Department of
Information Engineering
and Computer Science
DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive 14 - 38123 Povo - Trento (Italy)
http://www.disi.unitn.it

INCLUSION MATCHING IMPLEMENTATION OF AUTOMATA MODULO THEORY (AMT)

Fabio Massacci and Ida Siahaan

July 2009

Technical Report # DISI-09-073

Contents

Abstract

The traditional realm of formal methods is the off-line verification of formal properties of hardware and software. In this technical report we describe a different approach that uses formal methods (namely the integration of automata modulo theory with decision procedures) on-the-fly, at the time an application is downloaded on a mobile application such as PDA or a smart phone. We also describe its integration with decision solver based on MathSAT and NuSMV, and the results of our experiments on matching.

The idea behind security-by-contract is that a mobile application comes equipped with a signed contract describing the security relevant behavior of the application and such contract should be matched against the mobile platform policy. Both are specified as special kinds of automata and the operation is just an on-the-fly emptiness test over two automata modulo theories where edges are not just finite states of labels but rather expressions that can capture infinite transitions such as "connect only to urls starting with https://".

Keywords Access control · Language-based security · Malicious code · Security and privacy policies

1 Introduction

The paradigm of pervasive services [2] envisions a nomadic user seamlessly and constantly receiving services from other devices and sensors embedded in the environment. Beside this web-service-like model, a new model is emerging based on the notion of *pervasive client downloads* [8]: users download new (and likely untrusted) applications on their mobile in order to exploit the computational characteristic of the device.

A tourist landing in a large city can download at the airport a navigation application that can guide her to shopping centers or touristic sights. The application can query internet sites or bluetooth services to find the optimal routes or discover local services. Living Search by Microsoft and Navitime by DoCoMo [17] are primitive examples of these future applications. Peer-to-peer and Web 2.0 collaborative applications share the same features: Channel4 in the UK allows people to download video on demand if they also download a P2P servent.

Unfortunately, this business model is not supported by the current security architecture of Java [12] and .NET [16]:

- mobile code runs only if its origin is trusted (i.e. digitally signed by a trusted party);
- a pervasive download will likely be from small companies which cannot afford to obtain a mobile operator's certification and thus will not run as trusted code;
- then this application should be sandboxed, its interaction with the environment and the device's data should be limited;
- yet we made this pervasive download precisely to have lots of (controlled) interaction with the pervasive environment.

As it is now this is both a business opportunity but also a big security threat: Channel 4 naive users with a pay-as-you-go subscription to internet found out at their own expenses the "surprising" effect of hosting a P2P application for video on demand.

We need a better security model where the mobile code should be run only if it satisfies a user-defined policy. This is precisely the setting where we can use *formal methods on-the-fly*: before downloading the application we just verify that it complies with the user security policies.

Unfortunately, in the general case this is equivalent to arbitrary software verification which is not practical for pervasive downloads (remember this has to be done on your smart phone while you wait). However, the idea behind model-carrying code [21] and security-by-contract [7] is that code should come accompanied with a "digest" (a security model or a security contract) that represents its essential security behavior.

The question raised is how we know that the security claims are actually true on the code. One possible solution is to use proof carrying code [20] or trust relations and digital signatures. The PCC approach enables safe execution of code from untrusted sources. PCC is based on assumption that the code producer should know all the security policies that are of interest to consumers since the producer sends the safety proof together with the mobile code. This assumption can be impractical due to various security policies among different consumers. On the other hand, if we use only trust

relationship, i.e. digital signatures on mobile code, then we can only reject or accept the signature and no semantics attached to the signature. The security-by-contract proposed in [7] provides semantics to a digital signature, which was not presented beforehand. So that, when binding together the code and the contract the signer takes liability for the security claims ([23] describes mobile devices security architecture that supports integration of proof-carrying code, static verification and run-time monitoring). Then one only needs to match the contract against the platform security policies. However, whenever consumer does not trust the contract provided by the code producer then the overall architecture can take care that the code actually complies with the contract by run time monitoring(see [8] describes security by contract architecture).

The next question is *which is the best formal representation of such contract and policy*. Model carrying code papers [21] suggested finite automata. Unfortunately, finite state and even Büchi Automata are too simple to express any practical policy: already the rule "only allows connections to urls starting with `https://`" would generate an automaton with infinite transitions when instantiating urls. Languages for security-by-contract policies [1] are even more expressive.

The formal model considered for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ has been introduced in [18], which extends Büchi Automata (BA) by labeling transitions with expressions belong to decidable theories. It is suitable for formalizing systems with finitely many states but infinitely many transitions by leveraging on the power of satisfiability-modulo-theory (SMT) decision procedures. In this way we can represent the task of matching the contract with the policy as language containment problem between two automata. However, while [18] provides the theoretical framework, namely the on-the-fly matching algorithm and the complexity results of the operation, the actual implementation of the algorithm and the integration with a state-of-the-art theory solver is still left open.

