HAL
open science

A Correct Abstract Machine for Safe Ambients

Daniel Hirschkoff, Damien Pous, Davide Sangiorgi

To cite this version:

Daniel Hirschkoff, Damien Pous, Davide Sangiorgi. A Correct Abstract Machine for Safe Ambients. COORDINATION, 2005, Namur, Belgium. 10.1007/11417019_2 . hal-01441466

HAL Id: hal-01441466

https://hal.science/hal-01441466v1

Submitted on 19 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Correct Abstract Machine for Safe Ambients*

Daniel Hirschkoff[1], Damien Pous[1], and Davide Sangiorgi[2]

[1] ENS Lyon, France
[2] Università di Bologna, Italy

Abstract. We describe an abstract machine, called GcPan, for the distributed execution of Safe Ambients (SA), a variant of the Ambient Calculus (AC).
Our machine improves over previous proposals for executing AC, or variants of it, mainly through a better management of special agents (*forwarders*), created upon code migration to transmit messages to the target location of the migration.
We establish the correctness of our machine by proving a weak bisimilarity result with a previous abstract machine for SA, and then appealing to the correctness of the latter machine.
More broadly, this study is a contribution towards understanding issues of correctness and optimisations in implementations of distributed languages encompassing mobility.

Introduction

In recent years there has been a growing interest for core calculi encompassing distribution and mobility. In particular, these calculi have been studied as a basis for programming languages. Examples include Join [9], Nomadic Pict [19], Kells [2], Ambients [6], Klaim [16].

In this paper we study issues of correctness and optimisations in implementations of such languages. Although our technical work focuses on Ambient-based calculi, we believe that the techniques can be of interest for the study of other languages: those mentioned above, and more broadly, distributed languages with mobility.

The underlying model of the Ambient calculus is based on the notion of *location*, called ambient. Terms in Ambient-based calculi describe configurations of locations and sub-locations, and computation happens as a consequence of movement of locations. The three primitives for movement allow: an ambient to enter another ambient (In), an ambient to exit another ambient (Out), a process to dissolve an ambient boundary and obtain access to its content (Open).

A few distributed implementations of Ambient-like calculi have appeared [10, 11, 17]. The study of implementations is important to understand the usefulness of the model from a programming language point of view. Such studies have shown that the `open` primitive, the most original one in the Ambient model, is also the most difficult to implement.

* Work supported by european FET - Global Computing project Profundis.

Another major difficulty for a distributed implementation of an ambient-like language is that each movement operation involves ambients on different hierarchical levels. For instance, the ambients affected by an `out` operation are the moving ambient and its initial and final parents; before the movement is triggered, they reside on three different levels. In [4, 5] locks are used to synchronise all ambients affected by a move. In a distributed setting, however, this lock-based policy can be expensive. For instance, the serialisations introduced diminish the parallelism of the whole system. In [10] the synchronisations are simulated by means of protocols of asynchronous messages. The abstract machine Pan [11] has two main differences. The first is that the machine executes typed Safe Ambients [13] (SA) rather than untyped Ambients. Typed SA is a variant of the original calculus that eliminates certain forms of interference in ambients, called grave interferences. These arise when an ambient tries to perform two different movement operations at the same time, as for instance $n[\texttt{in}\ h.P \mid \texttt{out}\ n.Q \mid R]$. The second reason for the differences in Pan is the separation between the logical structure of an ambient system and its physical distribution. Exploiting this, the interpretation of the movement associated to the capabilities is reversed: the movement of the `open` capability is physical, that is, the location of some processes changes, whereas that of `in` and `out` is only logical, that is, some hierarchical dependencies among ambients may change, but not their physical location. Intuitively, In and Out reductions are acquisition of access rights, and Open is exercise of them.

In Pan, the implementation of Open exploits *forwarders* – a common technique in distributed systems – to retransmit messages coming from the inside of an ambient that has been opened. These lead to two major problems:

- *persistence:* along the execution of the Pan, some forwarders may become useless, because they will never receive messages. However, these are never removed, and thus keep occupying resources (very often in examples, the ambients opened are leaves, and opening them introduces useless forwarders).
- *long communication paths:* as a consequence of the opening of several ambients, forwarder chains may be generated, which induce a loss of performance by increasing the number of network messages.

In this paper, we introduce GcPan, an abstract machine for SA that is more efficient than Pan. The main improvements are achieved through a better management of forwarders, which in the GcPan enjoy the following properties:

- *finite lifetime:* we are able to predict the number of messages that will be transmitted by a forwarder, so that we can remove the latter once these messages have all been treated;
- *contraction of forwarder chains:* we enrich the machine with a mechanism that allows us to implement a union-find algorithm to keep forwarder chains short, so as to decrease the number of messages exchanged.

The basis of the algorithms we use (e.g., Tarjan's union-find algorithm [18]) are well-known. However, adapting them to Ambient-like calculi requires some care, due to the specific operations proposed by these languages.

We provide a formal description of our machine, and we establish a weak bisimilarity result between Pan and GcPan. We then rely on the correctness of the Pan w.r.t. the operational semantics of SA, proved in [11], to deduce correctness w.r.t. SA.

An original aspect of our analysis w.r.t. the proof in [11] is that we compare two abstract machines, rather than an abstract machine and a calculus. This involves reasoning modulo 'administrative reduction steps' on both sides of the comparison to establish the bisimulation results. However, the fact that, in the GcPan, chains of forwarders are contracted using the union-find algorithm prevents us from setting up a tight correspondence between the two machines. This moreover entails that standard techniques for simplifying proofs of weak bisimilarity results (such as those based on the expansion preorder and up-to techniques) are not applicable. As a consequence, the bisimulation proof in which the two machines are compared is rather long and complex. Still, deriving the correctness w.r.t. SA through a comparison with Pan is simpler than directly proving the correctness of our machine w.r.t. SA. This holds because Pan and GcPan are both abstract machines, with a number of common features.

