An efficient abstract machine for Safe Ambients

Daniel Hirschkoff, Damien Pous, Davide Sangiorgi

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

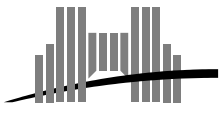
An efficient abstract machine for Safe Ambients

Daniel Hirschkoff, Damien Pous, Davide Sangiorgi

Décembre 2004

Research Report N° 2004-63

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



An efficient abstract machine for Safe Ambients

Daniel Hirschkoff, Damien Pous,
Davide Sangiorgi

Décembre 2004

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. Our machine improves over previous proposals for executing AC, or variants of it, mainly through a better management of special agents (the *forwarders*), created upon code migration to transmit messages to the target location of the migration. Well-known methods (such as reference counting and union-find) are applied in order to garbage collect forwarders, thus avoiding long – possibly distributed – chains of forwarders, as well as avoiding useless persistent forwarders.

The proof of correctness of GcPan, and a description of a distributed implementation of the abstract machine in OCaml are given.

Correctness is established by proving a weak bisimilarity result between GcPan and a previous abstract machine for SA, and then appealing to the correctness of the latter machine. This is simpler than comparing GcPan directly with SA, but it involves reasoning modulo 'administrative reduction steps' in both machines and therefore standard techniques for simplifying proofs of weak bisimilarity results are not applicable.

More broadly, this study is a contribution towards understanding issues of correctness and optimisations in implementations of distributed languages encompassing mobility.

Keywords: Mobile Ambients, abstract machine, forwarders, garbage collection, bisimulation

Résumé

Le calcul des Safe Ambients (SA) est une variable du calcul des Mobile Ambients qui permet, par le truchement d'un système de types, d'éviter certaines formes d'interférences appelées *interférences graves*.

Nous définissons ici une machine abstraite, appelée GcPan, pour l'exécution de processus SA typés. GcPan diffère de Pan, une machine abstraite introduite précédemment pour SA, par le fait qu'elle utilise des techniques classiques de programmation distribuée (comptage de références, union-find) afin d'améliorer les performances. En particulier, ces modifications permettent une meilleure gestion des agents de retransmission qui sont créés par la machine au moment où l'on fait migrer du code. Ces agents, dont le rôle est de faire suivre les messages vers la nouvelle destination d'un agent qui a migré, ne sont jamais supprimés dans la machine Pan. Dans le contexte d'une implantation distribuée, cela peut engendrer des chaînes de retransmetteurs arbitrairement longues, et passant par plusieurs sites physiques.

Au-delà de la comparaison avec la Pan, la GcPan se démarque des autres machines abstraites ayant été proposées pour l'exécution de calculs d'Ambients selon des critères de simplicité et d'efficacité. Le fait d'adopter le calcul des Safe Ambients plutôt que le calcul originel des Mobile Ambients, notamment, nous permet d'éviter d'avoir à gérer des interférences graves. Par ailleurs, nous découplons la structuration logique déterminée par un processus et la description de la répartition des sous-processus au niveau physique.

Nous présentons la preuve de correction de la machine proposée, et nous décrivons une implantation répartie que nous avons développée. La propriété de correction s'énonce comme un résultat de bisimilarité faible entre la GcPan et la Pan (qui, à son tour, est correcte vis-à-vis du calcul des Safe Ambients).

Mots-clés: ambients mobiles, machine abstraite, transmetteurs, glanage de cellules, bisimulation

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 [8], Nomadic Pict [23], Kells [1], Ambients [5], Klaim [16].

In this paper we study issues of correctness and optimisations in implementations of such languages. Although our technical work focuses on of 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-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 [9, 11, 17]. 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 [9] 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 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 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 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.

The basis of the algorithms we use (e.g., Tarjan's union-find algorithm [21]) 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 proof 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.

Not being able to apply standard proof techiques for bisimilarity, we have striven to make the proof highly-structured and readable, for instance isolating a number of basic results about forwarders and administrative reductions. Due to lack of space, only a few of them are reported in the paper.

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

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. [6, 8]). 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 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 a special kind of location named *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 [14], and the Pan abstract machine [11]. 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:

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

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*. For example, in the following overlapping redexes, the choice changes completely the behaviour of one of the ambients involved (here n).

$$\begin{array}{lcll} \mathtt{open}\ n \mid n[\overline{\mathtt{open}}\ n \mid \mathtt{in}\ m] \mid m[\overline{\mathtt{in}}\ m] & \longmapsto & \mathtt{in}\ m \mid m[\overline{\mathtt{in}}\ m] & \not\longmapsto \\ \mathtt{open}\ n \mid n[\overline{\mathtt{open}}\ n \mid \mathtt{in}\ m] \mid m[\overline{\mathtt{in}}\ m] & \longmapsto & \mathtt{open}\ n \mid m[n[\overline{\mathtt{open}}\ n]] & \not\longmapsto \end{array}$$

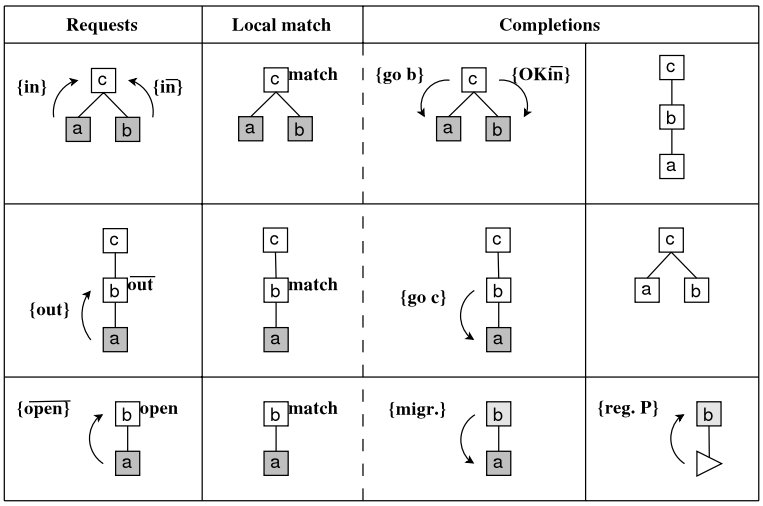


Figure 1: Simulation of the SA reductions by the Pan.

In presence of co-capabilities, one can define a type system for *single-threadedness*, that avoids such interferences [14]: 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 example, n is not single-threaded since it contains two threads, willing to execute $\overline{\texttt{open}}\ n$ and $\texttt{in}\ m$ respectively.

Of course, this restriction does neither remove concurrency (several ambients, each of which executes its own thread, may run concurrently), nor does it preclude non-determinism (consider for instance $a[\texttt{in}\ b] \mid b[\overline{\texttt{in}}\ b.P] \mid b[\overline{\texttt{in}}\ b.Q]$).

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 required to support grave interferences in a distributed setting [9] are not necessary.

In the sequel, when mentioning *well-typed* processes, this will be a reference to the type system of [14].

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: $a[]_{\texttt{root}} \parallel h_2$: $b[P]_{h_1} \parallel h_3$: $c[]_{h_2} \parallel h_4$: $d[Q]_{h_1}$. 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 ambients interaction: 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).

An ambient interaction in the Pan is basically decomposed into three steps: an ambient that wants to move first sends a *request message* to its parent and enters *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 on 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 from the design of Pan, in which the relation between parent and child

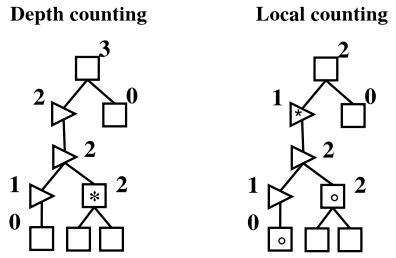


Figure 2: Depth and local counting.