1.1 The Contribution of this Paper

We discuss the overall implementation architecture and the integration issues with a state of the art decision procedure solver NuSMV [5] integrated with its MathSAT libraries [4]. 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.

To this extent we have decided to implement language inclusion as emptiness test as an on-the-fly procedure a-la-SPIN 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.

The second contribution is 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.

We begin in Section 2 by briefly recap the notion of Security-by-Contract (we refer the reader to [3] for more details). Next we present $\mathcal{AMT}$ and the corresponding automata

operations in (Section 3). We also expose some specific issues to be considered in $\mathcal{AMT}$.

Section 5 by discussing the overall implementation architecture and the integration issues with the procedure solver NuSMV [5] integrated with its MathSAT libraries [4]. 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 6 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 7 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.

2 Security by Contract in a nutshell

Security-by-contract (S×C)[7, 3] 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×Cframework [7, 3] 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.

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 may 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. Next, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a proof) and shipped as shown on Figure 2.

At *deployment time*, the target platform follows a workflow as depicted in Figure 1 [3]. This workflow is a modification of S×Cworkflow [3]) by adding optimization step. First, the correctness of the evidence of a code is checked. Such evidence can be a trusted signature [25] or a proof that the code satisfies the contract (one can use Proof-Carrying-Code (PCC) techniques to check it [20]). 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.

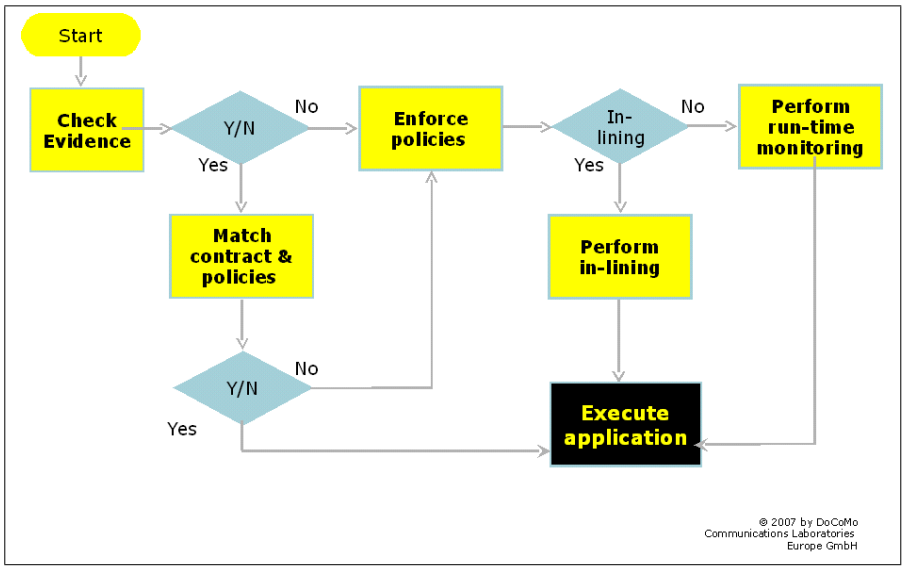


Figure 1: Workflow in Security-by-Contract

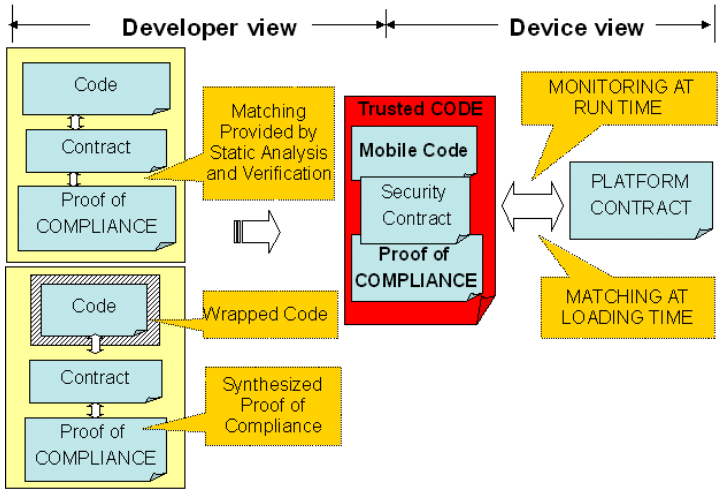


Figure 2: Mobile Code Components with Security-by-Contract

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 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 trustworthy code.

