Electronic Notes in Theoretical Computer Science 66 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 24 pages

Guardians for Ambient-Based Monitoring

G. Ferrari [1]

Dip. Informatica, Univ. di Pisa, Italy

E. Moggi [2]

Dip. Informatica e Scienze dell'Informazione, Univ. di Genova, Italy

R. Pugliese [3]

Dip. Sistemi e Informatica, Univ. di Firenze, Italy

Abstract

In the Mobile Ambients of Cardelli and Gordon an ambient is a unit for mobility, which may contain processes (data) and sub-ambients. Since the seminal work of Cardelli and Gordon, several ambient-based calculi have been proposed (Seal, Box-π, Safe Ambients, Secure Safe Ambients, Boxed Ambients), mainly for supporting security. At the operational level these (box- and) ambient-based calculi differ only in the capabilities of processes. We propose a way of extending ambient-based calculi, which embodies two principles: **an ambient is a unit for monitoring and coordination, the name of an ambient determines its (monitoring and coordination) policy**. More specifically, to each ambient we attach a guardian, which monitors the activity of sub-components (i.e. processes and sub-ambients) and the interaction with the external environment. In our proposal, guardians and processes play a dual role: guardians are centralized entities monitoring and inhibiting actions, while processes are decentralized entities performing actions. We exemplify the use of guardians for enforcing security properties.

1 Introduction

Specifying and verifying the security requirements of wide area network (WAN) applications is a difficult task. A WAN application may integrate heterogeneous computing environments having different security requirements. Moreover, the security policy cannot make any decision using knowledge of the

[1] Supported by MIUR project NAPOLI and EU project PROFUNDIS IST-2001-33100
[2] Supported by MIUR project NAPOLI and EU project DART IST-2001-33477
[3] Supported by MIUR project NAPOLI and by EU project AGILE IST-2001-32747

entire current state of the WAN application. Any realistic approach will identify which portion of the state of the WAN application is potentially relevant and may affect or be affected by the security policy decisions. Lastly, security policies are not static in general: the revocation of previously granted access rights must be supported. Hence, formal models of security must provide suitable mechanisms to specify and certify properties of a variety of security policies, including policy changes and dynamic policies.

In the last years, several research activities have explored approaches where security of WAN applications is enforced by programming language techniques: *language-based security*. Following Schneider, Morrisett and Harper [41], we can basically distinguish two different approaches to language-based security: *Program analysis*, and *Program transformation*.

The approach based on program analysis requires a full knowledge of the code of the application. The code is analyzed (either statically or dynamically) against the security policy. The approach ensures that only the code which does not violate the security policy will be granted the opportunity of being executed. The security types for access control of (see e.g. [11,18,29]) provide nice examples of this approach. Another interesting example is given by the (static and dynamic) techniques employed in modeling Java security permissions (see e.g. [48,38,3,25]).

In the approach based on program transformation the operations enforcing the security policies are merged into the code of the application. The underlying idea is that of a *reference monitor* which observes execution and halts the application whenever the application violates some security requirements. Here we can distinguish between *in-lined Reference Monitors* (IRM) and *Execution Monitoring* (EM). In the IRM approach the application code is enriched with some additional code performing security checks at run-time. This implies that the events that may affect the security policy are known in advance [21]. Execution monitoring applies when the application code is not available in advance. Hence the application code must be treated as a black-box and it must be wrapped inside a reference monitor [23].

This paper introduces a formal model capable of supporting specification of multiple dynamic security policies. The basic idea of the model is the separation between the computational mechanisms (communication and mobility primitives) and the managers, we call them *guardians*, that enforce the security policies. Each guardian basically defines a local security context that controls only a portion of the current state. However, a certain degree of coordination among guardians is needed to support inter-networking security policies. Our proposal is closely related to the idea of execution monitoring and guardians are like security automata [21] that specify the allowed interactions. The novelty of our approach is given by guardian coordination. The coordination of guardians enables to support a rich set of security policies. In other words, the security policies are programmable distributed entities.

The main features of our approach are summarized as follows

- The distinction between the computational mechanisms (processes act) and the security supports (guardians allow and deny) permits describing a variety of local security policies. Keeping the specification of actions and that of permissions separate is important also from a software engineering prospective.
- A guardian provides a unit to describe the local security policy: the set of permissions that may be granted to the agents entering, exiting and communicating in the monitored computational environment.
- Security decisions are the outcome of coordination activities among guardians which are location-aware. The coordinated security decisions control the propagation of access rights by ensuring that the permissions are in accordance with the local security policies.

In this paper, to demonstrate the applicability of the approach we introduce a calculus of guardians. We choose as starting point of our investigation the Boxed Ambient (BA) [5,6]. This variant of the mobile ambients calculus [13] drops the *open* capability and introduce fine-grain mechanisms for ambient interactions (inspired by the Seal calculus [15]). These changes provide better support for the specification of security policies. The calculus of guardians preserves the distinctive features of Boxed Ambients but it does not rely on a type system to enforce secure ambient interactions. In our approach ambients are equipped with guardians which monitor the activities of the components inside an ambient and their interactions with the external operational environment. Hence, each guardian defines and enforces a local security policy and guardian coordination fixes the permissions for ambient movements.

To illustrate our approach, we present a simple example of *resource access control* specified in BA. We consider a system s which consists of a resource r and a mobile process n that wants to use r

$$s[Q \mid n[P] \mid r[R]]$$

Let us assume that the resource r consists of a value v, thus R is the process $!\langle v\rangle.0$, and that using the resource amounts to read the value v. For instance, process P is $in\ r.(x)^{\uparrow}.P'$, i.e. n enters r and reads a value from the parent ambient r. We attach a guardian G_a to each ambient a, therefore in our calculus the system above becomes

$$s(G_s)[Q \mid n(G_n)[P] \mid r(G_r)[R]]$$

The expression $n(G)[P]$ indicates that ambient n runs process P under the control of guardian G. We exemplify the coordination among guardians by considering what happens when process P tries to perform the $in\ r$ action:

(i) G_n authorizes, within ambient s, migration (of ambient n) into r;

(ii) G_r authorizes, within ambient s, ambient n to enter inside (ambient r);

(iii) G_s authorizes the movement of ambient n inside ambient r.

In other words, the permission granted for the ambient movement is the result

of a coordination step of four entities: the process asking for the service and the three guardians G_n, G_r and G_s implementing the local security contexts of the ambients involved in the action. The agreement among four entities needed to perform ambient movements may appear to have a heavy run-time overhead. However, if one considers a low-level description of an ambient move (e.g. see [22,39]), it involves exactly three ambients and the process performing the movement action.

Notice that the coordination step adheres to the principles of *subsidiarity*, i.e. only the ambients directly involved in the action can forbid it, and *location awareness*, e.g. guardian G_n authorizes the movement only within the enclosing ambient s. The outcome of the move action is

$$s(G'_s)[Q \mid r(G'_r)[n(G'_n)[(x)^{\uparrow}.P'] \mid R]]$$

The reading of the value in r requires the coordination of four entities, namely the processes and guardians in r and n. In particular, the guardian G'_r must authorize the read from the child ambient n. The outcome of the read action is

$$s(G'_s)[Q \mid r(G''_r)[n(G''_n)[P'[x{:=}v]] \mid R]]$$

In this paper we will show that guardians are powerful enough to enforce dynamically a variety of security policies. For instance, focusing on the resource access control example, the guardian G_r can

- prevent certain ambients to enter depending on their name,
- allow a very limited pattern of interaction, e.g. a protocol of the form *read only once and then exit* (otherwise, the agent n could perform multiple read),
- perform check for intrusion detection , e.g. the agent n could be used as a Trojan horse by other ambients that want to enter r,
- discriminate among ambients having the same name by granting them different privileges.

