# A model checking-based approach for security policy verification of mobile systems

Chiara Braghin, Natasha Sharygina, Katerina Barone-Adesi

Under consideration for publication in Formal Aspects of Computing

# A Model Checking-based Approach for Security Policy Verification of Mobile Systems[1]

Chiara Braghin[1] and Natasha Sharygina[2] and Katerina Barone-Adesi[2]

[1]Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
[2]Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland

**Abstract.** This article describes an approach for the automated verification of mobile systems. Mobile systems are characterized by the explicit notion of *location* (e.g., sites where they run) and the ability to execute at different locations, yielding a number of security issues. To this aim, we formalize mobile systems as Labeled Kripke Structures, encapsulating the notion of *location net* that describes the hierarchical nesting of the threads constituting the system. Then, we formalize a generic *security-policy specification language* that includes rules for expressing and manipulating the code location. In contrast to many other approaches, our technique supports both access control and information flow specification.

We developed a prototype framework for model checking of mobile systems. It works directly on the program code (in contrast to most traditional process-algebraic approaches that can model only limited details of mobile systems) and uses abstraction-refinement techniques, based also on location abstractions, to manage the program state space. We experimented with a number of mobile code benchmarks by verifying various security policies. The experimental results demonstrate the validity of the proposed mobile system modeling and policy specification formalisms and highlight the advantages of the model checking-based approach, which combines the validation of security properties with other checks, such as the validation of buffer overflows.

**Keywords:** Software verification; Program security and safety; Mobile systems; Security policies; Access Control; Information Flow

*Correspondence and offprint requests to*: Chiara Braghin, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, via Bramante, 65 - I-26013 Crema. e-mail: chiara.braghin@unimi.it

[1] This paper is an extended version of [C. Braghin, N. Sharygina, K. Barone-Adesi, Automated Verification of Security Policies in Mobile Code, in: Proceedings of IFM, Lecture Notes in Computer Science, Volume 4591, p.37-53, (2007)][BSBA07].

## 1. Introduction

Mobile code technologies have been widely deployed via web browsers for several years. Moreover, the applet model is now being transferred to high-security embedded devices such as smart cards. Despite the promising applications of mobile code technologies, they have not yet been widely deployed. A major problem of mobile code is security: without appropriate security measures, a malicious applet could mount a variety of attacks against the local computer, such as destroying data (e.g., reformatting the disk), modifying sensitive data (e.g., registering a bank transfer via a home-banking software), divulging personal information over the network, or modifying other programs (e.g., Trojan horses attacks).

Moreover, programming over a wide area network such as the Internet not only introduces new issues to the field of multi-threaded programming and analysis, but also breaks many postulates commonly assumed in concurrent systems. For example, during the execution of a mobile program, a given thread may stop executing at a site, and continue executing at another site. That is, threads may jump from site to site while retaining their conceptual identity. The following issues distinguish mobile systems from a more general case of multi-threaded programs:

- threads may run at different locations (e.g., administrative domains, hosts, physical locations, etc.);
- thread migration takes into account their geographical distribution (e.g., migration can only occur between directly linked net locations);
- there exists distinction among local and global communication due to bandwidth fluctuations, node failures, etc.

This paper describes an approach for modeling and static verification of mobile systems. They are formalized as Labeled Kripke Structures (LKSs), which encapsulate the notion of *location* and unbounded thread creation typical to mobile systems. We define the semantics of mobile systems where threads are always confined to locations and have the *ability to move*. The LKS notation allows modeling both the data and the communication structures of the multi-threaded systems.

We formalize a language for specifying *general-purpose security policies*, and we show how mobile systems can be statically and exhaustively analyzed against any security policy by using model-checking techniques. To support features of mobile systems, the policy specification language defines rules for expressing and manipulating the code location. It works at the level of method calls and variable accesses, thus making it (in contrast to most other approaches) suitable for specifying *both* access control and information flow policies or a mixture of both.

The proposed modeling and security-policy specification formalisms are generic and are suitable for specification and model checking mobile programs implemented in a language of choice. We implemented and experimented with a prototype framework for modeling and verifying mobile systems written in C. C is actively used for mobile applications such as Wireless Sensor Networks (WSN) (for example, TinyOS[BC06], the most popular WSN operating system used today, is written in nesC, a dialect of the C programming language optimized for the memory limitations of sensor networks). Our experimental framework uses a model checker, SATABS [CKSY05], which implements a SAT-based counterexample-guided abstraction refinement framework (CEGAR for short) for ANSI-C programs. A policy configuration file, specifying what permissions (i.e., which types of system resource access) to deny, is given as an input to the model checker together with the program to be verified. Then, the mobile program is annotated with information related to the security policy in such a way that if and when the security policy is violated, the model checker returns a counter-example that led to such an error. In such a way, we are able to discover both implementation and security-related errors. To cope with the computational complexity of verifying mobile programs, we define projection abstractions, constructed by restricting a program path to actions or states satisfying certain conditions. In particular, by exploiting the explicit notion of locations, we define *location-based projections*, which allow efficient verification of location-specific security policies.

In summary, our approach to modeling and verifying mobile systems has several advantageous features:

- it explicitly models thread locations, location distribution and thread moving operations, which are essential elements of mobile systems;
- it preserves both data and communication structures of mobile systems;
- it defines a specification language for specifying generic security policies of mobile systems, which is suitable for specifying both access control and information flow properties;

- it integrates model-checking technologies to support exhaustive analysis of security policies;
- it defines location-specific abstractions which enable the efficient verification of large mobile code applications;
- it enables verification of mobile code, and not of a high level abstraction of a mobile system, as most approaches do.

We experimented with a number of mobile code benchmarks, instantiated in our mobile code framework, by verifying various security policies. The results of verifying security policies, dealing with both access permissions of system actions and tracking the location net with respect to permissible location configurations, demonstrated the applicability of the new formalisms and proven to scale to large instances of mobile programs.

The rest of the article is organized as follows. Section 2 gives an overview of the state of the art in the context of mobile systems' modeling and verification. Section 3 gives some preliminary information on mobile systems and security policies. In particular, it discusses the subtle differences between general multi-threaded programs and mobile systems (also called location-aware multi-threaded systems). It also outlines how the features specific to mobile systems are supported by programming languages. Section 4 formalizes in a language-independent way the features of mobile systems highlighted and defines formal semantics of mobile systems. In Section 5, we present a policy specification language that defines rules for expressing also the code location. Section 6 maps the modeling techniques to the SATABS-based model-checking approach, reporting also the experimental results obtained by verifying a number of mobile code benchmarks against various security policies. Section 7 concludes the paper with the summary of the contributions.

This article is an extended and revised version of [BSBA07]; with respect to the extended abstract, the article provides further details on location projections, has an extended background section describing the current languages used for describing mobile systems from which we drew ideas, and expands the description of the experimental results.

## 2. Related Work

**Modeling Mobile Systems.** The most common approach to modeling mobile systems is the process-algebra-based approach. Various location-aware calculi, with an explicit notion of *location*, have arisen in the literature to directly model phenomena such as the distribution of processes within different localities, their migrations, or their failures [CG00, HR98, DNFP98, FGL$^+$96, Ste03].

The process algebra that mostly influenced this work is the Mobile Ambient calculus [Car99, CG00]: this specification language provides a simple framework that encompasses mobile agents, the domains where agents interact and the mobility of the agents themselves. An ambient is a generalization of both agent and place notions. Like an agent, an ambient can move across places (also represented by ambients) where it can interact with other agents. Like a place, an ambient supports local undirected communication, and can receive messages (also represented by ambients) from other places [CG00]. The formal semantics we give to mobile systems draws many ideas from the ambient calculus.

Moreover, we benefited from the Kell calculus papers [BS03, BSS05, SS04], where the problems raised by using atomic actions (i.e., a sort of distributed synchronization) in wide-area networks are discussed. In particular, the authors highlight the need of modeling different forms of failure; we kept this in mind when giving the semantics of the moving capabilities.

Distributed process calculi have been introduced as foundations for distributed programming; for this reason they describe mobile system at high level, in a concise form, with only few key operators and programming primitives. Even though there exists process algebras such as Klaim [DNFP98] that deal with data structures, they often specify only limited details of the systems and do not extensively model data and data manipulation structures, since they concentrate on communication structures. In contrast to process algebra-based approaches, the advantage of our formalism (by means of Labeled Kripke Structures) is that it provides full support of data. In fact, the Labeled Kripke Structures give a low-level description of systems, closer to the code, allowing easy translation from any programming language.

**Verification of Mobile Systems.** The use of mobile systems raises a number of security issues, including access control (is the use of the resource permitted?), user authentication (to identify the valid users), data integrity (to ensure data is delivered intact), data confidentiality (to protect sensitive data), and auditing (to

track uses of mobile resources). All but the first category are closely coupled with research in cryptography and are outside of the scope of this paper. Our technique assumes that the appropriate integrity checking and signature validation are completed before the security access policies are checked.

Certified code [NL97] is a general mechanism for enforcing security properties. In this paradigm, untrusted mobile code carries annotations that allow a host to verify its trustworthiness. Before running the agent, the host checks the annotations and proves that they imply the host's security policy. Despite the flexibility of this scheme, so far, compilers that generate certified code have focused on simple type safety properties rather than more general security policies. The main difficulty is that automated theorem provers are not powerful enough to infer properties of arbitrary programs and constructing proofs by hand is prohibitively expensive.

Process-algebraic verification techniques usually deal with coarse overapproximations during the analysis of mobile systems. Overapproximations are useful to reduce the analysis complexity and guarantee that, if no errors are found in the abstract system, then no errors are present in the actual system. However, if errors are found, the verification techniques developed for process algebra fail to guarantee that they are real. There exists a number of efforts to model check process-algebra-based techniques. For example, [Dis00] and [CDZG$^+$02] propose techniques for model checking Mobile Ambients. In contrast to the process-algebraic approach, our technique (in the context of the abstraction-based model checking) simulates the errors on the actual system and, if the errors are found to be spurious, the approximated programs are refined. To the best of our knowledge, there are no abstraction-refinement techniques that would support the process-algebraic analysis techniques.

