Using Ambients to Control Resources*

David Teller[1], Pascal Zimmer[2], and Daniel Hirschkoff[1]

[1] LIP - ENS Lyon, France
{David.Teller,Daniel.Hirschkoff}@ens-lyon.fr
[2] INRIA Sophia Antipolis, France
Pascal.Zimmer@sophia.inria.fr

Abstract. Current software and hardware systems, being parallel and reconfigurable, raise new safety and reliability problems, and the resolution of these problems requires new methods. Numerous proposals attempt at reducing the threat of bugs and preventing several kinds of attacks. In this paper, we develop an extension of the calculus of Mobile Ambients, named *Controlled Ambients*, that is suited for expressing such issues, specifically Denial of Service attacks. We present a type system for Controlled Ambients, which makes resource control possible in our setting.

Introduction

The latest generation of computer software and hardware makes use of numerous new technologies in order to enhance flexibility or performances. Most current systems may be dynamically reconfigured or extended, allow parallelism or use it, and can communicate with other systems. This flexibility, however, induces the multiplication of subsystems and protocols. In turn, this multiplication greatly increases the possibility of bugs, the feasibility of attacks and the sensitivity to possible breakdown of individual subsystems.

This paper presents a formalism for *resource control* in parallel, distributed, mobile systems, called *Controlled Ambients* (CA for short). The calculus of CA is based on Mobile Ambients [2], and extends Safe Ambients [15], and is equipped with a type system to express and verify resource control policies.

In the first section, we present our point of view on the problem of resource control. We provide motivations for using ambient calculi to represent the notion of resource, and claim that a specific calculus should be designed for the purpose of guaranteeing some control on the use of resources. In Sec. 2, we introduce our calculus of Controlled Ambients and explain why it fits to our purposes. We then develop in Sec. 3 a type system which uses the specifics of this language to make resource control possible; we prove its correctness (i.e. that it does control the acquisition and release of resources), and use it to treat several examples. After this, we discuss some refinements of our type system, and, in the last section, we present possible extensions of this study as well as related works.

* Work supported by european project FET - Global Computing

L. Brim et al. (Eds.): CONCUR 2002, LNCS 2421, pp. 288–303, 2002.

1 Resource Control

We define a resource as an entity which may *at will* be acquired, used, then released. Thus, this notion encompasses ports, CPUs, computers or RAM, but not time, or (presumably) money. A *resource-controlled system* is a system in which no subsystem will ever require more resources than may be available.

In order to prevent problems such as Denial of Service attacks, we need a formalism making resource control possible. This formalism should in particular provide means to describe systems in terms of resource availability and resource requirement, and should also support the description of concurrent and mobile computations. Lastly, the model should provide some kind of entity that can be regarded as a resource. Ambient calculi can be used for these purposes.

Ambient Calculi. Ambient Calculi are based on locality: each *ambient* is a site. An ambient's evolution is controlled by the means of *capabilities*: $\mathsf{in}\ m$ and $\mathsf{out}\ m$ let an ambient move (resp. entering ambient m or leaving ambient m), while $\mathsf{open}\ m$ opens ambient m and releases its contents in the current ambient. To draw some analogies with real systems, in and out can represent the movement of data in a computer or in a network, while open could be used for cleaning memory, for reading data or for loading programs into memory. As for ambients, they could stand for computers, programs, data, components...

These correspondences open the way for a natural model of resource control, where each site may have a finite (or infinite) quantity of resources of a given category. Resources will be used for data, programs, ... In other words, each ambient has a given *capacity* and each subambient *uses* a part of this capacity. Basically, *controlling resources means checking the number of* **direct** *subambients (according to the amount of resources these are using) which may be present in one ambient at any time.*

Do note that we could have chosen different points of view and decided to count all subambients at all depths, or possibly only "leaf" ambients. Although these approaches seems equally valid, we have decided not to undertake them, since they did not seem more powerful, only slightly more complicated.

An example. Let us consider a cab protocol, as shown in Fig. 1, which we will use as our main running example. The system consists of one city, n sites, and several cabs and clients. Cabs may be either "anywhere in the city" or in a precise site. Each client may be either in a given site or in a cab. Any client may call a cab. If a cab is available, one (and only one) cab must come fetch the client and bring her to her destination. If we consider the unique passenger seat of a cab as a resource, the system will be resource-controlled if each cab contains at most one client at any time.

Fig. 1 presents the cab protocol as written in the original calculus of Mobile Ambients[1]. All entities involved in the protocol are represented by ambients. Ambient movements are used to simulate the movements in the protocol

[1] As a matter of fact, we are not exactly using the original MA calculus, since we have introduced the rec operator, for more readability.

Message emitted by client *client* at site *from* to call a cab

$$call\ from\ client \triangleq call[\mathsf{out}\ client.\mathsf{out}\ from.\mathsf{in}\ cab.\mathsf{in}\ from.loading[\mathsf{out}\ cab.\mathsf{in}\ client]]$$

Instructions given by client *client* going from site *from* to site *to*

$$trip\ from\ to\ c \triangleq trip[\mathsf{out}\ client.\mathsf{out}\ from.\mathsf{in}\ to.unloading[\mathsf{in}\ c]]$$

The client itself, willing to go from *from* to *to*

$$client\ from\ to \triangleq (\nu c)c[call\ from\ c \mid \mathsf{open}\ loading.\mathsf{in}\ cab.trip\ from\ to\ c \mid \mathsf{open}\ unloading.\mathsf{out}\ cab.bye[\mathsf{out}\ c.\mathsf{in}\ cab.\mathsf{out}\ to]]$$

The cab and the city

$$cab \triangleq cab[\mathsf{rec}\ X.\mathsf{open}\ call.\mathsf{open}\ trip.\mathsf{open}\ bye.X]$$

$$city \triangleq city[cab \mid cab \mid \cdots \mid site_1[client\ site_1 site_i \mid client\ site_1\ site_j \mid \cdots] \mid \cdots \mid site_i[\cdots]]$$

Fig. 1. Cab protocol - first attempt

(client entering a cab, cab moving from site to site,...). Some additional ambients are used to synchronise parties: *call* carries the client's call, *loading* (resp. *unloading*) tells *client* to enter (resp. exit) *cab*, *trip* carries the trip directions, and *bye* is used at the end of the trip.