Technically, a contract is a security automaton in the sense of Schneider [13], 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*(also *contract*) 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, 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 2.2. This example can be one of the rules of a policy. Both examples describe safety properties, which are common properties to be verified.

Example 2.1 *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 2.2 *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 [22] if we also want to cover liveness properties as shown in the following Example 2.3.

Example 2.3 *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.*

3 Automata Modulo Theory

The security behaviors, provided by the contract and desired by the policy, can be represented as automata, where transitions corresponds to invocation of APIs as suggested by

Erlingsson [9, p.59] and Sekar et al. [21]. Thus, the operation of matching the midlet's claim with platform policy can be mapped into classical problems in automata theory.

One possible mechanism to represent matching is *language inclusion*: given two automata Aut^C and Aut^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 Aut^C are a subset of the acceptable traces for Aut^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 other alternative is the notion of *simulation*: we have a match when every APIs invoked by Aut^C can also be invoked by Aut^P. In other words, every behavior of Aut^C is also a behavior of Aut^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. We pursue the language inclusion approach in [18] and in this technical report and refer to [19] for the simulation approach.

While this idea of representing the security-digest as an automaton is almost a decade old [21, 9], the practical realization has been hindered by a significant technical hurdle: we cannot use the naive encoding into automata for practical policies. Even the basic policies in Ex. 2.1 and Ex. 2.2 lead to automata with infinitely many transitions.

Fig.3a represents an automaton for Ex. 2.2. We start from state p_0 and stay in this state while PIM is not accessed (jop). As PIM is accessed, we move to state p_1 and stay in state p_1 only if the started connection javax.microedition.io.Connector.open(string url) method is a secure one (url starts with "https://") or we keep accessing PIM (jop). If we start an insecure connection javax.microedition.io.Connector.open(string url), for example url starts with "http://" or "sms://", then we enter state e_p.

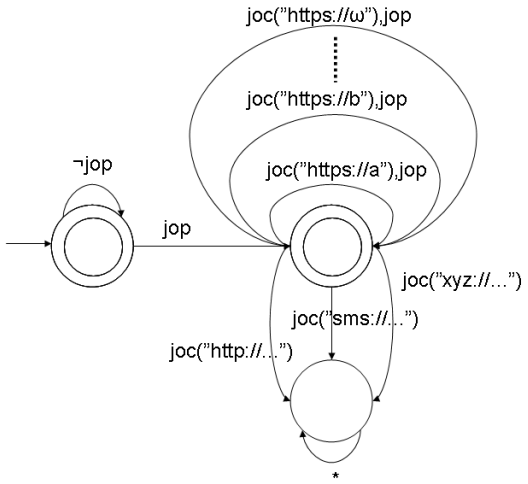
The examples presented are from a Java VM; since we do not consider it useful to invent our own names for API calls, we use the `javax.microedition` APIs (even though verbose) for the notation shown in Fig.3b.

Definition 3.1 (Automaton Modulo Theory ($\mathcal{AMT}$)) *An $\mathcal{AMT}$ is a tuple $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F\rangle$, where 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, $\Delta \subseteq S \times E \times S$ is a labeled transition relation, and $F \subseteq S$ is a set of accepting states.*

Figure 4 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 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

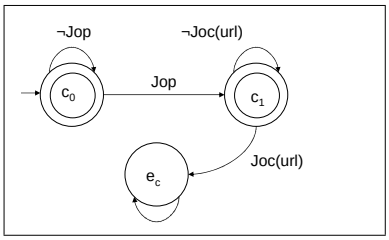


(a) An Infinite Automaton of Ex. 2.2

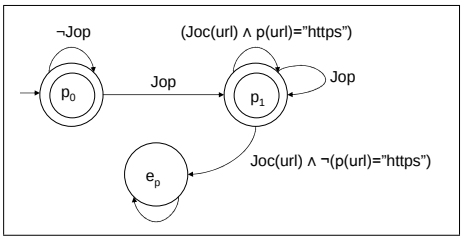
$$
\begin{aligned}
joc(url) &\doteq \texttt{javax.microedition.io.}\\
&\quad\ \texttt{Connector.open}(url)\\
jop &\doteq \texttt{javax.microedition.pim.}\\
&\quad\ \texttt{PIM.openPIMList}(\ldots)\\
p(url) = type &\doteq url.\texttt{startsWith}(type)
\end{aligned}
$$

(b) Abbreviations for Java APIs

Figure 3: Infinite Transitions Security Policies



(a) $\mathcal{AMT}$ rule from Example 2.1



(b) $\mathcal{AMT}$ rule from Example 2.2

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

`Joc,Jop` are predicate symbols representing respectively `joc(url),jop(`$x_1,\ldots,x_n$`)` APIs.