We believe that our study can also be of interest outside Ambient-based formalisms. For instance, the use of forwarders is common in distributed programming (see e.g. [7, 9]). However, little attention has been given to formal specification and correctness proofs of the algorithms being applied. The formalisation of the management and optimisations of forwarders that we provide and, especially, the corresponding correctness proof should be relevant elsewhere.

Outline of the paper. We present the design principles of the GcPan in Sect. 1. We then give the formal definition of the machine in Sect. 2, and describe the correctness proof in Sect. 3. Sect. 4 gives concluding remarks.

1 The Machine: Design Principles

We introduce the Safe Ambients (SA) calculus [14] and the Pan abstract machine [11]. We then present our ideas to remedy to some inefficiencies of Pan.

1.1 Safe Ambients

The SA calculus is an extension of the Mobile Ambients calculus [6] in which a tighter control of movements is achieved though *co-capabilities*. The four main reduction rules are:

$$\begin{array}{rcll} a[\mathtt{in}\ b.P \mid Q] \mid b[\overline{\mathtt{in}}\ b.R \mid S] & \longmapsto & b[a[P \mid Q] \mid R \mid S] & (IN) \\ b[a[\mathtt{out}\ b.P \mid Q] \mid \overline{\mathtt{out}}\ b.R \mid S] & \longmapsto & a[P \mid Q] \mid b[R \mid S] & (OUT) \\ \mathtt{open}\ b.P \mid b[\overline{\mathtt{open}}\ b.Q \mid R] & \longmapsto & P \mid Q \mid R & (OPEN) \\ \langle M \rangle \mid (x)P & \longmapsto & P\{M/x\} & (COM) \end{array}$$

Co-capabilities and the use of types (notably those for *single-threadedness*) make it possible to exclude *grave interferences*, that is, interferences among processes that can be regarded as programming errors (as opposed to an expected

form of non-determinism). A single-threaded (ST) ambient can engage in at most one external interaction, at any time its local process has only one *thread* (or active capability). In the sequel, when mentioning *well-typed* processes, this will be a reference to the type system of [14]. One of the benefits of the absence of grave interferences is that it is possible to define simpler abstract machines and implementations for ambient-based calculi: some of the synchronisation mechanisms needed to support grave interferences in a distributed setting [10] are not necessary (other possible benefits of SA, concerning types and algebraic theory, are discussed in [14]).

The modifications that yield typed SA have also computational significance. In the Mobile Ambient interaction rules, an ambient may enter, exit, or open another ambient. The latter ambient undergoes the action; it has no control on *when* the action takes place. In SA this is rectified: a movement is triggered only if both participants agree. Further, the modifications do not seem to exclude useful programming examples. In some cases the SA programs can actually be simpler, due to the tighter control over interferences. We refer to [14] for details.

1.2 The Pan

The Pan [11] separates the logical distribution of ambients (the tree structure given by the syntax) from their physical distribution (the actual sites they are running on). An ambient named n is represented as as a *located agent* h: $n[P]_k$, where h is the physical location, k the location of the parent of the ambient, and P is its local process. There can be several ambients named n, but a location h uniquely identifies an ambient. The physical distribution is flat, so that the SA process $a[b[c[] \mid P] \mid d[Q]]$ is represented by the parallel composition (also called *net*) h_1: $a[]_{\mathtt{root}} \parallel h_2$: $b[P]_{h_1} \parallel h_3$: $c[]_{h_2} \parallel h_4$: $d[Q]_{h_1}$. For the sake of simplicity, and when this does not lead to confusion, we sometimes use a to refer to the location of an ambient named a.

In the Pan, an ambient has only access to its parent location and to its local process: it does not know its sub-ambients. This simplifies the treatment of ambient interactions: communication between locations boils down to the exchange of asynchronous messages (while manipulating lists of child locations would mean setting many synchronisation points along computation).

In the Pan an ambient interaction is decomposed into three steps: an ambient that wants to move first sends a *request message* to its parent and enters in *wait state*. The father ambient then looks for a valid *match* to this request, and, upon success, sends appropriate *completion messages* back, using the location names contained in the request messages. The scenarios corresponding to the three kinds of movement are depicted in Fig. 1, where white squares (resp. grey squares) represent locations (resp. locations in wait state), and arrows indicate messages.

We remark that, for In and Out moves, the decision is taken by the parent of the moving ambient. Also note that in the Out move, the grandparent, that actually receives a new child, does not take part in any interaction: this follows

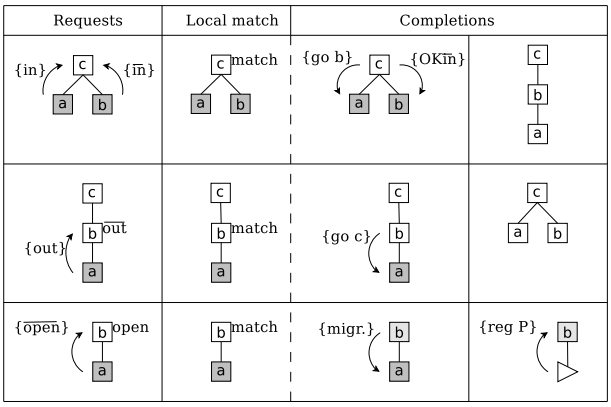

Fig. 1. Simulation of the SA reductions by the Pan.

the design of Pan, in which the relation between parent and child 'goes upwards'. Moreover, performing an In or Out movement does not trigger any physical migration in the Pan, only the logical distribution of ambients is affected.