Limitations. By examining the code of Fig. 1, one may see that several aspects of this implementation may lead to unwanted behaviors. The most visible flaw is the sending of ambient *bye*: if, for any reason, there are several *cab*s in the site, nothing guarantees that *bye* will reach the right *cab*. And if it does not, it may completely break the system by making one *cab* wait forever for its client to exit, although it already has left, while making the other *cab* leave its destination site with its unwilling *client*. In turn, the *client* may then get out of the cab about anywhere.

Although this problem is partly due to the way this implementation has been designed, its roots are deeply nested within the calculus of Mobile Ambients itself. One may notice that any malicious ambient may, at any time, enter the cab: in the calculus of Mobile Ambients, there is no such thing as a filtering of entries/exits. This lack of filtering and accounting is a security threat as well as an obstacle for resource control: for security, since it prevents modeling a system which could check and refuse entry to unwanted mobile code, and for control, since one cannot maintain any information about who is using which resources in a given ambient.

Towards a better control. Difficulties with security and control are due, for the greatest part, to the nature of capabilities in, out and open. Actually, the way these capabilities are used seems too simple: in any real system, arrival or departure of data cannot happen without the consent of the acting subsystem, much less go unnoticed, not to mention the opening of a program. In practice, if a program wishes to receive network information, it must first "listen" on some communication port. If a binary file is to be loaded and executed, it must have some executable structure and some given entry point.

A calculus derived from Mobile Ambients is presented in [15]; in this calculus of *Safe Ambients*, three *cocapabilities* are introduced, which we will note $\overline{\mathsf{SAin}}$, $\overline{\mathsf{SAout}}$ and $\overline{\mathsf{SAopen}}$. When executed in m, capability $\overline{\mathsf{SAin}}\ m$ allows an ambient to enter m (by execution of capability $\mathsf{in}\ m$). Similarly, $\overline{\mathsf{SAout}}\ m$ allows an ambient to leave m using $\mathsf{out}\ m$, while $\overline{\mathsf{SAopen}}\ m$ allows m's parent to open m using $\mathsf{open}\ m$. These cocapabilities make synchronizations more explicit and considerably decrease the risk of security breaches. Thinking of the example above, a rewritten cab may thus easily refuse entry right to parasites as long as it is not in any site, or while it contains a client. Moreover, a form of resource control is indeed possible, since a full ambient may refuse entrance of new subambients.

However, in this model, ambients are not always warned when they receive or lose subambients by some kind of side effect: in $h[m[n[\mathsf{out}\ m] \mid \overline{\mathsf{SAout}}\ m]]$, h receives n from m but is not made aware of this. Moreover, while $\overline{\mathsf{SAin}}\ m$ serves as a warning for m that it will receive a new subambient, m does not know which one. Since a subambient representing static data and another one modeling some internal message will not occupy the same amount of resources, this model is probably not sufficient for our purposes.

[11] offers an alternative to these cocapabilities, in order to further enhance systems' robustness: in this formalism, $\overline{\mathsf{in}}\ m$ does not allow *entering* m but rather m *to enter*. This approach solves one of our problems: identifying incoming data. Controlled Ambients may be considered as a development of [11] towards even more robustness as well as resource control. Let us also mention [16], where a different mechanism for the $\overline{\mathsf{SAout}}$ cocapability w.r.t. [15] is introduced. Our proposal subsumes the solutions of [16] and [15].

Embedding resource control. In Sec. 3, we equip our langage with a type system for resource control. Basically, the type of an ambient carries two informations:
– its *capacity* - how many resources the ambient offers to its subambients;
– its *weight* - how many resources it requires from its parent ambients.
The type system allows one to statically divide the available resources between parallel processes, and check that resources will be controlled along movements and openings of ambients.

2 The Language of Controlled Ambients

2.1 Syntax and Semantics

In CA, each movement is subject to a 3-way synchronization between the moving ambient, the ambient welcoming a new subambient and the ambient letting a subambient go. As for the opening of an ambient, it is subject to some synchronization between the opener and the ambient being opened. These forms of synchronization are somewhat reminiscent of early versions of Seal [21]. Interaction is handled using *cocapabilities*: $\overline{\mathsf{in}_\uparrow}$, $\overline{\mathsf{out}_\uparrow}$, $\overline{\mathsf{in}_\downarrow}$, $\overline{\mathsf{out}_\downarrow}$ and $\overline{\mathsf{open}}$.

$\overline{\text{in}}_{\uparrow}\ m$ the *up coentry*, welcomes m coming from a subambient;
$\overline{\text{in}}_{\downarrow}\ m$ the *down coentry*, welcomes m coming from the parent ambient;
$\overline{\text{out}}_{\uparrow}\ m$ the *up coexit*, allows m to leave the current ambient by exiting it;
$\overline{\text{out}}_{\downarrow}\ m$ the *down coexit*, allows m to leave by entering a subambient;
$\overline{\text{open}}\ \{m, h\}$ the *coopening*, allows the parent ambient h to open the current ambient m.

Do note that $\uparrow$ and $\downarrow$ are not necessary for resource control. We added them since we found they ease the task of specification in mobile ambients. We will return on the use of these annotations in Sec. 2.3.

The syntax of Controlled Ambients is presented in Fig. 2. We suppose we have two infinite sets of term variables, ranged over with capital letters (X, Y), and of names, ranged over with small letters $(m, n, h, x, \ldots)$. Name binders (input and restriction) are decorated with some type information, that shall be made explicit in the next section. While several proposals for Mobile Ambient calculi use replication, infinite behaviour is represented using recursion in CA. This is mostly due to the fact that recursion allows for an easier specification of loops, especially in the context of resource consumption. Note also that, compared to the original calculus of Mobile Ambients, we restrict ourselves to communication of ambient names only, and we do not handle communicated capabilities.

The null process $\mathbf{0}$ does nothing. Process $M.P$ is ready to execute M, then to proceed with P. $P|Q$ is the parallel composition of P and Q. $m[P]$ is the definition of an ambient with name m and contents P. The process $(\nu n : A)P$ creates a new, private name n, then behaves as P. The recursive construct $\mathsf{rec}\ X.P$ behaves like P in which occurences of X have been replaced by $\mathsf{rec}\ X.P$. Process $(n : A)Q$ is ready to accept a message, then to proceed with Q with the actual message replacing the formal parameter n. $\langle m \rangle$ is the asynchronous emission of a message m. In most cases, we omit the terminal $\mathbf{0}$ process. We say that a process is *prefixed* if it is of the form $M.P$, $\mathsf{rec}\ X.P$ or $(x : A)P$.