(c) Abbreviations for expressions

Figure 4: $\mathcal{AMT}$ Examples

expression is replaced by the set of transitions corresponding to all satisfiable instantiations of the expression. 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 3.2 ($\mathcal{AMT}$ symbolic run) *Let $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F\rangle$ be an $\mathcal{AMT}$. A*

symbolic run *of A is a sequence of states alternating with expressions $\sigma = \langle s_0 e_1 s_1 e_2 s_2 \ldots \rangle$, 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 $\langle s_0 e_1 s_1 e_2 s_2 \ldots s_{n-1} e_n s_n \rangle$. An* infinite symbolic run *is denoted by $\langle s_0 e_1 s_1 e_2 s_2 \ldots \rangle$. 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 which are actual system traces.

Definition 3.3 ($\mathcal{AMT}$ concrete run) *Let $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ be an $\mathcal{AMT}$. A* concrete run *of A is a sequence of states alternating with a valuation $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \ldots \rangle$, such that:*

1. $s_0 = \mathbf{s_0}$
2. *there exists expressions $e_{i+1} \in E$ 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 $\langle s_0 \nu_1 s_1 \nu_2 s_2 \ldots s_{n-1} \nu_n s_n \rangle$. An* infinite concrete run *is denoted by $\langle s_0 \nu_1 s_1 \nu_2 s_2 \ldots \rangle$. 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 $\sigma_C = \langle s_0 \nu_1 s_1 \nu_2 s_2 \ldots \rangle$ 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 [10] which is slightly different from the one we use in [18], where we use only states.

Example 3.1 *An example of an* accepting symbolic run *of $\mathcal{AMT}$ rule from Example 2.2 shown in Figure 4b is*

c_0 $Jop(jop, file, permission)$ c_1 $Joc(joc, url) \wedge p(url) =$ "https" c_1 $Jop(jop, file, permission)$ c_1 $Joc(joc, url) \wedge p(url) =$ "https" ...

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

$c_0(jop, PIM.CONTACT\_LIST, PIM.READ\_WRITE)$ c_1 $(joc,$ "https://www.esse3.unitn.it/")

$c_1(jop, PIM.CONTACT\_LIST, PIM.READ\_ONLY)$ c_1 $(joc,$ "https://online.unicreditbanca.it/login.htm") ...

Remark 3.1 *A symbolic run defined in Definition 3.2 is interpreted by a non empty set of concrete runs in Definition 3.3. This is a nature of our application domain where security policies define $\mathcal{AMT}$ 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 3.2 can be considered as an abstraction of concrete runs by Definition 3.3.*

The *alphabet* of $\mathcal{AMT}$ is defined as a set of valuations $\mathcal{V}$ that satisfy 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.

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

Definition 3.4 (Deterministic $\mathcal{AMT}$) *$A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ 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 $e_1, e_2 \in E$, 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 $\mathcal{T}$.*

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 [24] as follows:

Definition 3.5 ($\mathcal{AMT}$ Complementation) *Given a deterministic $\mathcal{AMT}$ $A = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ the* complement $\mathcal{AMT}$ automaton $A^c = \langle E, \mathcal{T}, \Sigma, S^c, {\mathbf{s_0}}^c, \Delta^c, F^c \rangle$ *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 $e \in E$*

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

Intersection of $\mathcal{AMT}$ $\mathcal{AMT}$ 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 [24] as follows:

Definition 3.6 ($\mathcal{AMT}$ Intersection) *Let $\langle E^a, \mathcal{T}^a, \Sigma^a, S^a, {\mathbf{s_0}}^a, \Delta^a_{\mathcal{T}}, F^a \rangle$ and $\langle E^b, \mathcal{T}^b, \Sigma^b, S^b, {\mathbf{s_0}}^b, \Delta^b_{\mathcal{T}}, F^b \rangle$ be (non) deterministic $\mathcal{AMT}$, the $\mathcal{AMT}$* intersection automaton *$A^{ab} = \langle E, \mathcal{T}, \Sigma, S, \mathbf{s_0}, \Delta, F \rangle$ is defined as follows:*

1. $E = E^a \cup E^b, \qquad \mathcal{T} = \mathcal{T}^a \cup \mathcal{T}^b, \qquad \Sigma = \Sigma^a \cup \Sigma^b,$

2. $S = S^a \times S^b \times \{1,2\}, \qquad \mathbf{s_0} = \langle {\mathbf{s_0}}^a, {\mathbf{s_0}}^b, 1 \rangle, \qquad F = F^a \times S^b \times \{1\},$

3.

