An efficient 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. Safe Ambients (SA) are a variant of the Ambient Calculus (AC) in which types can be used to avoid certain forms of interferences among processes called *grave interferences*.
An abstract machine, called GcPan, for a distributed implementation of typed SA is presented and studied. W.r.t. Pan, a previous abstract machine for SA, GcPan is more efficient mainly because it exploits well-known algorithms for distributed systems (e.g. algorithms based on reference counting and union-find). In particular these algorithms help improve the management of special agents (the *forwarders*), created upon code migration to transmit messages to the target location of the migration. (In Pan forwarders are never removed, and long – possibly distributed – chains of forwarders slow down the transmission of messages between agents.)
In addition to this, w.r.t. previous abstract machines for AC, GcPan is also simpler and more efficient because: the underlying calculus is typed Safe Ambients rather than the untyped AC and therefore does not present grave interferences; the logical structure of an ambient system and its physical distribution are separated.
The proof of correctness of GcPan, and a description of a distributed implementation of the abstract machine in OCaml are given. The correctness is obtained by establishing a weak bisimilarity between GcPan and Pan and then appealing to the correctness of Pan.

Introduction

Ambients-based calculi have been proposed as models for distributed and mobile computing based on the notion of *location*, called ambient. Terms in these 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-movement), an ambient to exit another ambient (Out-movement), a process to dissolve an ambient boundary thus obtaining access to its content (Open-movement).

A few distributed implementations of Ambient-like calculi have appeared [7, 8, 13]. The study of implementation 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.

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 [3, 4] locks are used to achieve a synchronisation among all ambients affected by a movement. 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 [7] the synchronisations are simulated by means of protocols based on asynchronous messages. The abstract machine Pan [8] has two main differences w.r.t. [7]. The first is that the machine executes typed Safe Ambients [10] (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` are acquisition of access rights, and `open` is exercise of them.

In Pan, the implementation of the `open` capability exploits *forwarders* – a standard technique in distributed systems – to retransmit messages coming from the inside of an ambient that has been opened. Forwarders 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 not 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 induces a loss of performances 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.

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 [8] to deduce correctness w.r.t. SA.

An original aspect of our proof w.r.t. that in [8] 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 result. 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. Moreover, because of the different management of forwarders, 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, since the two former abstract machines share a number of features.

Finally, we describe an implementation of our machine in OCaml. We have used the implementation to study the improvements brought by our modifications.

Outline of the paper. We start by describing the design principles of the GcPan in Section 1. We then give the formal definition of the machine in Section 2, and prove its correctness in Section 3. Section 4 is devoted to an extension of the GcPan to handle also immobile ambients. Section 5 gives the description of our implementation of the extended GcPan. Finally, we discuss related work and possible future work in Section 6.

1 The Machine: Design Principles

We briefly recall the main features of the Safe Ambients (SA) calculus [10], and the Pan abstract machine [8]. We then highlight the main problems of Pan in terms of efficiency, and present our modifications. We end with a discussion of some of the consequences brought by these modifications.

1.1 Safe Ambients

The Safe Ambients calculus is an extension of the Mobile Ambients calculus [5] in which a tighter control of movements is achieved though *co-capabilities.* The four main reduction rules are:

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

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

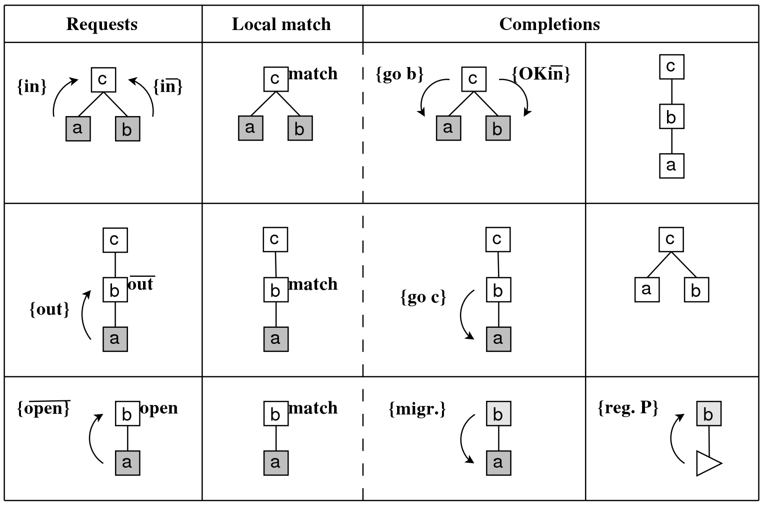

Fig. 1. Simulation of the SA reductions by the Pan.

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 [10]. 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 [7] are not necessary.