The operational semantics is defined in two steps. Structural congruence $(\equiv)$ is the least congruence relation satisfying monoidal laws for $(|, \mathbf{0})$ and the usual laws for restriction (permutation of consecutive restrictions, garbage collection of useless restrictions, name extrusion, and restrictions crossing ambient constructs). Reduction $(\longrightarrow)$ is defined by the rules of Fig. 3, plus the property that $\longrightarrow$ is defined modulo $\equiv$ and is preserved by parallel composition, restriction and ambient construct. $\longrightarrow^*$ stands for the reflexive transitive closure of $\longrightarrow$.

2.2 Examples

We omit in the examples given below type annotations in restrictions; these will be made explicit in the next section.

Renaming. Since movements in Controlled Ambients require full knowledge about the name of moving ambients (also in cocapabilites, which is not the case in Safe Ambients), renaming may be useful in order to comply with some protocols. One may write the renaming of ambient a to b as follows:

$P ::= \mathbf{0}$	null process	$M ::= \mathsf{in}\ m$	enter m
$\mid M.P$	capability	$\mid \mathsf{out}\ m$	leave m
$\mid m[P]$	ambient	$\mid \mathsf{open}\ m$	open m
$\mid P_1 \mid P_2$	parallel composition	$\mid \overline{\mathsf{in}_\uparrow}\ m$	m may climb in upwards
$\mid (\nu n : A)P$	restriction	$\mid \overline{\mathsf{in}_\downarrow}\ m$	m may climb in downwards
$\mid \mathsf{rec}\ X.P$	recursion	$\mid \overline{\mathsf{out}_\uparrow}\ m$	m may climb out upwards
$\mid X$	process variable	$\mid \overline{\mathsf{out}_\downarrow}\ m$	m may climb out downwards
$\mid (n : A)P$	abstraction	$\mid \overline{\mathsf{open}}\ \{m, h\}$	h may open m
$\mid \langle m \rangle$	message emission		

Fig. 2. Controlled Ambients – Syntax

$$m[\mathsf{in}\ n.P \mid Q] \mid n[\overline{\mathsf{in}_\downarrow}\ m.R \mid S] \mid \overline{\mathsf{out}_\downarrow}\ m.T \;\longrightarrow\; n[m[P \mid Q] \mid R \mid S] \mid T \quad (R-in)$$

$$n[m[\mathsf{out}\ n.P \mid Q] \mid \overline{\mathsf{out}_\uparrow}\ m.R \mid S] \mid \overline{\mathsf{in}_\uparrow}\ m.T \;\longrightarrow\; m[P \mid Q] \mid n[R \mid S] \mid T \quad (R-out)$$

$$h[\mathsf{open}\ m.P \mid Q \mid m[\overline{\mathsf{open}}\ \{m,h\}.R \mid S]] \;\longrightarrow\; h[P \mid Q \mid R \mid S] \quad (R-open)$$

$$\langle n \rangle \mid (x : A)P \;\longrightarrow\; P\{x \leftarrow n\} \quad (R-msg)$$

$$\mathsf{rec}\ X.P \;\longrightarrow\; P\{X \leftarrow \mathsf{rec}\ X.P\} \quad (R-rec)$$

Fig. 3. Controlled Ambients – Reduction

$$a\ be\ b.P \ \stackrel{\Delta}{=}\ b[\mathsf{out}\ a.\overline{\mathsf{in}_\downarrow}\ a.\mathsf{open}\ a] \mid \overline{\mathsf{out}_\uparrow}\ b.\mathsf{in}\ b.\overline{\mathsf{open}}\ \{a,b\}.P\,.$$

We then have $\overline{\mathsf{in}_\uparrow}\ b.\overline{\mathsf{out}_\downarrow}\ a \mid a[a\ be\ b.P] \;\longrightarrow^*\; b[P]$. This important example is also characteristic of Controlled Ambients, since $\overline{\mathsf{in}_\uparrow}\ b.\overline{\mathsf{out}_\downarrow}\ a$ illustrates a particular programming discipline: a's parent ambient must accept the replacement of a by b. This means that, at any time, the father ambient knows its own contents, that is both the number of subambients and their names.

Safe Ambients Cocapabilities. As mentioned above, Safe Ambients [15] introduce another kind of cocapabilities, similar to ours, though weaker. We concentrate here on the $\overline{\mathsf{SAin}}$ cocapability (the case of $\overline{\mathsf{SAout}}$ being symmetrical). Its semantics is defined by

$$a[\mathsf{in}\ b.P \mid Q] \mid b[\overline{\mathsf{SAin}}\ b.R \mid S] \;\longrightarrow\; b[R \mid S \mid a[P \mid Q]]\,.$$

By carrying on the idea behind renaming, we can approximate the working of this cocapability in CA. In other words, $a[\mathsf{in}\ b.P \mid Q] \mid b[\overline{\mathsf{SAin}}\ b.R \mid S]$ may be written

$$\begin{aligned}(\nu m, n)\ (a[&\overline{\mathsf{out}_\uparrow}\ m.\mathsf{in}\ b.(P \mid n[\mathsf{out}\ a.\overline{\mathsf{open}}\ \{n,b\}] \mid \overline{\mathsf{out}_\uparrow}\ n) \mid Q \\ &\mid m[\mathsf{out}\ a.\mathsf{in}\ b.\overline{\mathsf{open}}\ \{m,b\}.\overline{\mathsf{in}_\downarrow}\ a]] \\ &\mid b[\overline{\mathsf{in}_\downarrow}\ m.\mathsf{open}\ m.\overline{\mathsf{in}_\uparrow}\ n.\mathsf{open}\ n.R \mid S] \mid \overline{\mathsf{in}_\uparrow}\ m.\overline{\mathsf{out}_\downarrow}\ m.\overline{\mathsf{out}_\downarrow}\ a)\,.\end{aligned}$$