Unable to prove security properties statically, real-world security systems such as the Java Virtual Machine (JVM) have fallen back on run-time checking. Dynamic security checks are scattered throughout the Java libraries and are intended to ensure that applets do not access protected resources inappropriately. However, this situation is unsatisfying since tests rely on the implementation of monitors which are error-prone, and system execution is delayed during the execution of the monitor. For this reason, in [CA09], the authors address the problem of establishing at compile time that a given Java program preserves confidentiality of sensitive information. The approach is evaluated by model checking a set of Java (J2ME) methods.

Several research projects deal with the problem of mobile code verification of security properties, but they all restrict their working environment either to a specific programming language or to a not-so-general case study. MOBIUS (Mobility, Ubiquity and Security) [mob] investigates trust and security for small devices which are functioning as a part of global computers. The main objective is to enable proof-carrying code for Java on mobile devices. It works at Java byte-code level in order to overcome the problem of dealing at the source level. The drawback of the approach is the fact that it is highly language-dependent. In our case, the experimental framework is independent from the theoretical contribution, thus the modeling and verification techniques we propose can work with any programming language. S3MS (Security of Software and Services for Mobile Systems) [s3m] is a Specific Targeted Research Project (STReP) that develops a framework based on security-by-contract for trusted deployment and execution of mobile applications, in which the contract expresses the security features and requirements. The security properties that they intend to verify are highly dependent on the mobile application context, whereas our running environment and case studies are more heterogeneous.

**Verification of Multi-threaded Systems.** Formal verification of multi-threaded programs is an area of active research; see [Rin01] for an excellent survey. Pushdown automata have been used as tools for analyzing sequential programs with (recursive) procedures [BS95]. The expressive power of pushdown systems is equivalent to that of sequential programs with (possibly recursive) procedures where all variables have a finite data type. MOPED [ES01] and BEPOP [BR00], for example, are BDD-based symbolic model checkers for this class of languages. There has also been work on verifying concurrent software using pushdown automata with multiple stacks, e.g., see [CCK$^+$06]. As reachability is undecidable in this case, the existing implementations are not fully automated.

The class of programs considered in this paper can be viewed as an instance of a *parameterized system*, i.e., a system with a number of identical processes (threads in our case). Many approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques, network invariants, predicate abstraction or system symmetry (see an excellent overview in [CTTV04]). Methods that are most closely related to our work are based on abstraction (for example, an extension of Mur$\phi$ uses abstraction for replicated identical components [ID96]). In contrast to our approach, many of these methods are only

partially automated, requiring at least some human ingenuity to construct a process invariant or a closure process (for example, the TLPVS tool [PA03] is based on manual theorem proving).

Henzinger et al. use predicate abstraction in order to construct environment models from threads [HJM04, BCH+04]. When combined with a counter abstraction, an unbounded number of threads can be supported. Flanagan and Qadeer propose to use the idea of thread states in order to obtain environment models for *loosely-coupled* multi-threaded programs [FQ03]. In contrast to their algorithm, the model-checking approach we use addresses the spurious behavior introduced by this overapproximation by (safely) restricting the thread states that are passed, and by an automatic refinement procedure.

A number of tools for analysis of multi-threaded Java programs is available. While some of the tools compute abstract models automatically, most are restricted to explicit state space exploration. Representative examples of model checkers for Java are [Sto00] and JPF [HP00]. Yahav reports an implementation of a model checker for Java with an unbounded number of threads using three-valued logic [Yah01]. Similarly to the approach we use, an overapproximation is computed.

**Model checking code.** The modeling formalism and security policy specification language presented in this article allow model checking of mobile systems at the implementation level. These constructs are language independent and thus can be used with any software model checker. While there are no, to the best of our knowledge, tools for model checking mobile code, there is a number of general purpose software model checking tools working at the code level. Most of the code verifiers employ a counterexample guided abstraction refinement to cope with the complexity of verification of real programs (see, for example, [BR01], [STT09] describing respectively the basic and optimized abstraction-refinement techniques). Among such tools are SLAM [BCLR04], COMFORT [CISW05], BLAST [BCH+04] and SATABS [CKSY05] (each designed for C programming language with SATABS being the most complete tool as it provides both full support for ANSI-C and sound treatment of unbounded thread creation in concurrent programs) and JPF and a tool of Stoller for Java (described above). Our experimental framework for verification of mobile code uses SATABS. The mobile programs are annotated with the security invariants (as discussed in Section 6) and given to the model checker to validate the security policies. The annotation approach used in our framework is the approach of Schneider [Sch00]. The approach of Schneider was implemented in two software-verification projects before: (1) SLIC [BR02] and (2) BLAST [BCH+04]. However, this project provides major extensions and improvements over the two existing approaches: (a) SLIC and BLAST are all about pure C, whereas our framework allows an extension of C that is suitable for mobile code; (b) The previous approaches came with a general-purpose specification language, whereas our project comes with a very sophisticated language that is particularly tailored to the verification of security properties of mobile code; (c) Besides the programming language and specification language, also the verification engine is different: in this project, SATABS is used.

## 3. Background: Mobile Systems and Security Policies

### 3.1. Mobile Systems

In the last decade, the Web has rapidly evolved into a global computing platform, whose main characteristic is the geographical distribution. As a consequence, computer systems have evolved from centralized monolithic computing devices into client-server environments allowing complex forms of distributed computing. During this evolution process, limited forms of code mobility have arisen: Java applets downloaded from web servers into web browsers, JavaScript and Visual Basic scripts, ActiveX controls, Flash animations, and macros embedded within Office documents. A new phase of evolution is now under way that goes one step further, in the form of a software agent that can suspend its execution on a host computer, transfer itself to another agent-enabled host on the network, and resume execution on the new host. A number of new generation programming languages (e.g., Telescript, Obliq, Klaim, etc. ) viewing the network and its resources as a global computational environment have been proposed, although none is used in practice and is used only for academic purposes. They are characterized by the capability to provide some sort of code mobility, and to allow distributed multi-threading, i.e., threads running concurrently at different locations. This programming paradigm demonstrates features that make mobile programs different, and more complex, than the standard notion of multi-threaded computing. These features are essential and we believe that they should be explicitly modeled and verified in order to become usable in practice.

A spectrum of differing shades of mobility exists, corresponding to the possible variations of relocating

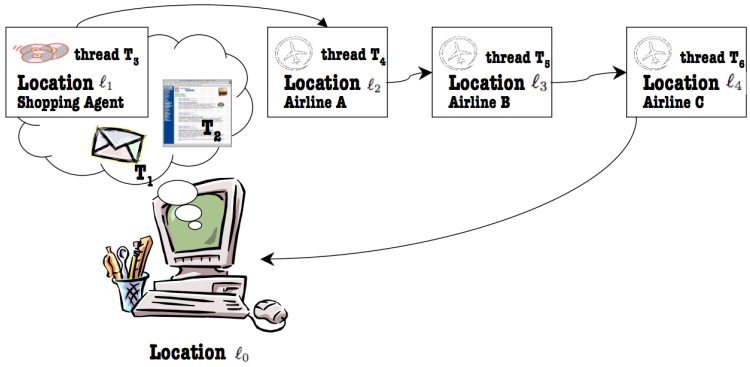


**Fig. 1.** A shopping agent. Here, $\ell_0$ to $\ell_4$ are locations representing sites, and $T_1$ to $T_6$ threads.

code and state information, including the values of instance variables, the program counter, execution stack, etc. For example, a Java applet [Mic95] has mobility of code through the movement of class files from a web server to a web browser. However, no associated state information is conveyed. In contrast, Aglets [Cor99], originally developed by the IBM Tokyo research laboratory and now hosted at SourceForge.net as an open source project, builds upon Java to allow the values of instance variables, but not the program counter or execution stack, to be conveyed along with the code as the agent relocates.

As an example demonstrating full mobility, we consider the Telescript language [Whi94]. Telescript is a rich, object-oriented language conceived for the development of large distributed applications, oriented in particular to the electronic market. It is an agent-based language that explicitly deals with locality and mobility: there are two kinds of entities, *agents*, i.e., threads that can move, and *places*, i.e., physical locations that offer services and can contain agents or places. A thread running on an interpreter (called engine in the Telescript jargon) is able to migrate autonomously to another engine run by a different machine by executing a `go` instruction. This operation causes the engine to suspend the thread, serialize it together with its state, and transmit it to the destination engine specified as argument. There, the thread will be unserialized, and its execution will be resumed from the instruction following the `go`. If a trip cannot be made, the `go` instruction fails and the thread throws an exception. Migration is transparent by default, i.e., only the final destination is given by the agent, but the `go` method allows the agent to request explicitly the route to follow. Agents do not communicate remotely with other agents; rather, they move to some location and communicate locally when they get there by executing a `meet` instruction.