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

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

4 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 Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy. A match is obtained when the language accepted by Aut^C (the execution traces of the midlet) is a subset of the language accepted by Aut^P (the acceptable traces for the policy). The matching problem can be reduced to an emptiness test: $\mathcal{L}_{\mathrm{Aut}^C} \subseteq \mathcal{L}_{\mathrm{Aut}^P} \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C} \cap \overline{\mathcal{L}_{\mathrm{Aut}^P}} = \emptyset \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C} \cap \mathcal{L}_{\overline{\mathrm{Aut}^P}} = \emptyset \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C \times \overline{\mathrm{Aut}^P}} = \emptyset$. In other words, there is no behavior of Aut^C which is disallowed by Aut^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. [6] while modification of this algorithm from Holzmann et al's [14] is considered as state-of-the-art (used in Spin [15]). Gastin et al [11] proposed two modifications to [6] for finding faster and minimal counterexample.

Remark 4.1 *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 1 $\cdots \wedge s^{\overline{\mathbf{P}}} = err^{\overline{\mathbf{P}}} \wedge \ldots$ can be replaced by accepting states from $\overline{Aut^P}$, 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.*

Algorithm 1 check_safety$(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x)$ Procedure

Input: state $s^{\mathbf{C}}$, state $s^{\overline{\mathbf{P}}}$, marker x;

1: $map(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x) :=$ `in_current_path`;
2: **for all** $((s^{\mathbf{C}}, e^{\mathbf{C}}, t^{\mathbf{C}}) \in \Delta^{\mathbf{C}})$ **do**
3: **for all** $((s^{\overline{\mathbf{P}}}, e^{\overline{\mathbf{P}}}, t^{\overline{\mathbf{P}}}) \in \Delta^{\overline{\mathbf{P}}})$ **do**
4: **if** $(DecisionProcedure(e^{\mathbf{C}} \wedge e^{\overline{\mathbf{P}}}) = SAT)$ **then**
5: $y := condition(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x, S^{\mathbf{C}}, S^{\overline{\mathbf{P}}})$
6: **if** $(map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) =$ `in_current_path` $\wedge ((\mathtt{s}^{\mathbf{C}} \in \mathtt{S}^{\mathbf{C}} \wedge \mathtt{s}^{\overline{\mathbf{P}}} = \mathtt{err}^{\overline{\mathbf{P}}} \wedge \mathtt{x} = 1) \vee (\mathtt{t}^{\mathbf{C}} \in \mathtt{S}^{\mathbf{C}} \wedge \mathtt{t}^{\overline{\mathbf{P}}} = \mathtt{err}^{\overline{\mathbf{P}}} \wedge \mathtt{y} = 1)))$ **then**
7: report policy violation;
8: **else if** $(map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) =$ `in_current_path` $\wedge ((\mathtt{s}^{\mathbf{C}} \in \mathtt{S}^{\mathbf{C}} \wedge \mathtt{s}^{\overline{\mathbf{P}}} \in (\mathtt{S}^{\overline{\mathbf{P}}} \backslash \{\mathtt{err}^{\overline{\mathbf{P}}}\}) \wedge \mathtt{x} = 1) \vee (\mathtt{t}^{\mathbf{C}} \in \mathtt{S}^{\mathbf{C}} \wedge \mathtt{t}^{\overline{\mathbf{P}}} \in (\mathtt{S}^{\overline{\mathbf{P}}} \backslash \{\mathtt{err}^{\overline{\mathbf{P}}}\}) \wedge \mathtt{y} = 1)))$ **then**
9: report availability violation;
10: **else if** $(map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) =$ `safe`$)$ **then**
11: **check_safety**$(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y)$;
12: **end if**
13: **end if**
14: **end for**
15: **end for**
16: **if** $(s^{\mathbf{C}} \in S^{\mathbf{C}} \wedge s^{\overline{\mathbf{P}}} \in S^{\overline{\mathbf{P}}} \wedge x = 1)$ **then**
17: **check_availability**$(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x)$;
18: $map(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x) :=$ `availability_checked`;
19: **else**
20: $map(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x) :=$ `safety_checked`;
21: **end if**

Algorithm 2 check_availability$(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x)$ Procedure

Input: state $s^{\mathbf{C}}$, state $s^{\overline{\mathbf{P}}}$, marker x;