As specified, this expression reduces to $b[R \mid S \mid a[P \mid Q]]$. As was the case for renaming, the father must accept the transaction with $\overline{\mathsf{in}_\uparrow}\ m.\overline{\mathsf{out}_\downarrow}\ m.\overline{\mathsf{out}_\downarrow}\ a$. This entails in particular that the father must know the existence of a.

$$\begin{array}{lll}
\mathit{call\ from} & \triangleq & \mathit{call}[\mathsf{out}\ \mathit{client}.\mathsf{out}\ \mathit{from}. \\
 & & \quad \mathsf{in}\ \mathit{cab}.\overline{\mathsf{open}}\ \{\mathit{call},\mathit{cab}\}.\mathsf{in}\ \mathit{from}.\overline{\mathsf{in}_{\downarrow}}\ \mathit{client}] \\
\mathit{trip\ from\ to} & \triangleq & \mathit{trip}[\mathsf{out}\ \mathit{client}.\overline{\mathsf{open}}\ \{\mathit{trip},\mathit{cab}\}.\mathsf{out}\ \mathit{from}. \\
 & & \quad \mathsf{in}\ \mathit{to}.\mathit{arrived}[\overline{\mathsf{open}}\ \{\mathit{arrived},\mathit{cab}\}.\mathit{end}[\overline{\mathsf{open}}\ \{\mathit{end},\mathit{cab}\}.\mathsf{out}\ \mathit{to}]]] \\
\mathit{client\ from\ to} & \triangleq & \mathit{client}[\mathit{call\ from} \mid \overline{\mathsf{out}_{\uparrow}}\ \mathit{call}.\mathsf{in}\ \mathit{cab}.\mathit{trip\ from\ to} \\
 & & \quad \mid \overline{\mathsf{out}_{\uparrow}}\ \mathit{trip}.\mathsf{out}\ \mathit{cab}] \\
\mathit{cab} & \triangleq & \mathit{cab}[\mathsf{rec}\ X.\overline{\mathsf{in}_{\downarrow}}\ \mathit{call}.\mathsf{open}\ \mathit{call}.\overline{\mathsf{in}_{\uparrow}}\ \mathit{trip}.\mathsf{open}\ \mathit{trip}.\mathsf{open}\ \mathit{arrived}. \\
 & & \qquad \overline{\mathsf{out}_{\uparrow}}\ \mathit{client}.\mathsf{open}\ \mathit{end}.X]
\end{array}$$

Fig. 4. Cab protocol – CA-style (see Fig. 1)

Firewall. We revisit the firewall example of [2], and consider a system f, protected by a firewall. Only agents aware of the password g are allowed in f. This may be modeled as:

$$\begin{array}{lll}
\mathit{Agent}\ P\ Q & \triangleq & \mathit{agent}[\mathsf{in}\ g.\overline{\mathsf{in}_{\downarrow}}\ \mathit{entered}.\mathsf{open}\ \mathit{entered}.P \mid Q] \\
\mathit{System} & \triangleq & (\nu f)f[\mathsf{rec}\ X.(g[\mathsf{out}\ f.\overline{\mathsf{in}_{\downarrow}}\ \mathit{agent}.\mathsf{in}\ f.\overline{\mathsf{open}}\ \{g,f\}] \\
 & & \mid \overline{\mathsf{out}_{\uparrow}}\ g.\overline{\mathsf{in}_{\downarrow}}\ g.\mathsf{open}\ g.(\mathit{entered}[\mathsf{in}\ \mathit{agent}.\overline{\mathsf{open}}\ \{\mathit{entered},\mathit{agent}\}] \\
 & & \mid \overline{\mathsf{out}_{\uparrow}}\ \mathit{entered}.X))] \\
 & & \mid \mathsf{rec}\ Y.\overline{\mathsf{in}_{\uparrow}}\ g.\overline{\mathsf{out}_{\downarrow}}\ \mathit{agent}.\overline{\mathsf{out}_{\downarrow}}\ g.Y
\end{array}$$

This system implements two authentifications: in the first place, the Agent must be named *agent* - it will not enter f by accident. In the second place, it must know the password. Note that this is not the Firewall described in the original paper on Mobile Ambients [2], which relied on the secrecy of three keys. This version uses only one key and takes advantage of the synchronization mechanism to execute correctly. The basic ideas are the following: *System* receives *agent* and then recovers its original structure thanks to rec . The structure of g guarantees that, at any time, g may only contain one *agent*. On the other hand, *System* may contain any number of *agent*s.

Cab. Fig. 4 presents a CA version of the cab protocol. We do not give definitions for the city or for the sites, which only need to contain all movement authorizations, in addition to *client*s and *cab*s. Using cocapabilities, synchronizations in CA are both easier than in Mobile Ambients and atomic. Additionally, the system is not subject to the interferences we have presented: only *client*s may enter the cab, not just any "parasite" ambient which happens to contain capability in *cab*. Similarly, sites only welcome *client*s, *cab*s and *call*s.

Note that in this version, all clients must be named *client* in order to enter a *cab*. In order to relax this constraint, one could use *renaming* or the approximation of $\overline{\mathsf{SAin}}$ (see above).

Additionally, as was planned, Controlled Ambients permit the control of resources such as available space in cabs. As opposed to the Mobile Ambients version, we may easily check that the cab may contain at most *only one pas-*

senger and possibly an auxiliary ambient *call*, *trip*, *arrived* or *end*. This will be expressed formally using our type system in Sec. 3.

2.3 Benefits

We believe that the formalism of Controlled Ambients is more reasonable than Mobile Ambients or Safe Ambients. More reasonable insofar as the implementation of movements in ambient calculi suggests this kind of three-way synchronization. Let us consider the following transition in Mobile Ambients:

$$h[m[\mathsf{in}\ n] \mid n[\mathbf{0}]] \longrightarrow h[n[m[\mathbf{0}]]]\,.$$

As shown in [8,19], a practical implementation of this rule requires that h must be aware of the presence of n, no matter how n may have entered h. More generally, the execution of this rule will need a synchronization between n (who is present), m (who looks for n) and h (who knows about m and n). Similarly, the opening of ambient m by ambient h requires some complex synchronization between m and h in order to recover all processes and subambients of m in h and update presence registers of h. A prototype implementation has been developped [9] in order to validate these assertions.