On the other hand, in an Open move, the code of the process that is local to the ambient being opened (a in Fig. 1) is sent to the parent ambient (via a `reg` message). Indeed, b has no access to its children, and hence it cannot inform them to send their requests to b instead of a. The solution adopted in the Pan is to use *forwarders*: any message reaching a will be routed to b by an agent represented by a triangle in Fig. 1, and denoted by '$h \triangleright k$' in the following (h and k being the locations associated respectively to a and b).

The logical structure of the Pan is hence a tree whose nodes are either located ambients or forwarders. Request (resp. completion) messages are transmitted upwards (resp. downwards) along the tree.

The design ideas that we have exposed entail two major drawbacks in the execution of the Pan: persistence of forwarders (even when there are no sub-ambients and therefore no message can reach the forwarder), and long forwarder chains which generate an overload in terms of network traffic.

1.3 The GcPan

We now explain how we address the problems exposed above, and what influence our choices have on the design of the Pan.

Counters. A forwarder can be thought of as a *service* provided to the children of an opened ambient. Our aim is to be able to bring this service to an end once there are no more children using it. At the same time, we wish to preserve asynchrony in the exchange. For this, GcPan agents are enriched with a kind of reference counter. Forwarders have a finite lifetime, at the end of which they are garbage collected. The lifetime of a forwarder intuitively corresponds to the number of locations that point to it. A counter is decremented each time a

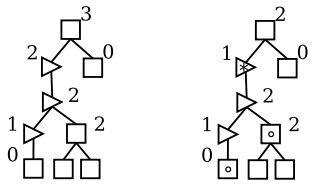

Fig. 2. Depth and local counting.

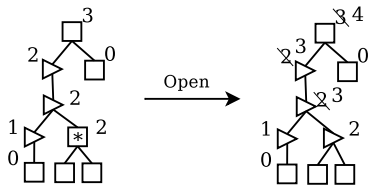

Fig. 3. Problem with depth counting.

message is forwarded. If the counter is zero, then the forwarder is a leaf in the logical structure of the net and can safely be removed.

We can think of two ways of associating a lifetime to a forwarder (Fig. 2):

- *(depth counting)* The most natural idea is probably to decorate each located ambient with the number of immediate sub-ambients it has. In doing this, we ignore forwarders, because request messages that are routed via forwarders can only be emitted by located ambients. This solution seems however difficult to implement, due to the asynchrony in the model. This is illustrated by Fig. 3: if the ambient marked '$*$' is opened, the counters along the whole forwarders chain should be updated before any of the children can send a message.
- *(local counting)* In our approach, we only consider the *immediate* children of a location (hence the name *local*), including forwarders. As a consequence, we may well have the situation where several sub-ambients are 'hidden' under a forwarder, so that the counter at a given location has no direct relationship with the number of sub-ambients. The difficulty described above does not arise in this setting: the forwarders chain remains unaffected by the opening, a located ambient becomes a forwarder, and this does not affect the counting.

Synchronisation problems and blocked forwarders. In the local approach, one has to be careful in transmitting request messages. Consider for instance the forwarder marked '$*$' on the right of Fig. 2: each ambient marked with a circle can send a request message. The intermediate forwarder cannot forward directly these two requests, since the '$*$-forwarder' is willing to handle only one message. In the GcPan, an agent can send only one message to a given forwarder, and whenever this message is sent, the agent commits to relocate itself if the agent it was talking to turns out to be a forwarder.

Implementing this policy is easy for located ambients, that enter a wait state just after emitting a request message. We only have to decorate completion messages with the appropriate information for relocation. For forwarders, we need to devise a similar blocking mechanism: once a forwarder has transmitted a request message, it enters a blocked state and waits for a $\mathsf{go}_{\triangleright}$ completion message, which contains the name of the location to which the next request should be forwarded. Fig. 4 illustrates this (blocked forwarders are represented by reversed, grey triangles): message $\{N\}$ is emitted by the grey ambient, and then routed towards the parent location, which has the effect of blocking forwarders along

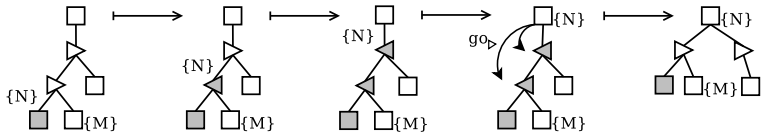

Fig. 4. Relocation of forwarders.

the way. When $\{N\}$ reaches the parent ambient, $\mathtt{go}_{\triangleright}$ messages are generated so that forwarders can resume execution, just below the parent ambient. This way, short communication paths between locations are maintained: at the end of the scenario, message $\{M\}$ is closer to its destination, without having been routed yet. The technique we use is based on Tarjan's union-find algorithm [18].

Remark 1 (Communication protocols). We comment on the way messages are transmitted in the GcPan:

- *(race situations)* Having blocked forwarders leads to race situations: consider the scenario of Fig. 4, where messages $\{M\}$ and $\{N\}$ are sent at the bottom of a chain of forwarders. When $\{N\}$ goes through the lowest forwarder, $\{M\}$ has to wait for the arrival of the former at the top of the chain, so that a $\mathtt{go}_{\triangleright}$ message is emitted to rearrange forwarders (following the union-find algorithm). The loss, from $\{M\}$'s point of view, is limited: once $\{N\}$ has entered the parent location, $\{M\}$ can reach the latter in three steps (the $\mathtt{go}_{\triangleright}$ message plus two routing steps).
- *(relocation strategy)* In the GcPan, the ambient that sits at the end of a forwarder chain broadcasts a relocation message ($\mathtt{go}_{\triangleright}$) to all blocked forwarders in the chain. In a previous version of our machine, this message was propagated back along the chain, unblocking the forwarders in a sequential fashion. We prefer the current solution because it brings more asynchrony (race situations introduced a delay of $n + 2$ because the relocation message had to go trough the whole chain in order to unblock all forwarders). On the other hand, request messages carry more information in our approach (we need to record the set of forwarders that have been crossed). However, in practise, we observe that long chains of forwarders are very unlikely to be produced in our machine, thanks to the contraction mechanism we adopt. Consequently, such messages have in most cases a rather limited size.