The rest of the paper is organized as follows. In Section 2 we review Boxed Ambients. Section 3 introduces the calculus of guardians for Boxed Ambients and its basic semantic theory. Section 4 discusses several examples that illustrate the expressive power of guardians to enforce a wide range of security protocols. Finally, comparisons with related work are in Section 5.

2 Boxed Ambients: an ambient-based language

In this section we present Boxed Ambients [5] (BA, in the rest of the paper), an ambient-based language inspired by Mobile Ambients [13]. BA uses the same mobility primitives as Mobile Ambients, but relies on a different communication model inspired by the Seal calculus [15]. This model results from dropping the *open* primitive and permitting communication across ambient boundaries, between parent and children, in addition to local communication.

Our presentation of syntax and operational semantics of BA differs from the original one for a few aspects, namely:

- We do not distinguish between the syntactic categories of values and (sequential) processes;
- We consider additional process actions corresponding to name, process and ambient creation. This refinement makes a difference when we introduce guardians that may forbid a process action to occur, even when in the calculus without guardians the action was asynchronous and non-blocking.
- For better programmability, we introduce also sequential composition (which subsumes concatenation of capabilities), conditional choice, and recursive definitions (instead of replication).
- We define the transition relation for nets, i.e. processes modulo structural congruence. Intuitively, sequential processes (i.e. values) are syntactic entities, while nets are semantic entities that describe the state of a system at a given moment of the computation.

The syntax of the calculus is given by the following BNF

$$
\begin{aligned}
n \in \mathsf{Name} &::= \ldots \\
x \in \mathsf{Var} &::= \ldots \\
X \in \mathsf{PId} &::= \ldots \\
b \in \mathsf{Bool} &::= \bot \mid b_1 \vee b_2 \mid \top \mid b_1 \wedge b_2 \mid \neg b \mid v_1 = v_2 \quad \text{name equality} \\
a \in \mathsf{Act} &::= \mathit{new} \mid \mathit{spawn} \mid \mathit{box} \mid \mathit{in} \mid \mathit{out} \mid \mathit{wr} \mid \mathit{wrup} \mid \mathit{wrdn} \mid \mathit{rd} \mid \mathit{rdup} \mid \mathit{rddn} \\
v, P \in \mathsf{Proc} &::= n \mid x \mid a(\overline{v}, ?\overline{x}).P \mid P_1; P_2 \mid 1 \mid \mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid X(\overline{v})
\end{aligned}
$$

Syntactically, a process P may also be a name n or a variable x, of course semantically these are values but not processes (thus they do not give rise to process transitions). Action prefix $a(\overline{v}, ?\overline{x}).P$ represents a process that can perform action a: $\overline{v}$ is a sequence of values (actual parameters), and $\overline{x}$ is a sequence of variables (formal parameters), which may occur free in P. The formal parameters $\overline{x}$ are replaced by actual parameters $\overline{v}'$ through synchronization (with other processes), thus $a(\overline{v}, ?\overline{x}).P$ binds the variables $\overline{x}$ in P. $P_1; P_2$ is sequential composition, and 1 denotes successful termination. Finally, we have conditional choice and process invocation $X(\overline{v})$, where we assume that for each process identifier X (of arity n) there exists a unique defining equation $X(\overline{x}) \triangleq P$ (where $\overline{x}$ has length n).

With respect to the original presentation of BA, we use three additional actions: $\mathit{new}(?x).P$ is a request for a fresh name to a name generator, thus it corresponds to $(\nu x)P$ of BA; $\mathit{spawn}(P_1).P_2$ activates P_1 in parallel with P_2, thus it corresponds to $P_1|P_2$ of BA; $\mathit{box}(n, P_1).P_2$ spawns P_1 in a new ambient named n, thus it corresponds to $n[P_1]|P_2$ of BA.

The remaining actions directly correspond to those of BA. The actions $in(n).P$ and $out(n).P$ are for ambients movement. The actions $wr(\overline{v}).P$ and $rd(?\overline{x}).P$ are for local communication: $wr(\overline{v}).P$ outputs the sequence of values $\overline{v}$, $rd(?\overline{x}).P$ inputs a sequence of values and binds them to the sequence of variables $\overline{x}$. The actions $wrup(\overline{v}).P$ and $rdup(?\overline{x}).P$ are the actions for communication with the parent and work similarly, while $wrdn(n, \overline{v}).P$ and $rddn(n, ?\overline{x}).P$ are for communication with a child named n.

In [5], the operational semantics of BA is defined in terms of a structural congruence and a reduction relation over processes. Here, the operational semantics is given by a transition relation $\Longrightarrow$ over *nets*, where a net N is basically a process modulo structural congruence. Net transitions $F, N \Longrightarrow F', N'$ are defined in term of process transitions $P \xrightarrow{a(\overline{v})} P'$. This distinction is convenient, since guardians are synchronized with process transitions (not with net transitions).

Process termination (denoted by $P\downarrow$) is a predicate which expresses whether P *may* successfully terminate. This predicate is exploited for defining the transitions of the sequential composition of processes.

$$1 \ \frac{}{1\downarrow} \qquad \text{rec}\ \frac{P[\overline{x}:=\overline{v}]\downarrow}{X(\overline{v})\downarrow}\ \text{if } X(\overline{x}) \stackrel{\Delta}{=} P \qquad ;\ \frac{P_1\downarrow \quad P_2\downarrow}{P_1; P_2\downarrow}$$

$$\text{ift}\ \frac{b\ \mathsf{true} \quad P_1\downarrow}{\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\downarrow} \qquad \text{iff}\ \frac{b\ \mathsf{false} \quad P_2\downarrow}{\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\downarrow}$$

Process transition $P \xrightarrow{a(\overline{v})} P'$, where $\overline{v}$ is a sequence of closed values (i.e. without free variables), is a predicate saying that P may perform action $a(\overline{v})$ and become P'. The rules for action prefix are of the form

$$\text{a}\ \frac{}{a(\overline{v}, ?\overline{x}).P \xrightarrow{a(\overline{v},\overline{v}')} P[\overline{x}:=\overline{v}']}$$

with restrictions on $\overline{v}$ and $\overline{v}'$ that depend on a.

$$\text{new}\ \frac{}{new(?x).P \xrightarrow{new(n)} P[x:=n]} \qquad \text{spawn}\ \frac{}{spawn(P_1).P_2 \xrightarrow{spawn(P_1)} P_2}$$

$$\text{box}\ \frac{}{box(n, P_1).P_2 \xrightarrow{box(n_1,P_1)} P_2}$$

$$\text{in}\ \frac{}{in(n).P \xrightarrow{in(n)} P} \qquad \text{out}\ \frac{}{out(n).P \xrightarrow{out(n)} P}$$

$$\text{wr}\ \frac{}{wr(\overline{v}).P \xrightarrow{wr(\overline{v})} P} \qquad \text{rd}\ \frac{}{rd(?\overline{x}).P \xrightarrow{rd(\overline{v})} P[\overline{x}:=\overline{v}]}$$

$$\text{wrup}\ \frac{}{wrup(\overline{v}).P \xrightarrow{wrup(\overline{v})} P} \qquad \text{rdup}\ \frac{}{rdup(?\overline{x}).P \xrightarrow{rdup(\overline{v})} P[\overline{x}:=\overline{v}]}$$

$$\text{wrdn}\ \frac{}{wrdn(n,\overline{v}).P \xrightarrow{wrdn(n,\overline{v})} P} \qquad \text{rddn}\ \frac{}{rddn(n,?\overline{x}).P \xrightarrow{rddn(n,\overline{v})} P[\overline{x}:=\overline{v}]}$$