Controlled Ambients are also more realistic as modeling tools. When a system receives informations, it must be by some action of his: the operating system "listens" on a device, the configuration server waits for a request by "listening" on some given TCP/IP port... Unfortunately, this listening aspect is not rendered at all by Mobile Ambients and only in half of the cases by Safe Ambients. Similarly, a system must be able to wait for several kinds of informations and to sort them according to their origin: the OS is able to differentiate data read on a disk from data read on the network or on the keyboard, while software may listen on several communication ports, for example. We can easily model such phenomena in CA, and if necessary take into account situations where some part of the system (like the network connexion itself) accepts data without listening for it, using renaming and infinite loops of cocapacities.

3 Typing Controlled Ambients

This section is devoted to the presentation of a basic type system for resource control in Controlled Ambients. We first describe the system and its properties, and then show the kind of information it is liable to capture on some examples.

3.1 The Type System

Type Judgments. The grammar for types is given in Fig. 5, and includes entries for the types of ambients, processes and messages ($\overline{\mathbb{N}}$ stands for $\mathbb{N} \cup \{\infty\}$).

Typing environments, ranged over with Γ, which are lists of associations of the form $x : A$ (for ambient names) or $X : U$ (for process variables). We write $\Gamma(x) = A$ (resp. $\Gamma(X) = U$) to represent the fact that environment Γ associates A (resp. U) to x (resp. X). $\Gamma, x : A$ stands for the extension of Γ with

$A ::= \mathrm{CA}\textsc{am}(s, e)[T]$	$s \in \overline{\mathbb{N}}, e \in \mathbb{N}$	ambients	$T ::= Ssh$	messages
$U ::= \mathrm{CA}\textsc{pr}(t)[T]$	$t \in \overline{\mathbb{N}}$	processes	$\mid\ t, A \ \ t \in \overline{\mathbb{N}}$	

Fig. 5. Types

the association $x : A$, possibly hiding some previous binding for x (and similarly for $\Gamma, X : U$).

The typing judgment for ambient names is of the form

$$\Gamma \ \vdash \ n : \mathrm{CA}\textsc{am}(s, e)[T] \,,$$

and expresses the fact that under assumptions Γ, n is the name of an ambient of *capacity* s, *weight* e, and within which messages carrying information of type T may be exchanged. The capacity s represents the number of resource units that are available within n (i.e. the space available for subambients - or *resources* for short), while e is the number of resources this ambient is occupying in its surrounding ambient. Note that while an ambient may have an infinite capacity ($s = \infty$), it cannot manipulate infinitely many resources ($e < \infty$). The type T for messages captures the kind of names being exchanged within n, similarly to Cardelli and Gordon's *topics of conversation* [3], augmented with an information t which represents a higher bound on the effect of exchanging messages within n (we shall come back to this below).

The typing judgment for processes is written

$$\Gamma \ \vdash \ P : \mathrm{CA}\textsc{pr}(t)[T] \,,$$

meaning that according to Γ, P is a process that may use up to t resources, and take part in conversations (that is, emit and receive messages) having type T.

Typing Rules. The rules defining the typing judgments are given on Fig. 6. We now comment on them. While typing (subjective) movements has no effect from the point of view of resources (rules $T - in$ and $T - out$), the rules $T - coin$ and $T - coout$, for the co-capabilities (in which δ ranges over a direction tag, which can be $\uparrow$ or $\downarrow$), express the meaning of types, according to the weight e of the moving ambient. Note that the number t of resources allocated to the process must remain positive after decreasing (rule $T - coout$). This is made possible by the subtyping property of the system (Lemma 1), together with rules $T - nil$ and $T - amb$, which allow one to allocate any number of resources to an inert process (inert from the point of view of the current ambient). This mechanism can be used for example to derive a typing for a process of the form $\overline{\mathsf{out}_\uparrow}\ n.\mathbf{0}$. Note also that the side condition $a \leq s$ in rule $T - amb$ expresses conformity with the capacity of the ambient.

When opening an ambient, we release the resources it had acquired (e), but at the same time we have to provide at least as many resources as its original capacity (s). The $\overline{\mathsf{open}}$ capability plays no role from the point of view of resource control, as illustrated by rule $T - coopen$ (note, still, that message types in the opening ambient and in the type of R are unified using this rule). We shall

$$\frac{\Gamma(n) = A}{\Gamma \vdash n : A}\ T-name \qquad \frac{\Gamma(X) = \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash X : \text{CA}\textsc{pr}(t')[T]}\ \begin{matrix} T-var \\ t' \geq t \end{matrix}$$

$$\frac{\Gamma, X : \text{CA}\textsc{pr}(t)[T] \vdash P : \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \mathsf{rec}\ X.P : \text{CA}\textsc{pr}(t')[T]}\ \begin{matrix} T-rec \\ t' \geq t \end{matrix}$$

$$\frac{\Gamma \vdash P : \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \mathsf{in}\ m.P : \text{CA}\textsc{pr}(t)[T]}\ T-in \qquad \frac{\Gamma \vdash P : \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \mathsf{out}\ m.P : \text{CA}\textsc{pr}(t)[T]}\ T-out$$

$$\frac{\Gamma \vdash P : \text{CA}\textsc{pr}(t)[T] \quad \Gamma \vdash m : \text{CA}\textsc{am}(s,e)[T']}{\Gamma \vdash \overline{\mathsf{in}_\delta}\ m.P : \text{CA}\textsc{pr}(t+e)[T]}\ T-coin$$

$$\frac{\Gamma \vdash P : \text{CA}\textsc{pr}(t)[T] \quad \Gamma \vdash m : \text{CA}\textsc{am}(s,e)[T']}{\Gamma \vdash \overline{\mathsf{out}_\delta}\ m.P : \text{CA}\textsc{pr}(t-e)[T]}\ \begin{matrix} T-coout \\ t \geq e \end{matrix}$$

$$\frac{\Gamma \vdash m : \text{CA}\textsc{am}(s,e)[T] \quad \Gamma \vdash P : \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \mathsf{open}\ m.P : \text{CA}\textsc{pr}(t-e+s)[T]}\ \begin{matrix} T-open \\ t-e+s \geq 0 \end{matrix}$$