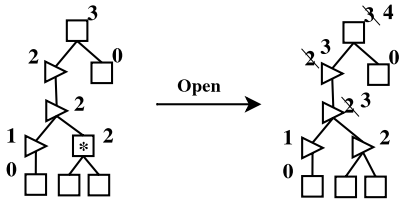


Figure 3: Depth counting: problem with the Open rule.

'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 on 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 a triangle on 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 how our choices affect the design of the Pan.

1.3.1 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.

A first remark is that we cannot predict the number of children that a created forwarder will have. Consider indeed the SA process

$$r[\texttt{open}\ a \mid a[\overline{\texttt{open}}\ a \mid b[\text{rec } X.(c[\texttt{out}\ b] \mid \overline{\texttt{out}}\ b.X)]]]\,.$$

(construction rec X binds the process variable X in the recursively defined process above). There is no way to predict the number of children that ambient a will have at the moment where it will get opened.

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 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 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.

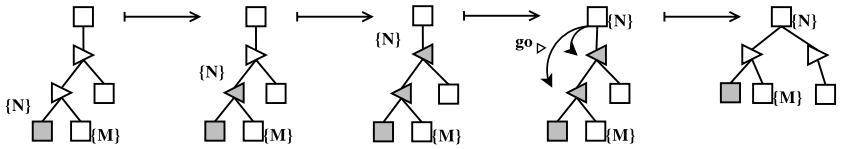


Figure 4: Relocation of forwarders.

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 value of 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 has no influence on the counting.

1.3.2 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 upwards, 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. 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, $\mathsf{go}_{\triangleright}$ 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 reminiscent of Tarjan's union-find algorithm [21].

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 $\mathsf{go}_{\triangleright}$ 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 $\mathsf{go}_{\triangleright}$ 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 ($\mathsf{go}_{\triangleright}$) 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 practice, 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.*

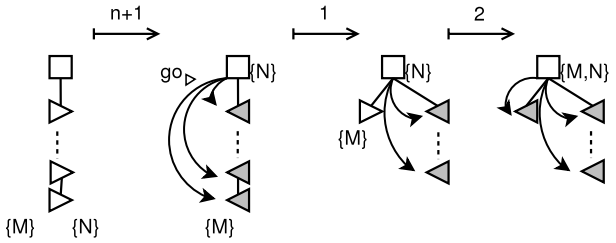


Figure 5: A race situation.

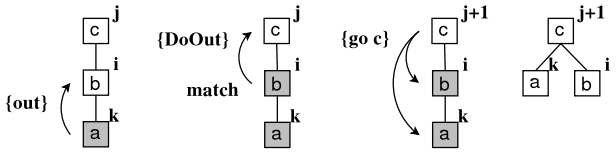


Figure 6: Counters along an Out move: first approach.

1.3.3 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: The overall result of the transition will be that c decrements its counter, and b increments its counter upon reception of the $\overline{\texttt{OKin}}$ completion.

Open: counters do not need to be modified. We introduce here an optimisation, when b's counter is equal to zero (i.e., b has no child at all): in this case, we avoid creating a useless forwarder (since there is no child to listen to).

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 on Figure 6. Adopting this protocol means introducing a new kind of message in the machine (message `DoOut` on Figure 6, from parent to grandparent), and having two agents in wait state (the child and the parent) while the control is at the grandparent's 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 on Figure 7: at b, we create a new forwarder that collects the parent (b) and the child (a) under a unique agent, so that the grandparent's 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's site, so that the extra communication between the parent and the forwarder will be local.

Remark 3 (The cost of message transmission) *After the opening of an ambient, the content migrates to the parent location in a* `register` *message, that can thus be arbitrarily large. In our machine, the parent location is in wait state until transmission of this message (this is mandatory in order to preserve a correct meaning to counters). The migrating process is informed of its target destination in the* `migrate` *message that precedes the physical movement. As a consequence,* `register` *messages are*

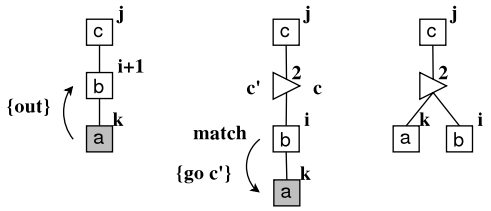


Figure 7: Counters along an Out move: our approach.

transmitted without being routed by forwarders, as is the case in the Pan *(which can be the cause of grave inefficiencies).*

On the other hand, in the Pan*, all messages have a fixed, small size, except for* `register` *messages that carry a migrating process. As hinted above (see Remark 2), the relocation method we use in the* GcPan *makes it necessary to decorate request messages with the list of forwarder locations they went trough. Hence, the size of such messages is a priori unbounded. Experiments performed using our implementation suggest however that the size of messages remains rather small.*

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: $n[P]_k$ 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's counter.

A message of the form $k\{req/E\}$ denotes the request req, located at k, and having been transmitted through the locations contained in the list of locations 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 2.3.

The definition of structural congruence is standard.

Definition 4 (Structural congruence) Structural congruence *is the smallest congruence* $\equiv$ *such that:*

1. *both kinds 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* $(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \equiv (\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P$*;*
4. $(\boldsymbol{\nu} p)\mathbf{0} \equiv \mathbf{0}$
5. rec $X.P \equiv P\{\text{rec } X.P/X\}$

Remark 5 (Extruding names across locations) *Like for the* Pan*, structural congruence does not allow a name restriction to be extruded out of a located ambient in a transparent way: the* GcPan *net* $h\colon n[(\boldsymbol{\nu} m)(\texttt{in}\ m)]_k$ *is not equivalent to* $(\boldsymbol{\nu} m)h\colon n[\texttt{in}\ m]_k$*. Since such a transformation is not innocuous from an implementation point of view, this phenomenon is described as a reduction step (see below).*

In order to establish the bisimulation result w.r.t. the Pan (and then compare a SA process with the GcPan net that represents it), we introduce a translation from SA processes to GcPan nets, and a notion of barb along the lines of the corresponding definitions for the Pan [11]. Intuitively, observables correspond to the In and Open interactions offered by a net (in SA, a context cannot trigger an Out movement).

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

Networks:

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

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

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

Processes:

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

Table 1: GcPan Syntax

Definition 6 (Translation from SA to GcPan) *Let $[\![.]\!]_{gc}$ be the translation of SA processes into GcPan nets defined by:*

$$[\![P]\!]_{gc} \triangleq \texttt{root}^0\colon \texttt{rootname}[P]_{\texttt{rootparent}}\,.$$

Definition 7 (Barbs) *Let A be a net, and n a name. We define the judgement 'A* exhibits barb *n' as follows: $A \downarrow_n$ iff $A \equiv (\boldsymbol{\nu}\overline{p})\left(\texttt{root}\colon \texttt{rootname}[\{M, h\} \mid P]_{\texttt{rootparent}} \parallel A'\right)$ where $M \in \{\overline{\texttt{in}}\ n,\ \overline{\texttt{open}}\ n\}$ and $n \notin \overline{p}$.*

2.2 Reduction Rules

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

Creation

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

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

Emission of request messages

[Req-In] $\texttt{in}\ m.P \xrightarrow[h:-]{k} \text{wait}.P \gg k\{\texttt{in}\ m, h\}$

[Req-Coin] $\overline{\texttt{in}}\ n.P \xrightarrow[h:n]{k} \text{wait}.P \gg k\{\overline{\texttt{in}}\ n, h\}$

[Req-Out] $\texttt{out}\ m.P \xrightarrow[h:-]{k} \text{wait}.P \gg k\{\texttt{out}\ m, h\}$

[Req-Coopen] $\overline{\texttt{open}}\ n.P \xrightarrow[h:n]{k} \text{wait}.P \gg k\{\overline{\texttt{open}}\ n, h\}$

Transmission of request messages

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

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

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

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

Local reductions

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

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

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

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

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

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

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

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

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

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

In the rules above, we have adopted the following notations:

$$P \xrightarrow[h:n]{k} P' \gg M \triangleq P \xrightarrow[h:n]{k} P' \gg^{0} M$$

$$h\{req\} \triangleq h\{req/[]\}$$

$$\left.\begin{array}{rcl} E\{M\} & \triangleq & e_1\{M\} \parallel \ldots \parallel e_i\{M\} \\ \#E & \triangleq & i \end{array}\right\} \text{ when } E = [e_1; \ldots; e_i]$$

Definition 8 (Communication relation, transitive closure) *We denote by $\longmapsto_c$ the reduction relation obtained by using only the transmission and inference rules. We denote by $\Longmapsto$ (resp. $\Longmapsto_c$) the reflexive, transitive closure of $\longmapsto$ (resp. $\longmapsto_c$).*

2.3 Comments on the Reduction Rules

2.3.1 About the shape 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 the $\gg$ records the increment that has to be brought to h's counter (cf. rule Proc-Agent, presented below). 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.

