Runtime Security Verification for Itinerary-Driven Mobile Agents

Zijiang Yang *
Western Michigan University
Kalamazoo, Michigan, USA
zijiang.yang@wmich.edu

Shiyong Lu †
Wayne State University
Detroit, Michigan, USA
shiyong@wayne.edu

Ping Yang
Binghamton University
Binghamton, New York, USA
pyang@cs.binghamton.edu

Abstract

We present a new approach to ensure the secure execution of itinerary-driven mobile agents, in which the specification of the navigational behavior of an agent is separated from the specification of its computational behavior. We empower each host with an access control policy so that the host will deny the access from an agent whose itinerary does not conform to the host's access control policy. A host uses model checking algorithms to check if the itinerary of the agent conforms to its access control policy written in μ-calculus, and if so, grant access permission. In order to address the state explosion problem for model checking itineraries, we propose an approach called Model Generation Code. In this approach, instead of verifying the itinerary itself, a host actually checks the conservative models of a mobile agent. If a conservative model does not satisfy the host's access control policy, the mobile agent will provide refined models for further verification. Our preliminary results show that this is a practical and promising approach to ensure the secure execution of mobile agents.

1 Introduction

Computing is increasingly characterized by the global scale of applications. Two main features of the *global ubiquitous computing* paradigm are *mobility*, where code is dispatched from site to site to increase flexibility and expressibility, and *openness*, which reflects the nature of global networks. Ubiquitous computing provides numerous advantages over the conventional distributed computing paradigm and starts to have applications in distributed computing, network management, and E-commerce. However, *mobility control* and *security* arise as two issues from massive code and resource migrations and openness, respectively.

*This work was supported by funds from the Faculty Research and Creative Activities Support Fund, Western Michigan University.

†This work was supported by Wayne State University Faculty Research Award 145768.

Although most mobile agent systems [18, 9, 16] provide the primitive for specifying mobility, they prescribe the navigational and computational behavior of a mobile agent in one mixed specification. To address the concern of mobility control, we propose an itinerary-driven system in which the specification of the navigational behavior of an agent is separated from the specification of the computational behavior. The benefit of this separation is immediate: not only we can reason about the correctness and security properties about itineraries even before the computation is defined, but also we can reuse itineraries and computations across different agents.

In this paper, we aim to provide a runtime verification mechanism to ensure the secure execution of itinerary-driven mobile agents. In particular, we empower each host with a control capability; the host will deny access to an agent whose *itinerary* does not conform to the host's *access control policy* (ACP). The conformance check is performed by model checking, a formal verification technique.

In order to address the state explosion problem for model checking itineraries, we propose an approach called *model generation code* (MGC), in which a host verifies conservative models instead of mobile agent. The mobile agent is safe if one of its model satisfies the ACP. In the case that a model violates the ACP, a mobile agent is allowed to refine its model. In this proof-of-concept paper, we develop model checking algorithms to verify the models of itineraries. However, The procedures to generate models from an itinerary (*model generation*) and refine a model after an ACP violation (*model refinement*) are being performed manually.

Related Work. To address the security concerns in mobile agent systems, several approaches have been developed. The two approaches that are the closest to ours are the proof-carrying code (PCC) [11] and the model-carrying code (MCC) [14].

PCC [11] enables the safe execution of mobile agents from unknown sources by requiring the mobile agent to carry a proof regarding its safety. A host can mechanically check the correctness of this proof and execute the code

only if the proof is correct. However, it is very difficult to develop machine checkable proofs. Although it is possible to generate proofs automatically for simple properties such as memory safety [12], automatic proof generation for more complex properties is a daunting problem. In addition, the mobile agent has to know all the ACPs at different hosts in order to generate proofs. This is an unrealistic assumption since security needs vary across different hosts.

MCC [14] offers an alternative solution by requiring a mobile agent to carry a model of its code. The model will be checked against the ACPs at the host side by using logic programs. However, the MCC approach still suffers from the following problems. First, the model is generated without the knowledge of the ACPs used by the host that the agent is going to visit. Since security needs vary across different hosts, the model can be too coarse for one host and too fine for another. Second, MCC places burden on hosts to refine ACPs manually if the security check fails. Such an approach is not practical because a user at the host may not know how to modify the ACPs or does not have privileges to do so.

Structure of the paper. Section 2 presents our novel mobile system architecture. The syntax and semantics of the itinerary language used in this paper are defined in Section 3. Section 4 describes the complete run-time verification system that includes model generation, model checking, model refinement, and specification language for access control policies. The experimental data is shown in Section 5. Finally we conclude the paper with Section 6.

2 Overall Mobile System Architecture

Figure 1 shows the overall system architecture for mobile agent systems in which the specification of the navigational behavior of a mobile agent is separated from the specification of its computational behavior.