The rest of the rules are fairly standard.

$$\text{rec}\ \frac{P[\overline{x}:=\overline{v}] \xrightarrow{a(\overline{v})} P'}{X(\overline{v}) \xrightarrow{a(\overline{v})} P'}\ \text{if } X(\overline{x}) \triangleq P$$

$$;\text{f}\ \frac{P_1 \xrightarrow{a(\overline{v})} P_1'}{P_1; P_2 \xrightarrow{a(\overline{v})} P_1'; P_2} \qquad ;\text{c}\ \frac{P_1 \downarrow \quad P_2 \xrightarrow{a(\overline{v})} P_2'}{P_1; P_2 \xrightarrow{a(\overline{v})} P_2'}$$

$$\text{ift}\ \frac{b\ \mathsf{true} \quad P_1 \xrightarrow{a(\overline{v})} P_1'}{\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{a(\overline{v})} P_1'} \qquad \text{iff}\ \frac{b\ \mathsf{false} \quad P_2 \xrightarrow{a(\overline{v})} P_2'}{\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{a(\overline{v})} P_2'}$$

The rules for conditional choice rely on the evaluation of boolean expressions b. The rules for deriving $b\ \mathsf{true}$ and $b\ \mathsf{false}$ are obvious, only those for name equality deserve to be mentioned

$$=\text{t}\ \frac{}{n = n\ \mathsf{true}} \qquad =\text{f}\ \frac{}{n_1 = n_2\ \mathsf{false}}\ \text{if } n_1 \neq n_2$$

Notice that the boolean expression $v_1 = v_2$ has a value only when v_1 and v_2 are names, since in the other cases the comparison makes no sense.

Remark 2.1 We do not define an observational congruence on processes, but any such congruence will validate at least the equations saying that $(1, ;)$ forms a monoid. □

We can now define net transitions for BA. Formally, a *net* $N \in \mathsf{Net} \triangleq \mu(\mathsf{Proc} + (\mathsf{Name} \times \mathsf{Net}))$ is a multi-set of processes P and named sub-nets $n[N]$. We define transitions on named nets $n[N]$, since we want to have always a top-level ambient n (with an associated guardian) that overlooks the evolution of the net (on the contrary in Ambient-like calculi, transitions are defined on generic processes). Net transitions are defined over configurations of the form $F, n[N]$, where $F \subseteq_{fin} \mathsf{Name}$ contains all the free names in $n[N]$. F keeps track of the names that are free in the whole net (i.e. names generated so far) and is needed to ensure global freshness of names. When a transition does not generate any fresh names, we write $n[N] \Longrightarrow n[N']$ instead of $F, n[N] \Longrightarrow F, n[N']$. Net transitions $F, n[N] \Longrightarrow F', n[N']$ are defined in

terms of a structural rule

$$\frac{F, n_2[N_2] \Longrightarrow F', n_2[N_2']}{F, n_1[N_1, n_2[N_2]] \Longrightarrow F', n_1[N_1, n_2[N_2']]}$$

and basic net transitions induced by process transitions. For each of these net transitions we name the ambients involved (actively or passively) in the transition (an ambient is active when one of its processes takes part to the action/interaction that causes the transition). The basic net transitions are classified in three groups.

Asynchronous and non-blocking transitions:

$$\text{new}\ \frac{P \xrightarrow{new(n')} P'}{F, n[N, P] \Longrightarrow F \cup \{n'\}, n[N, P']}\ n' \notin F \quad n \text{ is active}$$

$$\text{spawn}\ \frac{P \xrightarrow{spawn(P_1)} P_2}{n[N, P] \Longrightarrow n[N, P_1, P_2]}\quad n \text{ is active}$$

$$\text{box}\ \frac{P \xrightarrow{box(n_1, P_1)} P_2}{n[N, P] \Longrightarrow n[N, n_1[P_1], P_2]}\quad n \text{ is active}$$

Asynchronous and possibly blocking transitions:

$$\text{in}\ \frac{P \xrightarrow{in(n_2)} P'}{n[N, n_1[P, N_1], n_2[N_2]] \Longrightarrow n[N, n_2[n_1[P', N_1], N_2]]}$$

$$\text{out}\ \frac{P \xrightarrow{out(n_2)} P'}{n[N, n_2[n_1[P, N_1], N_2]] \Longrightarrow n[N, n_1[P', N_1], n_2[N_2]]}$$

In the rules above we assume that n_1 is active, and n, n_2 are passive.

Synchronous transitions:

$$\text{comm}\ \frac{P_1 \xrightarrow{wr(v)} P_1' \qquad P_2 \xrightarrow{rd(v)} P_2'}{n[N, P_1, P_2] \Longrightarrow n[N, P_1', P_2']}\quad n \text{ is active}$$

$$\text{rdup}\ \frac{P_1 \xrightarrow{wr(v)} P_1' \qquad P_2 \xrightarrow{rdup(v)} P_2'}{n_1[N_1, P_1, n_2[P_2, N_2]] \Longrightarrow n_1[N_1, P_1', n_2[P_2', N_2]]}\quad n_1 \text{ and } n_2 \text{ active}$$

$$\text{wrup}\ \frac{P_1 \xrightarrow{rd(v)} P_1' \qquad P_2 \xrightarrow{wrup(v)} P_2'}{n_1[N_1, P_1, n_2[P_2, N_2]] \Longrightarrow n_1[N_1, P_1', n_2[P_2', N_2]]}\quad n_1 \text{ and } n_2 \text{ active}$$

$$\text{rddn}\ \frac{P_1 \xrightarrow{rddn(n_2, v)} P_1' \qquad P_2 \xrightarrow{wr(v)} P_2'}{n_1[N_1, P_1, n_2[P_2, N_2]] \Longrightarrow n_1[N_1, P_1', n_2[P_2', N_2]]}\quad n_1 \text{ and } n_2 \text{ active}$$

$$\text{wrdn}\ \frac{P_1 \xrightarrow{wrdn(n_2,v)} P_1' \qquad P_2 \xrightarrow{rd(v)} P_2'}{n_1[N_1, P_1, n_2[P_2, N_2]] \Longrightarrow n_1[N_1, P_1', n_2[P_2', N_2]]}\quad n_1 \text{ and } n_2 \text{ active}$$

3 Guarded Boxed Ambients: an extension with guardians

In this section we describe Guarded Boxed Ambients (GBA for short), i.e. the extension of BA with guardians. The basic idea is that a guardian G perform transitions $G \xrightarrow{g(\overline{v})} G'$ (where $\overline{v}$ is a sequence of closed values) formally similar to process transitions. However, a guardian transition amounts to a permission for a process action (or interaction). The syntax of GBA is given by the grammar below. Here, we show only the new and extended syntactic categories, all the other categories are unchanged.