2.3.2 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, whose role is to inform the interacting agents that the transition has been fired.

  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 one could need to create an infinite number of ambients.
- The rules for *consumption of completion messages* describe how interacting agents resume computation when they are informed that a movement has occurred.

2.3.3 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 take into account:

- 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 ($k\{\mathtt{go}\ h\}$, $k\{\mathtt{go}_{\rhd} h\}$, ...).

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's 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).

2.3.4 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*:

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

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 $\texttt{go}_\rhd k$ 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 (h_1 and h_3) react to the relocation messages by moving to their new location, and resume computation.

3 Correctness of the Machine

In this section, we derive a bisimulation result relating *well-formed* Pan and GcPan nets. Intuitively, a net is well-formed if it corresponds to the evolution of the translation of a Safe Ambient process into the Pan (resp. GcPan) machine. Well-formedness of Pan nets is defined in [11], we present below the corresponding notion for the GcPan (Definition 13), and show that it is preserved by reductions (Theorem 18).

The correspondence we can make between two configurations of the PAN and the GCPAN respectively is rather coarse, because the machines route messages and manage forwarders differently. We could have hoped to be able to establish that the GCPAN always performs less reduction steps than the PAN. However, this is not the case, since in some situations the GCPAN is 'slower', due to the reductions that rearrange forwarders. Accordingly, we can only establish weak bisimilarity, and the candidate bisimulation relation we define (Definition 24) does not take forwarders into account at all.

Remark 9 (On the bisimulation up to expansion proof method) *To do the bisimulation proof, we rely on a few lemmas that basically say that both machines can route the same messages in the same way. Rather remarkably, we cannot use more powerful techniques, like up-to techniques, to abstract over this routing phase, in our setting.*

In [20], the authors establish the following lemma, that allows them to factorise the reasoning about forwarders when proving the adequacy of the PAN *w.r.t. SA:*

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

$\succeq$ *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.*

Unfortunately, we lack the corresponding expansion result in our setting. This is due to race situations that are introduced by the presence of blocked forwarders (see Appendix A for a detailed explanation).

3.1 Terminology

In our proof, we need to reason using some notions that are related to the logical structure of nets, such as the destination of a message or the parent of a location. We have been mentioning these notions in a rather informal way until now. The following definitions make these (as well as some other points related to the description of a state of the GCPAN) more precise.

A wait prefix in the local process of a located ambient indicates that this ambient is waiting for a completion message. However, these prefixes may actually indicate two different kinds of situation, that we call a *wait state* and a *frozen state*. These two cases are treated somehow differently in our proof. Below, a phrase like "there is a message M" is used to mean that in the state of the GCPAN we are considering, a message M can be found at top-level.

Definition 10 (Blocked ambients expecting a completion message)

- *A located ambient* h^i: $n[P]$ *is in* frozen state *if* $P \equiv (\boldsymbol{\nu}\overline{p})(\mathrm{wait}.Q|R)$ *for some* Q*,* R *and* $(\boldsymbol{\nu}\overline{p})$*, and there is a completion message of the form* $k\{\texttt{migrate}\ h\}$ *or* $h\{\texttt{register}^{\mathrm{s}}\ P\}$ *(the* wait *prefix comes in this case from application of the rule* LOCAL-OPEN*). Intuitively, an ambient is is frozen state while it is in the process of opening an immediate sub-ambient.*
- *A located ambient* h^i: $n[P]$ *is in* wait state *if* $P \equiv (\boldsymbol{\nu}\overline{p})(\mathrm{wait}.Q|R)$ *for some* Q*,* R *and* $(\boldsymbol{\nu}\overline{p})$*, and there is no completion message of the form* $k\{\texttt{migrate}\ h\}$ *or* $h\{\texttt{register}^{\mathrm{s}}\ P\}$ *(the* wait *prefix comes from the rules* REQ-$\star$*).*

Definition 11 (Naming and relating locations)

- *The* name *of a located ambient* h^i: $n[P]_k$ *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* $h\{\texttt{go}_{\rhd}\ k\}$
  - *undefined otherwise*
- *The* target location *of a message* $h\{M\}$ *is* h*, its* source location *is the location decorating the message (*`register` *completions have no source location).*

- *Given a net A, we can extract from A a relation over locations by saying that two locations are related if they are respectively 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: $h < k$ iff k is an ancestor of h according to the dependency relation.*

- *The* parent ambient location *(resp.* target ambient location*) of a location h (resp. a message $h\{M\}$) is the least (w.r.t. $<$) ambient location k such that $h < k$. h is a* child ambient location *of k if h is the home of a located ambient and $h < k$ (without restriction on k).*

Remark 12 (Handling of restrictions) *There are two usages of name restriction in the* Pan *(and the* GcPan*): the original SA name restriction, found in SA processes, and restriction on location names, allowing each site to create fresh location names.*

Restrictions on location names are introduced using rule New-Locamb*, and can always be brought at the top-level using structural congruence, so that we can ignore them in the proof. On the contrary, restrictions belonging to the SA process we want to simulate cannot escape a located ambient without using reduction rule* New-Res *(see Remark 5). However, once such restrictions are brought outside their defining location, they behave like restrictions on location names and can also be brought at top-level.*

In the following, we will implicitly pull up all restrictions located outside of located ambients, and omit them in our proofs.

3.2 Well-Formedness

The following definition 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.

Definition 13 (Well-formedness) *A* GcPan *net A is* well-formed *if* `root` *and* `rootparent` *are the only free locations of A, and the following properties hold:*

Blocked forwarders

1. *For each blocked forwarder $h\triangleleft^i$,*
   - *either $\exists!\ k\{M/E\}$ such that h belongs to E,*
   - *or $\exists!\ h\{\mathtt{go}_{\triangleright}\ k\}$.*
2. *For each request $h\{R/E\}$ 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`*;*
   - *the ambient locations in wait state (their parent locations are undefined);*
   - *the blocked forwarder locations h such that $pl(h)$ is undefined (the blocking request has not yet reached an ambient location).*
4. *For each location h, there is at most one agent located at h.*
5. *There is a special located ambient, called root, located at* `root`*, with name* `rootname`*, and with parent* `rootparent`*. The name* `rootname` *and the location* `rootparent` *cannot appear anywhere else.*

   The location `root` *cannot be:*

   (a) the source of a request message,

   (b) the target of a completion message that is not a `register` *message.*

Messages

6. *The source of every request message is a located ambient in wait state.*

7. *The target of every completion message is a located ambient in wait state, except for* `register` *messages, whose targets are frozen ambients.*

8. *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.*

9. *The source of every* `migrate` *message is a located ambient in frozen state.*

Single-Threadedness

10. *The process of every located ambient is single threaded.*

11. *The* wait *prefix can only appear as the only unguarded prefix of a local process of a located ambient.*

Counters

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

$$\begin{array}{rcll} \mu(h) & = & \#\{\ k^i\colon n[P]_h \ /\ P \text{ is not in a wait state}\ \} & (1)\\ & + & \#\{\ k \rhd^i h\ \} & (2)\\ & + & \#\{\ k\{\mathtt{go}_{\rhd} h\}\ \} & (3)\\ & + & \#\{\ h\{req/E\}\ \} & (4)\\ & + & \#\{\ k\{\mathtt{go}\ h\}\ \} & (5)\\ & + & \#\{\ k\{\mathtt{OK}\overline{\mathtt{in}}\ h\}\ \} & (6)\\ & - & \#\{\ h\{\mathtt{OK}\overline{\mathtt{in}}\ k\}\ \} & (7)\\ & + & \#\{\ h\{\mathtt{register}^1\ P\}\ \} & (8)\\ & + & \#\{\ k\{\mathtt{migrate}\ h\}\ \} & (9) \end{array}$$

where $\#\{p\}$ is the number of sub-terms of A matching the pattern p.