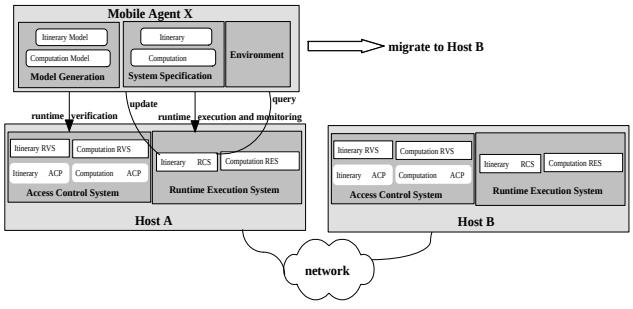

Figure 1. A novel architecture for an itinerary-driven mobile agent system

A mobile agent X consists of the following three components: (1) *System specification*. The system specification is a pair (i, c), where i is an itinerary and c is the computation of the agent that is specified with a traditional programming language such as Java. An agent is deployed from a home host, and after a host-dependent computation is completed, the agent will migrate to the next host based on an execution of the specified itinerary. (2)*Model generation*. In order to reduce the computational cost of runtime verification, model generators will produce sound abstractions of the system specification based on mobile agent and the access control policy at a host. (3)*Environment*. Environment consists of a set of environment variables which are used in the condition of a conditional itinerary.

Each host is equipped with a mobile agent virtual machine to support the secure execution of mobile agents. The virtual machine consists of the following two components: (1) *Runtime Verification System (RVS)*. The runtime verification system of a host contains itinerary ACPs and computation ACPs to control the host visit and resource access privileges of mobile agents. When a host B receives a visit request from a mobile agent X, host B's itinerary RVS will verify at runtime if the itinerary model provided by agent X satisfies host B's itinerary ACP. If it does, then agent X will be permitted to access host B; otherwise, the access request fails. After agent X migrates to host B, B's computation RVS will verify at runtime if the computation model provided by agent X satisfies host B's computation ACP. If it does, then agent X can execute its computation specification; otherwise, none of agent X's computation specification will be executed. (2)*Runtime Execution System (RES)*. The runtime execution system of a host consists of two subsystems: itinerary Runtime Compile System (RCS) and Computation RES. Itinerary RCS compiles and executes an itinerary at runtime, calculates its residue itinerary which corresponds to the unexecuted component of the original itinerary, and then changes the original itinerary to the residue itinerary. In addition, for a conditional itinerary, the itinerary RCS will query the environment component of the agent about the values of the environment variables that appear in the condition, and decides which branch of the itinerary should be executed. On the other hand, the computation RES executes the computation specification of a mobile agent. To prevent a malicious mobile agent from accessing resources of the host, the computation RES also monitors the execution with respect to the models so that the ACPs will never be violated.

3 Itinerary Language and Itinerary Access Control Policy

In this section we present the syntax and semantics of an itinerary language, and itinerary access control policy expressed using μ-calculus.

3.1 Itinerary Language

The itinerary language prototype considered in this paper is adopted from the MAIL language proposed in [10]. The following BNF notation shows the syntax.

$$\begin{aligned} i \quad ::= \quad & \text{end} \mid s \mid s?x \mid s!x \mid i_1; i_2 \mid i_1 \parallel i_2 \mid \text{comp} \\ & \text{if } c \text{ then } i_1 \text{ else } i_2 \mid \text{while } c \text{ do } i \end{aligned}$$

Basically, end models an empty itinerary; s represents a single host visit; $s?x$ represents that the agent receives a value from site s and $s!x$ represents that the agent sends a value to site s; $i_1; i_2$ and $i_1||i_2$ represent sequential and parallel visit patterns, respectively; comp denotes the syntax of computation such as $i = i + 1$, which is skipped here; if c then i_1 else i_2 represents a conditional visit and while c do i a visit loop. Although simple, the prototype is structured and compositional so that an itinerary can be constructed recursively from primitive constructs.

1.	Migration Rule:	$\frac{}{s \xrightarrow{\text{visit}(s)} end}$
2.	Input Rule:	$\frac{}{s?x \xrightarrow{s?x} end}$
3.	Output Rule:	$\frac{}{s!x \xrightarrow{s!x} end}$
4.	Sequential Rule 1:	$\frac{i \xrightarrow{\alpha} i'}{end; i \xrightarrow{\alpha} i'}$
5.	Sequential Rule 2:	$\frac{i_1 \xrightarrow{\alpha} i_1'}{i_1; i_2 \xrightarrow{\alpha} i_1'; i_2}$
6.	Condition Rule 1:	$\frac{c = true \wedge i_1 \xrightarrow{\alpha} i_1'}{\text{if } c \text{ then } i_1 \text{ else } i_2 \xrightarrow{\alpha} i_1'}$
7.	Condition Rule 2:	$\frac{c = false \wedge i_2 \xrightarrow{\alpha} i_2'}{\text{if } c \text{ then } i_1 \text{ else } i_2 \xrightarrow{\alpha} i_2'}$
8.	Parallel Rule 1:	$\frac{i_1 \xrightarrow{\alpha} i_1'}{i_1 \| i_2 \xrightarrow{\alpha} i_1' \| i_2}$
9.	Parallel Rule 2:	$\frac{i_1 \xrightarrow{\alpha} i_2'}{i_1 \| i_2 \xrightarrow{\alpha} i_1 \| i_2'}$
10.	Parallel Rule 3:	$\frac{i \xrightarrow{\alpha} i'}{end \| i \xrightarrow{\alpha} i'}$
11.	Parallel Rule 4:	$\frac{i \xrightarrow{\alpha} i'}{i \| end \xrightarrow{\alpha} i'}$
12.	Loop Rule 1:	$\frac{c = true}{\text{while } c \text{ do } i \xrightarrow{\epsilon} i; \text{while } c \text{ do } i}$
13.	Loop Rule 1:	$\frac{c = false}{\text{while } c \text{ do } i \xrightarrow{\epsilon} end}$