$$\begin{aligned}
W \in \mathsf{GId} &::= \ldots \\
g \in \mathsf{Perm} &::= \mathit{new} \mid \mathit{spawn} \mid \mathit{box} \mid \mathit{in}_1 \mid \mathit{in}_2 \mid \mathit{in}_3 \mid \mathit{out}_1 \mid \mathit{out}_2 \mid \mathit{out}_3 \mid \mathit{comm} \\
&\quad \mid \mathit{rdup}_1 \mid \mathit{rdup}_2 \mid \mathit{wrup}_1 \mid \mathit{wrup}_2 \mid \mathit{rddn}_1 \mid \mathit{rddn}_2 \mid \mathit{wrdn}_1 \mid \mathit{wrdn}_2 \\
G \in \mathsf{Gard} &::= g(\overline{x}) \ \mathsf{when}\ b \ \mathsf{then}\ G \mid G_1 \vee G_2 \mid W(\overline{v}) \mid \\
&\quad G \backslash g(\overline{x}) \ \mathsf{when}\ b \mid \bot \mid \top \mid G_1 \wedge G_2 \\
v, P \in \mathsf{Proc} &::= \ldots \mid G
\end{aligned}$$

Remark 3.1 Permissions are given by the syntactic category Perm. There is **one permission g for each basic net transition and each ambient n involved in the transition**. Intuitively, $g(\overline{v})$ means that the guardian G (of ambient n) permits the basic net transition, provided the parameters of the transitions (and the names of the other ambients involved) are those specified in $\overline{v}$. Therefore, **the role of guardians is exclusively to constrain the actions (and interactions) of processes**. □

We take as basic construct for guardians $g(\overline{x})$ when b then G, where g is a permission, $\overline{x}$ is a sequence of variables bound by the construct, b is a boolean expression, and G is a guardian. The other constructs for guardians are inspired by synchronous process calculi, like SCCS [33]. Of course, there are many other process algebra formalisms to choose from, but synchronous calculi appear particularly appropriate for stressing the *monolithic* nature of guardians, i.e. one guardian per ambient. A guardian G is fully specified by giving a system of equations $\langle W_i(\overline{x}) \triangleq G_i \mid i \in I \rangle$ defining the meaning of every W used in G and the G_i.

Remark 3.2 For defining guardians the *dynamic combinators* given in the first row of the BNF suffice. However, the *static combinators* given in the second row are quite convenient derived forms. □

The syntactic category Proc of processes and values is extended with a new production G, since values now include also guardians.

The operational semantics is given by a transition relation $\Longrightarrow$ over *nets with guardians*, which is defined in terms of process transitions $P \xrightarrow{a(\overline{v})} P'$ and guardian transitions $G \xrightarrow{g(\overline{v})} G'$.

- Process transitions are defined by the rules in Section 2, but the rule for name creation is now

$$\text{new}\ \frac{}{new(?x, G).P \xrightarrow{new(n,G[x:=n])} P[x := n]}$$

  In GBA the prefix $new(?x, G).P$ is a request for a fresh name, the guardian G is like a *type* for the new name, and the association name-guardian obeys the rules for static scoping.

- Guardian transitions are defined by the following rules

$$\text{allow}\ \frac{}{g(\overline{x})\ \mathsf{when}\ b\ \mathsf{then}\ G \xrightarrow{g(\overline{v})} G[\overline{x} := \overline{v}]}\ \overline{v}\ \text{closed}$$

$$\text{or}\ \frac{G_i \xrightarrow{g(\overline{v})} G'}{G_1 \vee G_2 \xrightarrow{g(\overline{v})} G'}\ i \in \{1, 2\} \qquad W\ \frac{G[\overline{x} := \overline{v}'] \xrightarrow{g(\overline{v})} G'}{W(\overline{v}') \xrightarrow{g(\overline{v})} G'}\ W(\overline{x}) \triangleq G$$

$$\text{deny}_1\ \frac{\neg b[\overline{x} := \overline{v}]\ \mathsf{true} \quad G \xrightarrow{g(\overline{v})} G'}{G \backslash g(\overline{x})\ \mathsf{when}\ b \xrightarrow{g(\overline{v})} G' \backslash g(\overline{x})\ \mathsf{when}\ b}$$

$$\text{deny}_2\ \frac{G \xrightarrow{g'(\overline{v})} G'}{G \backslash g(\overline{x})\ \mathsf{when}\ b \xrightarrow{g'(\overline{v})} G' \backslash g(\overline{x})\ \mathsf{when}\ b}\ g' \neq g$$

$$\text{and}\ \frac{G_1 \xrightarrow{g(\overline{v})} G_1' \quad G_2 \xrightarrow{g(\overline{v})} G_2'}{G_1 \wedge G_2 \xrightarrow{g(\overline{v})} G_1' \wedge G_2'} \qquad \top\ \frac{}{\top \xrightarrow{g(\overline{v})} \top}\ \overline{v}\ \text{closed}$$

Remark 3.3 We do not define an observational congruence on guardians, but any such congruence will validate at least the following equational properties:

- $(\bot, \vee)$ and $(\top, \wedge)$ form commutative idempotent monoids;
- distributivity of $(\bot, \vee)$ over $\wedge$, i.e. $\bot \wedge G = \bot$ and $(G_1 \vee G_2) \wedge G = (G_1 \wedge G) \vee (G_2 \wedge G)$;
- commutativity of $\_\backslash g(\overline{x})\ \mathsf{when}\ b$ with $\bot$, $\vee$ and $\wedge$, i.e. $\bot \backslash g(\overline{x})\ \mathsf{when}\ b = \bot$, $(G_1 \vee G_2) \backslash g(\overline{x})\ \mathsf{when}\ b = (G_1 \backslash g(\overline{x})\ \mathsf{when}\ b) \vee (G_2 \backslash g(\overline{x})\ \mathsf{when}\ b)$, and similarly for $\wedge$.

□

We can now define net transitions for GBA. Formally, a *net with guardians* $M \in \mathsf{GNet} \triangleq \mu(\mathsf{Proc} + (\mathsf{Name} \times \mathsf{Gard} \times \mathsf{GNet}))$ is a multi-set of processes P

and named sub-nets $n(G)[M]$ with guardians. Net transitions are defined over configurations of the form $F, n(G)[M]$, where $F\colon \mathsf{Name} \xrightarrow{fin} \mathsf{Gard}$ associates a guardian to the names generated so far. F is like a global environment, which is extended when a fresh name is generated, and is looked up when a new ambient is created (i.e. a *box* action is performed). When a transition does not generate any fresh names, we write $n(G)[M] \Longrightarrow n(G')[M']$ instead of $F, n(G)[M] \Longrightarrow F', n(G')[M']$.

Net transitions $F, n(G)[M] \Longrightarrow F', n(G')[M']$ are defined in terms of a structural rule (similar to that for BA)

$$\text{sub-net}\ \frac{F, n_2(G_2)[M_2] \Longrightarrow F', n_2(G_2')[M_2']}{F, n_1(G_1)[M_1, n_2(G_2)[M_2]] \Longrightarrow F', n_1(G_1)[M_1, n_2(G_2')[M_2']]}$$

and basic net transitions, classified in three groups (as before), *activated* by process transitions, but with additional premises corresponding to the permissions granted by the guardians of ambients *involved* in the transition.

Asynchronous and non-blocking transitions:

$$\text{new}\ \frac{P \xrightarrow{new(n_1,G_1)} P' \qquad G \xrightarrow{new(n_1,G_1)} G'}{F, n(G)[M,P] \Longrightarrow F[n_1 \mapsto G_1], n(G')[M,P']}\ n_1 \notin dom(F)$$

$$\text{spawn}\ \frac{P \xrightarrow{spawn(P_1)} P_2 \qquad G \xrightarrow{spawn(P_1)} G'}{n(G)[M,P] \Longrightarrow n(G')[M,P_1,P_2]}$$

$$\text{box}\ \frac{P \xrightarrow{box(n_1,P_1)} P_2 \qquad G \xrightarrow{box(n_1,P_1)} G'}{F, n(G)[M,P] \Longrightarrow F, n(G')[M, n_1(G_1)[P_1], P_2]}\ G_1 = F(n_1)$$

In (new) the process P requesting a fresh name must provide a guardian for it, thus the association name-guardian is established at name-generation time (and recorded in the partial function F). In (box) the guardian for the new ambient is recovered from F, while the process running in the new ambient is fixed at ambient-generation-time.