Updating counters along SA movements. Going back to the GcPan transitions corresponding to the basic SA moves (the *match* transitions of Fig. 1), we need to be able to maintain coherent counters along the three kinds of movement. This is achieved as follows (the names we use correspond to Fig. 1):

In: The overall result of the transition will be that c decrements its counter, and b increments its counter upon reception of the $\mathtt{OK}\overline{\mathtt{in}}$ completion.

Open: counters do not need to be modified.

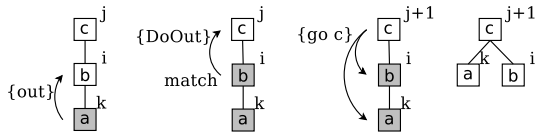

Fig. 5. Counters along an Out move: first approach.

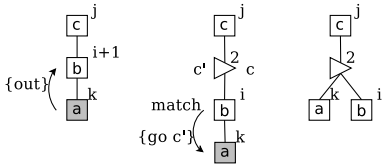

Fig. 6. Counters along an Out move: our approach.

Out: in the Pan, the match between the capability and the co-capability is done at b, and the grandparent c is not aware of the movement. In the GcPan, b decrements its counter, a is unaffected, but, a priori, c has to increment its counter, since it receives a new child, a.

A possibility would be to let b pass the control on the move along to c, that is then in charge of sending the completion messages: this solution is represented in Fig. 5. Adopting this protocol means introducing a new kind of message in the machine (message `DoOut` in Fig. 5, from parent to grandparent), and having two agents in wait state (the child and the parent) while the control is at the grandparent location.

We chose a different solution, that does not use an additional kind of message and in which interaction is more local and asynchronous. It is depicted in Fig. 6: at b, we create a new forwarder that collects the parent (b) and the child (a) under a unique agent, so that the grandparent counter does not need to be updated. It may seem rather counterproductive to add a new form of forwarder creation this way, considering that our goal in designing the GcPan is precisely to erase as many forwarders as possible. We can however observe that:

- the created forwarder has a lifetime of 2, which is short;
- from the point of view of the implementation, the forwarder is created on the parent site, so that the extra communication between the parent and the forwarder will be local.

2 Formal Definition of the Machine

2.1 GcPan Nets

The syntax of the terms of the GcPan (referred to as GcPan *nets*, or simply *nets*) is presented on Table 1. Agents in the GcPan are either located ambients

$a, b, m, n, .. \in Names \qquad h, k, .. \in Locations \qquad p, q, .. \in Names \cup Locations$
$i, j, .. \in \mathbb{N} \qquad x, y, .. \in Variables$

Networks:

$$\begin{array}{rll} A := & \mathbf{0} & \text{(empty net)} \\ \mid & Agent & \text{(agent)} \\ \mid & h\{Msg\} & \text{(emission)} \\ \mid & A_1 \parallel A_2 & \text{(composition)} \\ \mid & (\boldsymbol{\nu} p)A & \text{(restriction)} \end{array} \qquad \begin{array}{rll} Agent := & h \rhd^i k & \text{(forwarder)} \\ \mid & h \lhd^i & \text{(blocked forwarder)} \\ \mid & h^i\colon n[P]_k & \text{(located ambient)} \\ & & \\ Msg := & req/E & \text{(request)} \\ \mid & compl & \text{(completion)} \end{array}$$

$$\begin{array}{rll} req := & \mathtt{in}\ n, h & \text{(agent at } h \text{ wants to enter } n\text{)} \\ \mid & \overline{\mathtt{in}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts entrance)} \\ \mid & \mathtt{out}\ n, h & \text{(agent at } h \text{ wants to leave } n\text{)} \\ \mid & \overline{\mathtt{open}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts opening)} \end{array}$$

$$\begin{array}{rll} compl := & \mathtt{go}\ h & \text{(request completed, go to } h\text{)} \\ \mid & \mathtt{go}_{\rhd}\ h & \text{(relocate forwarder to } h\text{)} \\ \mid & \mathtt{OK}\overline{\mathtt{in}}\ h & \text{(request } \overline{\mathtt{in}} \text{ completed, go to } h\text{)} \\ \mid & \mathtt{mig}\ h & \text{(request } \overline{\mathtt{open}} \text{ completed, migrate to } h\text{)} \\ \mid & \mathtt{reg}^s\ P & \text{(add } P \text{ to the local processes)} \end{array}$$

Processes:

$$\begin{array}{l} P := \mathbf{0} \mid P_1 \mid P_2 \mid (x)P \mid (\boldsymbol{\nu} n)P \mid X \mid M.P \mid \mathrm{rec}\, X.P \mid M[P] \mid \mathrm{wait}.P \mid \langle M \rangle \mid \{req\} \\ M := x \mid n \mid \overline{\mathtt{out}}\ M \mid \mathtt{in}\ M \mid \mathtt{open}\ M \mid \overline{\mathtt{in}}\ M \mid \overline{\mathtt{open}}\ M \mid \mathtt{out}\ M \end{array}$$

Table 1. GcPan Syntax

($h^i\colon n[P]_k$ is the ambient $n[P]$ running at h, whose parent is located at k), blocked or running forwarders ($h \lhd^i$ is a blocked forwarder at h, while $h \rhd^i k$ is willing to transmit messages from h to k). In the three cases, the superscript $i \in \mathbb{N}$ represents the value of the agent counter.