$$\frac{\Gamma \vdash m : \text{CA}\textsc{am}(s,e)[T] \quad \Gamma \vdash R : \text{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \overline{\mathsf{open}}\ \{m,h\}.R : \text{CA}\textsc{pr}(t)[T]}\ T-coopen$$

$$\frac{}{\Gamma \vdash \mathbf{0} : U}\ T-nil \qquad \frac{\Gamma \vdash m : \text{CA}\textsc{am}(s,e)[T] \quad \Gamma \vdash P : \text{CA}\textsc{pr}(a)[T]}{\Gamma \vdash m[P] : \text{CA}\textsc{pr}(t)[T']}\ \begin{matrix} T-amb \\ a \leq s, e \leq t \end{matrix}$$

$$\frac{\Gamma, n : A \vdash P : U}{\Gamma \vdash (\nu n : A)P : U}\ T-res \qquad \frac{\Gamma \vdash P : \text{CA}\textsc{pr}(t)[T] \quad \Gamma \vdash Q : \text{CA}\textsc{pr}(t')[T]}{\Gamma \vdash P|Q : \text{CA}\textsc{pr}(t+t')[T]}\ T-par$$

$$\frac{\Gamma \vdash m : A}{\Gamma \vdash \langle m \rangle : \text{CA}\textsc{pr}(t')[t,A]}\ \begin{matrix} T-snd \\ t' \geq t \end{matrix} \qquad \frac{\Gamma, x : A \vdash P : \text{CA}\textsc{pr}(t)[t,A]}{\Gamma \vdash (x : A)P : \text{CA}\textsc{pr}(t')[t,A]}\ T-rcv$$

Fig. 6. Typing rules

present in Sec. 4 a richer system where a more precise typing of opening (and co-opening) permits a better control.

We now explain the typing rules for communication. Since reception of a message can trigger a process which will necessitate a certain amount of resources, we attach to the type of an ambient the maximum amount of resources needed by a receiving process running within it: this is information t in an ambient's topic of conversation. Put differently, messages are decorated with an integer representing at least as many resources as needed by the processes they are liable to trigger: we are thus somehow measuring an effect in this case. Note that our approach is based on the idea that one emission typically corresponds to several receptions. The dual approach could have been used, by putting in correspondence one reception and several concurrent emissions. Our experience in writing examples suggests that the first choice is more tractable.

Finally, rule $T - rec$ expresses the fact that a recursively defined process should run "in constant space".

3.2 Static Resource Control

We now present the main properties of the type system. Proofs are not given, but can be found in [20]. We start by some technical properties of typing derivations.

Lemma 1 (Subtyping). *Let P be a process and Γ an environment. Then:*
$$\text{if } \Gamma \vdash P : \mathrm{CA\small{PR}}(t)[T] \text{ then } \forall t' \geq t, \Gamma \vdash P : \mathrm{CA\small{PR}}(t')[T]\,.$$

Corollary 1 (Minimal typing). *If a process P is typeable in Γ with a topic type T, then there is a minimal $t \in \overline{\mathbb{N}}$ such that $\Gamma \vdash P : \mathrm{CA\small{PR}}(t)[T]$.*

Note that the minimal parameter t can be different for each possible value T (see for example rule $T - snd$).

Let us now examine resource control. In order to be able to state the properties we are interested in, we extend the notion of weight, which has been used for ambients, to processes, by introducing the notion of resource usage, together with a natural terminology:

Definition 1 (Resource policy and resource usage). *We call* resource policy *a typing context. Given a resource policy Γ, we define the* resource usage of a process P according to Γ, *written $Res_\Gamma(P)$, as follows:*

- *if $\Gamma(a) = \mathrm{CA\small{AM}}(s, e)[T]$, then $Res_\Gamma(a[P]) = e$;*
- *$Res_\Gamma(P_1|P_2) = Res_\Gamma(P_1) + Res_\Gamma(P_2)$;*
- *$Res_\Gamma((\nu n : A)\, P) = Res_{\Gamma,n:A}(P)$.*
- *in all other cases, $Res_\Gamma(P) = 0$;*

Note in particular that according to this definition, prefixed terms (capabilities, reception, recursion) do not contribute to a process' *current* resource usage (accordingly, their resource usage is equal to 0). We now define formally what it means for a process to respect a given resource policy.

Definition 2 (Resource policy compliance). *Given a resource policy Γ, we define the judgment $\Gamma \models P$ (pronounced "P complies with Γ"), as follows:*

- *$\Gamma \models n[P]$ iff $\Gamma \models P$ and $Res_\Gamma(P) \leq s$, where capacity s is given by $\Gamma(n) = \mathrm{CA\small{AM}}(s, e)[T]$;*
- *$\Gamma \models P_1|P_2$ iff $\Gamma \models P_1$ and $\Gamma \models P_2$;*
- *$\Gamma \models (\nu n : A)\, P$ iff $\Gamma, n : A \models P$;*
- *in all other cases, $\Gamma \models P$.*

The typing rules ensure that a typeable term complies with a resource policy:

Lemma 2 (Typeable terms comply with resource policies). *For any process P, resource policy Γ and process type U, if $\Gamma \vdash P : U$, then $\Gamma \models P$.*

The following theorem states that typability is preserved by the operational semantics of Controlled Ambients:

Theorem 1 (Subject reduction). *For any processes P, Q, resource policy Γ and type U, if $\Gamma \vdash P : U$ and $P \longrightarrow Q$, then $\Gamma \vdash Q : U$.*

As a direct consequence, we obtain our main result:

Theorem 2 (Resource control). *Consider a resource policy Γ and a process P such that $\Gamma \vdash P : U$ for some U. Then for any Q such that $P \longrightarrow^* Q$, it holds that $\Gamma \models Q$.*

3.3 Examples

We now revisit some examples of Sec. 2.2, and explain how they can be typed.
Notation: We shall write $a \rightsquigarrow_{\Gamma}^{T} (s, e)$ as a shorthand for $\Gamma(a) = \mathrm{CA}\textsc{am}(s, e)[T]$.