Asynchronous and possibly blocking transitions:

$$\text{in}\ \frac{P \xrightarrow{in(n_2)} P' \qquad G_1 \xrightarrow{in_1(n,n_2)} G_1' \qquad G_2 \xrightarrow{in_2(n,n_1)} G_2' \qquad G \xrightarrow{in_3(n_1,n_2)} G'}{n(G)[M, n_1(G_1)[P, M_1], n_2(G_2)[M_2]] \Longrightarrow n(G')[M, n_2(G_2')[n_1(G_1')[P', M_1], M_2]]}$$

$$\text{out}\ \frac{P \xrightarrow{out(n_2)} P' \qquad G_1 \xrightarrow{out_1(n,n_2)} G_1' \qquad G_2 \xrightarrow{out_2(n,n_1)} G_2' \qquad G \xrightarrow{out_3(n_1,n_2)} G'}{n(G)[M, n_2(G_2)[n_1(G_1)[P, M_1], M_2]] \Longrightarrow n(G')[M, n_1(G_1')[P', M_1], n_2(G_2')[M_2]]}$$

Synchronous transitions:

$$\text{comm}\ \frac{P_1 \xrightarrow{wr(v)} P_1' \qquad P_2 \xrightarrow{rd(v)} P_2' \qquad G \xrightarrow{comm(v)} G'}{n(G)[M, P_1, P_2] \Longrightarrow n(G')[M, P_1', P_2']}$$

$$\text{rdup}\ \frac{P_1 \xrightarrow{wr(v)} P_1' \qquad P_2 \xrightarrow{rdup(v)} P_2' \qquad G_1 \xrightarrow{rdup_2(n_2,v)} G_1' \qquad G_2 \xrightarrow{rdup_1(n_1,v)} G_2'}{n_1(G_1)[N_1, P_1, n_2(G_2)[P_2, N_2]] \Longrightarrow n_1(G_1')[N_1, P_1', n_2(G_2')[P_2', N_2]]}$$

$$\text{wrup}\ \frac{P_1 \xrightarrow{rd(v)} P_1' \qquad P_2 \xrightarrow{wrup(v)} P_2' \qquad G_1 \xrightarrow{wrup_2(n_2,v)} G_1' \qquad G_2 \xrightarrow{wrup_1(n_1,v)} G_2'}{n_1(G_1)[N_1, P_1, n_2(G_2)[P_2, N_2]] \Longrightarrow n_1(G_1')[N_1, P_1', n_2(G_2')[P_2', N_2]]}$$

$$\text{rddn}\ \frac{P_1 \xrightarrow{rddn(n_2,v)} P_1' \qquad P_2 \xrightarrow{wr(v)} P_2' \qquad G_1 \xrightarrow{rddn_1(n_2,v)} G_1' \qquad G_2 \xrightarrow{rddn_2(n_1,v)} G_2'}{n_1(G_1)[N_1, P_1, n_2(G_2)[P_2, N_2]] \Longrightarrow n_1(G_1')[N_1, P_1', n_2(G_2')[P_2', N_2]]}$$

$$\text{wrdn}\ \frac{P_1 \xrightarrow{wrdn(n_2,v)} P_1' \qquad P_2 \xrightarrow{rd(v)} P_2' \qquad G_1 \xrightarrow{wrdn_1(n_2,v)} G_1' \qquad G_2 \xrightarrow{wrdn_2(n_1,v)} G_2'}{n_1(G_1)[N_1, P_1, n_2(G_2)[P_2, N_2]] \Longrightarrow n_1(G_1')[N_1, P_1', n_2(G_2')[P_2', N_2]]}$$

Remark 3.4 In a net with guardians, when a new ambient with name n is created (i.e. when a *box* action is performed), we must provide a guardian for the new ambient. In GBA we have decided that the guardian for the new ambient is provided by the process that created the name n (by performing a *new* action). We review the alternatives, and discuss their pros and cons. There are four entities that may provide the guardian for a new ambient:

(i) the process P_n that did the *new* action for creating n;

(ii) the guardian G_n for process P_n;

(iii) the process P_b that does the *box* action spawning $n[P]$;

(iv) the guardian G_b for process P_b.

Options i and iii make processes aware of guardians. Options i and ii require a global environment to record the guardian to be used when an ambient named n is created. Option ii is too *generic*, since G_n does not know how P_n plan to use n, so it will provide a guardian that is the same for all new names. Option iii appears the most *flexible*, since the guardian for the new ambient is decided at ambient-generation-time. We have chosen option i, which is more rigid, but also more secure, and it embodies the principle **one name one policy**. □

Definition 3.5 [Erasure] Given $M \in \mathsf{GNet}$, the net $|M| \in \mathsf{Net}$ is obtained by removing all guardians occurring in M.

$|M|$ could be formally defined by induction on the structure of M. The following property says that the introduction of guardians has only an inhibitory effect on net transitions.

Proposition 3.6 *If* $M \Longrightarrow M'$, *then* $|M| \Longrightarrow |M'|$

We conclude this section by giving a few simple examples of guardians, that are dynamic surrogates for static types:

• $\bot$, nothing is permitted. In $n(\bot)[M]$ the sub-nets are still able to evolve,

but in isolation.

- $\top$, everything is permitted.
- $Immobile = \top\backslash out_1(\overline{x})\backslash in_1(\overline{x})$ forces immobility: the net $n(Immobile)[M]$ cannot move.
- $Shh = \top\backslash comm(x)$ forbid local communications,
- $Comm(\tau) = \top\backslash comm(x)$ when $\neg x{:}\,\tau$ restricts local communications to values of type τ (we need a predicate $\_{:}\,\tau$, that performs dynamic type-checking).

There is no way to define a guardian corresponding to the single-thread types of Safe Ambients [31], however one can prevent the creation of new threads by forbidding the *spawn* action.

Remark 3.7 One may wonder how much run-time overhead is introduced by the execution monitoring performed by guardians. In the case of ambient movements (in) and (out), the agreement among four entities is a faithful representation of what happens in the distributed abstract machines for Mobile Ambients [22] and for Safe Ambients [39]. The asynchronous message-passing protocol of [22] could be used to split the synchronization among four entities required in rules (in) and (out) into multiple synchronization steps each involving only two entities. The splitting could give rise to a number of blocking situations, that were not present in the original semantics but can be permissible because we are mainly interested to safety properties. Similarly, the implementation of boxed ambient communications would require multi-parts cooperations. In all other cases the synchronization with guardians represents a real overhead. This overhead could be reduced by adopting asynchronous communication, which amounts to having one data repository per ambient (thus the synchronization with the guardian corresponds to the interaction with the data repository).

The rules (new) and (box) introduce an overhead related to the extension and access to the environment F mapping names to guardians. This environment can be extended, but an association is never overwritten, thus some simple caching technique can substantially reduce the overhead.