**Example 1.** As an implementation example demonstrating the features of mobile programs, consider the Telescript implementation of the shopping agent example [Whi94]. The shopping agent scenario depicted in Figure 1 consists of different *locations* representing a user's personal computer and the servers hosting various airline Web-sites. The shopping agent is a *thread* that migrates from its home location (i.e., the personal computer) to the airline locations to collect informations on their flights in order to find the best airfare. The agent is given various requirements, such as departure and destination time restrictions. After querying the airline databases, it reports back the information to the user who made the request. In the Telescript programming language, migrating threads are called agents, whereas locations are called places. Moreover, user-defined classes for agents and places are subclasses of the predefined `Place` and `Agent` classes. In our example, the shopping thread corresponds to a Telescript agent, and two place classes must be defined: one for the airline locations (in the example they all offer the same services), and one for the home location of the thread. In the following, we give the definition of the three classes, omitting the details which are out of the scope of the paper. The `Airline_Location` class is defined as follows:

```
Airline_Location: class (Place) =
(
     public
        flightSearch: op (src, dest, date: String)
                      List =  {  /*  body  */  }
     property
        flightCatalog: Dictionary[String, Entry];
```

```
)
```

Note that within a class definition the three keywords `public`, `private` and `property` distinguish among public and private methods, and local or global variables, respectively. In addition, `List` and `Dictionary` are predefined types with the usual meaning. The meaning of the code is rather intuitive: an `Airline_Location` offers a method to look for flights, given the desired source, destination and date of the trip. The method searches within the `flightCatalog` and returns a, possibly empty, list of flights available. Each location $\ell_1$ to $\ell_n$ from Figure 1 will be an object of this class, while $\ell_0$ will be an instantiation of the following class:

```
Home_Location: class (Place) =
(
     private
       orderFlight: op (src, dest, date: String) =
        { Shopping_Agent (src, dest, date: String);
          /* check returned list of flights */  }
)
```

This class has only one private method that creates an agent which goes to search in the Web for a cheap flight. The places where to migrate must be given in the omitted body code.

```
Shopping_Agent: class (Agent) =
(
     public
        showReport: op () = {  ...  };
     private
        goShopping: op () = {  ...  };
        checkPrice: op () = {  ...  };
        goHome: op () = {  ...  };
     property
        homeAddress, clientName: String;
        desiredPrice, actualPrice: Integer;
        source, destination, date: String;
)
```

The `goShopping` and `goHome` methods are based on the predefined method `go`. The `goShopping` method contains the list of Airline locations to visit and possibly some of the route details specified by the user (e.g., the period of time to do the research, the price range, etc.). ◇

We can infer, as for example in the above examples, that the key characteristics of mobile programs, disregarding any particular implementation language, are:

- an explicit notion of *location* (e.g., physical locations, hosts, administrative domains where threads run, etc.);
- locations are structured to capture the distributed environment of the Web (e.g., an application running on several sites);
- the capability for a thread to migrate;
- the distinction between *local* and *global* thread communication to depict communication within a single administrative domain or among different domains.

## 3.2. Security Policies and Models

There has been much debate in the security literature as to what exactly is meant by the terms *security policy* or *security model* and indeed what, if any, is the distinction. A consensus has been reached by considering a *policy* a set of rules and conditions that state which actions are permitted and which actions are prohibited, whereas a *model* as a formal description of a security policy: it precisely and unambiguously conveys those aspects of the security policy that are enforced by the system [McL94].

In the following subsections we illustrate different security models that have been proposed in the literature, not pretending to be exhaustive. Such security models will be used as an input when defining the

language for specifying location-aware security policies. In the descriptions, we consider a computer system consisting in *subjects*, i.e., active entities requesting access to objects, such as a user or a process, and *objects*, i.e., passive entities storing information, such as a file or a magnetic storage.

### *3.2.1. Access Control Policies*

Access control is the process of mediating every request to the resources maintained by a system and determining whether a request should be granted or denied. The aim is to limit disclosure of classified data and to guarantee that only authorized access can take place. Access control policies [DoD85] can be broadly classified into mandatory, discretionary and rôle-based access control policies depending on who is in charge of specifying access rights and to whom permissions are given/assigned.

In discretionary access control (DAC) models, access control is at the discretion of the owner: the owner of a resource decrees who is allowed to have access, by specifying the subjects, and the rights of the subjects to objects. The file management of Unix is a classic example of DAC.

In mandatory access control (MAC) models, access regulations are mandated by a central system-wide authority. This policy is used with highly confidential data, such as military or government. The most common for mandatory access control policy is the *multi-level security policy*, based on the classification of subject and object into clearance levels.

The essence of rôle-based access control (RBAC) models lies with the notions of user, rôle and permission. Within an organization, rôles are created for various job functions, and these rôles are assigned permissions. Staff are made members of appropriate rôles and thus acquire the permissions assigned to those rôles. This leads to a greatly simplified administration of permissions. For example, a staff member can be immediately assigned a new rôle when changing department, rather than closing all existing access, and creating a new set of access controls. RBAC is more common in database management systems, and it is attracting increasing attention for commercial applications.

### *3.2.2. Information Flow Models*

**The Bell-LaPadula Model.** The model proposed by Bell and LaPadula (BLP) is one of the earliest and best known models [BLP76]. It provides a framework for handling data of different classifications, and for this reason is also called *multi-level security* model.

Security or clearance levels are arranged into a lattice $\langle L, \leq \rangle$ where $\ell_1 \leq \ell_2$ means that $\ell_1$ has a security level lower than $\ell_2$, i.e., $\ell_2$ dominates $\ell_1$. A particularly simple example is a linear hierarchy $L = \{\mathsf{unclassified}, \mathsf{confidential}, \mathsf{secret}, \mathsf{top\ secret}\}$ with $\mathsf{unclassified} \leq \mathsf{confidential} \leq \mathsf{secret} \leq \mathsf{top\ secret}$. More elaborate, non-linear lattices representing partial orders are possible and indeed common. The security level associated with an object reflects the sensitivity of the information contained in the object, i.e., the potential damage that could result from the unauthorized disclosure of the information.

Then, a function $f_c : S \cup O \to L$ is defined assigning to each member of the set of subjects $S$ and objects $O$ a clearance level. The modes of access are usually two: `read` and `write`[2], however it can be extended depending on the type of the resource. Three basic rules are imposed by the model:

**Simple Security Property** – A subject $s$ is allowed `read` access to an object $o$ if and only if $f_c(s) \geq f_c(o)$, i.e., $f_c(s)$ dominates $f_c(o)$. That is, a subject has read access only to objects whose security level is equal or below the subject's current clearance level. This prevents a subject from getting access to information available in security levels higher than its current clearance level. This property is also called *no read up*.

***-Property** – A subject $s$ is allowed `write` access to an object $o$ if and only if $f_c(s) \leq f_c(o)$, i.e., $f_c(o)$ dominates $f_c(s)$. That is, a subject has write access to objects whose security level is equal to or higher than its current clearance level. This prevents a subject from passing information to levels lower than its current level. This property is also called *no write down*.

**Tranquility Property** – The tranquility property states that the security level of an object cannot be changed while it is being processed by a computer system.

[2] In Bell and LaPadula's formulation of the model, the `append` and `write` access modes are considered, but our simplification does not affect the overall model.

The purpose of the model is to confine sensitive data at its correct level: "no read up" prevents users from accessing information for which they are not cleared to access; "no write down" prevents users (or more importantly software) from taking more sensitive information and writing it into a less sensitive document. In this manner, the Bell-LaPadula model guarantees that data from a higher security level can never flow to a lower security level. Still, it would be possible to transmit *indirectly* information through system side effects. These indirect ways of transmitting information are called *covert channels* and are not prevented by the BLP model.

**Example 2.** Let us consider the shopping agent of Example 1 again. In this case, following a multi-level security model, it would be safe to classify the shopping agent and the user personal computer as secret since they record highly confidential information such as the user credit card number or personal information, whereas all the airline Web-sites should be labelled as confidential since they need to use some of the highly confidential information of the user in order to be able to book a ticket. All other sites should be classified as unclassified since they are untrusted.

A mandatory access rule implementing the multi-level security model should not allow agents/sites labelled either as secret or confidential to write/send confidential information to an unclassified site.

## 4. Formal Semantics of Mobile Programs

In this section we formalize in a language-independent way the features of mobile systems highlighted in Section 3.1 and we formally describe the semantics of mobile programs.

### 4.1. Mobile Programs

This section gives the syntax of mobile programs using a C-like programming language (which we believe is one of the most popular general-purpose languages). We extend the standard definition of multi-threaded programs with an explicit notion of *location* and moving actions. The syntax of a mobile program is defined using a finite set of *variables* (either local to a thread or shared among threads), a finite set of *constants*, and a finite set of *names*, representing constructs for thread synchronization, similar to the Java `wait` and `notify` constructs. It is specified by the following grammar:

| | | | |
|---|---|---|---|
| $LT$ | ::= | | location-aware threads |
| | | $\ell[\![ T ]\!]$ | single thread |
| | \| | $LT_1 \parallel LT_2$ | parallel composition |
| $T$ | ::= | | threads |
| | | $T_1 \mid T_2$ | parallel composition |
| | \| | $Instr$ | sequential execution |
| $Instr$ | ::= | | instructions |
| | | $Instr_1$ `;` $Instr_2$ | sequential execution |
| | \| | `x :=` $Expr$ | assignment |
| | \| | `if (`$Expr$ `!= 0)` $Instr$ | condition |
| | \| | `while (`$Expr$ `!= 0)` $Instr$ | loop |
| | \| | `skip` | skip |
| | \| | `m` | synchronised call |
| | \| | `new` | thread creation |
| | \| | $M\_Instr$ | moving action |
| $Expr$ | ::= | | expressions |
| | | `c` | constant |
| | \| | $Expr_1$ ( `+` \| `-` \| `*` \| `/` ) $Expr_2$ | arithmetic operation |
| $M\_Instr$ | ::= | | moving actions |
| | \| | `go_in(`$\ell$`)` \| `go_out(`$\ell$`)` | move in/out |

In the grammar, `x` ranges over variables, `c` over constants, and `m` over the names of synchronization constructs. The meaning of the constructs for expressions and instructions is rather intuitive: an expression can be either a constant or an arithmetic operation (i.e., sum, difference, product and division). The instruction set mainly

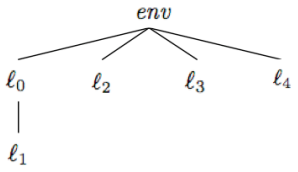


**Fig. 2.** The location net of Example 3.

consists of the standard instructions for imperative languages: a sequential composition operator (`;`), the assignment instruction, the control flow instructions `if` and `while`, and the `skip` statement. The instructions specific to the threads package are the `new` instruction, which spawns a new thread that is an exact copy of the thread executing the `new` instruction, and the call to a synchronization method `m`.