1.2 The Pan

The Pan [8] 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 name contained in the request message. The scenarios corresponding to the three kinds of movement are depicted in Figure 1, where white squares (resp. grey squares) represent locations (resp. locations in wait state), and arrows indicate travelling 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 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 Figure 1) is sent to the parent ambient (via a `register` 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

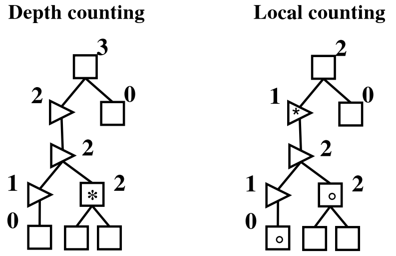

Fig. 2. Depth and local counting.

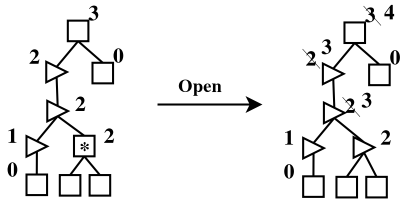

Fig. 3. Depth counting: problem with the Open rule.

a triangle in Figure 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 a Pan net is therefore a tree whose nodes are either a located ambient or a forwarder. 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 maintain the asynchrony in the exchange. For this, in the 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 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 (Figure 2):

Depth counting. The most natural idea is probably to decorate each located ambient with the number of 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 Figure 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 count 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 Figure 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.

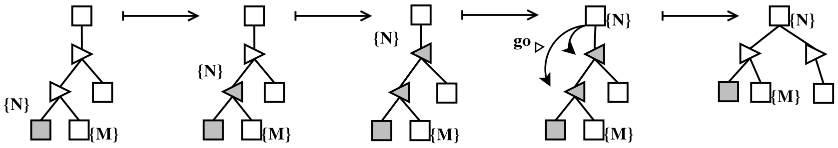

Fig. 4. Relocation of forwarders.

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 $\mathtt{go}_{\triangleright}$ completion message, which contains the name of the location to which the next request should be forwarded. Figure 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 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 [16].

Before explaining how counters are updated along the transmission of messages in the GcPan, we make some observations about the communication protocols that we have just described.

Remark 1 (Race situations). Having blocked forwarders leads to race situations: consider the scenario of Figure 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, $\{N\}$ can reach the latter in three steps (the $\mathtt{go}_{\triangleright}$ message plus two routing steps).

Remark 2 (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 Figure 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 Figure 1):

- In: c increments its counter when finding the match, b decrement its counter upon reception of the $\mathtt{OK}\overline{\mathtt{in}}$ completion.
- Open: counters do not need to be modified.
- 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 Figure 5. Adopting this protocol means introducing a new kind of message in the machine (message `DoOut` in Figure 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.

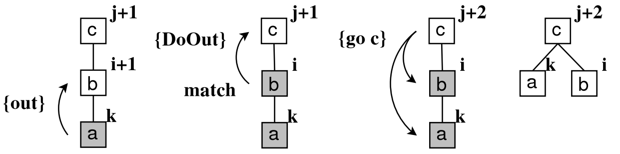

Fig. 5. Counters along an Out move: first approach.

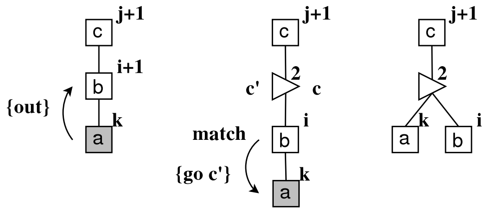

Fig. 6. Counters along an Out move: our approach.

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 Figure 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 decide to add a forwarder 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 ($h^i\colon n[P]_k$ is the ambient $n[P]$ running at h, whose parent is located at k), blocked or running forwarders ($h\triangleleft^i$ is a blocked forwarder at h, while $h \triangleright^i k$ 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 an abstraction. 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 2.3.

Definition 1. Structural congruence *is the smallest congruence* $\equiv$ *such that:*

1. *both forms of parallel composition,* $\mid$ *and* $\parallel$*, together with* $\mathbf{0}$*, form abelian monoids;*
2. $((\boldsymbol{\nu} p)A) \parallel B \equiv (\boldsymbol{\nu} p)(A \parallel B)$ *if* p *not free in* B*, and* $((\boldsymbol{\nu} a)P) \mid Q \equiv (\boldsymbol{\nu} a)(P \mid Q)$ *if* a *not free in* Q*;*
3. $(\boldsymbol{\nu} p)(\boldsymbol{\nu} q)A \equiv (\boldsymbol{\nu} q)(\boldsymbol{\nu} p)A$*, and* $(\boldsymbol{\nu} p)(\boldsymbol{\nu} q)P \equiv (\boldsymbol{\nu} q)(\boldsymbol{\nu} p)P$*;*
4. $(\boldsymbol{\nu} p)\mathbf{0} \equiv \mathbf{0}$
5. rec $X.P \equiv P\{\text{rec } X.P/X\}$

Structural congruence does not allow a name restriction to be extruded out of a located ambient in a transparent way: the net $h\colon n[(\boldsymbol{\nu} m)(\texttt{in } m)]_k$ is not equivalent to $(\boldsymbol{\nu} m)h\colon n[\texttt{in } m]_k$. Such a transformation is handled at the level of reduction since it is not harmless from an implementation point of view.