E denotes a list of locations. A message of the form $k\{req/E\}$ denotes the request req, located at k, and having been transmitted through the locations contained in E. $k\{req\}$ is an abbreviation for $k\{req/[]\}$, and we write $h ::E$ to denote the list obtained by adding h to E. Reception ($(x)P$) and restriction ($(\boldsymbol{\nu} x)P$) are binders. Given a process P, we let $FL(P)$ stand for the set of free locations of P. An occurrence in a process P is *guarded* if it appears under a prefix or a reception. We suppose that in every process of the form rec $X.P$, all occurrences of X in P are guarded.

Other aspects of the syntax of messages are explained in Subsection 2.2.

The definition of structural congruence, $\equiv$, is mostly standard, and omitted. The only peculiarity is that $\equiv$ does not allow a name restriction to be extruded out of a located ambient in a transparent way: the net $h\colon n[(\boldsymbol{\nu} m)(\mathtt{in}\ m)]_k$ is not equivalent to $(\boldsymbol{\nu} m)h\colon n[\mathtt{in}\ m]_k$. Such a transformation is handled using reduction, and not as a structural congruence rule, because at the level of im-

plementation, generating names that are fresh even for possibly distant agents involves a nontrivial distributed protocol.

The GcPan (resp. Pan) encoding of a SA process P is written $[\![P]\!]_{gc}$ (resp. $[\![P]\!]$). $[\![P]\!]$ is defined in [11], and $[\![P]\!]_{gc}$ is defined as follows:

Definition 1 (Translation from SA to GcPan). *Given an SA process P, we define:* $[\![P]\!]_{gc} \triangleq \texttt{root}^0\colon \texttt{rootname}[P]_{\texttt{rootparent}}$.

2.2 Reduction Rules

Fig. 7 presents the operational semantics of GcPan nets. The following explanations should help in reading the rules and understanding how they implement the ideas we have discussed above.

Form of the rules: rules for emission of request messages and for local reductions have the shape $P \xrightarrow[h:n]{k} P' \gg^i M$, to denote the fact that process P, running in ambient n at location h, may liberate message M and evolve into process P', k being the parent location of h. Integer i decorating $\gg$ records the increment that has to be brought to h's counter (cf. rule Proc-Agent below). $\gg$ is an abbreviation of $\gg^0$. When n, h or k are unimportant, we replace them with '-'. We do the same in the rules for consumption of completion messages, when the parent location of a located ambient is not important.

In rule Local-Com, $P\{x\backslash M\}$ denotes process P in which x is substituted with M. In rule Loc-Rcv, we use the following notations, for $E = [e_1; \ldots; e_i]$: $E\{M\}$ stands for $e_1\{M\} \parallel \ldots \parallel e_i\{M\}$, and $\#E$ is i.

Six kinds of rules govern the behaviour of a GcPan net, according to the way SA transitions are implemented in our model.

- Before being able to start interacting, a process might have to allocate new resources for the creation of new names and for the spawning of new ambients: this is handled by the rules for *creation*.
- The translation of a prefixed SA process starts with emitting a request for interaction, which is expressed by the corresponding four rules for *emission of request messages*.
- Request messages are transmitted through forwarders and reach their destination location via the rules for *transmission of request messages*.
- *Local reductions* describe the steps that correspond to SA transitions. Such reductions do the matching between a capability and the corresponding co-capability, and generate completion messages.
  Notation $\ggg$ is introduced similarly to $\gg$, in order to handle the Out movement, that is achieved using rule Proc-Agent'. The subscript k' denotes the source location of the created forwarder (we have to adopt a special treatment for this case because the newly created forwarder is outside the 'active location').
- Some rather standard *inference rules* are used to transform a local reduction into a transition of the whole GcPan net.

Creation

$$[\textsc{New-Locamb}]\quad h^i\colon m[n[P]\,|\,Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu}k)(k^0\colon n[P]_h) \quad k \notin FL(P)$$

$$[\textsc{New-Res}]\quad h^i\colon m[(\boldsymbol{\nu}n)P]_k \longmapsto (\boldsymbol{\nu}n)(h^i\colon m[P]_k)$$

Emission of request messages

$$[\textsc{Req-In}]\quad \mathtt{in}\ m.P \xrightarrow[h:-]{k} \mathrm{wait}.P \gg k\{\mathtt{in}\ m, h\}$$

$$[\textsc{Req-Coin}]\quad \overline{\mathtt{in}}\ n.P \xrightarrow[h:n]{k} \mathrm{wait}.P \gg k\{\overline{\mathtt{in}}\ n, h\}$$

$$[\textsc{Req-Out}]\quad \mathtt{out}\ m.P \xrightarrow[h:-]{k} \mathrm{wait}.P \gg k\{\mathtt{out}\ m, h\}$$

$$[\textsc{Req-Coopen}]\quad \overline{\mathtt{open}}\ n.P \xrightarrow[h:n]{k} \mathrm{wait}.P \gg k\{\overline{\mathtt{open}}\ n, h\}$$

Transmission of request messages

$$[\textsc{Fw-Send}]\quad h \rhd^{i+1} k \parallel h\{req/E\} \longmapsto h \lhd^{i} \parallel k\{req/h{::}E\}$$

$$[\textsc{Fw-SendGC}]\quad h \rhd^{1} k \parallel h\{req/E\} \longmapsto k\{req/E\}$$

$$[\textsc{Fw-Reloc}]\quad h \lhd^{i} \parallel h\{\mathtt{go}_{\rhd}\ k\} \longmapsto h \rhd^{i} k$$

$$[\textsc{Loc-Rcv}]\quad h^{i+1}\colon n[P]_k \parallel h\{req/E\} \longmapsto h^{i+\#E}\colon n[P\,|\,\{req\}]_k \parallel E\{\mathtt{go}_{\rhd}\ h\}$$