1: **for all** $((s^{\mathbf{C}}, e^{\mathbf{C}}, t^{\mathbf{C}}) \in \Delta^{\mathbf{C}})$ **do**
2: **for all** $((s^{\overline{\mathbf{P}}}, e^{\overline{\mathbf{P}}}, t^{\overline{\mathbf{P}}}) \in \Delta^{\overline{\mathbf{P}}})$ **do**
3: **if** $(DecisionProcedure(e^{\mathbf{C}} \wedge e^{\overline{\mathbf{P}}}) = SAT)$ **then**
4: $y := condition(s^{\mathbf{C}}, s^{\overline{\mathbf{P}}}, x, S^{\mathbf{C}}, S^{\overline{\mathbf{P}}})$
5: **if** $(map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) =$ `in_current_path`$)$ **then**
6: **if** $(t^{\overline{\mathbf{P}}} = err^{\overline{\mathbf{P}}})$ **then**
7: report policy violation;
8: **else**
9: report availability violation;
10: **end if**
11: **else if** $(map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) =$ `safety_checked`$)$ **then**
12: $map(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y) :=$ `availability_checked`
13: **check_availability**$(t^{\mathbf{C}}, t^{\overline{\mathbf{P}}}, y)$;
14: **end if**
15: **end if**
16: **end for**
17: **end for**

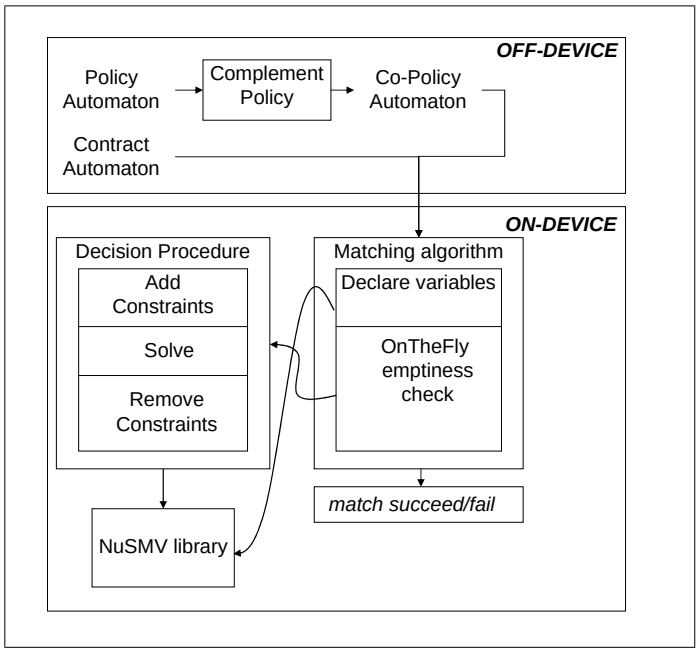


Figure 5: Contract-Policy Architecture

5 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 [5] integrated with its MathSAT libraries [4]. We provide an overview of how the prototype is implemented for 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. Detailed class diagram is available on Appendix as Figure 8.

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 [23] 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 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 6.

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 [1] 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 $\mathcal{AMT}$ś 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.

6 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 irrelevant.

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 ex-

pressions 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. 6d) of the interactions between the solver and the on-the-fly emptiness check algorithm.

Table 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

SC: Number of States Contract TC: Number of Transitions Contract
SP: Number of States Policy TP: Number of Transitions Policy

(a) Abbreviations

7 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 1 some sample possible combinations of policy-contract (mis)matching pairs. For instance, the contract pimNoConn_contract.pol represents Example 2.1 and the policy pimSecConn_policy.pol corresponds to Example 2.2.

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 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 $\mathcal{AMT}$ for each contract and policy in Table 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 2. That is why we only present here the number of reachable states in Table 1 and number of visited states during on-the-fly running in Table 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 2. The results for all design alternatives are mapped into diagram shown in Figure 6a for matching problems and Figure 6c for not matching problems. Notice that we only provide the cumulative running time that is

Table 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

ART: Average Runtime for 10 runs SV: Number of Visited States
CRT: Cumulative Average Runtime TV: Number of Visited Transitions

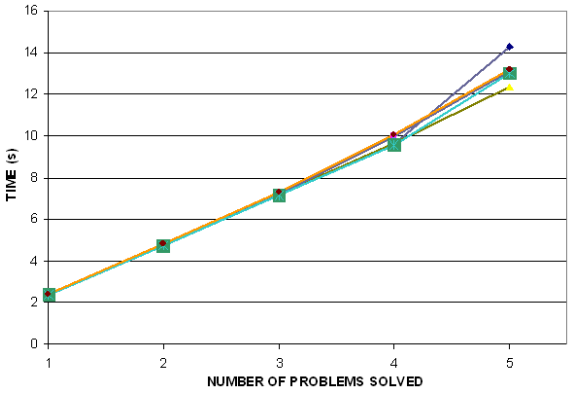
(a) Abbreviations

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.