The GcPan (resp. Pan) encoding of a SA process P is written $[\![P]\!]_{gc}$ (resp. $[\![P]\!]$). $[\![P]\!]$ is defined in [8], and $[\![P]\!]_{gc}$ is defined as follows (note that the syntax of SA is included in the syntax of GcPan):

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

Networks:

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

$$\begin{array}{rll} \mathit{req} := & \texttt{in}\ n, h & \text{(agent at } h \text{ wants to enter } n\text{)} \\ \mid & \overline{\texttt{in}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts someone in)} \\ \mid & \texttt{out}\ n, h & \text{(agent at } h \text{ wants to leave } n\text{)} \\ \mid & \overline{\texttt{open}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts to be opened)} \end{array}$$

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

Processes:

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

Table 1. GcPan Syntax

Definition 2 (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

The following are the rules defining the operational semantics of GcPan nets. We explain them below.

Emission of request messages

$$\begin{array}{lrcl} \text{[Req-In]} & \texttt{in}\ m.P & \xrightarrow[h:-]{k} & \mathrm{wait}.P \gg k\{\texttt{in}\ m, h\} \\ \text{[Req-Coin]} & \overline{\texttt{in}}\ n.P & \xrightarrow[h:n]{k} & \mathrm{wait}.P \gg k\{\overline{\texttt{in}}\ n, h\} \\ \text{[Req-Out]} & \texttt{out}\ m.P & \xrightarrow[h:-]{k} & \mathrm{wait}.P \gg k\{\texttt{out}\ m, h\} \\ \text{[Req-Coopen]} & \overline{\texttt{open}}\ n.P & \xrightarrow[h:n]{k} & \mathrm{wait}.P \gg k\{\overline{\texttt{open}}\ n, h\} \end{array}$$

Transmission of request messages

$$\begin{array}{lrcl} \text{[Fw-Send]} & h \rhd^{i+1} k \parallel h\{\mathit{req}/E\} & \longmapsto & h \lhd^i \parallel k\{\mathit{req}/h{::}E\} \\ \text{[Fw-SendGC]} & h \rhd^1 k \parallel h\{\mathit{req}/E\} & \longmapsto & k\{\mathit{req}/E\} \\ \text{[Fw-Reloc]} & h \lhd^i \parallel h\{\texttt{go}_{\rhd}\ k\} & \longmapsto & h \rhd^i k \end{array}$$

[Loc-Rcv] $h^{i+1}\colon n[P]_k \parallel h\{req/E\} \ \longmapsto\ h^{i+\#E}\colon n[P \mid \{req\}]_k \parallel E\{\mathtt{go}_{\triangleright}\ h\}$

Local reductions

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

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

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

[Local-Open] $\mathtt{open}\ n.P \mid \{\overline{\mathtt{open}}\ n, h\} \ \xrightarrow[h':-]{-}\ \text{wait}.P \ \gg^1 h\{\mathtt{migrate}\ h'\}$

Inference rules

[Proc-Agent] $$\frac{P \xrightarrow[h:n]{k} P' \gg^s M \qquad Q \text{ does not have unguarded ambients}}{h^i\colon n[P \mid Q]_k \ \longmapsto\ h^{i+s}\colon n[P' \mid Q]_k \parallel M}$$

[Proc-Agent'] $$\frac{P \xrightarrow[h:n]{k} P' \ggg_{k'} M \qquad k' \notin FL(P|Q) \qquad Q \text{ does not have unguarded ambients}}{h^i\colon n[P \mid Q]_k \ \longmapsto\ (\boldsymbol{\nu}k)'(k' \triangleright^2 k \parallel h^i\colon n[P' \mid Q]_{k'} \parallel M)}$$

[Par-Agent] $$\frac{A \ \longmapsto\ A'}{A \parallel B \ \longmapsto\ A' \parallel B}$$

[Res-Agent] $$\frac{A \ \longmapsto\ A'}{(\boldsymbol{\nu}p)A \ \longmapsto\ (\boldsymbol{\nu}p)A'}$$

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

Consumption of completion messages

[Compl-Parent] $h\{\mathtt{go}\ k\} \parallel h^i\colon n[P \mid \text{wait}.Q]_{-} \ \longmapsto\ h^i\colon n[P \mid Q]_k$