Figure 2. Operational semantics

The structural operational semantics is shown in Figure 2. The semantics describes the operational behavior of an agent with respect to a given itinerary. Each rule in the semantics is of the form $\frac{\text{premises}}{\text{conclusion}}$. The premise is a condition or is specified in terms of a labeled transition system $(\mathcal{S}, \Sigma, \rightarrow)$, where $\mathcal{S}$ is a set of states (i.e. itinerary expressions), Σ is a set of labels (actions), and $\rightarrow \subseteq S \times \Sigma \times S$ is a set of transitions. The conclusion is specified in terms of a labeled transition system. We define two actions: ϵ which represents an empty action, and visit(s) which represents that the agent visits site s. Rule 1 is an axiom (empty premise). It says that, given a primitive itinerary s, the agent performs action visit(s) and migrates to site s. Rules 2 and 3 say that the agent performs an input action $s?x$ and an output action $s!x$, respectively. Rule 4 says that the itinerary $end; i$ is equivalent to i where end signifies the end of an itinerary (or a sub-itinerary) in the transition. Rule 5 describes the semantics of executing a sequential composition of two itineraries i_1 and i_2. The rule says that if an agent can perform an α transition according to the itinerary i_1 and the updated itinerary is i_1', then, given an itinerary $i_1; i_2$, the agent can perform an α transition and the itinerary is updated to $i_1'; i_2$. Rules 6 and 7 say that the agent will use i_1 as the itinerary if the condition c holds, and use i_2 otherwise. Rules 8 and 9 enable an agent to visit different sites concurrently by making clones of the agent; the order of the visits does not matter. Rules 10 and 11 deal with the merging of cloned agents after they complete the tasks. Rules 12 and 13 state that the loop will continue if c holds and terminate otherwise.

3.2 Itinerary Access Control Policy

Access control policies are specified using alternation-free fragment [6] of the modal μ-calculus [8], which is an expressive temporal logic. A μ-calculus formula is alternation-free if no mutual recursion exists between greatest and least fixed-point operations. Let $\mathcal{F}$ denote the set of formula expressions; $\mathcal{A}$ the set of actions; $\mathcal{C}$ the set of conditions; and $\mathcal{E}$ the set of fixed-point equations. The syntax of the μ-calculus is as follows:

$$\begin{aligned} \mathcal{A} \quad ::= \quad & \text{visit}(s) \mid \epsilon \\ \mathcal{F} \quad ::= \quad & tt \mid ff \mid \mathcal{C} \mid \mathcal{F} \vee \mathcal{F} \mid \neg F \mid \mathcal{F} \wedge \mathcal{F} \mid \\ & \langle \mathcal{A} \rangle \mathcal{F} \mid [\mathcal{A}] \mathcal{F} \mid X \\ \mathcal{E} \quad ::= \quad & \mu X.\mathcal{F} \mid \nu X.\mathcal{F} \end{aligned}$$

tt and ff stand for propositional constants true and false, respectively. The operators $\wedge$, $\vee$ and $\neg$ are boolean connectives. $\langle\ \rangle$ and $[\ ]$ are dual operators, which represent diamond (existential) and box (universal) modalities, respectively. For instance, $\langle\alpha\rangle\varphi$ means that, possibly after an α transition, the formula φ holds. $[\alpha]\varphi$ means that, necessarily after an α transition, the formula φ holds. X is a formula variable.