Then for each agent whose counter is equal to i ($h^i\colon n[P]_k,\ h \rhd^i k$ or $h \lhd^i$), $i = \mu(h)$

As can be seen, several kinds of properties have to be checked for a GcPan net to be well-formed. These are basically the same as for the corresponding definition in [11], plus some conditions on counters in localities, forwarders and messages, that insure a form of 'well-countedness'. As a matter of fact, some of the well-formedness properties that were given in [11] hold as a consequence of the latter requirements about counters, and are thus omitted in the GcPan version.

Remark 14 (Trees vs forests of locations) *In the* GcPan*, when an agent sends a request message, it commits to relocate itself on the location that will be transmitted along the completion message. This is mandatory because the link it keeps to its parent location is no longer safe, since this location may contain a forwarder bound to be garbage collected. Indeed, we would otherwise observe the following (erroneous) transition:*

$$h^i\colon n[\mathrm{wait}.P]_k \parallel k\{M\} \parallel k \rhd^1 k' \parallel A \overset{\text{Fw-SendGC}}{\longmapsto} h^i\colon n[\mathrm{wait}.P]_k \parallel k'\{M\} \parallel A\,.$$

The forwarder located at k gets garbage collected while h keeps a link to this location.

This leads us to consider the logical structure of a GcPan *net as a forest rather than as a tree (in the* Pan*, this problem does not arise since forwarders are persistent): a blocked agent is considered as disconnected from the main tree until it relocates itself.*

Lemma 15 *If A is a well-formed net and $A \equiv A'$ then A' is well-formed.*

Proof: Straightforward. ♢

The fact that well-formedness is closed w.r.t. $\equiv$ is not surprising. The important property is that well-formedness is preserved by reduction. In order to prove this, we need the two following lemmas.

Lemma 16 *Let $A \equiv\ h \rhd^1 k \parallel h\{req/E\} \parallel B$ be a well-formed net, then h can only appear in B as the parent location of a ambient location in wait state.*

Proof: From property 7, there is no completion message matching the *negative* (negative in the sens that it involves a subtraction) pattern 12.7. From property 12, $\mu(h) = 1$ so that there can only be one occurrence of the other patterns, and this occurrence is $h\{req/E\}$. Properties 2, 7 and 4 allow us to exclude all other potential occurrences of h. ♢

Lemma 17 *Let* $A \equiv \ h^0\colon n[P|\text{wait}.Q]_{k'} \parallel h\{\texttt{migrate}\ k\} \parallel B$ *be a well-formed net, then* h *can only appear in* B *as the parent location of an ambient location in wait state.*

Proof: From property 8, there is no completion message matching the negative pattern 12.7. From property 12, $\mu(h) = 0$ so that there is no occurrence of the other patterns. property 8 prevents the existence of other potential occurrences of h (completion messages of target h). ♢

Theorem 18 *If* A *is a well-formed net and* $A \longmapsto A'$ *then* A' *is well-formed.*

Proof:

We proceed by case analysis on the rule used to derive $A \longmapsto A'$ (up to the inference rules). Each time, we just mention the properties that are affected by the reduction.

- Local-Com

$$h^i\colon n[\langle M\rangle \mid (x).P \mid Q]_k \longmapsto h^i\colon n[P\{M/x\} \mid Q]_k$$

  The subject reduction property for ST types [14] ensures that property 10 holds.

- Local-In

$$h'^i\colon n[\{\texttt{in}\ n, h\} \mid \{\overline{\texttt{in}}\ n, k\} \mid Q]_{k'} \longmapsto h'^{i+1}\colon n[Q]_{k'} \parallel h\{\texttt{go}\ k\} \parallel k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$$

  Property 7 (resp. 5b) comes from property 6 (resp. 5a) for A. The two local request messages that are consumed ensured property 8, the two emitted completion messages ensure that this property is preserved. We have to check the counters of ambient localities h' and k (k contains an ambient from property 6):

  - k: the new occurrence of 12.5 ($h\{\texttt{go}\ k\}$) is compensated by a new occurrence of 12.7 ($k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$),
  - h': the new occurrence of 12.6 ($k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$) is handled by the increment brought via the reduction rule.

- Local-Out

$$h'^i\colon n[\{\texttt{out}\ n, h\} \mid \overline{\texttt{out}}\ n.P \mid Q]_k \longmapsto (\boldsymbol{\nu} k')(h'^i\colon n[P \mid Q]_{k'} \parallel k' \rhd^2 k \parallel h\{\texttt{go}\ k'\})$$

  The full dependency tree is modified but still respects properties 3 and 4. Property 7 (resp. 5b) comes from property 6 (resp. 5a) for A. Like in the previous case, property 8 is kept. We have to check the counters of k and k':

  - k: from property 11, since the ambient located at h' contains an unguarded prefix ($\overline{\texttt{out}}\ n.P$), this location cannot be in wait state, so that 12.1 is used for location k. In A', it is replaced by a child forwarder located at k' which matches 12.2.
  - k': there are only two children to count: the relocation completion message (12.5) and the relocated non-wait ambient h' (12.1).

- Local-Open

$$h'^i\colon n[\texttt{open}\ n.P \mid \{\overline{\texttt{open}}\ n, h\} \mid Q]_k \longmapsto h'^{i+1}\colon n[\text{wait}.P \mid Q]_k \parallel h\{\texttt{migrate}\ h'\}$$

  Property 7 (resp. 5b) comes from property 6 (resp. 5a) for A. The ambient at location h' enters in frozen state, ensuring property 9. As in the previous cases, property 8 is preserved. We have to check the counters of k and h'

  - k: although a wait prefix is added to the ambient child h', the latter enters frozen state (i.e., *not* wait state), the counter of k thus remains unchanged.

– h': the `migrate` completion, matching 12.9, is handled by the incrementation done with the rule.

- FW-SEND

$$h \rhd^{i+1} k \parallel h\{req/E\} \longmapsto h \lhd^{i} \parallel k\{req/h::E\}$$

Properties 1 and 2 hold for the newly blocked forwarder and the transmitted request. In the dependency relation, the sub-tree beyond h gets disconnected and becomes a new root in the forest; since h is a blocked forwarder with $pl(h)$ undefined, 3 holds. We have to check the counters of locations h, k and the transmitted request

– h: the lost request matching 12.4 is taken into account by the rule.

– k: the lost forwarder child matching 12.2 is matched by the transmitted request (12.3).

- FW-SENDGC

$$h \rhd^{1} k \parallel h\{req/E\} \longmapsto k\{req/E\}$$

Properties 1 and 2 hold because the request remains unchanged. From 16, h is a leaf in the dependency relation of A', and it is safe to remove it: property 3 is preserved. The counter of k is unaffected: the forwarder child matching 12.2 becomes a request matching 12.4.