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

[Compl-Migr] $h\{\mathtt{migrate}\ k\} \parallel h^{i+1}\colon n[P \mid \text{wait}.Q]_{-} \ \longmapsto\ h \triangleright^{i+1} k \parallel k\{\mathtt{register}^0\ (P \mid Q)\}$

[Compl-MigrGC] $h\{\mathtt{migrate}\ k\} \parallel h^0\colon n[P \mid \text{wait}.Q]_{-} \ \longmapsto\ k\{\mathtt{register}^1\ (P \mid Q)\}$

[Compl-Reg] $h\{\mathtt{register}^s\ R\} \parallel h^{i+s}\colon n[P \mid \text{wait}.Q]_k \ \longmapsto\ h^i\colon n[P \mid Q \mid R]_k$

Creation

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

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

2.3 Comments on the Reduction Rules

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, presented below). $\gg$ is an abbreviation of $\gg^0$. When n or 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.

- 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 of the creation of the forwarder outside the 'active location').
- Some rather standard *inference rules* are used to transform a local reduction into a transition of the whole GCPAN net.

  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 interacting agents resume computation when they are informed that a movement has occurred.
- Finally, when a process resumes computation after consuming a prefix, it 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.*

Counting: the notion of *well-formed net*, that will be defined in the next section, expresses the laws that are satisfied by counters. 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 $k^i\colon n[P]_h$);
- 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 ($k\{\texttt{go}\ h\}$, $k\{\texttt{go}_{\rhd} h\}$, ...).

This is how the principles that govern our accounting are preserved along execution:

IN: In an IN move, the parent ambient loses a child, and so it seems that its counter should be decremented. This is not so: in our setting, when we use rule LOCAL-IN, the two interacting child locations (h and k) are in wait state, and the parent (h') does no longer take them into account (ambients in wait state are pending, and have no parent location). Therefore, the parent actually has to *increment* its counter: the role of the completion message $k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$ is to bring k back under h'.

Similarly, h, that will receive h' as a new child (message $h'\{\texttt{go}\ h\}$ and rule COMPL-PARENT), also increments its counter, upon reception of message $\texttt{OK}\overline{\texttt{in}}$ (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 so 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 `migrate` 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 `register` message (rule COMPL-REG).

Also note that sub-ambients belonging to local processes are only taken into account once they become a located ambient (rule NEW-LOCAMB, that accordingly increments the counter of the parent location).

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*:

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

First, the message gets transmitted by forwarder h_1: h_1 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 (rule Fw-SendGC), 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 (i.e., the number of blocked forwarders that will eventually become direct sons of k) *minus one* (as one of these forwarders was already under k). Finally, the blocked forwarders (h_1 and h_3) 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 [8], there are important differences. First of all, we compare two abstract machines, rather than a machine and a calculus as in [8]. 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 [8] 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 [8], 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 result 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 (see Subsection 1.3). 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 a process P expands Q if Q is 'better than P' *at every step* (a formal proof of the non-expansion result is given in [9]).

The notion of equivalence we adopt is barbed bisimulation [12, 15], that we denote $\approx$. It can be used to compare states of the same transition system, or even of different transition systems. The only requirement is that the transition systems should be equipped with a reduction relation and observability predicates.

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 [8]).

Our main results are the following:

Theorem 1 (Equivalence between Pan and GcPan). *For any well-typed SA process P, we have $[\![P]\!] \approx [\![P]\!]_{gc}$.*

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

Proof: *[8] establishes that for any P, $[\![P]\!] \approx P$, which is enough to conclude using Theorem 1.* ◇

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

Proof of Theorem 1

We now sketch the main steps of the proof of Theorem 1.

Definition 3, presented below, states the properties that a GcPan net should satisfy in order to correspond to a configuration that makes sense from the point of view of the execution of a single threaded Safe Ambient process. We first need to introduce some terminology.

Terminology (Naming and relating locations) In the sequel, we shall use the following notions:

- In the net $h^i\colon n[(\boldsymbol{\nu}\overline{p})(\mathrm{wait}.Q|R)]_k \parallel A$, the ambient located at h is in *frozen state* if $A \equiv A' \parallel C$ with $C \in \{k\{\texttt{migrate}\ h\},\ h\{\texttt{register}^s\ P\}\}$; it is in *wait state* otherwise.
- The *name* of a located ambient $h^i\colon n[P]_k$ is n, its *home location* is h, the *parent location* of h (denoted by $pl(h)$) is k if the located ambient is not in wait state, and is undefined otherwise.
- The *home location* of a forwarder $h \rhd^i k$ is h, its *parent location* is k.
- The *home location* of a blocked forwarder $h\lhd^i$ is h, its *parent location* is k if there is a completion message $h\{\texttt{go}_{\rhd}\ k\}$, and is undefined otherwise.
- The *target location* of a message $h\{M\}$ is h, its *source location* is the location decorating the message.
- Given a net A, we can extract from A a relation over locations by saying that two locations are related if they are the home and parent location of the same agent in A. We call this the *dependency relation of A*. When the graph of this relation is a forest, we define a partial order over locations: $h <_A k$ iff k is an ancestor of h according to the dependency relation.