$\mu X.\mathcal{F}$ and $\nu X.\mathcal{F}$ represent the least and greatest fixed point formulas, respectively. The greatest and least fixed points can be used to specify *safety* (something bad never happens) and *liveness* (something good eventually happens) properties, respectively. For instance, the greatest fixed point formula $\nu X.\varphi \wedge [\alpha]X$ specifies the property "φ is true along every α-path". The least fixed point formula

1:	$[\![\mathcal{C}]\!]\mathcal{V}$	=	$\begin{cases} \{S \mid S \in \mathcal{S}\} & \text{if } \mathcal{C} \text{ holds} \\ \emptyset & \text{otherwise.} \end{cases}$
2:	$[\![\varphi_1 \vee \varphi_2]\!]\mathcal{V}$	=	$[\![\varphi_1]\!]\mathcal{V} \cup [\![\varphi_2]\!]\mathcal{V}$
3:	$[\![\varphi_1 \wedge \varphi_2]\!]\mathcal{V}$	=	$[\![\varphi_1]\!]\mathcal{V} \cap [\![\varphi_2]\!]\mathcal{V}$
4:	$[\![\langle\alpha\rangle\varphi]\!]\mathcal{V}$	=	$\{P \mid \exists P'.P \xrightarrow{\alpha} P' \wedge P' \in [\![\varphi]\!]\mathcal{V}\}$
5:	$[\![[\alpha]\varphi]\!]\mathcal{V}$	=	$\{P \mid \forall P'.P \xrightarrow{\alpha} P' \Rightarrow P' \in [\![\varphi]\!]\mathcal{V}\}$
6:	$[\![\neg\varphi]\!]\mathcal{V}$	=	$\mathcal{S} - [\![\varphi]\!]$
7:	$[\![(\mu X.\varphi)]\!]\mathcal{V}$	=	$\bigcap\{S \subseteq \mathcal{S} \mid S \subseteq [\![\varphi]\!]_{\mathcal{V}[X:=S]}\}$
7:	$[\![(\nu X.\varphi)]\!]\mathcal{V}$	=	$\bigcup\{S \subseteq \mathcal{S} \mid S \supseteq [\![\varphi]\!]_{\mathcal{V}[X:=S]}\}$

Figure 3. Semantics of the μ-Logic

$\mu X.\varphi \vee \langle\alpha\rangle X$ specifies the property "there exists an α-path such that φ eventually holds". The technical details and more examples on fixed points are provided in [3].

Example 1 *Some examples of ACPs in μ-calculus are as follows:*

- *Policy $\mu X.[visit(e)]tt \vee [visit(f)]tt$ grants access permission to an agent iff the agent will visit either e or f next.*
- *Policy $\nu X.([visit(e)]ff \wedge [-]X)$ grants access permission to an agent iff the agent will never visit e in the future.*

The semantics of the modal μ-calculus is given in Figure 3. The semantics function $[\![\varphi]\!]\mathcal{V} \subseteq \mathcal{S}$ returns the set of itineraries that satisfy φ and is given by induction on the structure of the formula φ. Function $\mathcal{V}$ is an environment that maps formula variables to subsets of states $\mathcal{S}$.

Satisfaction relation $\models$ between a set of itineraries P and an ACP φ is defined as $P \models \varphi$. $P \models \varphi$ if and only if $P \in [\![\varphi]\!]\mathcal{V}$ for all $\mathcal{V}$.

4 Runtime Security Verification

In this section, we introduce our runtime security verification mechanism. Although the architecture presented in Section 2 considers access control for both itinerary and computation code, in this paper we focus on the run-time verification system for itineraries and itinerary ACPs.

4.1 Runtime Verification Flow

Figure 4 shows a flow of our runtime security verification with dynamic model generation, which is described as follows.

1. When visiting a host, the mobile agent first requests for the ACP P of the host.

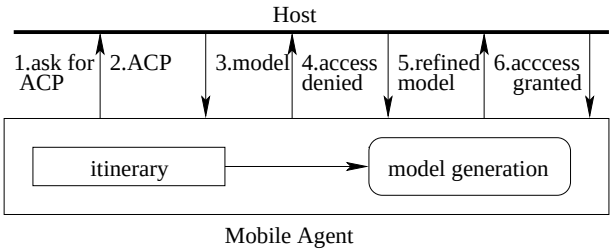

Figure 4. Runtime security verification flow

2. After receiving P, the agent creates a model M_0 based on the ACP and its own itinerary I. Let $B(M_0)$ denote the set of behaviors of M_0 and $B(I)$ the set of behaviors of I, it is guaranteed that $B(I) \subseteq B(M_0)$.
3. M_0 is submitted to the host for verification.
4. The access request is rejected because $M_0 \not\models P$. Along with the decision, host provides a behavior b_0 of M_0 that violates P as the reason for the rejection.
5. Since M_0 is an abstraction of I, it is possible that $M_0 \not\models P$ and $I \models P$. If it discovers that $b_0 \in M_0$ and $b_0 \notin I$, the mobile agent refines M_0 to get M_1 such that $b_0 \notin M_1$. That is, the agent will not be rejected again for the *same* reason. On the other hand, if $b_0 \in I$, the rejection is final. In this example, the refined model M_1 is submitted to the host for verification.
6. The model checking engine at the host proves $M_1 \models P$. Since M_1 is a conservative model ($B(I) \subseteq B(M_1)$), $M_1 \models P \implies I \models P$. The access request is granted.