Renaming. The expression of renaming given in Sec. 2.2 is typeable as soon as there exists a typing environment Γ and a conversation type T such that

$$a \rightsquigarrow_{\Gamma}^{T} (s, e), , \quad b \rightsquigarrow_{\Gamma}^{T} (s, e) \text{ with } s \geq e\,, \quad \text{and} \quad \Gamma \vdash P : \mathrm{CA}\textsc{pr}(s)[T]\,.$$

We can actually slightly relax the conditions on types. One can show that the least set of conditions to type the renaming is

$$t_P \leq s_a, \quad e_b \leq s_a, \quad s_a \leq s_b, \quad \text{and} \quad e_a \leq s_b\,,$$

where $a \rightsquigarrow_{\Gamma}^{T} (s_a, e_a)$, $b \rightsquigarrow_{\Gamma}^{T} (s_b, e_b)$ and $\Gamma \vdash P : \mathrm{CA}\textsc{pr}(t_P)[T]$.

Firewall. Similarly, the firewall in Controlled Ambients, as defined in subsection 2.2 can be typed in a context Γ such that:

$$\mathit{agent} \rightsquigarrow_{\Gamma}^{T} (a_P + a_Q, 1), \quad \mathit{entered} \rightsquigarrow_{\Gamma}^{T} (0, 0),$$
$$f \rightsquigarrow_{\Gamma}^{T} (\infty, 0), \quad \text{and} \quad g \rightsquigarrow_{\Gamma}^{T} (1, 0)\,.$$

In particular, the typing of the recursive process $\mathsf{rec}\ X. \ldots$ in *System* entails a constraint of the form $\mathrm{CA}\textsc{pr}(t)[T] = \mathrm{CA}\textsc{pr}(t{+}1)[T]$. This is possible if and only if $t = \infty$, and as a consequence the capacity of f should also be ∞, so that the firewall is supposed to have infinite size. This is no surprise, since it may actually receive any number of external ambients. However, these ambients are contained in the firewall. Hence, one may still integrate this firewall as a component in a system with limited resources and resource control.

Cab. Let us consider an environment Γ such that:

$$\mathit{client} \rightsquigarrow_{\Gamma}^{T} (0, 1), \quad \mathit{call} \rightsquigarrow_{\Gamma}^{T} (1, 0), \quad \mathit{trip} \rightsquigarrow_{\Gamma}^{T} (0, 0), \quad \mathit{arrived} \rightsquigarrow_{\Gamma}^{T} (0, 0),$$
$$\mathit{end} \rightsquigarrow_{\Gamma}^{T} (0, 0), \quad \mathit{cab} \rightsquigarrow_{\Gamma}^{T} (1, 0), \quad \mathit{site}_i \rightsquigarrow_{\Gamma}^{T} (\infty, 0), \quad \text{and} \quad \mathit{city} \rightsquigarrow_{\Gamma}^{T} (0, 0)\,.$$

Note in particular that this resource policy specifies that among the ambients that may enter the cab, only those named *client* are actually “controlled”: this corresponds to the property we focus on when analyzing the cab. With these assumptions, the complete cab system is typeable. This means that resources are statically controlled in cabs: *at any step of its execution, the cab may contain at most one client*. Moreover, by changing our resource policy in such a way that ambients *call*, *trip*, *arrived* and *end* have weight 1 while *client* has weight 0, we can type the cab as having size 1. This typing lets us control the number of “auxiliary” ambients: at any time, at most one of those may be present in *cab*.

4 Refining the Resource Policy

While the basic system we have presented so far allows one to type many interesting examples, some relatively simple examples show its limitations. We illustrate these on two examples; consider the terms:

$$T_1 \triangleq a[\overline{\mathsf{open}}\ \{a,b\}.\mathsf{rec}\ X.(X \mid b[\mathbf{0}])] \ \mid\ \mathsf{open}\ a$$
$$T_2 \triangleq h[\mathsf{rec}\ X.(m[\overline{\mathsf{in}_\downarrow}\ n.\overline{\mathsf{out}_\uparrow}\ n.\overline{\mathsf{open}}\ \{m,h\}] \mid \overline{\mathsf{out}_\downarrow}\ n.\overline{\mathsf{in}_\uparrow}\ n.\mathsf{open}\ m.X) \mid n[\mathsf{rec}\ Y.\mathsf{in}\ m.\mathsf{out}\ m.Y]]$$

Suppose for T_1 that the weight of b is not 0. The construction $\mathsf{rec}\ X.(X \mid b[\mathbf{0}])$ then requires infinite resources. Although the execution would not use any resource inside a, the typing will require a to have an infinite capacity. Similarly, if in T_2 the weight of n is not 0, by following the evolution of this term, one may notice that a finite capacity for h *should be sufficient*. However, the typing rules we have presented compell the capacity of h to be infinite. In both cases, the typing system is not refined enough to express a resource control property, essentially because the rules for open and $\overline{\mathsf{open}}$ control resources too strictly. In order to be able to type such programs as "resource conscious", we have defined an extended version of our type system, in which ambient types are of the form $\mathrm{CA}\textsc{am}(s,e,r,z)[T]$, $r \in \overline{\mathbb{N}}, z \in \mathbb{N}$ and $z \leq s$. The rules are almost the same as above, except for the following ones:

$$\frac{\Gamma \vdash m : \mathrm{CA}\textsc{am}(s,e,r,z)[T] \quad \Gamma \vdash P : \mathrm{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \mathsf{open}\ m.P : \mathrm{CA}\textsc{pr}(t-e+z+r)[T]} \quad \begin{matrix} T-open \\ t-e+z+r \geq 0 \end{matrix}$$

$$\frac{\Gamma \vdash m : \mathrm{CA}\textsc{am}(s,e,r,z)[T] \quad \Gamma \vdash R : \mathrm{CA}\textsc{pr}(t)[T]}{\Gamma \vdash \overline{\mathsf{open}}\ \{m,h\}.R : \mathrm{CA}\textsc{pr}(t')[T]} \quad \begin{matrix} T-coopen \\ t \leq r, t' \geq s-z \end{matrix}$$

For lack of space, we do not discuss further this enriched type system. Let us just stress that the results of Sec. 3.2 also hold for it, and that "intermediate" type systems adopting only one of the two additional parameters could also be defined (for more details, see [20]).

5 Conclusion

The language of Controlled Ambients has been introduced to analyze resource control in a distributed and mobile setting through an accurate programming of movements and synchronisations. We have enhanced our formalism with a type system for the static control of resources, and extensions of the basic type system have also been presented. Further, examples show that indications on the maximal amount of resources needed by a process match rather closely the actual amount of resources which may be reached in the worst case, which suggests that the solution we propose could serve as the basis for a study of resource control properties on a larger scale.