Finally, we discuss the rule (spawn). A BA/GBA process can be thought of as a thread executing one action at a time. Therefore, the action $spawn(P)$ corresponds to a system call requesting to start a new thread executing P (thus the synchronization with the guardian corresponds to the interaction with the operating system). At present a guardian cannot discriminate between complex values (only between names). However, by extending boolean expressions with new predicates on values (besides name equality), one could envisage that the permission for spawning a new thread is based on a run-time analysis of P. □

4 Examples

In this section we provide several examples that show the expressivity of guardians to dynamically enforce security policies. To this purpose we will describe in our calculus some ambient interaction protocols that have been already discussed in the literature to point out and discuss advantages and weaknesses of ambient-like calculi. In the examples, we may use the derived forms $g(\overline{p})$ then G and $G\backslash g(\overline{p})$, where $p ::= x \mid n \mid \_$ are patterns. The meaning of the derived forms is the obvious one, e.g.

- $g(x)$ then G corresponds to $g(x)$ when $\top$ then G
- $g(n)$ then G corresponds to $g(x)$ when $x = n$ then G, with $x \notin \mathrm{FV}(G)$
- $g(\_)$ then G corresponds to $g(x)$ when $\top$ then G, with $x \notin \mathrm{FV}(G)$.

The guardians presented in the examples seem to fall into two classes: everything is allowed unless explicitly forbidden (exploiting the "deny" combinator), only a very limited pattern of interaction is allowed (exploiting the dynamic combinators: "allow", "choice" and "recursion"). The "and" combinator is not used in the examples. It could be used to compose guardians implementing elementary policies into more structured ones, as it is the case for *passive* composition of wrappers [23].

4.1 Firewall

The firewall protocol (see [12,31,6]) has been introduced to examine which are the activities needed to control accesses through a firewall. In this protocol, ambient f describes a firewall and we assume that the name of the firewall must always remain unknown. An external agent a can enter the firewall f through a pilot ambient k, however the external agent has no permissions to exit from the firewall f (thus carrying confidential information, like the name f, outside). The following net (without guardians) describes the initial configuration: $a[in(k).out(k).P_a], f[k[X_k(f)], P_f]$ where $X_k(f) \stackrel{\Delta}{=} out(f).in(f).X_k(f)$. The intended behavior of the firewall protocol corresponds to the following sequence of transitions:

$a[in(k).out(k).P_a], f[k[X_k(f)], P_f]$	k exits from f
$a[in(k).out(k).P_a], k[in(f).X_k(f)], f[P_f]$	a enters into k
$k[a[out(k).P_a], in(f).X_k(f)], f[P_f]$	k enters back into f
$f[k[a[out(k).P_a], X_k(f)], P_f]$	a exits from k
$f[k[X_k(f)], a[P_a], P_f]$	k is back in the initial state, and a can interact with P_f

However, there are other sequences of transitions from the initial configuration that give rise to a violation of the protocol. In particular a could get out of f

(even if P_a does not know the name f) by getting a lift back from k.

We now describe two simple guardians for f and k (W_f and W_k) that dynamically enforce the intended security policy. No assumptions are made on the guardian for a, since a malicious agent may have also a malicious guardian.

- $W_k \stackrel{\Delta}{=} out_1(\_, x_f)$ then $in_2(\_, \_)$ then $in_1(\_, x_f)$ then $out_2(x_f, \_)$ then W_k
  The guardian W_k describes a security policy that forces the following sequencing of actions:
  (i) k exits from the enclosing ambient f (and x_f is bound to f)
  (ii) k allows one ambient to enter
  (iii) k goes back into the ambient f
  (iv) k allows one ambient to exit (while staying within f)
  When these actions are terminated the pilot ambient k is back in its original state.
- $W_f \stackrel{\Delta}{=} Immobile \backslash in_2(\_, x)$ when $x \neq k \backslash out_2(\_, x)$ when $x \neq k$
  The guardian W_f ensures immobility of the firewall and that only ambients with name k are allowed to cross the firewall.

Notice that W_k does not explicitly use the name f of the firewall, while W_f uses the name k for pilot ambients. Since the association name-guardian is made at name-generation-time, this means that the name k must be generated before name f. However, this is not a strong constraint since we want to keep f private, and thus it is better to generate it as late as possible, (i.e. just before generating the ambient with name f).

One could argue that guardian W_f is too restrictive. In particular, one could adopt a simpler guardian for the firewall f that does not use k, e.g. $W_f \stackrel{\Delta}{=} Immobile$. The requirement that the only ambient having the permission to cross the firewall have name k might be ensured by other mechanisms. However, this seems to require a careful design of P_f (having much more global assumptions), hence it does not appear to us as a viable option.

4.2 Train Movement

The train protocol (see [8]) can be described as follows: there is a train t moving between two stations a and b, a passenger p can get in and out of the train when it is in a station, but not when it is moving between stations. If passenger p wants to get off at a certain station, he needs to be informed when the train has reached that station. In [8] the example is modeled by exploiting a new primitive for *ambient renaming*. Intuitively, the train ambient takes a suitable name to implicitly inform the passengers when it has arrived at a certain station, while it takes a name unknown to passengers when it is moving (in this way passengers cannot get in or out of the train). The train protocol provides a nice pictorial example of access control in presence of mobility. We will model the access control decision by a coordination activity

with a suitable ambient called *announcement*. Passengers are informed of the arrival at a certain station by the ambient announcement ($@a$ or $@b$) that is generated by the train when it arrives at a station (a or b respectively).

Remark 4.1 In Mobile Ambients [13], ambient renaming can be expressed in terms of the calculus primitives, although the translation does not preserve atomicity and can give rise to unwanted interferences. On the contrary, in GBA, ambient renaming is not internally representable in the calculus, but we refrain from introducing it as a primitive since it is at odds with the principle "one name one security policy". □

The following net (without guardians) describes the initial configuration: $a[], b[], t[X_t], p[X_1]$, where

- $X_t \triangleq in(a).box(@a, out(t).0).out(a).in(b).box(@b, out(t).0).out(b).X_t$, i.e. the train t moves between stations a and b.
- X_1 models a generic passenger, e.g. $X_1 \triangleq in(a).in(t).in(@b).out(@b).X_2$ means: enter station a, get on train t, listen for the announcements $@b$, get off at station b. After this, the passenger could exit station b, or get on another train, say t', to reach a third station c. This second behavior is modeled by $X_2 \triangleq in(t').in(@c).out(@c).X_3$.

We now describe three guardians: W_s for the stations a and b, $W_{@s}$ for the announcements $@a$ and $@b$, and W_t for the train t. No assumptions are made on the guardian for a passenger p:

$$\begin{aligned} W_s &\triangleq Immobile \\ W_{@s} &\triangleq in_2(\_,\_) \text{ then } W_{@s} \vee \\ &\qquad out_1(\_,\_) \text{ then } W'_{@s} \\ W'_{@s} &\triangleq out_2(\_,\_) \text{ then } W'_{@s} \end{aligned}$$

The guardian $W_{@s}$ forces the following sequencing of actions: first passengers enter $@s$, then $@s$ moves out of the train, and finally passengers exit $@s$.

$$\begin{aligned} W_t &\triangleq in_1(\_, x_s) \text{ then } box(x_{@s}, \_) \text{ then } W_t(x_s, x_{@s}) \\ W_t(x_s, x_{@s}) &\triangleq in_2(\_,\_) \text{ then } W_t(x_s, x_{@s}) \vee \\ &\qquad in_3(\_, x_{@s}) \text{ then } W_t(x_s, x_{@s}) \vee \\ &\qquad out_2(\_, x_{@s}) \text{ then } out_1(\_, x_s) \text{ then } W_t \end{aligned}$$

The guardian W_t describes the security policy of the train. First t arrives at a station s and makes an announcement $@s$, then passengers get in (i.e. enter t) or get off (i.e. enter $@s$), finally $@s$ exit t and t leaves station s.