Two things may happen if the mobile agent is malicious. Firstly, a model provided by the agent is not a sound abstraction of the actual code. This cannot happen by using our model generation and refinement algorithms. However, a malicious agent may choose to deliberately modify the model. To deal with this problem, the Runtime Execution system shown in Figure 1 is equipped with a runtime monitoring mechanism which intercepts model-relevant itinerary events and matches them against models. The idea was proposed in [15, 2] and experiments showed less than a 5% overhead to the execution time of most programs. Although their runtime monitoring system operates on system calls, we can adapt it to work on itinerary in our system. With the runtime monitoring mechanism, we can detect the discrepancy if the actual behavior of the agent does not follow the provided model and therefore terminate the execution. Secondly, a malicious agent might keep submitting models to cause denial-of-service at the host. To prevent this attack, we limit the number of times N that an agent is allowed to

refine its model. If N is greater than a predefined threshold, the agent will not be allowed to refine the model further.

There are three possible outcomes when a mobile agent tries to access a host. Firstly, the model generated by the agent satisfies the ACP. In this case, the mobile agent is safe and can be accepted. Secondly, the model violates the ACP and the mobile agent cannot provide a refined model. In this case the mobile agent violates the ACP and the agent has to be rejected by the host. Finally, the verification engine at the host cannot give a conclusive result. This is due to either one verification instance requires too much computational resource that the host cannot afford for, or there are too many verification requests from a mobile agent. In this case, the mobile agent will be rejected.

4.2 Dynamic Model Generation

We illustrate the idea of dynamic model generation and refinement by a running example. Algorithm 1 shows the itinerary of a buyer agent X who wants to buy products at either host S_1 or S_2 at a low price. X's strategy is to negotiate price with both S_1 and its competitor S_2, and use their offers to negotiate further with each other. Since the quality at S_1 is a little better than that at S_2, X slightly prefers S_1.

Algorithm 1 NEGOTIATE_ITINERARY()

```
1:  visit_1 = visit_2 = visit_B = 0;
2:  S_1 ? price_1, quant_1;
3:  visit_1 + +;
4:  S_2 ? price_2, quant_2;
5:  visit_2 + +;
6:  B ! price_1, quant_1, price_2, quant_2;
7:  visit_B + +;
8:  while price_1 > price_2 AND visit_1 == visit_2 do
9:      S_1 ? price_1;
10:     visit_1 + +;
11:     B; α ! price_1;
12:     visit_B + +;
13:     if price_1 ≤ price_2 then
14:         S_2 ? price_2;
15:         visit_2 + +;
16:         B ! price_2;
17:         visit_B + +;
18:     end if
19: end while
20: B;
21: visit_B + +;
```

The variables $visit_1, visit_2$ and $visit_B$, initially set to 0(line 1), are used to keep track of how many times the hosts S_1, S_2 and B have been visited, respectively. Then X gets the price and quantity information at both hosts S_1, S_2, and increase the values of $visit_1, visit_2$ accordingly (lines 2-5). Next the information is reported back to its home host B (lines 6-7). After obtaining the initial information, X continuously uses S_2's low price to negotiate for even lower price from S_1(lines 8-19). Each time X obtains a new price from S_1, it reports the new information to B (lines 9-12); and if the new price from S_1 becomes lower than the previous offer from S_2, it will use the information to negotiate a better deal from S_2 (lines 13-18). After X cannot get better deal, it returns home (lines 20-21).

In this example, S_1 allows a mobile agent to visit its competitor. However, it does not allow an agent to visit itself twice in a row without visiting any other hosts in between. The reason is obvious: S_1 does not want to beat its own price. The ACP at S_1 can be represented as the negation of the following μ-calculus formula: $\mu X.\langle visit(s)\rangle\langle visit(s)\rangle tt \vee \langle -\rangle X$. Here, $\langle -\rangle$ is a wildcard action.

4.2.1 Itinerary Slicing

The first step in model generation is *itinerary slicing*, in which the itinerary code that is irrelevant to a policy is removed. Similar to program slicing, we consider control and data dependency in an itinerary.

A statement m is *control dependent* on a conditional statement n (e.g. `if` n or `while` n) in an itinerary if there exist two paths from n such that one visits m and the other dose not. A statement m is *data dependent* on a statement n in an itinerary if the value of any variable used in m is updated in n and there exist a path from n to m.

Our itinerary slicing algorithm first marks only those hosts that appear in an ACP, then recursively marks all the statements that are in the control or data dependency closure. Any statements that are not marked when the algorithm terminates will be removed. Range analysis techniques [19] can also be applied in itinerary slicing to reduce the range of variables.

Algorithm 2 NEGOTIATE_ITINERARY()

```
1:  visit_1 = visit_2 = 0;
2:  S_1 ? price_1;
3:  visit_1 + +;
4:  S_2 ? price_2;
5:  visit_2 + +;
6:  while price_1 > price_2 AND visit_1 == visit_2 do
7:      S_1 ? price_1;
8:      visit_1 + +;
9:      if price_1 ≤ price_2 then
10:         S_2 ? price_2;
11:         visit_2 + +;
12:     end if
13: end while
```

In our example, the values of $quant_1, quant_2$, and the visit to home host B in Algorithm 1 will not be marked. Algorithm 2 shows the sliced itinerary with respect to property P. Note that the sliced itinerary is equivalent to the original itinerary with respect to the ACP.