Definition 3 (Well-formedness). *A* GcPan *net A is* well-formed *if the following properties hold:*

Blocked forwarders

1. *For each blocked forwarder $h\lhd^i$, either there is a unique message $k\{M/E\}$, for some k, M, E, such that h belongs to E, or there is a unique message of the form $h\{\texttt{go}_{\rhd}\ k\}$.*
2. *For each request $h\{req/E\}$ and location e belonging to E, the agent located at e is a blocked forwarder.*

Dependency relation between locations

3. *The graph of the dependency relation defined by A forms a forest, whose roots are* `rootparent` *and the locations whose parent location is undefined.*
4. *For each location h, there is at most one agent located at h.*

Messages

5. *The source (resp. target) of every request (resp. completion) message is an ambient in wait state.*
6. *For each ambient location h in wait state, there is exactly one request message whose source is h or completion message whose target is h.*
7. *The source of every* `migrate` *message is an ambient in frozen state.*

Single-Threadedness

8. *The process of every located ambient is single threaded.*
9. *A* wait *prefix can only appear as the unique unguarded prefix of the local process of a located ambient.*

Counters

10. *Let μ be the function from localities to integers defined by:*

$$\begin{aligned}\mu(h) = {} & \#\{k^i\colon n[P]_h \ /\ P \text{ is not in a wait state}\} + \#\{k \rhd^i h\} + \#\{h\{req/E\}\} \\ & + \#\{k\{\texttt{go}\ h\}\} + \#\{k\{\texttt{go}_{\rhd} h\}\} + \#\{k\{\texttt{OK}\overline{\texttt{in}}\ h\}\} - \#\{h\{\texttt{OK}\overline{\texttt{in}}\ k\}\} \\ & + \#\{h\{\texttt{register}^1\ P\}\} + \#\{k\{\texttt{migrate}\ h\}\}\end{aligned}$$

where $\#\{p\}$ is the number of sub-terms of A matching the pattern p.
Then for each agent with counter i ($h^i\colon n[P]_k$, $h \rhd^i k$ or $h\lhd^i$), $i = \mu(h)$

Proposition 1. *If P is a well-typed SA process, then $[\![P]\!]_{gc}$ is well-formed. If A is a well-formed* GcPan *net and $A \longmapsto A'$ then A' is well-formed.*

This result allows us to assume that all nets we consider in the sequel are well-formed.

Since forwarder trees have very different behaviours in the two machines, we are compelled to abstract over them. Well-formedness allows us to guess, for each message in a net, what its 'true' destination is (in the current state of the net), which is how, intuitively, we define the bisimulation candidate (Definition 5). We first need to introduce some auxiliary notions.

To each location h of a GcPan net C, we associate its *real location* $\widetilde{h}$, that corresponds to the ambient location where messages whose target is h are routed to (in particular, if h is the location of an ambient, $\widetilde{h} = h$). We moreover define $\prec_C$ as the relation between ambient locations such that $h \prec_C \widetilde{k}$ whenever $k = pl(h)$ or k is either the target of the request emitted from h or the source of the completion sent to h. Intuitively, $\prec_C$ expresses the hierarchy of ambients in the SA process being executed.

Definition 4 (Message sets). *For each ambient location* h: $n[P]_k$ *of a* GcPan *net* B*, we let* $p_B(h)$ *stand for the process obtained by removing all request messages from* P*, and we further define:*

$$\begin{aligned} tr_B(h) &= \{req \text{ s.t. } B \equiv B_0 \parallel h'\{req/E\} \text{ with } h' = \widetilde{h}\} \\ ar_B(h) &= \{req \text{ s.t. } P \equiv P_0 \mid \{req\}\} \end{aligned} \qquad r_B(h) = ar_B(h) \cup tr_B(h)$$

$$\begin{aligned} c_B(h) = &\{\mathtt{OK}\overline{\mathtt{in}}\ \widetilde{k} \text{ s.t. } B \equiv B_0 \parallel h\{\mathtt{OK}\overline{\mathtt{in}}\ k\}\} \cup \{\mathtt{migrate}\ \widetilde{k} \text{ s.t. } B \equiv B_0 \parallel h\{\mathtt{migrate}\ k\}\} \\ &\cup \{\mathtt{register}\ Q \text{ s.t. } B \equiv B_0 \parallel h\{\mathtt{register}^s\ Q\}\} \cup \{\mathtt{go}\ \widetilde{k} \text{ s.t. } B \equiv B_0 \parallel h\{\mathtt{go}\ k\}\} \end{aligned}$$

Intuitively, $tr_B(h)$ is the set of requests that will eventually reach h, $ar_B(h)$ is the set of requests that have already reached h, and $c_B(h)$ is the set of completions whose target is h. We also introduce the analogue of these definitions for the Pan (see [9] for a detailed presentation).