4.3 Routable packet

In this protocol, from [13,30], a packet is used to drive some information to various destinations. In our formulation, agent a uses a routable packet p to be carried to its destination. The ambient p supports the following interactions: it allows an agent a to enter p and to inform p of the route M for reaching its final destination. Then p executes the sequence of movements specified by M, upon completion a exits from p and p returns in its original state.

The security requirements of this protocol are the following:

- The routable packet p cannot perform actions specified in M that are not movement actions;
- The agent a should leave the packet only when the final destination has been reached;
- No other agent should be allowed to enter inside the packet p until the protocol has been completed.

The behavior of the agent and of the Routable packet are given below.

- $X_a \triangleq in(p).wrup(M).out(p).P$,
  where the action $wrup$ spawns process M in the parent ambient of a (i.e. in the routable packet p).
- $X_p \triangleq rd(?x).new(?z, \bot).x; box(z, 1).X_p$,
  where the box action after x is exploited to inform the guardian G_p when destination has been reached. Notice that since z is a fresh name, it is unknown to x.

We now describe a guardian for the routable packet p. This guardian enforces the sequencing of actions specified by the protocol, no matter how malicious a is:

$$\begin{aligned} G_p &= in_2(\_, x_a) \text{ then } wrup_2(x_a) \text{ then } new(z, \_) \text{ then } G_p(z, x_a) \\ G_p(z, x_a) &= in_1(\_, \_) \text{ then } G_p(z, x_a) \vee \\ &\quad out_1(\_, \_) \text{ then } G_p(z, x_a) \vee \\ &\quad box_1(z) \text{ then } out_2(\_, x_a) \text{ then } G_p \end{aligned}$$

G_p guarantees that, e.g., two different routes cannot enter the packet p and interfere with the path to follow. G_p is not able to guarantee that eventually p reaches the destination specified by M, unless all the ambients that are used by M are immobile. Indeed, we cannot rely on the agent a to ensure this property. However, we can modify G_p to allow only movements in and out of ambients that G_p knows to be immobile (this means that the names of such ambients must have been created before the name p).

5 Related work

Policy enforcement mechanisms.

Several alternative approaches have been exploited to enforce security policies in networked computing systems. The approaches may differ in the level of trust required, the flexibility of the enforced security policy and their costs to components producers and users. A comprehensive security framework could result from the combination of complementary issues.

As pointed out in [42], approaches like *code signing* and *sand-boxing* (for instance consider the Java implementation of these concepts [26,24]) have low costs but cannot enforce flexible security policies (signed components may behave in arbitrary ways and the user must trust the component producer, sand-boxed components are isolated and cannot interact with each other).

Three sensible and flexible *language-based* techniques — type systems, in-lined reference monitors, and certifying compilers — are pointed out and evaluated in [41]. *Type systems* (like, e.g., those used in [45,28]) statically analyze the application code against the wanted security policy; they require a full knowledge of the code, that must be written in conformance with the type system, and usually do not need any run-time overhead. *In-lined reference monitors* postpone (some of) the security checks at run-time, by inserting additional code for enforcing the wanted security policy into the target application. This technique has been used in Software Fault Isolation (see, e.g., [46]), and two implementations using *security automata* as a mechanism for specifying security policies are presented in [21]. In [43,44], *type specialization* is used to remove (in some cases) all run time safety checks from programs transformed for in-lined monitored execution while guaranteeing the safety of the transformed code. *Certifying compilers* produce object code along with a certificate, i.e. a machine checkable evidence that the object code respects certain security policies. This technique has been used, e.g, for generating Proof-Carrying Code [34,35] and enforcing standard security policies (like type and memory safety). In [47], it is shown that this technique can be used to enforce any security policy that can be specified by a security automaton [4] (given a security automaton specifying the policy to be enforced, it is shown how to insert run-time security checks during program compilation and how to verify that the compiled code obeys the security policy).

Execution monitoring (EM, [40]) is a general class of enforcement mechanisms that work by monitoring execution steps of a target application and by terminating execution that is about to violate the security policy being enforced. EM mechanisms rely on security automata: the monitored application is executed in tandem with a simulation of the automaton defining the wanted security policy. EM mechanisms include, e.g., reference monitors and security kernels, and are flexible and expressive (they permit to enforce at best *safety policies*). EM mechanisms are rather flexible, since they are also

[4] Security automata are very expressive, in fact they can specify every *safety property* [40].

applicable to components for which only untyped object code is available and whose internal structure cannot be analyzed.

Wrappers, i.e. code that encapsulates possibly untrusted components and can intercept the communications between each such component and the rest of the system, are significant examples (see, e.g., [23,42]) of EM mechanisms. In [23], a *wrapper definition language* for composing COTS software components is defined along with a framework for wrappers management. In [42], box-π, a process calculus obtained as a minimal extension of the π-calculus with encapsulation, is defined and used for specifying wrappers that rigorously implements security policies; a *causal* type system is then used for statically capturing the allowed information flows between wrapped, possibly badly typed, components. Our proposal is closely related to the idea of execution monitoring, and guardians are like security automata that specify the allowed process actions and interactions.

Security techniques for Ambients-based languages.

A number of analysis techniques have been developed for the *mobile ambients* (MA, [13]) and its variants. Some of these techniques are proof systems and modal logics [7], abstract interpretation [27], control and data flow analysis [37,36,19], denotational models [16,17]. However, the techniques closer to security policies enforcement are based on type systems.

Many type disciplines have been developed for MA. In [12], *exchange types* are introduced to discipline the exchange of values in communications. In [9], *immobility* and *locking* annotations have been used for ensuring that only ambients that are deemed mobile will move and that only ambients that are deemed openable will be opened. In [10], by introducing the notion of *group* names, the type of an ambient is refined to enable controlling the set of ambients it can cross and the set of ambients it can open. Finally, in [11], all these type systems are reviewed and assessed in a common framework.

The *safe ambients calculus* (SA, [31,30]) is a variant of MA obtained by adding co-capabilities to the base calculus thus enabling the ambient target of a movement or object of an open action to control the interaction. A distributed implementation of SA is described in [39]. SA permits defining powerful and accurate type systems for controlling mobility. In [31,30], a type discipline is introduced for controlling ambient mobility and removing *grave interferences* by relying on *single-threaded* ambients (i.e. ambients which at every step offer at most one interaction with external and internal ambients). In [20], a type discipline is introduced that, by relying on a statically fixed hierarchy of security levels, ensures that ambients at a given security level can only be crossed or opened by ambients at a greatest or equal security level. In [4], a type system for expressing and verifying behavioral invariants of ambients (written in Secure SA, a typed variant of SA) is defined, by accounting not only for immediate behavior but also for behavior resulting from capabilities acquired by interacting with the surrounding environment. In order

to develop equational theories to support reasoning on SA terms, in [32], a labeled bisimilarity for a variant of SA enriched with *passwords* (and with a different semantics for *out*) has been developed and proved to coincide with barbed congruence.

The *seal calculus* [15] is variant of MA where the *open* primitive has been dropped, ambient communication relies on located channels and can also take place across ambient boundaries (between parent and children), and ambient mobility is *objective* (an ambient is moved by a process in the parent ambient) and implemented in terms of agents (higher-order) communication. In [14], *interface types* are introduced for describing the requests that an ambient may accept from its surrounding environment and are then used to type mobile ambients and "their mobility", i.e., to allow one to declare, for each ambient, the type of the ambients that can enter or exit it.