4.2.2 Abstraction

Although itinerary slicing has resulted in a *cone of influence* of P that is smaller than the original code, in many cases a

model checking on the cone is still costly in a run-time environment. In the case there are infinite type variables, such as $visit_i$ and $price_i$ in Algorithm 2, our model checking algorithm cannot be applied directly.

Predicate abstraction [7, 5, 1], a special form of abstraction interpretation [4], is a technique that is used to prove properties of infinite state systems. In our itinerary language, the infiniteness comes from the environment variables. Since only hosts appear in ACPs, the effect of environment variables on policies is through the conditions of the branch statements. If we consider both the possibilities at a branch statement, we are actually checking a superset of actual itinerary behaviors. Therefore, a conservative finite state abstraction is generated if we replace the conditions that involve infinite variables with non-determinism.

Algorithm 3 NEGOTIATE_ITINERARY()

```
1: S1;
2: S2;
3: while * do
4:     S1;
5:     if * then
6:         S2;
7:     end if
8: end while
```

Algorithm 3 shows the abstraction M_0 of the sliced itinerary. Here * denotes a non-deterministic value tt or ff. Note the state space of the abstraction is finite (* is Boolean), while the state space of the sliced itinerary is still infinite ($price_1$ and $price_2$ have infinite domain integer). Our model checking algorithms proves that $M_0 \not\models P$ because the trace $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 4$ violates the ACP.

4.2.3 Refinement

The models obtained after abstraction serve as sound abstractions of the actual itinerary, which means that a security violation reported by a model checker might be spurious. Let $\mathcal{B}(I), \mathcal{B}(M_0)$ and $\mathcal{B}(P)$ denote the set of all behaviors of the itinerary I, abstract model M_0 and property P respectively. Our abstraction algorithm guarantees that $\mathcal{B}(I) \subseteq \mathcal{B}(M_0)$. Therefore, if the model checking procedure shows $\mathcal{B}(M_0) \subseteq \mathcal{B}(P)$, the itinerary is safe and can be accepted by the host. However, if $\mathcal{B}(M_0) \not\subseteq \mathcal{B}(P)$, the result is inconclusive since we can conclude neither $\mathcal{B}(I) \subseteq \mathcal{B}(P)$ nor $\mathcal{B}(I) \not\subseteq \mathcal{B}(P)$.

In this case, we refine the abstraction M_0. The refinement procedure first determine whether the counterexample exists in I. If it does, I violates P and therefore has to be rejected. Otherwise, the procedure eliminates the spurious bug and refine the abstract model.

An example of the refinement procedure is shown in Figures 5. The rectangle marked by ACP shows the set of behaviors allowed by an ACP. The small circle marked

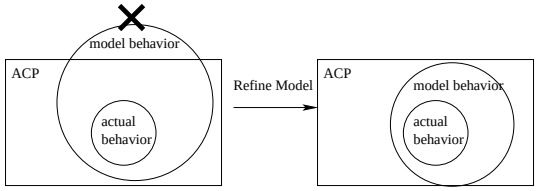

Figure 5. Model refinement

by `actual behavior` shows the set of behaviors of an itinerary. Since the behaviors of the itinerary is a subset of the ACP, there is no violation. However, because the actual itinerary cannot be checked directly, an abstraction is created (big circle marked by `model behavior`). As indicated on the left in Figure 5, the abstract model violates the ACP because the model behavior is not fully contained in the allowed behavior. However, the behavior that violates the policy is spurious since it is introduced in the abstraction procedure. The goal of the refinement procedure is to eliminate spurious behaviors from the model while preserving simplicity. The right figure illustrates an ideal algorithm that removes all the offensive behaviors from the initial abstract model and results in a refined model whose behavior is a subset of the ACP. Since the refined model is safe and it is a superset of the itinerary, the actual itinerary is safe with respect of the ACP.

Algorithm 4 NEGOTIATE_ITINERARY()

```
1: equal_visit = tt;
2: S1;
3: equal_visit = ff;
4: S2;
5: equal_visit = tt;
6: while * AND equal_visit do
7:     S1;
8:     equal_visit = ff;
9:     if * then
10:        S2;
11:        equal_visit = tt;
12:    end if
13: end while
```

Algorithm 4 shows the refined model M_1 with a predicate $equal\_visit$ introduced. The value of $equal\_visit$ is tt iff hosts S_1 and S_2 have been visited the same number of times, i.e. $equal\_visit = \{visit_1 == visit_2\}$. After the refinement, the offensive behavior found in the previous section is no longer a valid trace. Our model checker proves $\mathcal{B}(M_1) \not\subseteq \mathcal{B}(P)$. Since $\mathcal{B}(I) \subseteq \mathcal{B}(M_1) \subseteq \mathcal{B}(M_0) \subseteq \mathcal{B}(P)$, I is safe with respect to P.