We further assume a set of location names *Loc*, and we let $\ell, \ell_1, \ell_2, \ldots$ range over *Loc*. A thread is $\ell[\![ T ]\!]$, with $\ell$ being the location name of thread $T$. More than one thread may be bounded to the same location, that is $\ell[\![ T_1 \mid T_2 ]\!]$ (examples will be shown later). A mobile program is defined by the parallel composition of multiple threads. A location can thus be seen as a bounded place, where mobile computation happens.

Conceptually, thread locations represent the geographical distribution of the Web. To capture this fact, we use a special structure, called a *location net*, which encapsulates the hierarchical nesting of the Web. We define the location net as a tree, whose nodes are labeled by unique location names, and the root labeled by the special location name *env*, representing the external environment of the system under analysis. A tree $t_\ell$ is identified with the set of its paths, i.e., $t_\ell$ will be a finite subset of $Loc^*$.

**Example 3.** As a running example consider again the *shopping agent* program of Example 1. For simplicity, let's assume that the system is composed of threads $T_1 ... T_6$ which are distributed among various locations: $Loc = \{env, \ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}$ and that a single thread is sent out. Here, $\ell_2, \ell_3, \ell_4$ are the locations of various websites; $\ell_1$ is the location of the agent, $\ell_0$ is the program sending out the agent, and *env* the generalized environment location. Clearly, some of the locations are nested, and the location net corresponds to a tree, which can be defined by the set of its paths, i.e., $t_\ell = \{env.\ell_0.\ell_1, env.\ell_2, env.\ell_3, env.\ell_4\}$, or can be depicted as in Figure 2. In the rest of the paper, when referring to nodes of the location net, we borrow standard vocabulary to define the relationship among tree nodes, such as father, child and sibling. For instance, in our example, $\ell_2$ and $\ell_3$ are siblings, whereas $\ell_0$ is the father of $\ell_1$ (and $\ell_1$ is the child of $\ell_0$).

In the Mobile Ambient calculus, the calculus from which we drew most of the "mobility features", the hierarchy among localities (i.e., ambients) is inferred directly from the nesting among the ambient processes. Thus, the location net depicted in Figure 2 would be formalized with the following ambient process:

$$env[\![\, \ell_0[\![\, \ell 1[\![ \mathbf{0} ]\!] \,]\!] \mid \ell_2[\![ \mathbf{0} ]\!] \mid \ell_3[\![ \mathbf{0} ]\!] \mid \ell_4[\![ \mathbf{0} ]\!] \,]\!].$$

In order to highlight only the hierarchical structure of the ambients we intentionally defined the internal threads as null processes. ◇

The location net represents the topology of thread locations. In fact, it implicitly represents the distribution of threads. Location-aware threads can perform moving actions to change this distribution. These actions are the moving instructions, `go_in` and `go_out`. The explicit notion of location and the existence of moving actions affect the interaction among concurrent threads as follows (the formal definition will be given in Section 4.2):

- There are two types of composition: the parallel composition among threads bounded to the same location (i.e., $\ell[\![ T_1 \mid T_2 ]\!]$), and the parallel composition among threads bounded to different locations (i.e., $\ell_1[\![ T_1 ]\!] \parallel \ell_2[\![ T_2 ]\!]$) - see the example below.
- The execution of moving actions changes the location net, i.e., mobility can be described by updates of the location net.
- The execution of moving actions is constrained by the structure of the location net, i.e., moving actions can be performed only if the thread location and the target location has the father-child or siblings relationship.

**Example 4.** For example, if threads $T_1$ and $T_2$ represent a mail server and a browser running at site $l_0$,

thread $T_3$ the agent, and threads $T_4 ... T_6$ are each running at sites $l_1 ... l_4$, then the shopping agent program of Example 3 can be formalized as follows:

$$\ell_0[\![ T_1 \mid T_2 ]\!] \;\|\; \ell_1[\![ T_3 ]\!] \;\|\; \ell_2[\![ T_4 ]\!] \;\|\; \ell_3[\![ T_5 ]\!] \;\|\; \ell_4[\![ T_6 ]\!]$$

In this program, threads $T_1$ and $T_2$ are running in parallel *locally* since $\ell_0[\![ T_1 \mid T_2 ]\!]$. On the contrary, $T_3$ and $T_4$ are running *remotely* since $\ell_1[\![ T_3 ]\!] \;\|\; \ell_2[\![ T_4 ]\!]$. Let us now suppose that thread $\ell_1[\![ T_3 ]\!]$ is defined as

$$T_3 = \texttt{go\_out}(\ell_0); \texttt{go\_in}(\ell_2); T_3'$$

and that the location net is as depicted in Figure 2. In this case, the moving instruction $\texttt{go\_out}(\ell_0)$ can be executed since location $\ell_0$ and location $\ell_1$ are father and child, i.e., location $\ell_1$ is nested inside $\ell_0$, thus a location-aware thread bounded to location $\ell_1$ can *go out* of location $\ell_0$.

In the Mobile Ambient calculus the full process would be formalized as follows:

$$env[\![\; \ell_0[\![ T_1 \mid T2 \mid \ell 1[\![ \textbf{out}\; \ell_0.\textbf{in}\; \ell_2.T_3' ]\!] ]\!] \mid \ell_2[\![ T_4 ]\!] \mid \ell_3[\![ T_5 ]\!] \mid \ell_4[\![ T_6 ]\!] \;]\!].$$

Notice that the hierarchies among localities are less evident than with our formalization. ◇

## 4.2. The Computational Model

In this section we formalize the semantics of mobile programs. We first define the semantics of a single thread, and then extend it to the case of a multi-threaded system. As done in the examples of the previous section, when discussing multi-threaded systems consisting of $n$ threads, we will use $i$, with $1 \leq i \leq n$, as a unique identifier of each thread $T$ (i.e., we will write $T_i$).

**Definition 1 (Location-aware Thread).** A thread is defined as a Labeled Kripke Structure [CCO+04] $T = (S, Init, AP, \mathcal{L}, \Sigma, \mathcal{R})$ such that:

- $S$ is a (possibly infinite) set of states;
- $Init \in S$ is the initial state;
- $AP$ is the set of atomic propositions;
- $\mathcal{L} : S \to 2^{AP}$ is a state-labeling function;
- $\Sigma$ is a finite set (alphabet) of *actions*;
- $\mathcal{R} \subseteq S \times \Sigma \times (S \cup \{ S \times S \})$ is a total *labeled* transition relation.

A state $s \in S$ of a thread is defined as a tuple $(V_l, V_g, pc, \varphi, \eta)$, where $V_l$ is the evaluation of the set of local variables, $V_g$ is the evaluation of the set of global variables, $pc$ is the program counter, $\varphi : Loc \hookrightarrow Loc$ is a partial function denoting the *location net* (where $Loc$ is the set of location names as defined in Section 4.1), and $\eta : \mathbb{N} \hookrightarrow Loc$ is a partial function denoting the *thread location*. More specifically, $\varphi$ describes the location net at a given state by recording the father-child relationship among all nodes of the net ($\perp$ in the case of $env$), whereas $\eta(i)$ returns the location name of $T_i$ (i.e., the thread identified by $i$).

**Example 5.** Consider again the shopping agent program and its location net as defined in Example 3. In this case, the location net function is $\varphi(\ell_0) = env, \varphi(\ell_1) = \ell_0, \varphi(\ell_2) = env, \varphi(\ell_3) = env, \varphi(\ell_4) = env$. In addition, the thread location function for threads $T_1 \cdots T_6$ is defined as $\eta(1) = \ell_0, \eta(2) = \ell_0, \eta(3) = \ell_1, \eta(4) = \ell_2, \eta(5) = \ell_3, \eta(6) = \ell_4$. ◇

The transition relation $\mathcal{R}$ is labeled by the *actions* of which there are four types: *moving*, *synchronization*, *thread creation*, and $\tau$ actions, which are contained in the mutually disjoint sets $\Sigma^M, \Sigma^S, \Sigma^T, \Sigma^\tau$, respectively. We use $\Sigma$ to identify the set of all actions. $\tau$ represents a generic action such as an assignment, a function call, etc. We write $s \xrightarrow{a} s'$ to mean $(s, a, s') \in \mathcal{R}$, with $a \in \Sigma$. Moreover, we write $s \xrightarrow{a}_i s'$ to specify which thread performed the action. Note that, since we allow thread creation, if thread $T_i$ performs a `new` action, $s'$ can be defined as a pair of states s.t. $s \xrightarrow{\texttt{new}}_i (s', \overline{s})$, where $s'$ is the next state of $s$, and $\overline{s} = Init_i$ is an initial state of the newly created thread (which corresponds to the initial state of $T_i$).