Definition 5 (Bisimulation candidate). *Let* $\mathcal{R}$ *be the relation between* Pan *and* GcPan *nets defined by* $A \mathcal{R} B$ *iff:*

1. *A and B are well-formed, according to the respective definitions for the* Pan *and the* GcPan*;*
2. $\prec_A \ = \ \prec_B$*;*
3. *for each ambient location h, we have $r_A(h) = r_B(h)$, $c_A(h) = c_B(h)$, and $p_A(h) = p_B(h)$.*

Relation $\mathcal{R}$ satisfies the following properties:

Lemma 1 (GcPan communications). *Let* $\longmapsto_c$ *be the reduction relation obtained by using only the transmission and inference rules. Suppose* $A \mathcal{R} B$ *and* $B \longmapsto_c B'$*, then* $A \mathcal{R} B'$*.*

Lemma 2 (GcPan requests). *Suppose* $A \mathcal{R} B$ *and* $A \equiv h$: $n[P|\{M\}]_k \parallel A_0$*. Then there is a net* B' *such that* $B \Longmapsto B'$, $A \mathcal{R} B'$*, and* $B' \equiv h^i$: $n[p_B(h) \mid ar_B(h) \mid \{M\} \mid M_a]_{k'} \parallel B_0$*, for some set of messages* $M_a \subset tr_B(h)$*.*

In the last lemma, $\{M\} \,|\, M_a$ stands for the parallel composition of $\{M\}$ and of messages in M_a.

We also prove similar lemmas for Pan nets. The set of messages that is called M_a in Lemma 2 is specific to the GcPan, and does not appear in the Pan counterpart of these results. Indeed, elements of M_a are the messages that have to be brought to their destination in order to unblock forwarders when routing message $\{M\}$. An example of this appears in Figure 4: after the first transition, if we want to move $\{M\}$ upwards, we first need to route $\{N\}$ (and reach the state described in the last configuration). By contrast, no such race situation appears in the Pan.

To prove that $\mathcal{R}$ is a weak barbed bisimulation, we use Lemma 2 (together with its Pan counterpart) to show that related states exhibit the same observables. Lemma 1 (and its Pan version) is needed to play the bisimulation game, in order to route messages when necessary (a full proof is available in [9]). Theorem 1 then follows by remarking that for any P, $[\![P]\!] \mathcal{R} [\![P]\!]_{gc}$.

4 Immobile Ambients

Besides ST ambients, the other main type for SA processes [10] is that of *immobile* ambients (IM). An immobile ambient is an ambient that can neither move (in our out other ambients), nor be opened ($\overline{\mathtt{open}}$ co-capability). Such an ambient is not necessarily single-threaded, e.g. the following is an IM ambient: $N \triangleq n[!\overline{\texttt{in}}\ n \mid !\overline{\texttt{out}}\ n \mid !\texttt{open}\ m]$. We explain how to extend GcPan to handle immobile ambients.

Using the GcPan as it is to execute IM ambients is not sound. Typically, an immobile ambient can enter a wait state by emitting a $\overline{\texttt{in}}$ request and then simultaneously: (*i*) send other $\overline{\texttt{in}}$ requests, and (*ii*) let ambients out (by exercising $\overline{\texttt{out}}$ co-capabilities). If the parent location of this ambient is a forwarder, we run into non-legal configurations, since an unpredictable number of messages can be transmitted 'upwards'.

Regarding (*i*), we remark that we can send $\overline{\texttt{in}}$ requests in a sequential way, and wait for a $\texttt{OK}\overline{\texttt{in}}$ completion message between each emission. Similarly, we can let Out moves happen in a sequential fashion (case (*ii*)). However, we cannot interleave smoothly these two behaviours: there is no reason for an Out to be blocked until the arrival of an $\texttt{OK}\overline{\texttt{in}}$ message, in particular since the corresponding $\overline{\texttt{in}}$ request could well remain unmatched forever. Our solution is to introduce *persistent forwarders*, denoted $\triangleright^{\infty}$. As the notation suggests, a persistent forwarder has an infinite lifetime (we set $\infty + 1 = \infty$ for rule Fw-Send in the case of persistent forwarders).