There are two advantages to generate abstract models dynamically: (1) compared to model checking, model generation/abstraction is a low cost operation; and (2) the quality of generated models will directly affect the efficiency of model checking. Therefore, a careful study on how to exploit the knowledge of both the itinerary and ACP will be essential on the quality of models and eventually the effi-

ciency of the model checker.

4.3 Model Checking

A number of model checkers have been developed for model checking alternation-free modal μ-calculus. In this work, we used XMC, a logic-programming based model checker. XMC directly encodes the semantics in Figure 3 using XSB tabled logic programming system [17]. XSB extends SLD resolution in Prolog with tabled resolution which enables XSB to terminate more often than the standard Prolog and to avoid redundant sub-computations.

We have also implemented the structural operational semantics in Figure 2 using XSB. The transitions are specified using a ternary predicate `trans`(S, A, T) where S is the source state, A is the action, and T is the target state. Below, we present the encoding of transition rules:

```
1. Migrate Rule   trans(pitin(S), visit(S), end).
2. Input Rule     trans(in(S,X), in(S,X), end).
3. Output Rule    trans(out(S,X), out(S,X), end).
4. Sequential Rule 1
   trans(pref(end,S), A, T) :- trans(S,A,T).
5. Sequential Rule 2
   trans(pref(S1,S2),A,pref(T1,S2)) :- trans(S1,A,T1).
6. Condition Rule 1
   trans(cond(C,S1,S2), A, T) :- C, trans(S1,A,T).
7. Condition Rule 2
   trans(cond(C,S1,S2), A, T) :- not(C), trans(S2,A,T).
8. Parallel Rule 1
   trans(par(end,S), A, T) :- trans(S,A,T).
9. Parallel Rule 2
   trans(par(S,end), A, T) :- trans(S,A,T).
10. Parallel Rule 3
   trans(par(S1,S2), A, par(T,S2)) :- trans(S1,A,T).
11. Parallel Rule 4
   trans(par(S1,S2), A, par(S1,T)) :- trans(S2,A,T).
12. Loop Rule 1
    trans(while(C,I), epsilon, pref(I,while(C,I))) :- C.
13. Loop Rule 2
    trans(while(C,I), epsilon, end) :- not(C).
```

Identifiers beginning with uppercase letters are Prolog variables. Function symbols are denoted by identifiers beginning with lowercase letter. A term denoted by t is constructed from function symbols and variables. In the above, term `pitin(S)` represents the primitive itinerary `S`. `in(S,X)` and `out(S,X)` represent $s?x$ and $s!x$, respectively. Function symbols `pref`, `cond`, `par`, and `while` represent the sequential composition, condition, parallel composition, and loop operators, respectively. A logic program is a set of Horn clauses, in which each clause is of the form $p(\vec{t}) : -G$. Here $p(\vec{t})$ is the head of the clause with a list of terms as parameters and G is a conjunction of atoms (terms with predicate at and only at the root). $p(\vec{t}) : -G$ means that if G is true, then $p(\vec{t})$ is true. $p(\vec{t}) : -true$ is a fact, and can also be written as $p(\vec{t})$. For instance, the encoding of the Migrate rule is a fact which means that given a primitive itinerary `pitin(S)`, the agent can perform a `visit(S)` action. The encoding of the Sequential Rule 1 is a Horn clause, which means that if an agent can perform an A action according to itinerary S then it can perform an A action according to itinerary $end; S$.

Table 1. Data for Algorithms 1,2

#iter	#states$_1$	#tran$_1$	time$_1$	#states$_2$	#tran$_2$	time$_2$
10_{60}	1843	32K	39.1	1131	29K	28.2
100_{60}	5371	72K	82.8	3483	69K	70.2
10_{100}	3203	95K	186	1971	89K	135
100_{100}	14951	333K	587	9803	323K	526

The $\models$ relation is encoded as Prolog predicate `models(P,F)` which checks if an itinerary expression P satisfies a μ-calculus formula F. Here we present the encoding of some of the rules in Figure 3. The details of the model checker are presented in [13]. Rule 2 is encoded as `models(P, fOr(F1, F2)) :- models(P, F1); models(P, F2)`, which means that if a process P satisfies the formulas F_1 or F_2, then P satisfies `fOr(F1,F2)`. The Diamond rule (Rule 4) is encoded as: `models(P,fDiam(A,F)) :- trans(P,A,P1), models(P1,F).`, which means that if P can perform an A transition and then behaves as `P1`, and $P1$ models F, then `P` satisfies `fDiam(A,F)`. Since the semantics of logic programs are based on minimal models, XSB directly computes least fixed points. To compute the greatest fixed point formula, XMC makes use of the duality $\nu X.\varphi = \neg\mu.X\neg\varphi$ and reduces the problem of checking whether a state satisfies a greatest fixed point formula to the problem of checking whether the state satisfies the negation of the corresponding least fixed point formula.