Table 1 gives the inference rules for the labeled transition relation in the case of moving actions ($\texttt{go\_in}(\ell)$, $\texttt{go\_out}(\ell)$), thread creation action, `new`, and the synchronization action `m`. For the rules corresponding to the generic operations the reader is referred to [CKS05]. The premises of the rules presented in Table 1

**Table 1.** Inference rules for the labeled transition relation $\mathcal{R}$ for thread $T_i$.

(New-action)

$$\frac{Instr(s.pc) = \texttt{new}}{s \xrightarrow{\texttt{new}}_i (s', \overline{s})\ \ [s'.pc = s.pc + 1; \overline{s} = Init_i]}$$

(**in**-action)

$$\frac{Instr(s.pc) = \texttt{go\_in}(\ell) \wedge (\exists \ell_1.\ell_1 := s.\eta(i) \wedge s.\varphi(\ell_1) = s.\varphi(\ell))}{s \xrightarrow{\texttt{go\_in}(\ell)}_i s'\ \ [s'.pc = s.pc + 1; s'.\varphi = s.\varphi \cup \{\ell_1 \mapsto \ell\}]}$$

(**out**-action)

$$\frac{Instr(s.pc) = \texttt{go\_out}(\ell) \wedge (\exists \ell_1.\ell_1 := s.\eta(i) \wedge s.\varphi(\ell_1) = \ell)}{s \xrightarrow{\texttt{go\_out}(\ell)}_i s'\ \ [s'.pc = s.pc + 1; s'.\varphi = s.\varphi \cup \{\ell_1 \mapsto s.\varphi(\ell)\}]}$$

(Sync-action)

$$\frac{Instr(s.pc) = \texttt{m}}{s \xrightarrow{\texttt{m}}_i s'\ \ [s'.pc = s.pc + 1]}$$

**Table 2.** The labeled transition relation for the parallel composition of two threads.

(Sync-action)

$$\frac{a \in \Sigma_1^S \wedge s^1 \xrightarrow{a}_1 s'^1 \wedge a \in \Sigma_2^S \wedge s^2 \xrightarrow{a}_2 s'^2 \wedge s^1.\eta(1) = s^2.\eta(2)}{(s^1, s^2) \xrightarrow{a} (s'^1, s'^2)}$$

(L-Par)

$$\frac{a \in \Sigma_1^M \wedge s \xrightarrow{a}_1 s'^1}{(s^1, s^2) \xrightarrow{a}_1 (s'^1, s^2)}$$

(R-Par)

$$\frac{a \in \Sigma_2^M \wedge s^2 \xrightarrow{a}_2 s'^2}{(s^1, s^2) \xrightarrow{a}_2 (s^1, s'^2)}$$

represent guarded conditions for the execution of the actions. All rules check the value of $Instr(s.pc)$, which determines the instruction to be executed by the running thread. Then, depending on the type of the action, they check further guarding conditions. In the consequences of the inference rules, we describe (within square brackets) the updates of the thread state caused by the execution of an action. We use the standard notation $\varphi \cup \{\ell_1 \mapsto \ell_2\}$ (with $\ell_1, \ell_2 \in Loc$) to indicate the update to function $\varphi$, i.e., the updates to the location net.

In the case of a "`new`" action, thread $T_i$ spawns a new thread that is an exact copy of itself. As a consequence, the program counter of $T_i$ is updated, and a new thread is created with an initial state $\overline{s}$. The initial state is a copy of the initial state of $T_i$.

In the case of a "`go_in(`$\ell$`)`" action, if $\ell$ is a sibling location to thread $T_i$'s location (i.e., $s.\varphi(s.\eta(i)) = s.\varphi(\ell)$), then the thread makes a transition and changes the state accordingly: the program counter *pc* is incremented, and the location net is updated ($\ell$ is now the father location of $T_i$ location). If $\ell$ is not a sibling location, then the action is not performed because the guard does not hold.

In the case of a "`go_out(`$\ell$`)`" action, if $\ell$ is the father location to thread $T_i$'s location (i.e., $s.\varphi(s.\eta(i)) = \ell$), then the thread makes a transition and changes the state accordingly: the program counter *pc* is incremented, and the location net is updated ($\ell$ is now a sibling location of $T_i$'s location). If $\ell$ is not the father location, then the action is not performed because the guard does not hold.

Note that the subtle features of mobile programs (namely, location, location net and unbounded thread creation) are modeled explicitly.

Let $T_1, \cdots, T_n$ be a set of threads initially present in the mobile program $\mathcal{P}$, then $\mathcal{P} = T_1 \parallel \cdots \parallel T_n$. The parallel composition operation is defined as follows.

**Definition 2 (Mobile Program).** Let thread $T_1 = (S_1, Init_1, AP_1, \mathcal{L}_1, \Sigma_1, \mathcal{R}_1)$ and thread $T_2 = (S_2, Init_2,$

$AP_2, \mathcal{L}_2, \Sigma_2, \mathcal{R}_2)$ be two Labeled Kripke structures. Then their composition is defined as follows: $T_1 \parallel T_2 = (S_1 \times S_2, Init_1 \times Init_2, AP, \mathcal{L}, \Sigma_1 \cup \Sigma_2, \mathcal{R})$ with the labeled transition relation defined in Table 2.

In Table 2, a single state belonging to thread $T_i$ is denoted by $s^i$, i.e., with $i$ as superscript to indicate the thread number. When needed, we also use a subscript (and variable $j$) to indicate the position of an element in the path of execution steps (see Definition 3). For example, $s_1^i$ is the initial state of thread $T_i$. Given a state $s^i \in S_i$ of thread $T_i$, $s^i.V_l$, $s^i.V_g$, $s^i.pc$, $s^i.\varphi$ and $s^i.\eta$ are the values of local variables $V_l$, of global variables $V_g$, of program counter $pc$, of $\varphi$ and of $\eta$, respectively. Moreover, $Instr(s^i.pc)$ denotes the instruction pointed by $pc$ in thread $T_i$ at state $s^i$. Note that $\forall i, j, i \neq j, \Sigma_i \cap \Sigma_j = \Sigma^S$, that is, threads share only synchronization actions. In other words, threads proceed independently on local actions and synchronize on shared actions ($m \in \Sigma^S$), or on shared data (by definition of $S_i$, $S_1 \cap S_2 \neq \emptyset$). This notion of composition is derived from CSP [Ros98].

The definition of a path of a mobile program reflects the possibility of unbounded thread creation during the execution of the `new` instruction.

**Definition 3 (Path).** A *path* $\pi = \langle (s_1^1, s_1^2, \ldots, s_1^{n_1}), a_1, (s_2^1, s_2^2, \ldots, s_2^{n_2}), a_2, \ldots \rangle$ of a mobile program is an alternating (possible infinite) sequence of tuples of states and events such that:

(i) $n_j \in \mathbb{N}$ and, $\forall i, j \geq 1, s_1^i = Init_i$, $s_j^i \in S_i$, and $a_j \in \cup_i \Sigma_i$;

(ii) either $s_j^i \xrightarrow{a_j} s_{j+1}^i$ or $s_j^i = s_{j+1}^i$ for $1 \leq i \leq n_{j+1}$;

(iii) if $a_j =$ `new`:

- then $n_{j+1} = n_j + 1$ and $s_{j+1}^{n_{j+1}} = Init_k$ with $s_j^k \xrightarrow{a_j} s_{j+1}^k$
- else $n_{j+1} = n_j$.

A path includes *tuples* of states, rather than a single state. The reason for that is that when a `new` operation is executed, the state of the newly created thread must be recorded. Our notation indicates each state $s_i^j$ by two indices, $i$ and $j$, one to indicate the thread number, the other one to indicate the position in the path, respectively. The size of the tuple of states (i.e., the number of the currently existing threads) increases only if a `new` is executed, otherwise it remains unchanged (case (*iii*)). In case of a `new`, index $k$ identifies the thread that performed the action. Thus, the state of the newly created thread is a copy of the initial state of thread $T_k$. Moreover, depending on the type of action (i.e., shared or local) one or more threads will change state, whereas the others do not change (case (*ii*)).

**Example 6.** Consider the following location-aware thread:

```
go_in(ℓ); m; go_out(ℓ); new; new;
```

Then, a possible path $\pi$ formalizing the semantics of the thread is:

$$\langle s_1^1, \texttt{go\_in(l)}, s_2^1, \texttt{m}, s_3^1, \texttt{go\_out(l)}, s_4^1, \texttt{new}, (s_5^1, s_1^2), \texttt{new}, (s_6^1, s_2^2, s_1^3) \rangle.$$

◇

## 5. Generic Security-Policies Specification Language

In order to support features of mobile systems, we devised a policy specification language that defines rules for expressing also the code location. In this case, the active entities (i.e., subjects) requesting access to objects are the running threads, whereas the resources (i.e., objects) that may be accessed are methods and variables. The language is able to express rules in which an agent (i.e., a running thread) may be granted or denied to perform a specific method call, or to invoke a method call with specific arguments (as in the mandatory access control model). (Global) variables may be marked as having a high security level, and they cannot be assigned to variables of a lower level; it is also possible to specify the security level of the arguments of a method (the no read up and no write down rules of the information flow model). In this way, it is possible to express both access control policies and information flow with the same language.

The BNF specification of the language is depicted in Figure 3, where terminals appear in Courier, non terminals are enclosed in angle brackets, optional items are enclosed in square brackets, items repeated one

⟨policy⟩ ⟶ {⟨sec_levels⟩ | ⟨operation_def⟩ | ⟨deny statement⟩}
⟨deny statement⟩ ⟶ `deny␣to` ⟨deny_target⟩ [⟨code base⟩] [⟨code origin⟩]
`{` ⟨permission entry⟩ {`,` ⟨permission entry⟩} `}`
⟨deny_target⟩ ⟶ `public` | ⟨entity list⟩
⟨entity list⟩ ⟶ ⟨entity_id⟩ {`,` ⟨entity_id⟩}
⟨entity_id⟩ ⟶ ⟨location_id⟩
⟨location_id⟩ ⟶ ⟨identifier⟩
⟨identifier⟩ ⟶ (⟨letter⟩ | ⟨symbol⟩) {⟨letter⟩ | ⟨digit⟩ | ⟨symbol⟩}
⟨symbol⟩ ⟶ `_` | `.`
⟨code base⟩ ⟶ `codeBase` ⟨IPv4 addr⟩
⟨code origin⟩ ⟶ `codeOrigin` (⟨location⟩ | `remote`)
⟨location⟩ ⟶ ⟨location_id⟩ {: ⟨location_id⟩}
⟨permission entry⟩ ⟶ `permission` ⟨action⟩
⟨action⟩ ⟶ ⟨function⟩ | ⟨operation⟩
⟨function⟩ ⟶ `function` ⟨function_id⟩ ⟨parameters⟩
⟨function_id⟩ ⟶ ⟨identifier⟩
⟨parameters⟩ ⟶ ⟨actual par⟩ | ⟨formal par⟩ | `high` | $\varepsilon$
⟨actual par⟩ ⟶ `"` ⟨string⟩ `"`
⟨formal par⟩ ⟶ `args` ⟨vars⟩ | `"` ⟨location_id⟩ `"` [`*`]
⟨vars⟩ ⟶ ⟨identifier⟩ {`,` ⟨identifier⟩}
⟨operation_def⟩ ⟶ ⟨operation ⟩ `{` ⟨function_id⟩ {`,` ⟨function_id⟩} `}`
⟨operation⟩ ⟶ `operation` ⟨operation_id⟩
⟨operation_id⟩ ⟶ ⟨identifier⟩
⟨sec_levels⟩ ⟶ `High={` ⟨vars⟩ `}`

**Fig. 3.** The Policy Specification Language.

or more times are enclosed in curly brackets, and alternative choices in a production are separated by the | symbol.