- FW-RELOC

$$h \lhd^{i} \parallel h\{\mathtt{go}_{\rhd}\ k\} \longmapsto h \rhd^{i} k$$

Property 2 holds for A' from the uniqueness in Property 1. $pl(h) = k$ in both A and A', so that the dependency relation does not change. The $\mathtt{go}_{\rhd}$ message counted as 12.3 for k becomes a forwarder taken into account in 12.2.

- LOC-RCV

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

The consumed request blocked the forwarders at locations found in E, the messages $e\{\mathtt{go}_{\rhd}\ k\}$ ensures that property 1 still holds for these blocked forwarders, which were roots of the dependency relation and thus are moved under the location h. We check that this new forest satisfies property 3. For property 12, there are exactly j new occurrences of 12.3 ($\mathtt{go}_{\rhd}$ messages), and one occurrence of 12.4 less (consumed request).

- REQ-$\star$

$$h^{i}\colon n[M.P \mid Q]_k \longmapsto h^{i}\colon n[\mathrm{wait}.P \mid Q]_k \parallel k\{M, h\}$$

The ambient located at h enters in wait state, and the location h becomes a new root of the dependency relation, which remains a valid forest. h cannot be `root`, so that property 5a holds. Property 6 holds since h has just entered in wait state. Properties 7 and 6 ensure that there were no completion having h as a target nor request having h as source, so that the emitted request validate property 8. The lost child ambient location matching 12.1 for k becomes a new occurrence of 12.4 (emitted request).

- COMPL-PARENT

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

The location h, which was a root of the dependency relation, becomes a child of location k. For k, this new occurrence of 12.1 is compensated by the consumption of the completion, which was an occurrence of 12.5.

Note: the following holds for the three next rules.

From property 7, h is in wait state in A; from property 8, h can neither be the source of a request message, or the target of a completion different from the consumed one; this ensures the conservation of properties 6 and 7 although h leaves the wait state. For property 9, we use fact that wait states and frozen states are exclusive, thus forbidding h to be the source of a `migrate` message.

- Compl-Coin
$$h\{\texttt{OK}\overline{\texttt{in}}\ k\} \parallel h^i\colon n[P \mid \text{wait}.Q]_- \longmapsto h^{i+1}\colon n[P \mid Q]_k$$
The location h, which was a root of the dependency relation, becomes a child of location k. For k, this new occurrence of 12.1 is compensated by the consumption of the completion, which was an occurrence of 12.6. We need to increment the counter of h because the consumption of the completion also removes an occurrence of 12.7 (which is negative) for h.

- Compl-Migr
$$h\{\texttt{migrate}\ k\} \parallel h^{i+1}\colon n[P \mid \text{wait}.Q]_- \longmapsto h \rhd^{i+1} k \parallel k\{\texttt{register}^0\ (P \mid Q)\}$$
The location h, which was a root of the dependency relation, becomes a child of location k (as a forwarder). For k, this new new occurrence of 12.1 is compensated by the consumption of the completion, which was an occurrence of 12.9. As the ambient location h becomes a forwarder location, its counter is decremented accordingly to the property 12. For property 9, we also have to ensure that k remains in frozen state, this is the case because the `migrate` completion becomes a `register` completion.

- Compl-MigrGC
$$h\{\texttt{migrate}\ k\} \parallel h^0\colon n[P \mid \text{wait}.Q]_- \longmapsto k\{\texttt{register}^1\ (P \mid Q)\}$$
From Lemma 17, h is a leaf in the dependency relation of A', and it is safe to remove it. For k, the occurrence of 12.9 (consumed completion) becomes an occurrence of 12.8. As previously, k remains in frozen state.

- Compl-Reg
$$h\{\texttt{register}^s\ R\} \parallel h^{i+s}\colon n[P \mid \text{wait}.Q]_k \longmapsto h^i\colon n[P \mid Q \mid R]_k$$
From property 9, h is in frozen state in A, so that the full dependency relation remains unchanged. From property 8, h cannot be the source of a `migrate` completion; this ensure the conservation of property 9 although h leaves the frozen state. For properties 6 and 7, like previously, we use fact that wait states and frozen states are exclusive, thus forbidding h to be the source of a request message or the target of a completion message. The counter of k does not change since h is not in wait state, in A as well as A'. If $s = 1$, the counter of h is decremented so that the occurrence of 12.8 is taken into account.

- New-Locamb
$$h^i\colon m[n[P] \mid Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu} k)(k^0\colon n[P]_h)$$
The uniqueness of locations is kept since the name k is protected by the restriction $(\boldsymbol{\nu} k)$. The dependency relation is valid: we just added a new child to the location h. The counter of h is incremented for the new occurrence of 12.1; the counter of k is set to zero: k is fresh and $\mu(k) = 0$.

♢

From now on, we assume that all nets we consider are well-formed. This is in particular the case for the translation of a SA processes.

Lemma 19 *Let P be a single-threaded SA process, then $[\![P]\!]_{gc}$ is well-formed.*

Proof: Straightforward. ♢

3.3 Establishing Weak Bisimilarity

We define a relation between Pan and GcPan nets, which we prove to be a weak barbed bisimulation. We then show that, given a SA process, its two encodings in Pan and GcPan are related by this relation.

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). This will be the basis of the definition of our candidate bisimulation relation. To make this precise, we rely on the following definitions.

Definition 20 (Real location)

- *Let A be a* PAN *net, and h a location of A, we denote by $\widetilde{h}$ the* real location *of h, defined by:*
  - *$\widetilde{h} = h$ if the agent located at h is $h\colon n[P]_k$*
  - *$\widetilde{h} = \widetilde{k}$ if the agent located at h is $h \rhd k$*
- *Let B be a* GCPAN *net, and h a location of B, we similarly define $\widetilde{h}$, the* real location *of h, as follows:*
  - *$\widetilde{h} = h$ if the agent located at h is $h\colon n[P]_k$*
  - *$\widetilde{h} = \widetilde{k}$ if either*
    - *the agent located at h is $h \rhd k$, or*
    - *the agent located at h is $h{\lhd^i}$ and there is a request message $k\{req/E\}$ such that $h \in E$, or a relocation message $h\{\mathtt{go}_{\rhd}\ k\}$.*

Note that the well-formedness of B gives sense to the second definition above. We now introduce the *ambient dependency relation* defined by a net, not to be confused with the *dependency relation* (Definition 11).

Definition 21 (Ambient dependency relation) *For both abstract machines, given a net C, we call* ambient dependency relation of C *the relation defined on ambient locations of C by $h \prec_C k$ iff:*

- *either the ambient located at $\widetilde{h}$ is in wait state, in which case k is the target (resp. source) of the request (resp. completion) message given by clause 8 of Definition 13*[1]*.*
- *or the ambient located at $\widetilde{h}$ is not in wait state, so that we have a subterm of the form $\widetilde{h}\colon n[P]_{k'}$: we set $k = \widetilde{k'}$.*

Informally, $\prec_C$ captures the tree formed by ambient locations.