Local reductions

$$[\textsc{Local-Com}]\quad \langle M\rangle\,|\,(x).P \xrightarrow[-:-]{-} P\{x\backslash M\} \gg \mathbf{0}$$

$$[\textsc{Local-In}]\quad \{\mathtt{in}\ n, h\}\,|\,\{\overline{\mathtt{in}}\ n, k\} \xrightarrow[h':-]{-} \mathbf{0} \gg^1 h\{\mathtt{go}\ k\} \parallel k\{\mathtt{OK}\overline{\mathtt{in}}\ h'\}$$

$$[\textsc{Local-Out}]\quad \{\mathtt{out}\ n, h\}\,|\,\overline{\mathtt{out}}\ n.P \xrightarrow[-:n]{-} P \ggg_{k'} h\{\mathtt{go}\ k'\}$$

$$[\textsc{Local-Open}]\quad \mathtt{open}\ n.P\,|\,\{\overline{\mathtt{open}}\ n, h\} \xrightarrow[h':-]{-} \mathrm{wait}.P \gg^1 h\{\mathtt{mig}\ h'\}$$

Inference rules

$$[\textsc{Proc-Agent}]\quad \frac{P \xrightarrow[h:n]{k} P' \gg^s M \qquad Q \text{ has no unguarded ambient}}{h^i\colon n[P\,|\,Q]_k \longmapsto h^{i+s}\colon n[P'\,|\,Q]_k \parallel M}$$

$$[\textsc{Proc-Agent'}]\quad \frac{P \xrightarrow[h:n]{k} P' \ggg_{k'} M \quad Q \text{ has no unguarded ambient}, k' \notin FL(P|Q)}{h^i\colon n[P\,|\,Q]_k \longmapsto (\boldsymbol{\nu}k)'(k' \rhd^2 k \parallel h^i\colon n[P'\,|\,Q]_{k'} \parallel M)}$$

$$[\textsc{Par-Agent}]\quad \frac{A \longmapsto A'}{A \parallel B \longmapsto A' \parallel B} \qquad \frac{A \longmapsto A'}{(\boldsymbol{\nu}p)A \longmapsto (\boldsymbol{\nu}p)A'}\quad [\textsc{Res-Agent}]$$

$$[\textsc{Struct-Cong}]\quad \frac{A \equiv A' \qquad A' \longmapsto A'' \qquad A'' \equiv A'''}{A \longmapsto A'''}$$

Consumption of completion messages

$$[\textsc{Compl-Parent}]\quad h\{\mathtt{go}\ k\} \parallel h^i\colon n[P\,|\,\mathrm{wait}.Q]_{_} \longmapsto h^i\colon n[P\,|\,Q]_k$$

$$[\textsc{Compl-Coin}]\quad h\{\mathtt{OK}\overline{\mathtt{in}}\ k\} \parallel h^i\colon n[P\,|\,\mathrm{wait}.Q]_{_} \longmapsto h^{i+1}\colon n[P\,|\,Q]_k$$

$$[\textsc{Compl-Migr}]\quad h\{\mathtt{mig}\ k\} \parallel h^{i+1}\colon n[P\,|\,\mathrm{wait}.Q]_{_} \longmapsto h \rhd^{i+1} k \parallel k\{\mathtt{reg}^0\ P\,|\,Q\}$$

$$[\textsc{Compl-Migr'}]\quad h\{\mathtt{mig}\ k\} \parallel h^0\colon n[P\,|\,\mathrm{wait}.Q]_{_} \longmapsto k\{\mathtt{reg}^1\ P\,|\,Q\}$$

$$[\textsc{Compl-Reg}]\quad h\{\mathtt{reg}^s\ R\} \parallel h^{i+s}\colon n[P\,|\,\mathrm{wait}.Q]_k \longmapsto h^i\colon n[P\,|\,Q\,|\,R]_k$$

Fig. 7. Reduction rules

The premises about unguarded ambients insure that all sub-ambients of an ambient are activated as soon as possible (rule NEW-LOCAMB), before any local reduction takes place — here we exploit the fact that recursions are guarded, otherwise there could be an infinite number of ambients to create.
- The rules for *consumption of completion messages* describe how agents resume computation when they are informed that a movement has occurred.

Counting: counters have to be kept coherent along the transitions of a net. Intuitively, to understand the counting for an agent located at h, in a given GCPAN configuration, we have to consider:

- the number of non waiting ambient locations that are immediate children of h (of the form k^i: $n[P]_h$);
- the number of child forwarders ($k \rhd^i h$);
- the number of request messages emitted to h ($h\{req/E\}$);
- the number of completion or relocation messages whose effect will be to increment the number of immediate children of h ($k\{\mathtt{go}\ h\}$, $k\{\mathtt{go}_{\rhd} h\}$, ...).

We explain below how our accounting is preserved along the moves:

IN: The two brother ambients taking part in an IN move (h and k) are in wait state at the moment when the parent ambient (h') matches the corresponding requests. Ambients in wait state are pending, and hence are not taken into account by the counter of h'. As a consequence, h' has to *increment* its counter in rule LOCAL-IN. The role of the completion message $k\{\mathtt{OK}\overline{\mathtt{in}}\ h'\}$ is to bring k under h' (which was its original father in case there was no forwarder between h and h'). Similarly, h, that will receive h' as a new child (message $h'\{\mathtt{go}\ h\}$ and rule COMPL-PARENT), also increments its counter, upon reception of message $\mathtt{OK}\overline{\mathtt{in}}$ (rule COMPL-COIN).