A policy might contain: (*i*) the definition of *security levels*; (*ii*) a list of *operation* definitions; and (*iii*) a list of *deny* statements, each one including one or more *permission* entries. In the definition of security levels, we enumerate the high-level variables to specify a multi-level security policy. The definition of an *operation* collects together functions with the same meaning or side-effect (e.g., scanf and fread). A deny statement specifies which types of actions are not allowed to entities. By default, in the absence of deny statements, all actions are allowed to every possible user.

The *entities* to deny permissions to consist of processes (e.g., agents), identified by their current location. The keyword `public` means that the permission is denied to all entities. As we are dealing with mobile systems, an entity can also be identified by the host address (via `codeBase`), or by the location (via `codeOrigin`) it came from. The keyword `remote` identifies non-local locations.

A *permission* entry must begin with the keyword `permission`. It specifies *actions* to deny. An action can be a function (either user-defined or from the standard library), or an operation (a collection of functions). If it is a function, it is possible to also specify (*i*) formal parameters (variable names), (*ii*) actual parameters (the value of the arguments passed), (*iii*) an empty string, denying access to the function regardless of the arguments to it, or (*iv*) the keyword `high` (no high variables can be passed as arguments to this function). Notably, an actual parameter may be a location (a trailing `*` prevents not only the location, but all sublocations too).

Consider the Java sandbox: it is responsible for protecting a number of resources by preventing applets

from accessing the local hard disk and the network. In our language, a sketch of this security policy could be expressed as:

```
operation read_file_system { fread, read, scanf, gets}
deny to public codeOrigin remote
{ permission function connect_to_location,
  permission operation read_file_system }
```

This is an example of access control policy, where the request to either open a TCP connection or to access a file in the local file system is granted or denied depending on the credentials of the requester. In this case the decision is taken according to the origin of the code making the request.

With the same language, it is possible to express a multi-level security policy as:

```
High={confidential_var, x}
deny to public codeOrigin remote
{ permission function fopen high}
```

In this case, two variables are labeled as high (i.e., they contain confidential information), whereas a code coming from a remote and possibly unknown site is considered untrusted, thus labeled as low. The deny statement expresses the "no read up" rule of the multi-level security policy described in the previous section.

## 6. A Model-Checking Framework for Verification of Security Policies

We implemented a prototype framework for the security analysis of mobile programs (shown in Figure 4). It uses a model checker for exhaustive and static analysis of mobile systems. A mobile program, $P$, and a security policy, $S$, are provided as an input to the model-checking engine. It is also possible to provide a configuration file giving details on the initial network configuration (i.e., the *location net*). If the file is not given, the policy is checked against all possible network configurations.

These inputs are processed, creating a new program, $P'$, annotated with the security invariants. The procedure for annotating the program with security invariants is a multi-step process. First, the intersection of methods in the security policy and methods used within the agent to verify is found. Then, a wrapper method is created for each of these methods. This wrapper contains an `assert(0)`, either unconditionally, or within a guard, based on the policy (this may check where the agent came from, and/or the arguments being passed to the method). The handling of high-variable access is more complex (due to scoping and syntax), but analogous. This annotating procedure is the implementation of Schneider's security automata [Sch00]. In fact, the annotated program $P'$ consists of program $P$ with inlined the reference monitor that enforces the security policy $S$. It has the following property: an assertion `assert(0)` (a security invariant) is not reachable in $P'$ if and only if $P$ enforces the security policy $S$. Thus, it is sufficient to give $P'$ as an input to a model checker to statically determine whether or not an `assert(0)` is reachable in $P$.

We experimented with mobile programs written in a C-based mobile language. The C programming language is a popular choice for programming mobile middleware. For example, TinyOS, an operating system designed for wireless embedded sensor networks, is a dialect of C with support for concurrency model and component-based programming. We developed a mobile code framework from which different benchmarks of different complexity can be instantiated[3]. In particular, it can be used for instantiation of mobile code agents of a shopping client-server system [Whi94] and an updating system [BNL02] (the benchmarks are described in more detail later in this section). The mobile agents can be instantiated in two types of communication model: a *pull* model (i.e., client/server), where a client asks for code to run from a server, and then downloads and executes it, and a *push* model, where a location listens for incoming code (i.e., agents), and then executes them. Since serialization is not possible in C, our framework only allows code mobility (i.e., applet sending), running code cannot migrate. It is straightforward to extend our approach to benchmarks written in other programming languages (e.g., Telescript, Klaim, Java, JavaScript, etc.) by implementing/using a different front-end of a favorite model checker targeted to the language of choice and with full mobility capabilities.

Our framework uses a model-checking toolset, SATABS [CKSY05], which automatically extracts LKSs from ANSI-C programs. Applying model checking to the analysis of mobile and multi-threaded systems is

[3] The mobile code framework is available at http://www.verify.inf.unisi.ch/projects/AVSPMC/tool

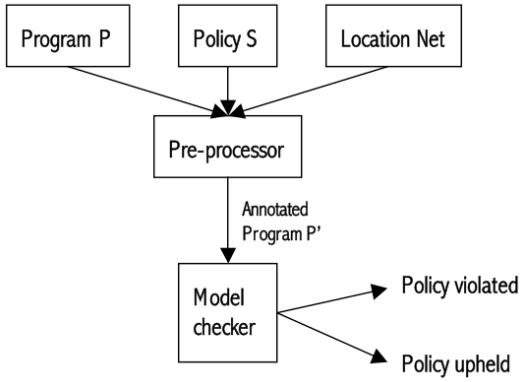


**Fig. 4.** The experimental framework.

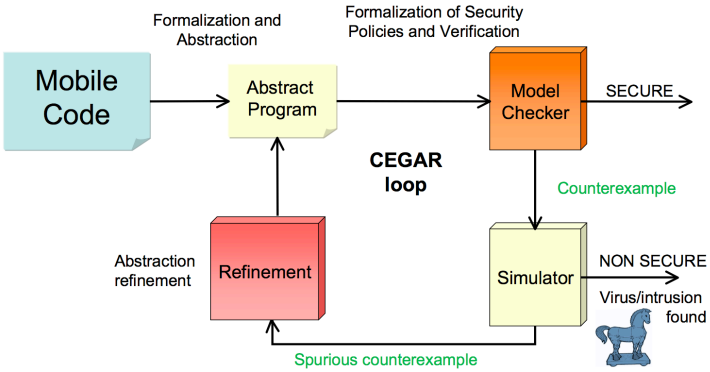


**Fig. 5.** The CEGAR loop.

complicated by several factors, ranging from the perennial scalability problems to thread creation that is potentially unbounded and that thus leads to infinite state space. *Predicate abstraction* is one of the most popular and widely applied methods for systematic state-space reduction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an overapproximation of the original program. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm [Kur95]. SATABS implements the abstraction refinement loop by computing and refining abstract programs (see Figure 5). The procedure for the location-specific projections described in Section 6.1 can be seen as the extension of SATABS's abstraction procedures. Among various techniques employed by SATABS, there is a model checker for Boolean programs (computed by the SATABS abstraction engine), Boppo[CKS05] that handles unbounded thread creation. The execution of a `new` action potentially leads to the creation of an unbounded number of new threads. SATABS implements a symbolic algorithm for overapproximating reachability in Boolean programs to support arbitrary thread creation which is guaranteed to terminate [CKS06]. The devised algorithm is used as the underlying reachability engine in the CEGAR framework and is efficient. The SATABS ability to handle programs with arbitrary thread creation was the key reason for using it as a model-checking engine of our security framework.

The initial configuration file, when provided, is used to test whether the policy is upheld in specific network configurations (where the agent came from, both on the underlying network and the location network, and where it's currently running). Several functions exist within the mobile code framework to check these values; there is a dynamic version to be used at run-time, and a static version which is generated from the specified initial configuration. To check whether the policy holds under all possible conditions, it suffices to not provide these function definitions to the model checker, which then treats the results as non-deterministic; this can

be accomplished by telling the model checker to use the run-time version of the definitions, not providing an initial configuration, or by not providing the generated location file as an input to the verifier.

### 6.1. Security and Projection

To cope with the computational complexity of verifying mobile programs, we define *projection* abstractions. Given a path of a multi-threaded program $T_1 \parallel \cdots \parallel T_n$, one can construct projections by restricting the path to the actions in the alphabet of threads, or to states satisfying some conditions. We exploit the explicit notion of locations and define the location-based projections, which allow efficient verification of location-specific security policies (security policies in which `codeOrigin` or `codeBase` is present). With a location-specific policy, only processes which run on the indicated location need to be verified.

In the following, we assume only paths of finite length, as they are computed by the symbolic fix-point algorithm to handle verification of systems with an unbounded number of threads. In addition, we write $\langle\rangle$ for the empty path, and we use the dot notation to denote the concatenation of sequences. The concatenation of sequences will be used in the inductive definitions of projections to concatenate subsequences of paths. Notice that . is the concatenation operator for sequences of characters, thus it is not affected by the presence of mismatched parentheses.