Definition 22 (Message sets)

- *For each ambient location of a* PAN *net A, we define:*

$$\begin{array}{rcll} tr_A(h) & = & \{req & / \ A \equiv A_0 \parallel h'\{req\} \quad \text{with } \widetilde{h'} = h\} \\ c_A(h) & = & \{\mathtt{OK}\overline{\mathtt{in}}\ k & / \ A \equiv A_0 \parallel h\{\mathtt{OK}\overline{\mathtt{in}}\} \quad \text{with } h \prec_A k\} \\ & \cup & \{\mathtt{migrate}\ k & / \ A \equiv A_0 \parallel h\{\mathtt{migrate}\} \quad \text{with } h \prec_A k\} \\ & \cup & \{\mathtt{register}\ Q & / \ A \equiv A_0 \parallel h\{\mathtt{register}\ Q\}\} \\ & \cup & \{\mathtt{go}\ \widetilde{k} & / \ A \equiv A_0 \parallel h\{\mathtt{go}\ k\}\} \end{array}$$

- *For each ambient location of a* GCPAN *net B, we define:*

$$\begin{array}{rcll} tr_B(h) & = & \{req & / \ B \equiv B_0 \parallel h'\{req/E\} \quad \text{with } \widetilde{h'} = h\} \\ c_B(h) & = & \{\mathtt{OK}\overline{\mathtt{in}}\ \widetilde{k} & / \ B \equiv B_0 \parallel h\{\mathtt{OK}\overline{\mathtt{in}}\ k\}\} \\ & \cup & \{\mathtt{migrate}\ \widetilde{k} & / \ B \equiv B_0 \parallel h\{\mathtt{migrate}\ k\}\} \\ & \cup & \{\mathtt{register}\ Q & / \ B \equiv B_0 \parallel h\{\mathtt{register}^s\ Q\}\} \\ & \cup & \{\mathtt{go}\ \widetilde{k} & / \ B \equiv B_0 \parallel h\{\mathtt{go}\ k\}\} \end{array}$$

- *For each ambient location $h\colon n[P]_k$ of a net C (*PAN *or* GCPAN*), we define:*

$$\begin{array}{rcl} ar_C(h) & = & \{req \ / \ P \equiv P_0 \mid \{req\}\} \\ p_C(h) & = & P \text{ from which all request messages have been removed} \\ r_C(h) & = & ar_C(h) \cup tr_C(h) \end{array}$$

[1] The notion of wait state and property 8 in Definition 8 can be defined for PAN nets like we did for GCPAN.

Intuitively, $tr_C(h)$ is the set of requests that will eventually reach h, $ar_C(h)$ is the set of requests that have already reached h, and $c_C(h)$ is the set of completions whose target is h. When C is a PAN net, we need to decorate the completions so that we can relate them with completions in the GCPAN. Since we abstract over the structure of forwarder chains, we also replace each location with its real location.

In Definition 22, we indeed introduce *sets* of messages: while, technically, a net is a multi-set rather than a set, these two notions coincide in this case. The reason is that, because of single-threadedness, a given location cannot send two requests, so that all sent requests are different. Similarly, clause 8 in the definition of well-formedness guarantees that there is at most one completion with a given target location ($\#c_C(h) \leq 1$).

In the following, we will sometimes write $n[P \mid M]$, where $M = \{M_1, \ldots, M_k\}$ is a set of messages, in order to denote the process $n[P \mid \{M_1\} \mid \ldots \mid \{M_k\}]$.

Definition 23 (Weak barbed bisimulation) *A relation $\mathcal{R}$ between* PAN *and* GCPAN *nets is a* weak barbed bisimulation *if $A \mathcal{R} B$ implies:*

- *$A \downarrow_n$ implies $B \Longmapsto \downarrow_n$, and the symmetrical condition on B.*
- *whenever $A \longmapsto A'$, there is B' s.t. $B \Longmapsto B'$, and symmetrically.*

Weak barbed bisimilarity, written $\approx$, is the greatest weak barbed bisimulation.

Definition 24 (Bisimulation candidate) *Let $\mathcal{R}$ be the relation between* PAN *and* GCPAN *nets defined by $A \mathcal{R} B$ iff:*

1. *A is well-formed (Definition 5.1 in [11]);*
2. *B is well-formed (Definition 13);*
3. *$\prec_A = \prec_B$;*
4. *for each ambient location h,*
   (a) *$r_A(h) = r_B(h)$;*
   (b) *$c_A(h) = c_B(h)$;*
   (c) *$p_A(h) = p_B(h)$.*

Note that the last point makes sense since the third one implies that A and B have the same ambient locations.

We now prove that $\mathcal{R}$ is a weak barbed bisimulation. We need several lemmas for this: the first two lemmas show that a net remains in the same $\mathcal{R}$-class when performing communications-only reductions. The other lemmas allow us to 'pull up' request messages when needed: since two $\mathcal{R}$-related nets do not necessarily have the same messages in the same locations, we need to be able, when one of the nets uses one of the LOCAL rules, to bring the corresponding message to the corresponding location and to perform the same local reduction.

Lemma 25 (PAN communications) *Suppose $A \mathcal{R} B$ and $A \longmapsto A'$, using the* PAN *reduction rules* CONSUME-REQ *or* FW-MSG. *Then $A' \mathcal{R} B$.*

Proof: Since A is well-formed, from Lemma 5.5 of [11], A' is well-formed. Obviously, $\prec_A = \prec'_A$, $c_A(h) = c_{A'}(h)$ and $p_A(h) = p_{A'}(h)$ for all location h. We then check that $r_A(h) = r_{A'}(h)$. ♢

Lemma 26 (GCPAN communications) *Suppose $A \mathcal{R} B$ and $B \longmapsto_c B'$, then $A \mathcal{R} B'$.*

Proof: Theorem 18 ensures that B' is well-formed. As previously, the sets $r_A(h)$, $c_A(h)$ and the processes $p_A(h)$ are preserved by these reductions. The ambient dependency relation also remains the same: for rule FW-SENDGC, Lemma 16 ensures that we can safely remove the forwarder. ♢

Lemma 27 (PAN requests) *Suppose $A \mathcal{R} B$ and $B \equiv h^i\colon n[P \mid \{M\}]_k \parallel B_0$. Then there is a net A' such that*

1. $A \Longmapsto A'$;
2. $A' \equiv h\colon n[p_A(h) \mid ar_A(h) \mid \{M\}]_{k'} \parallel A_0$;
3. $A' \mathcal{R} B$.

Proof: If $M \in ar_A(h)$, we take $A' = A$.
Otherwise, from property 4a, $M \in tr_A(h)$, that is,

$$A \equiv h_j\{M\} \parallel h_j \rhd h_{j-1} \parallel \ldots \parallel h_1 \rhd h \parallel h\colon n[Q]_{k'} \parallel A_0$$

We show by induction on j,

$$A \Longmapsto A' = h_j \rhd h_{j-1} \parallel \ldots \parallel h_1 \rhd h \parallel h\colon n[Q|\{M\}]_{k'} \parallel A_0$$

only using rules Consume-Req and Fw-Msg. From Lemma 25, $A' \mathcal{R} B$ holds, and $A \mathcal{R} B$ gives $Q \equiv p_A(h) \mid ar_A(h)$. ♢

In order to establish a similar result on the GcPan side, we need the following lemma, which explains the behaviour of forwarders in the GcPan. As depicted on Figure 8, when we need to route a message $\{M\}$ in the GcPan, we are in general compelled to 'bring up' other messages as well ($\{N\}$ on the figure, which correspond to M_a in the statement of the lemma below). Such messages correspond to the forwarders that have been blocked above $\{M\}$ in the chain that leads to the destination location. We have to unblock these forwarders, which means bringing the messages in M_a to their destination. This involves a reconfiguration of the chains of forwarders (note that this does not happen on the Pan side (Lemma 27), since forwarders are never blocking).

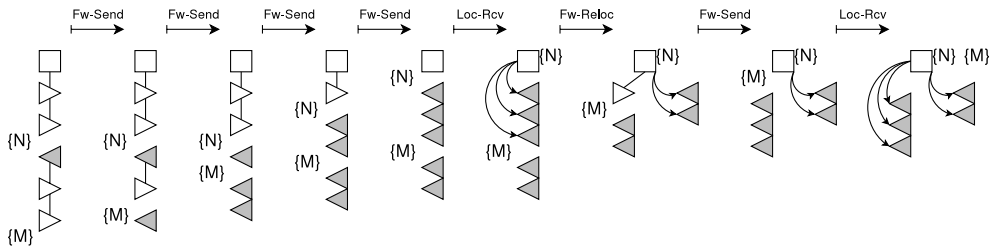


Figure 8: GcPan forwarders behaviour

Lemma 28 (Transmission of messages in the GcPan) *Let $B \equiv h\{M/E\} \parallel k^i\colon n[P] \parallel B_0$ be a well-formed net such that $k = \widetilde{h}$. Then there is a net B' and a set M_a of request messages such that:*

- $B \Longmapsto_c B'$;
- $B' \equiv E\{\mathsf{go}_{\rhd}\ k\} \parallel k^{i'}\colon n[P \mid \{M\} \mid M_a)] \parallel B_1$;
- $M_a \subset tr_B(k)$.