The reduction rules of Subsection 2.2 are adapted as described below. All rules but the last one are added to the system to address the execution of IM ambients, the corresponding already existing rule (without prime) being kept and decorated with the side condition specifying that the ambient which is called n should be ST. Rule Proc-Agent is instead *replaced* by the rule shown below.

$$[\textsc{Local-Out'}] \qquad \{\texttt{out}\ n, h\} \mid \overline{\texttt{out}}\ n.P \xrightarrow[h':n]{k} P \gg h\{\texttt{go}\ k\} \qquad n \text{ is IM}$$

$$[\textsc{Compl-Coin'}] \qquad h\{\texttt{OK}\overline{\texttt{in}}\ -\} \parallel h^i\colon n[P \mid \text{wait}.Q]_{k'} \longmapsto h^{i+1}\colon n[P \mid Q]_{k'} \qquad n \text{ is IM}$$

$$[\textsc{Local-In'}] \qquad \{\texttt{in}\ n, h\} \mid \{\overline{\texttt{in}}\ n, k\} \xrightarrow[h':-]{-} \mathbf{0} \gg^0 h\{\texttt{go}\ k\} \parallel k\{\texttt{OK}\overline{\texttt{in}}\ h'\} \qquad n \text{ is IM}$$

$$[\textsc{New-Locamb'}] \quad h^i\colon m[n[P] \mid Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu} k, k')(k^0\colon n[P]_{k'} \parallel k' \triangleright^{\infty} h) \qquad \begin{array}{r} n \text{ is IM} \\ k, k' \notin FL(Q) \end{array}$$

$$[\textsc{Proc-Agent}] \qquad \frac{P \xrightarrow[h:n]{k} P' \gg^s M \qquad \begin{array}{l} P' \mid Q \text{ has at most one wait prefix} \\ Q \text{ does not have unguarded ambients} \end{array}}{h^i\colon n[P \mid Q]_k \longmapsto h^{i+s}\colon n[P' \mid Q]_k \parallel M}$$

Every IM ambient is associated upon creation to a persistent forwarder (cf. rule New-Locamb' — we could have avoided introducing persistent forwarders by adopting a special notation for IM ambients, but we preferred this approach for the sake of uniformity in the presentation). Ambients exiting an IM ambient are located under the associated persistent forwarder (compare rule Local-Out' with Local-Out). Upon reception of a $\texttt{OK}\overline{\texttt{in}}$ completion message, an IM ambient does not relocate itself; the associated persistent forwarder, however, relocates itself when necessary, just as any forwarder. This explains the IM versions of the Compl-Coin and Local-In rules: in the former rule, ambient n remains at location k', and in the latter, we generate a null increment (relation $\gg^0$) because ambient n will not be relocated. The additional premise in rule Proc-Agent ensures that $\overline{\texttt{in}}$ requests are sent sequentially.

Remark 3 (No counters for immobile ambients). An IM ambient is by definition persistent, i.e., it will never become a forwarder, and hence its counter is useless. As a first consequence, we can get rid of counters for IM ambients: this is what we actually do in the implementation. (We do not present the corresponding refined syntax and reduction rules for the sake of brevity.) A second consequence is that we can spawn IM ambients inside IM ambients without creating the persistent forwarder associated to the inner ambients: the side condition of rule New-Locamb could thus be relaxed to "n is ST or m is IM".

The proof of Theorem 1 can be adapted to systems that include also IM ambients:

Theorem 2. *For any well typed (using ST and IM types) SA process P, $[\![P]\!] \approx [\![P]\!]_{gc}$.*

5 An Implementation of the GcPan

We have developed a distributed implementation of the GcPan, written in OCaml (the sources are available for download at [1]). Our implementation supports both ST and IM ambients. The execution of GcPan locations can be spawned on different sites, in an orthogonal way w.r.t. the logical hierarchy of locations. A first layer of our system consists in a general purpose library for channel communication, that transparently handles intra-site communication, via shared memory (when two GcPan locations are actually running on the same site), and inter-site communication, via TCP. On top of this layer, the code is organised in such a way that the rules of Section 2 can be written almost literally in a file. Not only does this give confidence about the fact that the implementation actually executes the machine we described, but this also allows us to obtain an implementation of the Pan 'for free', by writing a different file with the Pan rules.

We thus obtain a system in which we can experiment with GcPan programs and compare their execution with the corresponding Pan version. Some tracing facilities allow us to learn about the performance of the GcPan. Up to now, we have been experimenting with a few simple examples, and the results show that forwarders are indeed collected in an efficient way in the GcPan (see [1] for some commented examples). These observations have to be completed by further experiments, involving larger examples. So far, our impression is that on systems running for more than a dozen reductions, having shorter forwarder chains as in GcPan rapidly leads to gains in terms of number of messages exchanged, which compensates for the additional cost incurred by the union-find algorithm.