**Definition 4 (Location Projection, $\pi \downarrow \ell$).** Let $\mathcal{P}$ be $T_1 \parallel \cdots \parallel T_n$ and $\ell \in Loc$ be a location. The projection function $Proj_\ell : L(\mathcal{P})^* \rightarrow L(\mathcal{P})^*$ is defined inductively as follows (we write $\pi \downarrow \ell$ to mean $Proj_\ell(\pi)$):

1. $\langle\rangle \downarrow \ell = \langle\rangle$
2. If $s^i.\eta(i) = \ell$ then $(\langle s^i\rangle.\pi) \downarrow \ell = \langle s^i\rangle.(\pi \downarrow \ell)$
3. If $s^i.\eta(i) \neq \ell$ then $(\langle s^i\rangle.\pi) \downarrow \ell = \pi \downarrow \ell$
4. If $a \in \Sigma_i$, with $i$ s.t. $s^i.\eta(i) = \ell$, then $(\langle a\rangle.\pi) \downarrow \ell = \langle a\rangle.(\pi \downarrow \ell)$
5. If $a \notin \Sigma_i$, with $i$ s.t. $s^i.\eta(i) \neq \ell$, then $(\langle a, (s^1, s^2, \ldots, s^n)\rangle.\pi) \downarrow \ell = \pi \downarrow \ell$

This projection traces the execution of threads for a particular location. The following information is collected: (i) states of threads whose location is $\ell$ (i.e., threads $T_i$ such that $s^i.\eta(i) = \ell$), and (ii) actions that are performed by the threads whose location is $\ell$ (i.e., actions $a$ such that $a \in \Sigma_i$, with $\ell[\![ T_i ]\!]$). Here, the concatenation is done on each state element of the path, since each single state is examined to satisfy condition (i) (rules 2-3). On the contrary, once an action does not satisfy condition (ii), the next tuple is erased (rule 4).

With respect to what happens at a particular location during execution of a mobile program, there is no loss of precision in this projection-based abstraction. The projection removes only states and actions which are irrelevant to the particular location. Moreover, since security policies are defined in terms of a single location, this abstraction does not introduce spurious counterexamples during the verification of security policies using the `codeOrigin` entry.

**Definition 5 (Moving Projection, $\pi \downarrow_M i$).** Let $\mathcal{P}$ be $T_1 \parallel \cdots \parallel T_n$. For $1 \leq i \leq n$, the projection function $Proj_i^M : L(\mathcal{P})^* \rightarrow L(\mathcal{P})^*$ is defined inductively as follows (we write $\pi \downarrow i$ to mean $Proj_i(\pi)$):

1. $\langle\rangle \downarrow_M i = \langle\rangle$
2. $(\ \langle(s_1^1, s_1^2, \ldots, s_1^n)\rangle.\pi\ ) \downarrow_M i = \langle s_1^i\rangle.(\pi \downarrow_M i)$
3. If $a \in \Sigma_i^M$ then $(\langle a_j, (s_j^1, s_j^2, \ldots, s_j^n)\rangle.\pi) \downarrow_M i = \langle a_j, s_j^i\rangle.$ $(\pi \downarrow_M i)$
4. If $a \notin \Sigma_i^M$ then $(\langle a_j, (s_j^1, s_j^2, \ldots, s_j^n)\rangle.\pi) \downarrow_M i = \pi \downarrow_M i$

Here $\Sigma^M = \{$ `go_in(l)`, `go_out(l)` $\}$. This allows keeping track of all the moving actions executed by thread $T_i$. Therefore, our framework is able to track the route followed by each thread.

Many other location-specific properties can be specified, e.g., which thread interacts with the other threads, in what order and under which conditions, etc. This ability is of paramount importance since mobility requires more than the traditional notion of authorization to run and to access information in certain domains: it involves the authorization to enter or exit certain domains. Thus we can exploit our verification framework for security analysis.

**Definition 6 (Thread Projection, $\pi \downarrow i$).** Let $\mathcal{P}$ be $T_1 \parallel \cdots \parallel T_n$. For $1 \leq i \leq n$, the projection function $Proj_i : L(\mathcal{P})^* \rightarrow L(\mathcal{P})^*$ is defined inductively as follows (we write $\pi \downarrow i$ to mean $Proj_i(\pi)$):

1. $\langle\rangle \downarrow i = \langle\rangle$
2. $(\ \langle(s_1^1, s_1^2, \ldots, s_1^n)\rangle.\pi\ ) \downarrow_M i = \langle s_1^i\rangle.(\pi \downarrow_M i)$
3. If $a \in \Sigma_i$ then $(\langle a_j, (s_j^1, s_j^2, \ldots, s_j^n)\rangle.\pi) \downarrow i = \langle a_j, s_j^i\rangle.$ $(\pi \downarrow i)$.
4. If $a \notin \Sigma_i$ then $(\langle a_j, (s_j^1, s_j^2, \ldots, s_j^n)\rangle.\pi) \downarrow i = \pi \downarrow i$.

Intuitively, the projection $\pi \downarrow i$ of $\pi = \langle(s_1^1, s_1^2, .., s_1^n), a_1, \ldots\rangle$ on $T_i$ consists of the (possibly infinite) subsequence $\langle s_1^i, a_1, \ldots\rangle$ obtained by removing all pairs $\langle a_j, (s_j^1, s_j^2, \ldots, s_j^n)\rangle$ for which $a_j \notin \Sigma_i$ and by removing all states $s_j^t$ where $1 \leq t \leq n$ and $t \neq i$ (i.e., those states that do not belong to $T_i$).

As a consequence of Definition 6, the following theorem extends similar standard results obtained for CSP, and enables efficient compositional analysis of multi-threaded programs: abstraction, counter-example validation, and refinement can all be done one thread at a time.

**Theorem 6.1 (Soundness).** Let $\pi$ be a path, and $\parallel$ be defined as in Table 2, then:

1. Parallel composition is (well defined and) associative and commutative up to $\sim$-equivalence.
2. Let $T_1, \ldots, T_n$ be LKSs, and let $A_1, \ldots, A_n$ be the respective abstractions of the $T_i$: for each $i$, $T_i \sqsubseteq A_i$. Then, $T_1 \parallel \cdots \parallel T_n \sqsubseteq A_1 \parallel \cdots \parallel A_n$.
3. Let $T_1, \ldots, T_n$ be LKSs with alphabets $\Sigma_1, \ldots, \Sigma_n$, and let $\pi$ be a path of $T_1 \parallel \cdots \parallel T_n$. Then, $\pi \in L(T_1 \parallel \cdots \parallel T_n)$ iff, for each $i$, there exists $\pi_i' \in L(T_i)$ such that $\pi \downarrow i$ is a prefix of $\pi_i'$.

*Sketch of the Proof.* The correctness of the theorem follows directly from the definitions 3 and 6, and from the properties of set union. For the full proof we refer the reader to [Ros98].

The theorem states that parallel composition preserves the abstraction relation, and that it is possible to check whether a path belongs to the language of a mobile program by projecting and examining the path on each individual thread separately.

## 6.2. Experimental Results

To validate the theoretical concepts presented in this paper, an experimental mobile code framework was developed, for which a number of examples of mobile code agents were generated. The mobile code agents were our running example on a shopping agent [Whi94] and an updating agent [BNL02]. The shopping example deals with a shopping query client, which sends several agents out to query simulated airline services in order to find available airfares. The agent is run on a simulated airline server, which is a distinct location on the location net from the original query client, and may be on a remote host. When the agent receives a reply, or fails to, it then tries to report back to the shopping query client.

The updating example specifies a central update server and several clients. The clients contact the server, and updates are sent as an executable agent whenever an update is available. This represents a way to keep the client software up to date, without forcing the client to poll the update server.

Both sets of benchmarks were parameterized in the number of clients. The mobile code framework was used to create mobile programs of various complexity to test the scalability of the verification framework. In both examples the mobile agent contains a possibly "malicious" action, that is, opening a connection to the location named "bad" for the shopping agent, and opening the /etc/passwd file for the updating agent example.

We verified the two examples against a number of security policies ranging from file access control to policies that conditionally allowed the use of mobile code APIs based on the `codeOrigin`, as well as with different initial location configurations. The security policies were either general or application-dependent. Moreover, we were able to validate our technique on systems of different complexities, by changing the number of agents instantiated.

The policies we used to test the updating agent, which opens the /etc/passwd file, are:

1. policy `none`, verifying the agent without any security policy, i.e., allowing any kind of action;
2. policy `a`, the Java-like policy described in Section 5:

```
operation read_file_system { fread, read, scanf, gets}
deny to public codeOrigin remote
{ permission function connect_to_location,
  permission operation read_file_system }
```

3. policy `b`, disallowing the reading of the /etc/passwd file if and only if the agent came from a remote location, whereas every other argument to fopen is allowed:

```
deny to public codeOrigin remote
{ permission function fopen "/etc/passwd"}
```

4. policy `c`, the security-level example policy also described in Section 5:

```
High={confidential_var, x}
deny to public codeOrigin remote
{ permission function fopen high}
```

The policies we used for the shopping agent, where an attempt to open a connection to the location named "bad" is done, are:

1. policy `none`, verifying the agent without any security policy, i.e., allowing any kind of action;
2. policy `no effect`, i.e., a policy with no effect on this example. It demonstrates that there is no performance penalty when no checks need to be added.
3. policy `a`, disallowing the opening of a TCP connection to any agent coming from location `airc`, which is the name of the site which sent out the shopping agent:

```
deny to public codeOrigin airc
{ permission function connect_to_location}
```

4. policy `b local`, disallowing the opening of a TCP connection to agent coming from anywhere, i.e., non local. In the location net specified in the initialization file, the agent's `codeOrigin` is local, thus the action must be allowed.

```
deny to public codeOrigin remote
{ permission function connect_to_location}
```

5. policy `b remote`, which is identical to the previous one. What changes is the location net specified in the initialization file. In this case the shopping agent's code origin is remote, thus the action must be denied.
6. policy `c codeBase`, specifying `codeBase` (an IPv4 origin address) instead of `codeOrigin`, and tailored to the "malicious action" found in the shopping agent (i.e., calls to `connect_to_location` with argument `bad` are denied, calls with all other arguments are accepted):

```
deny to public codeBase 127.0.0.1
{ permission function connect_to_location bad}
```

7. policy `c`, a policy exactly like the previous one, except that it does not specify a `codeBase` (and so, applies to all possible `codeBase`s):

```
deny to public
{ permission function connect_to_location bad}
```

Whereas in the updating agent example we did not give any initial configuration file, the initial location net given in input to the model checker for the shopping agent example is as follows:

```
location net: (air0 [127.0.0.1 3000] air1 [127.0.0.1 4000] airc [127.0.0.1 3333]
    airq [127.0.0.1 3555] bad  [127.0.0.1 4444])
current execution location: airc:a0
codebase host: 127.0.0.1
code origin: airc
```

The configuration file specifies the location net (both in terms of location identifiers and IP addresses), the execution location, the code origin and the codebase of the agent to be verified. In our experiments, those were thread *a0*, located in location *airc* with its IP address `127.0.0.1`.