Finally, a few type disciplines have been developed also for the boxed ambients calculus. In [5], a type system is defined that permits to fully control the types of the values exchanged in ambients interactions. In [6], a security type system is introduced that permits to control resources access by implementing policies for mandatory access control that rely on a security level associated to each ambient.

6 Conclusions and future work

In this paper we have presented GBA, an extension of Boxed Ambients with guardians. Similar extensions can be given for other ambient-based calculi. Indeed, we have massaged BA to make apparent the general pattern for these extensions. Also the open primitive of Mobile Ambients can be handled consistently with the general principle "one name one policy", simply by dropping the guardian of the ambient destroyed by open. We envisage three directions for further research on GBA:

- Proof techniques for establishing security properties. More specifically, assuming complete knowledge of certain guardians and processes (the trusted ones), we would like to derive properties of a complete system, which may include unreliable processes and guardians. [41] proposes a general framework for defining security policies. Since guardians are a particular case of execution monitoring, the only properties that we can hope to enforce are *safety properties*.
- Implementation issues. The distributed abstract machines for Mobile Ambients [22] and for Safe Ambients [39] represent useful starting point. Probably, one may have to consider a more restricted language for guardians, identified after a more careful analysis of relevant examples.
- Language extensions with more powerful guardians. Two kinds of extension appear of interest: increase the monitoring ability of guardians, give to guardians the ability to act. In both cases the syntax for processes should

be unchanged, while the syntax of guardians (and nets with guardians) may change. One could envisage another kind of extension, namely give to processes the ability to modify guardians. However, this could introduce security problems. In fact, guardians are the entities which dynamically enforce security policies, thus it is dangerous to give to (untrusted) processes the ability to modify (trusted) guardians.

We list some of the language extensions that make guardians more powerful:

- Associate to each sub-ambient a unique local stamp, which is visible only to the guardian of the parent ambient, when an ambient moves it receives a new local stamp from the destination ambient. In this way a guardian can discriminate between sub-ambients with the same name.
- Partition processes into local regions, which are visible only to the guardian of the ambient. In this way the guardian can give different privileges to processes in different regions.
- Give to a guardian the ability to react to attempts by processes to perform actions that are denied by it or other guardians.
- Give to a guardian the ability to spawn processes or to augment and transform the actions of processes under its control.

Acknowledgement

The authors thank Cedric Fournet for helpful comments.

References

[1] ACM. *27th ACM Symposium on Principles of Programming Languages (Boston, MA)*, 2000.

[2] ACM. *28th ACM Symposium on Principles of Programming Languages (London, UK)*, 2001.

[3] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. In *International Conference on Concurrency and Coordination*, number 54 in ENTCS, 2001.

[4] Michele Bugliesi and Giuseppe Castagna. Secure safe ambients. In *Proceedings of POPL '01* [2], pages 222–235.

[5] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 38–63. Springer, 2001.

[6] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 102–120. Springer, 2001.

[7] L. Cardelli and A. Gordon. Anytime, anywhere — modal logics for mobile ambients. In *Proceedings of POPL '00* [1], pages 365–377.

[8] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999.

[9] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiří Wiederman, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 1999.

[10] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Proceedings of TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 333–347. IFIP, Springer, 2000. Full version to appear in jic.

[11] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 2002.

[12] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92. ACM, 1999.

[13] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, number 1378 of Lecture Notes in Computer Science, pages 140-155, Springer, 1998.

[14] Giuseppe Castagna, Giorgio Ghelli, and Francesco Zappa Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 82–101. Springer, 2001.

[15] Giuseppe Castagna and Jan Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer, 1999.

[16] M. Coppo and M. Dezani-Ciancaglini. A fully abstract model for mobile ambients. In *TOSCA'01*, ENTCS 62. Elsevier Science B. V., 2001.

[17] M. Coppo and M. Dezani-Ciancaglini. A fully abstract model for higher-order mobile ambients. In *VMCAI'02*, LNCS. Springer, 2002.

[18] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[19] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *ASIAN Computing Sciece Conference - ASIAN'00*, volume 1961 of *LNCS*, pages 199–214. Springer, 2000.

[20] M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *ASIAN Computing Sciece Conference - ASIAN'00*, volume 1961 of *LNCS*, pages 215–236. Springer, 2000.

[21] Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*. ACM Press, 1999. Full version as Technical Report TR99-1758, Department of Computer Science, Cornell University, Jul 1999.

[22] Cedric Fournet, Jean-Jacques Levy, and Alan Schmitt. An asynchronous, distributed implementation of mobile ambients. In *IFIP TCS*, pages 348–364, 2000.

[23] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[24] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, Reading, MA, USA, 1999.

[25] A. Gordon and C. Fournet. Stack inspection: Theory and variant. In *Proceedings of POPL '01* [2].

[26] G. Mc Graw and E. Felten. *Securing Java*. Wiley, 1999.

[27] Rene Rydhof Hansen, Jacob Grydholt Jensen, Flemming Nielson, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. In *Static Analysis Symposium (SAS'99)*, number 1694 in Lecture Notes in Computer Science, pages 134–148. Springer, 1999.

[28] N. Heintze and J.G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of POPL '98*. ACM, 1998.

[29] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier, 1998. To appear in Information and Computation.

[30] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. Full version of [31].

[31] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00* [1], pages 352–364.

[32] M. Merro and M. Hennessy. Bisimulation congruences in Safe Ambients. In *Proceedings of POPL '02*. ACM, 2002.

[33] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[34] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL '97*, pages 106–119. ACM, 1997.

[35] G.C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, number 1419 in Lecture Notes in Computer Science, pages 61–91. Springer, 1998.

[36] Flemming Nielson and Hanne Riis Nielson. Shape analysis for mobile ambients. In *Proceedings of POPL '00* [1], pages 135–148.

[37] Flemming Nielson, Hanne Riis Nielson, Rene Rydhof Hansen, and Jacob Grydholt Jensen. Validating firewalls in mobile ambients. In Jos C.M. Baeten and Sjouke Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 463–477. Springer, 1999.

[38] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In *ESOP'01*, number 2028 in LNCS. Springer, 2001.

[39] Davide Sangiorgi and Andrea Valente. A distributed abstract machine for safe ambients. In *Proceedings of ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 408–420. Springer, 2001.

[40] Fred B. Schneider. Enforceable security policies. *ACM Transation on Information and System Security*, 2(4), 2000. Also appeared as Technical Report TR99-1759, Department of Computer Science, Cornell University, Jul 1999.

[41] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back. Conference on the Occasion of Dagstuhl's 10th Anniversary*, number 2000 in Lecture Notes in Computer Science, pages 86–101. Springer, 2000.

[42] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box pi, wrappers, and causality types. *To appear in Journal of Computer Security*, 2002. Full version as Technical Report 478, Computer Laboratory, University of Cambridge, November 1999.

[43] Peter Thiemann. Enforcing safety properties using type specialization. In D. Sands, editor, *European Symposium on Programming (ESOP'01)*, number 2028 in Lecture Notes in Computer Science, pages 62–76. Springer, 2001.

[44] Peter Thiemann. Program specialization for efficient monitored execution. 2001.

[45] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[46] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *ACM Synposium on Operating Systems Principles (SOSP'93)*, pages 203–216. ACM Press, 1993.

[47] D. Walker. A Type System for Expressive Security Policies. In *Proceedings of POPL '00* [1].

[48] D. S. Wallach, A. W. Appel, and E. W. Felten. Safkasi: a security mechanism for language-based systems. *ACM TOSEM*, 2000.