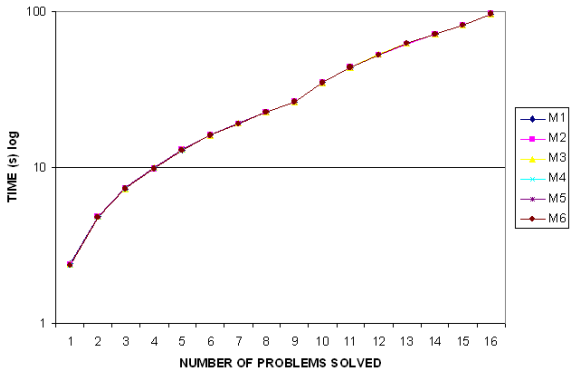
We singled out P100 as a challenging artificial problem because it has a large number of states compared to the others: essentially this happened because we draw an automaton modulo theory with 100 states and which traverse from one state to another by adding 1 to the number of SMS sent.

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, and 100 for both contract and policy. The data of the experiment is given on Appendix B. The generated cases cumulative running time of implementation is propositional to the number of problems solved (see Figure 6b). In this case the difference among M1 until M8 is negligible as can be seen from Figure 6b that the results construct almost a line.

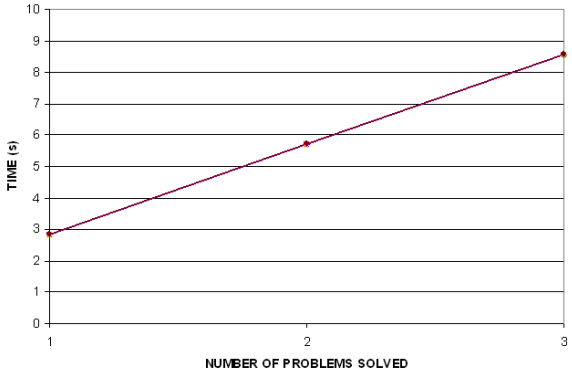
All methods seem to perform equally well because the problems are not stressful enough for the different configurations. This is actually a promising result for the deployment to the resource constrained in mobile device domain. Therefore, we have implemented the same algorithm for the mobile platform HTC P3600 (3G PDA phone). We run the problem suit of P1-P8 and P100 with MUTEX_MC ONE_INSTANCE CACHING_SOLVER configuration. Table 2 shows the results on device, where the runtime of every single problem running is longer than on Desktop PC. This result is obvious 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 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;



(a) Match succeeds for real policies



(b) Matches among synthetic contracts and policies

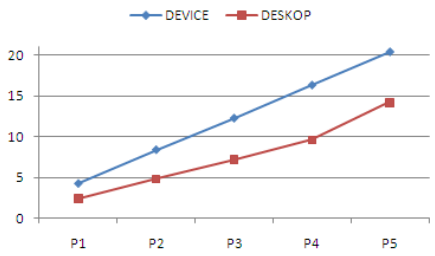


(c) Match fails for real policies

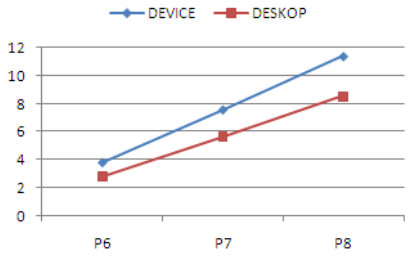
M1: MUTEX_MC ONE_INSTANCE CACHING_SOLVER
M2: MUTEX_SOLVER ONE_INSTANCE CACHING_SOLVER
M3: PRIORITY_MC ONE_INSTANCE CACHING_SOLVER
M4: MUTEX_MC ONE_INSTANCE CACHING_MC
M5: MUTEX_SOLVER ONE_INSTANCE CACHING_MC
M6: PRIORITY_MC ONE_INSTANCE CACHING_MC

(d) Abbreviations for Configurations

Figure 6: Cumulative response time of matching algorithm on Desktop PC



(a) Match succeeds



(b) Match fails

Figure 7: Cumulative response time of matching algorithm on the Mobile Device

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. 7a for matching and Figure 7b for not matching.

Current implementation uses PRIORITY_MC ONE_INSTANCE CACHING_MC configuration. PRIORITY_MC is preferred due to the nature of rules in policies which is *priority or*, also

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 to save calls to solver for the already solved rules.

Acknowledgement

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.
The EU-FP6-IST-STREP-S3MS project for partly supporting this research.