**Table 3.** Agent benchmarks (pv = policy violated, ua = updating agent, sa = shopping agent).

| policy | time (s) | # iterations | # predicates | pv? |
|---|---|---|---|---|
| ua - none | 0 | 1 | 0 | no |
| ua - a | 10.888 | 2 | 11 | yes |
| ua - b | 34.812 | 14 | 18 | yes |
| ua - c | 0.194 | 1 | 3 | yes |
| sa - none | 0.001 | 1 | 0 | no |
| sa - no_effect | 0 | 1 | 0 | no |
| sa - a | 151.644 | 7 | 17 | yes |
| sa - b local | 100.234 | 12 | 36 | no |
| sa - b remote | 524.866 | 5 | 15 | yes |
| sa - c codeBase | 340.011 | 12 | 22 | yes |
| sa - c | 108.564 | 6 | 16 | yes |

The results of the experiments, with a location projection (where $\ell$=the agent's location) on the whole system, are given in Table 3, where we report: (*i*) the total verification time (in sec); (*ii*) the number of CEGAR loop iterations; (*iii*) the number of predicates indicating the complexity of the abstracted models; and (*iv*) in the column named `pv?`, if the policy to be verified is violated or not. The results reported in Table 3 demonstrate the applicability and scalability of our tool, although we intend to enlarge the set of benchmarks.

The benchmarks we used in our experiments were parameterized in the number of the shopping agent clients (for the shopping agent system) and in the number of clients (for the updating benchmark). The complexity of analysis grew exponentially with the number of agents/clients and was problematic (without using the location projection). We set a timeout limit of two hours for the model checker and it reached the timeout for experiments with cases consisting more than four clients/agents. Statistics provided in Table 3 is given for cases when the location projection was used to reduce the complexity of verification. The data reported in Table 3 corresponds to benchmarks with five clients/agents.

It demonstrates that the used location projects reduced the complexity considerably as witnessed by time needed to complete analysis.

Our tool correctly detected every security policy violation with no false positives. The experiments were run on a machine with Redhat Linux 7.3, kernel 2.4.18-27.7.xsmp, dual processor 2.791MHz machine with 3.78 GB RAM.

## 7. Conclusion

In this article, we introduced a framework for the modeling and verification of mobile programs. The mobile system semantics is defined as Labeled Kripke Structures, which encapsulate the essential features of mobile programs: namely, thread location, location distribution and thread-moving operations. We introduced a new data structure, called location net, which captures the hierarchical nesting of the thread distribution during the execution of mobile program. The LKS formalism preserves both the data and the communication structures of mobile systems, which outperforms the classical process-algebraic approach mostly focusing on the communication behavior only. The explicit modeling of these features enables the specification of generic security policies for mobile systems, making both access control and information flow policies possible to define. We implemented and validated our approach. The verification framework builds on an integrated model checking framework to support exhaustive analysis of security policies. It exploits abstraction-refinement techniques that not only allow the handling of unbounded state space, but also deal effectively with large systems.

**Acknowledgments.** This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA).

## References

[BC06] Paolo Bellavista and Antonio Corradi. *The Handbook of Mobile Middleware*. Auerbach Publications, Boston, MA, USA, 2006.

[BCH+04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The BLAST Query Language for Software Verification. In *LNCS*, pages 2–18. Springer, 2004.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[BLP76] David Elliot Bell and Leonard J. La Padula. Secure Computer Systems: Unified Exposition and Multics Interpretation. ESD-TR-75-306, MITRE MTR-2997, MITRE Corporation, March 1976.

[BNL02] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Software Update Via Mobile Agent Based Programming. In *SAC*, pages 32–36. ACM, 2002.

[BR00] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[BS95] O. Burkart and B. Steffen. Composition, Decomposition and Model Checking of Pushdown Processes. *Nordic Journal of Computing*, 2:89 – 125, 1995.

[BS03] P. Bidinger and J.B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, 2005, Proceedings*, volume 2884 of *Lecture Notes in Computer Science*. Springer, 2003.

[BSBA07] Chiara Braghin, Natasha Sharygina, and Katerina Barone-Adesi. Automated Verification of Security Policies in Mobile Code. In *Integrated Formal Methods (IFM07)*, volume 4591 of *LNCS*, pages 37–53, 2007.

[BSS05] Philippe Bidinger, Alan Schmitt, and Jean-Bernard Stefani. An Abstract Machine for the Kell Calculus. In *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, 2005, Proceedings*, volume 3535 of *Lecture Notes in Computer Science*, pages 43–58, June 2005.

[CA09] Pavol Cerný and Rajeev Alur. Automated Analysis of Java Methods for Confidentiality. In *Computer Aided Verification (CAV09)*, pages 173–187, 2009.

[Car99] Luca Cardelli. Wide Area Computation. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proc. of 26th Int. Colloquium in Automata, Languages and Programming (ICALP'99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 10–24. Springer–Verlag, Berlin, 1999. Invited Paper.

[CCK+06] Sagar Chaki, Edmund M. Clarke, Nick Kidd, Thomas W. Reps, and Tayssir Touili. Verifying Concurrent Message-Passing C Programs with Recursive Calls. In Holger Hermanns and Jens Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.

[CCO+04] Sagar Chaki, Edmund Clarke, Joel Ouaknine, Natasha Sharygina, and Nishant Sinha. State/Event-Based Software Model Checking. In *IFM 2004*, pages 128–147, 2004.

[CDZG+02] Witold Charatonik, Silvano Dal Zilio, Andrew D. Gordon, Supratik Mukhopadhyay, and Jean-Marc Talbot. Finite-control Mobile Ambients. In *Proc. of European Symposium on Programming (ESOP02)*, number 2305 in Lecture Notes in Computer Science, pages 295–313. Springer–Verlag, Berlin, 2002.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[CISW05] Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt C. Wallnau. The comfort reasoning framework. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 164–169. Springer, 2005.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *Proceedings of SPIN*, pages 75–90. Springer Verlag, 2005.

[CKS06] Byron Cook, Daniel Kroening, and Natasha Sharygina. Over-Approximating Boolean Programs with Unbounded Thread Creation. In *FMCAD 06: Formal Methods in System Design*. Springer-Verlag, 2006.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer Verlag, 2005.

[Cor99] IBM Corporation. Aglet Software Development Kit, 1999.

[CTTV04] Edmund Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by Network Decomposition. In *CONCUR 04*, pages 276–291. Springer-Verlag, 2004.

[Dis00] Dino Distefano. A Parametric Model for the Analysis of Mobile Ambients. In *3rd Asian Symposium on Programming Languages and Systems (APLAS 2005). Tsukuba, Japan. LNCS 3780, pp. 401-417, Springer 2005*, pages 305–326. Kluwer Academic Publishers, 2000.

[DNFP98] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[DoD85] US Department of Defence. *DoD Trusted System Evaluation Criteria, (The Orange Book)*, volume DOD 5200.28-STD. jun 1985.

[ES01] J. Esparza and S. Schwoon. A BDD-Based Model Checker for Recursive Programs. In *CAV*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *Proc. of the 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer–Verlag, Berlin, 1996.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-Modular Model Checking. In *Proceedings of the 10th International Workshop on Model Checking Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race Checking by Context Inference. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–13. ACM, 2004.

[HP00] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[HR98] Matthew Hennessy and James Riely. Resource Access Control in Systems of Mobile Agents. In *HLCL '98*, Journal of TCS, pages 3–17. Elsevier, 1998.

[ID96] C.N. Ip and D.L. Dill. Verifying Systems with Replicated Components in Mur$\phi$. In *Proceedings of CAV*, volume 1102, pages 147–158. Springer Verlag, 1996.

[Kur95] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, 1995.

[McL94] John McLean. Security Models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, Inc., 1994.

[Mic95] Sum Microsystems. The Java Language Specification, 1995.

[mob] MOBIUS (Mobility, Ubiquity and Security) European Project, `http://mobius.inria.fr`.

[NL97] George C. Necula and Peter Lee. Research on Proof-Carrying Code for Untrusted-Code Security. In *IEEE Symposium on Security and Privacy*, page 204, 1997.

[PA03] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[Rin01] Martin Rinard. Analysis of multithreaded programs. *Lecture Notes in Computer Science*, 2126, 2001.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[s3m] S3MS (Security of Software and Services for Mobile Systems) European Project, `http://www.s3ms.org`.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1), February 2000.

[SS04] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2004.

[Ste03] J.B. Stefani. A Calculus of Kells. *ENTCS*, 85(1), 2003.

[Sto00] Scott Stoller. Model-checking multi-threaded distributed Java programs. In *SPIN 00: International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, 2000.

[STT09] Natasha Sharygina, Stefano Tonetta, and Aliaksei Tsitovich. The synergy of precise and fast abstractions for program verification. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 566–573. ACM, 2009.

[Whi94] J.E. White. Telescript Technology: The Foundation of the Electronic Marketplace. Technical report, General Magic Inc., 1994.

[Yah01] Eran Yahav. Verifying Safety Properties of Concurrent Java Programs using 3-valued Logic. In *POPL*, pages 27–40, 2001.