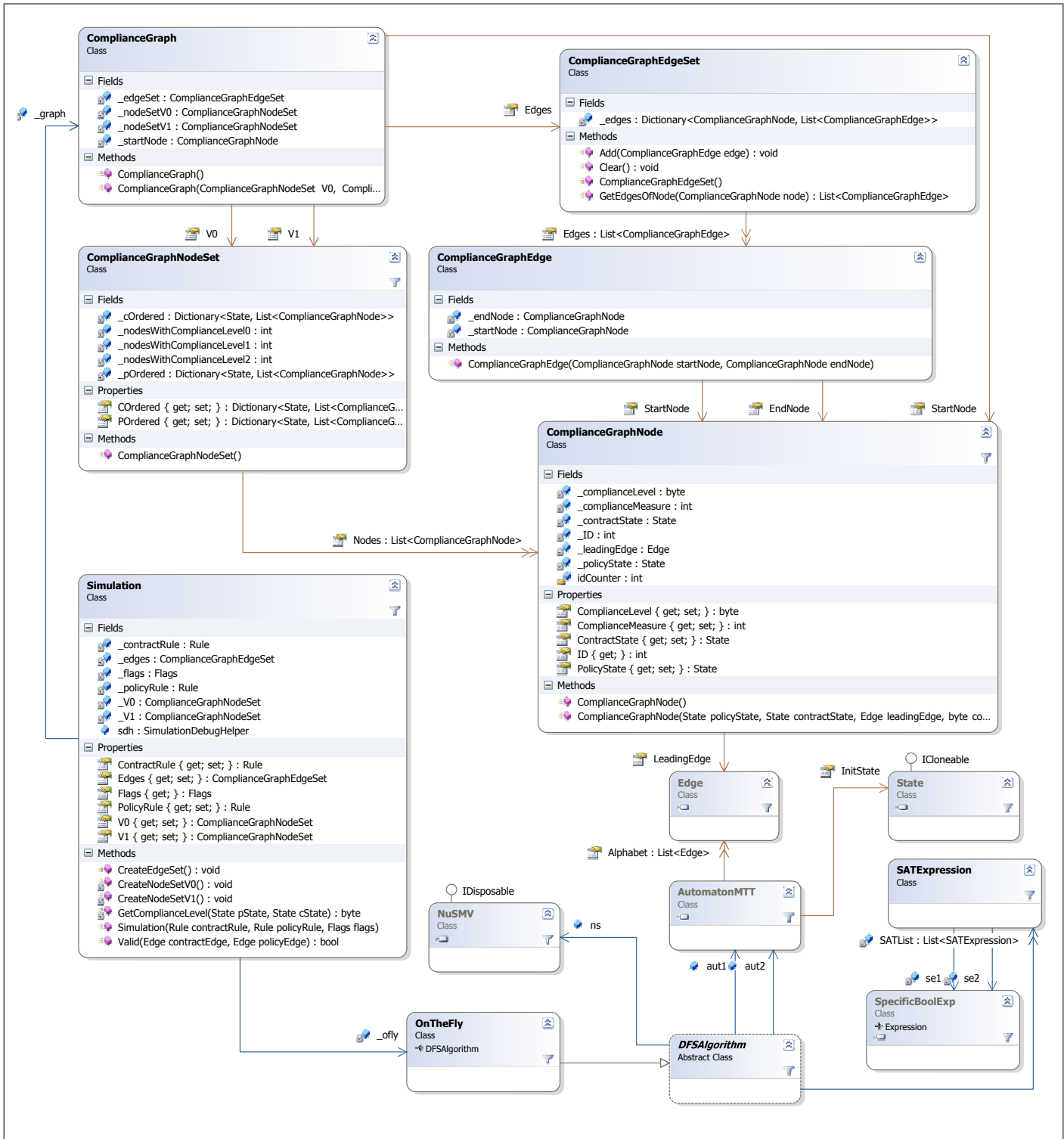


Figure B.1: Simulation Class Diagram

Appendix C

On-the-fly Matching Prototype Experiments

Table C.1: Problems Suit

Problem	Contract	Policy
P100-100	100SMS_contract.pol	100SMS_policy.pol
P100-10	100SMS_contract.pol	10SMS_policy.pol
P100-1	100SMS_contract.pol	1SMS_policy.pol
P100-NO	100SMS_contract.pol	noSMS_policy.pol
P10-100	10SMS_contract.pol	100SMS_policy.pol
P10-10	10SMS_contract.pol	10SMS_policy.pol
P10-1	10SMS_contract.pol	1SMS_policy.pol
P10-NO	10SMS_contract.pol	noSMS_policy.pol
P1-100	1SMS_contract.pol	100SMS_policy.pol
P1-10	1SMS_contract.pol	10SMS_policy.pol
P1-1	1SMS_contract.pol	1SMS_policy.pol
P1-NO	1SMS_contract.pol	noSMS_policy.pol
PNO-100	noSMS_contract.pol	100SMS_policy.pol
PNO-10	noSMS_contract.pol	10SMS_policy.pol
PNO-1	noSMS_contract.pol	1SMS_policy.pol
PNO-NO	noSMS_contract.pol	noSMS_policy.pol

Table C.2: Average Running Problem Suit 10 Times (s)

Problem	M1	M2	M3	M4	M5	M6	Result
P100-100	15.219	15.478	15.19	15.335	15.219	15.187	Match
P100-10	9.468	10.086	9.355	9.372	9.391	9.429	Not Match
P100-1	8.824	8.951	8.91	8.927	8.953	8.871	Not Match
P100-NO	8.83	8.835	8.798	8.716	8.847	8.852	Not Match
P10-100	9.846	9.77	9.831	9.781	9.684	9.818	Match
P10-10	3.847	3.821	3.854	3.797	3.783	3.834	Match
P10-1	3.192	3.12	3.192	3.194	3.189	3.162	Not Match
P10-NO	3.042	3.058	3.065	3.041	3.051	3.042	Not Match
P1-100	9.309	8.714	9.308	9.329	9.187	9.234	Match
P1-10	3.286	3.286	3.271	3.301	3.241	3.275	Match
P1-1	2.444	2.446	2.462	2.432	2.457	2.423	Match
P1-NO	2.573	2.595	2.582	2.571	2.596	2.566	Not Match
PNO-100	9.259	9.16	9.211	9.202	9.117	9.122	Match
PNO-10	3.197	3.16	3.188	3.173	3.155	3.179	Match
PNO-1	2.5	2.502	2.513	2.525	2.523	2.522	Match
PNO-NO	2.427	2.386	2.395	2.38	2.405	2.379	Match

Appendix D

Simulation Matching Prototype Experiments

Table D.1: Average Running Problem Suit 10 Times (s)

Problem	M7	M8	Result
P100-100	3.668	5.528	Match
P100-10	5.465	7.259	Not Match
P100-1	9.106	7.419	Not Match
P100-NO	7.228	6.385	Not Match
P10-100	5.308	7.531	Match
P10-10	3.446	2.59	Match
P10-1	2.308	2.165	Not Match
P10-NO	2.18	2.105	Not Match
P1-100	6.184	4.696	Match
P1-10	2.26	2.15	Match
P1-1	1.918	1.886	Match
P1-NO	1.854	1.87	Not Match
PNO-100	5.387	4.434	Match
PNO-10	2.372	2.077	Match
PNO-1	1.995	1.838	Match
PNO-NO	1.822	1.838	Match