OUT: As previously, the intuition is that the parent (h') loses a child (h), and has to decrement its counter, but since this child is in wait state, there is nothing to do. The freshly created forwarder allows us to keep the grandparent counter unaffected: the forwarder hides both parent and child (and hence the value of its counter is set to two).

OPEN: The opening location (h') increments its counter to take into account the creation of the forwarder (rule COMPL-MIGR, that lets h, the opened location, react to a `mig` completion message). In the case where the counter of h is null, h has no child: there is no need for such a forwarder, and we avoid creating it (rule COMPL-MIGRGC). We must be careful, though, to let h' know that it has to undo the increment of its counter, which is achieved using the flag s decorating the `reg` message (rule COMPL-REG).

Forwarders behaviour is defined by the rules for transmission of request messages. We illustrate these by the following reductions, that show the behaviour of a message carrying request R traversing three forwarders h_1, h_2 and h_3 to reach its *real target*:

$$
\begin{array}{lll}
 & h_1\{R/[]\} \parallel h_1 \rhd^3 h_2 \parallel h_2 \rhd^1 h_3 \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & \\
\longmapsto & h_1 \lhd^2 \parallel h_2\{R/[h_1]\} \parallel h_2 \rhd^1 h_3 \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & [\textsc{Fw-Send}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3\{R/[h_1]\} \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & [\textsc{Fw-SendGC}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3 \lhd^3 \parallel k\{R/[h_3{::}h_1]\} \parallel k^2\colon n[P] & [\textsc{Fw-Send}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3 \lhd^3 \parallel h_3\{\mathtt{go}_{\rhd}k\} \parallel h_1\{\mathtt{go}_{\rhd}k\} \parallel k^3\colon n[P \mid \{R\}] & [\textsc{Loc-Rcv}] \\
\longmapsto & h_1 \lhd^2 \parallel h_1\{\mathtt{go}_{\rhd}k\} \parallel h_3 \rhd^3 k \parallel k^3\colon n[P \mid \{R\}] & [\textsc{Fw-Reloc}] \\
\longmapsto & h_1 \rhd^2 k \parallel h_3 \rhd^3 k \parallel k^3\colon n[P \mid \{R\}] & [\textsc{Fw-Reloc}]
\end{array}
$$

First, the message gets transmitted by forwarder h_1, which decrements its counter, adds its name to the list decorating the message before transmission to h_2, and blocks. In the second step of transmission, since h_2's counter is equal to one, h_2 gets garbage collected, and the message is passed to h_3, which transmits it to k (along the lines of the first step). Then the target location k receives the message, and reacts by broadcasting a $\mathtt{go}_{\rhd}k$ relocation message to each agent that has been registered in the list decorating the message. k's counter is incremented by the size of this list *minus one*: all forwarders except the uppermost one will become new direct children of the parent location (note that in the case of an empty chain of forwarders, we decrement the counter because the direct child is in wait state, and hence pending). Finally, the blocked forwarders react to the relocation messages by moving to their new location, and resume computation.

3 Correctness of the Machine

We establish the correctness of our machine by showing a weak barbed bisimilarity result with the Pan. Although the overall structure of the proof has similarities with [11], there are important differences. First of all, we compare two abstract machines, rather than a machine and a calculus as in [11]. The correspondence we can make between two configurations of the Pan and the GcPan is fairly coarse (barbed bisimilarity), because the machines route messages and manage forwarders differently.

Also, a few results, that are crucial in the proof for Pan [11] do not hold for GcPan. For instance in Pan, we have

$$(\boldsymbol{\nu} h)(h \rhd k \parallel A) \;\succeq\; A\{k\backslash h\}\,,$$

where $\succeq$ stands for *expansion*, a behavioural preorder that guarantees that, intuitively, if $P \succeq Q$, P exhibits the same behaviour as Q modulo some extra internal computation (expansion is not explicitly mentioned in [11], but the technique is essentially equivalent). This makes it possible, using weak bisimulation up to expansion, to factorise reasoning about forwarders and to considerably reduce the size of the relations needed to establish bisimilarity results.

Unfortunately the corresponding expansion law does not hold in our setting. This is due to the way the union-find algorithm works: rearranging forwarders entails an initial cost, and generates race situations. This cost is later compensated by the fact that messages are transmitted on shorter chains. This kind of delayed improvement cannot be captured using expansion because $P \succeq Q$ if Q is 'better than P' *at every step* (see [12] for a proof of the non-expansion result).

The notion of equivalence we adopt is barbed bisimulation [15], that we denote $\approx$. Here we use it to compare states belonging to different transition systems.

In GcPan the observability predicates $\Downarrow_n$ (where n is any name) are defined as follows. A is *observable at* n means, intuitively, that A contains an agent n that accepts interactions with the external environment. Formally: $A \downarrow_n$ if $A \equiv (\boldsymbol{\nu}\overline{p})\,\big(\mathtt{root}\colon \mathtt{rootname}[\{M, h\} \mid P]_{\mathtt{rootparent}} \parallel A'\big)$ where $M \in \{\overline{\mathtt{in}}\ n,\ \overline{\mathtt{open}}\ n\}$ and $n \notin \overline{p}$ (here $\overline{p}$ stands for a set of names or localities). Then, using $\Longmapsto$ for the reflexive and transitive closure of $\longmapsto$, we write $A \Downarrow_n$ if $A \Longmapsto \downarrow_n$. In SA and Pan, observability is defined similarly (see [11]). Our main results are:

Theorem 1. *For any well-typed SA process P, we have $[\![P]\!] \approx [\![P]\!]_{gc}$.*

Corollary 1. *Let P be a well-typed SA process, then $[\![P]\!]_{gc} \approx P$.*

Proof: *By [11], we have $[\![P]\!] \approx P$. Theorem 1 allows us to conclude.* $\diamondsuit$

The above corollary implies, for instance, that for all n, $P \Downarrow_n$ iff $[\![P]\!]_{gc} \Downarrow_n$.