Proof: The condition $k = \widetilde{h}$ ensures that B_0 contains a chain of forwarders from h to k, we proceed by induction on the length of this chain, and by case analysis on the agent located at h:

- $h^i\colon n[P]$: the length is zero, and $h = k$. We use rule Loc-Rcv and take $M_a = \emptyset$:

$$B \longmapsto_c E\{\mathsf{go}_{\rhd}\ k\} \parallel k^{i+\#E-1}\colon n[P \mid \{M\}] \parallel B_0$$

- $h \rhd^1 h'$, using the rule Fw-SendGC, we bring the message up so that we can apply the induction hypothesis (IH below):

$$\begin{array}{rll} B & \longmapsto_c & h'\{M/E\} \parallel H' \parallel k^i\colon n[P] \parallel B_1 \qquad [\text{Fw-SendGC}] \\ & \Longmapsto_c & E\{\mathsf{go}_{\rhd}\ k'\} \parallel k^{i'}\colon n[P \mid \{M\} \mid M_a)] \parallel B_2 \qquad (IH) \end{array}$$

- $h \rhd^{u+1} h'$, using rule Fw-Send, we bring the message up so that we can apply the induction hypothesis, and conclude using rule Fw-Reloc:

$$\begin{array}{lll} B & \longmapsto_c & h \lhd^{u-1} \parallel h'\{M/h{::}E\} \parallel H' \parallel k^i\colon n[P] \parallel B_1 \qquad \text{[Fw-Send]} \\ & \Longmapsto_c & h \lhd^{u-1} \parallel h\{\mathtt{go}_\rhd\ k'\} \parallel E\{\mathtt{go}_\rhd\ k'\} \parallel k^{i'}\colon n[P \mid \{M\} \mid M_a] \parallel B_2 \qquad (IH) \\ & \longmapsto_c & h \rhd^{u} k' \parallel E\{\mathtt{go}_\rhd\ k'\} \parallel k^{i'}\colon n[P \mid \{M\} \mid M_a] \parallel B_2 \qquad \text{[Fw-Reloc]} \end{array}$$

- $h \lhd^u$, from property 1 of well-formedness, B_0 contains:
  - either a relocation message $h\{\mathtt{go}_\rhd k'\}$. From the definition of $\widetilde{h}$, $k = \widetilde{h} = \widetilde{k'}$, and we are back to one of the two last cases using rule Fw-Reloc:

$$\begin{array}{lll} B & \equiv & h\{M/E\} \parallel h \lhd^u \parallel h\{\mathtt{go}_\rhd\ k'\} \parallel k^i\colon n[P] \parallel B_1 \\ & \longmapsto_c & h\{M/E\} \parallel h \rhd^u k' \parallel k^i\colon n[P] \parallel B_1 \qquad \text{[Fw-Reloc]} \\ & \Longmapsto_c & E\{\mathtt{go}_\rhd\ k\} \parallel k^{i'}\colon n[P \mid \{M\} \mid M_a] \parallel B_2 \qquad \text{(previous cases)} \end{array}$$

  - or a request message $k'\{N/h{::}E'\}$ (modulo the index of h in E). As previously $k = \widetilde{k'}$, so that we can apply the induction hypothesis with the message $\{N\}$, and get back to the previous case:

$$\begin{array}{lll} B & \equiv & h\{M/E\} \parallel h \lhd^u \parallel k'\{N/h{::}E'\} \parallel k^i\colon n[P] \parallel B_1 \\ & \Longmapsto_c & h\{M/E\} \parallel h \lhd^u \parallel h\{\mathtt{go}_\rhd\ k\} \parallel E'\{\mathtt{go}_\rhd\ k'\} \parallel k^{i_0}\colon n[P \mid \{N\} \mid N_a] \parallel B_2 \qquad (IH) \\ & \Longmapsto_c & E\{\mathtt{go}_\rhd\ k\} \parallel k^{i'}\colon n[P \mid \{M\} \mid \{N\} \mid N_a] \parallel B_3 \qquad \text{(previous)} \end{array}$$

    We check that $N \in tr_B(k)$ and take $M_a = \{N\} \cup N_a$.

♢

Lemma 29 (GcPan requests) *Suppose $A \mathcal{R} B$ and $A \equiv h\colon n[P|\{M\}]_k \parallel A_0$. Then there is a net B' and a set of request messages such that:*

1. $B \Longmapsto B'$;
2. $B' \equiv h^i\colon n[p_B(h) \mid ar_B(h) \mid \{M\} \mid M_a]_{k'} \parallel B_0$;
3. $M_a \subset tr_B(h)$;
4. $A \mathcal{R} B'$.

Proof: If $M \in ar_B(h)$, we take $B' = B$.
Otherwise, from property 4a, $M \in tr_B(h)$, that is:

$$B \equiv h'\{M/E\} \parallel h^i\colon n[Q]_{k'} \parallel B_0\,.$$

Lemma 28 gives B' such that $B \Longmapsto_c B'$ and

$$B' \equiv h^{i'}\colon n[Q \mid \{M\} \mid M_a]_{k'} \parallel B_1, \text{ with } M_a \subset tr_B(h)\,.$$

Then Lemma 26 ensures that $A \mathcal{R} B'$, and we check that $Q \equiv p_B(h) \mid ar_B(h)$. ♢

Theorem 30 *$\mathcal{R}$ is a weak barbed bisimulation.*

Proof: Suppose $A \mathcal{R} B$.

- Lemma 27 (resp. 29) ensures that whenever $A \downarrow_n$ (resp. $B \downarrow_n$), then $B \Longmapsto \downarrow_n$ (resp. $A \Longmapsto \downarrow_n$).
- Suppose $A \overset{\lambda}{\longmapsto} A'$ (we use this notation to indicate that the derivation of $A \longmapsto A'$ is obtained by applying rule named λ), we show that $B \Longmapsto B'$ with $A' \mathcal{R} B'$:
  - $\lambda \in \{$Local-Com, New-Locamb, New-Res, Req-$\star$, Compl-Reg$\}$:
    we check easily that $B \overset{\lambda}{\longmapsto} B'$ with $A' \mathcal{R} B'$.

– $\lambda \in \{\text{Fw-Msg}, \text{Consume-Req}\}$: Lemma 25 ensures that $A' \mathcal{R} B$, so that we can take $B' = B$.

– $\lambda \in \{\text{Local-In}, \text{Local-Open}\}$: we consider the first case, the second one being treated similarly. From Lemma 29, $B \Longmapsto B_0$ such that $A \mathcal{R} B_0$ and the requests $\{\mathtt{in}\ n, h\}$ and $\{\overline{\mathtt{in}}\ n, k\}$ have reached the ambient located at h'. Therefore $B_0 \overset{\text{Local-In}}{\longmapsto} B'$, and we check that $A' \mathcal{R} B'$.