6 Conclusions

6.1 Related Work

The comparison with the PAN [8] has been discussed in detail in sections 1 (design) and 3 (proof).

Cardelli [3, 4] 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 [7], 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 [8] for a detailed comparison. Other differences are due to the fact that in AtJ there is no separation between the logical and physical distribution of an ambient system: in AtJ the `in` and `out` capabilities produce physical movements, whereas in Pan and GcPan only `open` gives physical movement.

[13] presents a distributed abstract machine for the Channel Ambients calculus, a variant of Boxed Ambients [2]. In Channel Ambients the `open` primitive, which is the most challenging primitive in the implementation of Ambient calculi, does not exist (`open` is dropped in favour of a form of inter-ambient communication). Further, there is only *logical movement* in the machine of [13], in the sense that there is no migration of code among sites. Hence the main problems that we have encountered in the design and implementation of our abstract machine do not appear in Channel Ambients.

In the Distributed Join calculus [6], migrating join definitions are replaced in the source space with a forwarder, to route local messages to the definition at its new location. This phenomenon is reminiscent of the execution of Open reductions in our machine.

The framework we have presented relies on the absence of grave interferences to execute processes. In our case, this is guaranteed by SA type system for ST and IM ambients. We believe that our machine could be adapted to handle other proposals for Ambient-based calculi, as long as grave interferences are ruled out. This should be possible for Controlled Ambients [17], a model in which a three-way synchronisation involving an additional co-capability is used to trigger In and Out operations. On the contrary, for some other SA-like models, getting rid of grave interferences seems more problematic. This is the case in particular for SAP [11] – where the parent ambient (b) is not involved in an Out move – or the Push and Pull Ambient Calculus [14] – where all moves are objective.

6.2 Future work

Reasoning about code mobility. Code mobility arises as a consequence of an Open operation in our setting. We believe that the way we analyse mobility goes beyond Ambient-like calculi. Indeed, to implement this kind of mobility in real systems, one needs to be able to: (i) passivate the code and resume computation after migration, and (*ii*) preserve the coherence of the execution environment at the source location. In the GcPan, (*i*) is made possible by the single-threadedness property, and (*ii*) is achieved via forwarders. Commonly used techniques in distributed programming, such as forwarders and forwarder chains contractions, are rarely supported by formal studies. In the present work, we have given a formal account and a proof for some of these techniques, as they appear in GcPan. We plan to extend our analysis to other implementations of distributed systems and other aspects of such implementations.

Types. We have preliminary results about a type-inference procedure to check that a given ambient is typable with either an ST type or an IM type. We would like to study whether an enriched type system would allow us to improve further on the management of forwarders in the machine.

Correctness result. It would be interesting to see whether it is possible to have a more refined statement of the relationship between Pan and GcPan. Such statement should formally express the improvement in efficiency of GcPan. In process calculi results of this kind are usually given by means of the expansion preorder (as opposed to, say, bisimilarity). As already discussed, in our case we cannot use expansion because the union-find algorithm requires some initial additional administrative work. A more relaxed form of expansion would be needed.

Correctness proof. Despite our efforts, we have not been able to simplify the correctness proof for the machine. Our proof is lengthy, which has to be the case, in some sense, because we reason about a complex system that we describe at a fine-grained level. We believe that several simplifications and more modularity could be introduced in the proof if we had the more relaxed expansion preorder discussed above. This would seem important also in order to apply our approach to more sophisticated implementations of distributed systems.

References

1. GcPan webpage. `http://perso.ens-lyon.fr/damien.pous/gcpan`.
2. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proc. TACS 2001*, LNCS 2215, pages 38–63. Springer Verlag, 2001.
3. L. Cardelli. Ambit, 1997. `http://www.luca.demon.co.uk/Ambit/Ambit.html`.
4. L. Cardelli. Mobile ambient synchronisation. Technical Report 1997-013, Digital SRC, 1997.
5. L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of FOSSACS'98*, volume 1378 of *LNCS*, pages 140–155. Springer Verlag, 1998.
6. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, 1998.
7. 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.
8. 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.
9. D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients. long version, available from [1].
10. F. Levi and D. Sangiorgi. Mobile Safe Ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003. ACM Press.
11. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients (extended abstract). In *Proc. of POPL '02*, pages 71–80, 2002.
12. 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.
13. 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.
14. I. Phillips and M. G. Vigliotti. On Reduction Semantics for the Push and Pull Ambient Calculus. In *Proc. of IFIP TCS'02*, pages 550–562. Kluwer, 2002.
15. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.
16. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.
17. D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*, pages 288–303. Springer Verlag, 2002.