Among extensions of the present work, we are currently enriching the language and type system to include communication of capabilities, as in the original Mobile Ambients calculus [2]. We are also studying type inference for our system.

It seems that by requiring the recursion variables to be explicitely typed, type inference is decidable, and a rather natural algorithm can compute a minimal type for a given process, if it exists. In particular, the "message" component of terms leads to a classical unification problem. The question becomes more problematic if no information is given for recursion variables: one can compute a set of inequalities (resembling those given for the example of renaming in Sec. 3), but solving it in the general case would require more work.

We plan to study whether our approach can be adapted to other formalisms for mobile and distributed computation such as the distributed π-calculus [18] or the distributed join-calculus [7]. Along the same ideas, in some process calculi without any primitive notion of site, we could choose to regard name creation as a form of resource allocation, and find out whether (some of) our ideas can be adapted to this setting.

We could also consider combining our type system for resource control with other typing disciplines, adapted from the Single Threadness types of [15], or the Mandatory Access Control of [1]. It seems that Controlled Ambients could also be used to approximate some of the analyses done in [6,**?**], where, in a context where *security levels* are associated with processes, types are used to check that no agent can access an information having a security level higher than its own. In the simple case where we have two security levels, we could attach weight 0 to agents of high level, and 1 to low-level agents, and store high-level information in ambients of size 0: in such a framework, our type system can guarantee that only high-level processes can enter high-level data. Of course this is a very rough approximation, and a more refined account of access control in Controlled Ambients needs further investigation.

We have not addressed the issue of behavioural equivalences for CA. A possible outcome of such a study could be to validate a more elaborate treatment of resources involving operations like garbage collection, which would allow one to make available uselessly occupied resources. An example is the perfect firewall equation of [10]: when $c \notin \mathrm{fn}(P)$, process $(\nu c)\, c[P]$ may manipulate some resources while being actually equivalent to $\mathbf{0}$.

Other Related Works. Other projects aim at controlling resources in possibly mobile systems without resorting to mobile process algebras. [14] presents a modified ML language with sized types in which bounds may be given to stack consumption. Like in our framework, resources are releasable entities; however, this approach seems more specialized than ours, and moreover concentrates on a sequential model. Similarly, [5] introduces a variant of the Typed Assembly Language *"augmenting TAL's very low-level safety certification with running-time guarantees"*, while Quantum, [17] may be used to describe distributed systems from the point of view of their resource consumption. In contrast to our approach, both systems consider non-releasable resources. Another programming language, PLAN [13], has been designed specifically for active networks, and also handles some form of resource bounds. Although PLAN accounts for both releasable (space, bandwidth) and non-releasable (time) resources, it handles neither recursion nor concurrency on one node.

These works all focus on resource control; however, none of these approaches can be directly compared to ours. It might be interesting to study if and how our methods could be integrated to these works, in order to combine several forms of resource control.

Another form of accounting on mobile ambients is introduced in [4]. In a calculus with a slightly different form of recursion than in CA (and without cocapabilities), the authors introduce a type system to count the number of active outputs and ambients (at any depth) in a process. This analysis, however, is not aimed at resources: it tries and isolate a finite-control fragment of mobile ambients on which model checking w.r.t. the Ambient Logic is decidable through state-space exploration.

Acknowledgements

We would like to thank Davide Sangiorgi for suggesting the original idea behind CA and providing insightful suggestions along this work.

References

1. Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *Proc. TACS 2001*, LNCS 2215, pages 38–63. Springer Verlag, 2001. 301
2. Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of FOSSACS'98*, volume 1378, pages 140–155. Springer Verlag, 1998. 288, 294, 300
3. Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Symposium on Principles of Programming Languages (POPL'99)*, pages 79–92. ACM Press, 1999. 296
4. Witold Charatonik, Andrew D. Gordon, and Jean-Marc Talbot. Finite-control mobile ambients. In *Proc. of ESOP'02*, volume 2305 of *LNCS*, pages 295–313, 2002. 302
5. Karl Crary and Stephanie Weirich. Resource bound certification. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 184–198. ACM Press, 2000. 301
6. Mariangiola Dezani-Ciancaglini and Ivano Salvo. Security types for mobile safe ambients. In *Proc. of ASIAN'00*, LNCS 1961, pages 215–236. Springer Verlag, 2000. 301
7. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. of CONCUR'96*, pages 406–421. Springer Verlag, 1996. 301
8. Cédric Fournet, Jean-Jacques Lévy, and Alan Schmitt. A distributed implementation of mobile ambients. In *Proc. of IFIP TCS'00*, pages 348–364. Springer Verlag, 1872. 295
9. T. Gazagnaire and D. Pous. Implémentation des Controlled Ambients en JoCaml. Students project – Magistère d'Informatique ENS Lyon, 2002. 295
10. Andrew D. Gordon and Luca Cardelli. Equational properties of mobile ambients. In *Proc. of FOSSACS'99*, volume 1578 of *LNCS*, pages 212–226. Springer Verlag, 1999. 301

11. Xudong Guan, Yiling Yang, and Jinyuan You. Making ambients more robust. In *Proc. of the International Conference on Software: Theory and Practice*, pages 377–384, 2000. 291
12. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98*, number 16.3 in ENTCS, pages 3–17. Elsevier, 1998.
13. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proc. ICFP'99*, pages 86–93. ACM Press, 1999. 301
14. John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999. 301
15. Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Symposium on Principles of Programming Languages*, pages 352–364. ACM Press, 2000. 288, 291, 293, 301
16. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *Proc. of POPL'02*, pages 71–80. ACM Press, 2002. 291
17. Luc Moreau. A distributed garbage collector with diffusion tree reorganisation and mobile objects. In *Proc. of ICFP'98*, pages 204–215. ACM Press, 1998. 301
18. James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proc. of POPL'98*, pages 378–390. ACM Press, 1998. 301
19. D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. In *Proc. of ICALP'01*, 2001. 295
20. D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. Technical Report 2002-16, LIP - ENS Lyon, 2002. 298, 300
21. Jan Vitek and Giuseppe Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, volume 1686 of *LNCS*. Springer Verlag, 1999. 291