For lack of space, we only give the main intuitions behind the proof of Theorem 1 (the reader is referred to [12] for details). The first step is to introduce a notion of *well-formed net*, and to show that it is preserved by reduction. Well-formedness allows us to express which nets are 'reasonable', in particular w.r.t. the destination of messages and the value of counters.

In Pan and GcPan, the routing of messages is deterministic and does not change the bisimilarity class of a net. Therefore, the main idea in introducing the candidate bisimulation relation to establish Theorem 1 is to define a kind of normal form for nets, in which all messages are routed to their destination and the nets in both machines can be compared directly. Based on this, we derive some preliminary lemmas to show that whenever a message is routed to its destination in a given configuration of one of the machines, the other machine can do the same (this might involve some additional transitions in the GcPan, because, as seen above, race conditions may prevent a message from being 'directly routable'). These lemmas are then used in a modular way to construct the bisimulation proof, that amounts to show that by definition, processes related by the candidate bisimulation exhibit the same observables and preserve this property.

4 Final Remarks

Developments of our machine. Besides ST ambients, the other main type for SA processes [14] is that of *immobile* ambients (IM). An immobile ambient is an ambient that can neither move (in or out other ambients), nor be opened ($\overline{\mathtt{open}}$ co-capability). Such an ambient is not necessarily single-threaded. We have designed an extension of the GcPan [12] to handle immobile ambients as well.

We have also developed a prototype OCaml implementation of the (extended) GcPan, that is described at [1]. We plan to exploit it to further evaluate the improvements in terms of efficiency brought by our machine.

Related Work. Cardelli [4, 5] has produced the first implementation, called *Ambit*, of an ambient-like language; it is a single-machine implementation of the untyped Ambient calculus, written in Java. The algorithms are based on locks: all the ambients involved in a movement (three ambients for an In or Out movement, two for an Open) have to be locked for the movement to take place.

In [10], a JoCaml implementation of an abstract machine for Mobile Ambients, named AtJ, is presented. In Mobile Ambients, there are no co-capabilities, movements are triggered using only capabilities, and grave interferences arise. These differences enable considerable simplifications in abstract machines for SA (Pan, GcPan) and in their correctness proof — see [11] for a detailed comparison. Other differences are related to the distinction between logical and physical movements: in AtJ physical movements are triggered by the execution of `in` and `out` capabilities, whereas in GcPan only `open` induces physical movement.

[17] presents a distributed abstract machine for the Channel Ambients calculus, a variant of Boxed Ambients [3]. In Channel Ambients the `open` primitive – one of the most challenging primitives for the implementation of Ambient calculi – does not exist (`open` is dropped in favour of a form of inter-ambient communication). Although in the implementation [17] actual movement of code arises as a consequence of movement of ambients, the phenomenon is not reflected in the definition of the Channel Ambient calculus. Therefore, the main problems we have been focusing on do not appear in that setting.

In the Distributed Join calculus [8], migrating join definitions are replaced in the source space with a forwarder, to route local messages to the join definition at its new location. This phenomenon is reminiscent of the execution of Open reductions in our machine.

References

1. GcPan webpage. `http://perso.ens-lyon.fr/damien.pous/gcpan`.
2. P. Bidinger and J.-B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, pages 109–123. Springer Verlag, 2003.
3. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proc. TACS 2001*, LNCS 2215, pages 38–63. Springer Verlag, 2001.
4. L. Cardelli. Ambit, 1997. `http://www.luca.demon.co.uk/Ambit/Ambit.html`.
5. L. Cardelli. Mobile ambient synchronisation. Technical Report 1997-013, Digital SRC, 1997.
6. L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of FOSSACS'98*, volume 1378 of *LNCS*, pages 140–155. Springer Verlag, 1998.
7. F. Le Fessant, I. Piumarta, and M. Shapiro. An Implementation for Complete, Asynchronous, Distributed Garbage Collection. In *Proc. of PLDI'98*, ACM Sigplan Notices, pages 152–161, 1998.
8. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, 1998.

9. C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *Proc. of Advanced Functional Programming 2002*, volume 2638 of *LNCS*, pages 129–158. Springer Verlag, 2002.
10. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Proc. of IFIP TCS'00*, volume 1872 of *LNCS*, pages 348–364. Springer Verlag, 2000.
11. P. Giannini, D. Sangiorgi, and A. Valente. Safe Ambients: abstract machine and distributed implementation, 2004. submitted; an extended abstract appeared in Proc. ICALP'01, volume 2076 of LNCS, pages 408–420, Springer Verlag.
12. D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients. Technical Report 2004–63, LIP – ENS Lyon, 2004.
13. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proc. 27th POPL*. ACM Press, 2000.
14. F. Levi and D. Sangiorgi. Mobile Safe Ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003. ACM Press.
15. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. 19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.
16. R. De Nicola, G.L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.
17. A. Phillips, N. Yoshida, and S. Eisenbach. A Distributed Abstract Machine for Boxed Ambient Calculi. In *Proc. of ESOP'04*, LNCS, pages 155–170. Springer Verlag, 2004.
18. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.
19. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructure for Mobile Computation. In *Proc. of 28th POPL*, pages 116–127. ACM Press, 2001.