A On-the-fly Matching Prototype Class Diagram

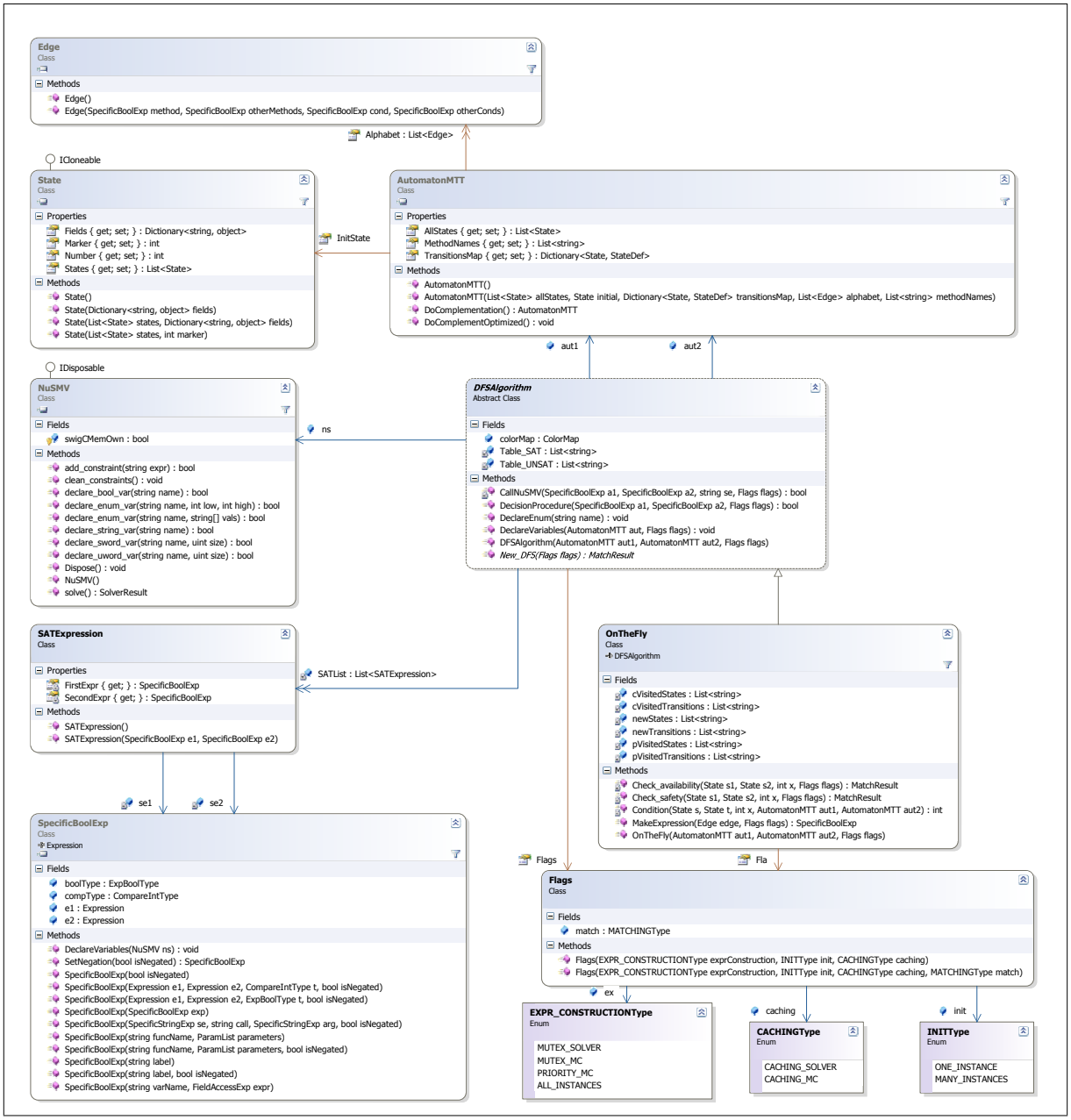


Figure 8: On-the-fly Class Diagram

B On-the-fly Matching Prototype Experiments

Table 3: Problems Suit

Problem	**Contract**	**Policy**
P100-100	100SMS_contract.pol	100SMS_policy.pol
P100-10	100SMS_contract.pol	100SMS_policy.pol
P100-1	100SMS_contract.pol	100SMS_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 4: 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

References

[1] 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.

[2] J. Bacon. Toward pervasive computing. *IEEE Pervasive Comp. Magazine*, 1(2):84, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] N. Dragoni, F. Massacci, C. Schaefer, T. Walter, and E. Vetillard. A security-by-contracts architecture for pervasive services. In *Proc. of the 3rd Int. Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*. IEEE Press, 2007.

[9] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2004.

[10] 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.

[11] 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.

[12] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*. Addison-Wesley Professional, 2003.

[13] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[14] 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.

[15] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[16] B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.

[17] S. Konomi M. Arikawa and K. Ohnishi. Navitime: Supporting pedestrian navigation in the real world. pages 21–29, 2007.

[18] 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.

[19] 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.

[20] 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.

[21] R. Sekar, V.N. Venkatakrishnan, 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.

[22] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.

[23] 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.

[24] 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.

[25] 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.