Our model checking engine produces a counter-example that shows $M_0 \not\models P$. The trace of the counterexample is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 4$. By studying the counterexample and Algorithm 3 we can see S_1 is visited twice at line 4 without a visit to S_2 in between.

5 Experimental Results

In this section, we provide the experimental results for model checking Algorithms 1-4 shown in Section 4. For algorithms 1 and 2, we cannot apply model checking directly without knowing the bounds of the variables. In order to do the comparison, we limit the ranges of $price_1, price_2$ to [1..60] or [1..100], and $quant_1, quant_2$ to [1..2]. Although Algorithms 1-4 allow infinite loops, in practice the developer of the mobile agent will always want the mobile agent stop the negotiation after sufficient attempts. In this experiment we limit the number of iterations up to 100. The ACP under verification is that no agent visits site s_1 more than once before visiting some other sites. All reported performance data were obtained on a 1.4GHz Pentium M machine with 512MB of memory running Red Hat Linux 9.0.

Table 1 shows the results for model checking the itinerary Algorithms 1 and 2, where the subscripts at each

Table 2. Data for Algorithms 3,4

#iter	#states$_3$	#tran$_3$	time$_3$	#states$_4$	#tran$_4$	time$_4$
10	203	301	0.01	203	202	0.01
100	-	-	-	2003	2002	0.07

row (60 or 100) indicating the ranges of $price_1$ and $price_2$. Table 2 shows the results for Algorithms 3 and 4. In both tables, `#iter, #states, #tran, time` specifies the number of maximal iterations, the number of states, the number of transitions and CPU time in seconds, respectively. The columns with subscript i contain experimental data for Algorithms i.

As shown in Table 1, the itinerary slice used in Algorithm 2 improve the performance (CPU time) by about 10%-28% over Algorithm 1. The gain is due to reduced number of states and transitions after removing irrelevant information before model checking. It can also be observed that incrementing in the range of variables (from 60 to 100) will increase the time usage substantially. Even with an assumption of a small range for the variables, the table indicates that it is not practical to use formal verification in an environment where quick decisions are expected. On the other hand, as shown in Table 2, it takes merely 0.01 second for Algorithm 3 to report an ACP violation. Note the violation is detected at before the 10th iteration, so no data for 100th iteration is listed in the table. However, the reason provided by the model checker turns out to be a spurious counter-example. After refinement, the model checkers uses only 0.07 seconds up to 100 iterations, which is orders of magnitude improvement over Algorithms 1/2.

6 Conclusions and Future Work

We presented a new approach to address the security of mobile agents with the following salient features: (1)*an itinerary-driven mobile system architecture*, where the specification of the navigational behavior of a mobile agent is separated from the specification of the computational behavior; (2)a *run-time verification system*, where model checking techniques are used to formally verify whether an itinerary satisfies an ACP; (3)*model generation code*, where abstract models, instead of itineraries, are verified to gain performance.

Currently model generation and model refinement are being performed manually. One future work is to develop algorithms to automate these two procedures. A second future work is to perform a similar run-time security verification on computational code in addition to itineraries. Finally, we will apply this framework to scientific workflows where data-intensive computation requires the technique of mobile agent.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, 2001.

[2] R. Bowen, D. Chee, M. Segal, R. Sekar, P. Uppuluri, and T. Shanbag. Building survivable systems: An integrated approach based on intrusion detection and confinement. In *DARPA Information Security Symposium*, 2000.

[3] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction. *Handbook of Process Algebra*, 2001.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages*, pages 238–252, 1977.

[5] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, 1999.

[6] E. A. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science*, pages 267–278, 1986.

[7] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, pages 333 – 354, 1983.

[9] D. Lange and M. Oshima. *Program and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[10] S. Lu and C. zhong Xu. A formal framework for mobile agent itinerary specification, safety reasoning, and logic analysis. In *Proc. of International Workshop on Mobile Distributed Computing (MDC05)*, 2005.

[11] G. Necula. Proof carrying code. In *ACM Principles of Programming Languages*, 1997.

[12] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, 1998.

[13] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Computer-Aided Verification*, pages 143–154, 1997.

[14] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *New Security Paradigms Workshop (NSPW)*, 2001.

[15] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[16] A. Tripathi, N. Karnik, T. Ahmed, and et al. Design of the Ajanta system for mobile agent programming. *Journal of Systems and Software*, May 2002.

[17] XSB. The XSB logic programming system v2.7, 2005. Available from `http://xsb.sourceforge.net`.

[18] C.-Z. Xu. Naplet: A flexible mobile agent framework for network-centric applications. In *Proc. of the 2nd Int'l Workshop on Internet Computing and E-Commerce*, April 2002.

[19] A. Zaks, I. Shlyakhter, F. Ivancic, S. Cadambi, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Using range analysis for software verification. In *4th International Workshop on Software Verification and Validation*, 2006. to appear.