– $\lambda = \text{Local-Out}$:

$$A \equiv A_1 \parallel h'\colon n[\{\mathtt{out}\ n, h\} \mid \overline{\mathtt{out}}\ n.P \mid Q]_k \longmapsto A_1 \parallel h'\colon n[P \mid Q]_k \parallel h\{\mathtt{go}\ k\} \equiv A'$$

Like previously, we use Lemma 29 to ensure that the request $\{\mathtt{out}\ n, h\}$ has reached the ambient at h'. Then we obtain B' using rule Local-Out:

$$\begin{array}{rcl} B \Longmapsto_c B_0 & \equiv & B_1 \parallel h'\colon n[\{\mathtt{out}\ n, h\} \mid \overline{\mathtt{out}}\ n.P \mid Q']_{k'} \\ & \longmapsto & B_1 \parallel ((\boldsymbol{\nu} k''))(h'\colon n[P \mid Q']_{k''} \parallel h\{\mathtt{go}\ k''\} \parallel k'' \rhd^1 k') \equiv B' \end{array}$$

$A \mathcal{R} B$ gives $\widetilde{k} = \widetilde{k'}$ and by definition $\widetilde{k''} = \widetilde{k'}$, therefore we have $\widetilde{k''} = \widetilde{k}$ so that we can check that $\prec_{A'} = \prec_{B'}$ and then $A' \mathcal{R} B'$.

– $\lambda = \text{Compl-Parent}$:

$$\begin{array}{l} A \equiv A_1 \parallel h\colon n[P \mid \mathrm{wait}.Q]_k \parallel h\{\mathtt{go}\ k'\} \\ B \equiv B_1 \parallel h^i\colon n[P' \mid \mathrm{wait}.Q]_- \parallel h\{\mathtt{go}\ k''\} \end{array} \quad \overset{\text{Compl-Parent}}{\longmapsto} \quad \begin{array}{l} A_1 \parallel h\colon n[P \mid Q]_{k'} \equiv A' \\ B_1 \parallel h^i\colon n[P' \mid Q]_{k''} \equiv B' \end{array}$$

The hypothesis $\widetilde{k'} = \widetilde{k''}$ given by $A \mathcal{R} B$ allows us to check that $A' \mathcal{R} B'$.

– $\lambda = \text{Compl-Coin}$:

$$\begin{array}{l} A \equiv A_1 \parallel h\colon n[P \mid \mathrm{wait}.Q]_k \parallel h\{\mathtt{OK}\overline{\mathtt{in}}\} \\ B \equiv B_1 \parallel h^i\colon n[P' \mid \mathrm{wait}.Q]_- \parallel h\{\mathtt{OK}\overline{\mathtt{in}}\ k'\} \end{array} \quad \overset{\text{Compl-Parent}}{\longmapsto} \quad \begin{array}{l} A_1 \parallel h\colon n[P \mid Q]_k \equiv A' \\ B_1 \parallel h^{i+1}\colon n[P' \mid Q]_{k'} \equiv B' \end{array}$$

As previously, $A \mathcal{R} B$ gives $\widetilde{k} = \widetilde{k'}$ and allows us to conclude. We can remark that since k does not appear in the $\mathtt{OK}\overline{\mathtt{in}}$ completion of the Pan, we had to introduce a way to recover this information in Definition 22. For this reason, a definition that would just erase extra decorations of GcPan completions would not be sufficient.

– $\lambda = \text{Compl-Migr}$:

$$A \equiv A_1 \parallel h\colon n[P \mid \mathrm{wait}.Q]_k \parallel h\{\mathtt{migrate}\} \longmapsto A_1 \parallel h \rhd k \parallel k\{\mathtt{register}\ P|Q\} \equiv A'$$

Depending on the counter of the ambient located at h in B, we use either the rule Compl-Migr or Compl-MigrGC:

* $B \equiv B_1 \parallel h^{i+1}\colon n[P \mid \mathrm{wait}.Q]_- \parallel h\{\mathtt{migrate}\ k'\} \longmapsto B_1 \parallel h \rhd^{i+1} k' \parallel k'\{\mathtt{register}^0\ P|Q\}$
* $B \equiv B_1 \parallel h^0\colon n[P \mid \mathrm{wait}.Q]_- \parallel h\{\mathtt{migrate}\ k'\} \longmapsto B_1 \parallel k'\{\mathtt{register}^1\ P \mid Q\}$

As previously, $\widetilde{k} = \widetilde{k'}$. In the latter case, in order to prove $\prec_{A'} = \prec_{B'}$, we use Lemma 17 to ensure that there cannot be any real location under h in B and thus neither in A since $\prec_A = \prec_B$.

- Now suppose $B \overset{\lambda}{\longmapsto} B'$, we show that $A \Longmapsto A'$ with $A' \mathcal{R} B'$. The proof is quite similar to the symmetric case:

– $\lambda \in \{\text{Local-Com}, \text{New-Locamb}, \text{New-Res}, \text{Req-}\star, \text{Compl-Reg}\}$:

we check easily that $A \overset{\lambda}{\longmapsto} A'$ with $A' \mathcal{R} B'$.

– $\longmapsto_c$: Lemma 26 ensures that $A \mathcal{R} B'$, so that we can take $A' = A$.

– $\lambda \in \{\text{Local-}\star\}$: from Lemma 27, $A \Longmapsto A_0$ such that $A_0 \mathcal{R} B$ and the matching request messages have reached the ambient located at h'. Therefore $A_0 \overset{\lambda}{\longmapsto} A'$, and we check like previously that $A' \mathcal{R} B'$.

– $\lambda \in \{\text{Compl-Parent}, \text{Compl-Coin}, \text{Compl-Migr}\}$: as in the corresponding case above, we only have to be careful in handling the decoration of the completion message.
– $\lambda = \text{Compl-MigrGC}$: using rule Compl-Migr, $A \longmapsto A'$, Lemma 17 ensures that there cannot be any real location under h in B, so that the absence of forwarder in B' does not prevent relations $\prec_{A'}$ and $\prec_{B'}$ to coincide.

♢

We can now state our main result on the GcPan (we recall from Section 1 that we refer to [11] for the typing of SA processes):

Corollary 31 (Adequacy) *Let P be a well-typed SA process, then $[\![P]\!]_{gc} \approx P$.*

Proof: We check that $[\![P]\!] \mathcal{R} [\![P]\!]_{gc}$, so that from Theorem 30, $[\![P]\!] \approx [\![P]\!]_{gc}$, the result then comes from the adequacy of the Pan [11]: $[\![P]\!] \approx P$. ♢

It can be remarked that this proof actually exploits three notions of weak bisimilarity, to relate behaviours in SA, in the Pan and in the GcPan. These are defined along the lines of Definition 23, based on a notion of observable and a notion of reduction in the corresponding calculus.

4 Immobile Ambients

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 our out other ambients), nor be opened ($\overline{\texttt{open}}$ 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 $\rhd^\infty$. 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 (rules that are not mentioned are kept unchanged).

[Local-Out]	$\{\texttt{out}\ n, h\} \mid \overline{\texttt{out}}\ n.P \xrightarrow[h':-]{-} P \gg\!\gg_{k'} h\{\texttt{go}\ k'\}$	n is ST
[Local-Out']	$\{\texttt{out}\ n, h\} \mid \overline{\texttt{out}}\ n.P \xrightarrow[h':n]{k} P \gg h\{\texttt{go}\ k\}$	n is IM
[Compl-Coin]	$h\{\texttt{OK}\overline{\texttt{in}}\ k\} \parallel h^i\colon n[P \mid \text{wait}.Q]_{k'} \longmapsto h^{i+1}\colon n[P \mid Q]_k$	n is ST
[Compl-Coin']	$h\{\texttt{OK}\overline{\texttt{in}}\ -\} \parallel h^i\colon n[P \mid \text{wait}.Q]_{k'} \longmapsto h^{i+1}\colon n[P \mid Q]_{k'}$	n is IM
[Local-In]	$\{\texttt{in}\ n, h\} \mid \{\overline{\texttt{in}}\ n, k\} \xrightarrow[h':-]{-} \mathbf{0} \gg^1 h\{\texttt{go}\ k\} \parallel k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$	n is ST
[Local-In']	$\{\texttt{in}\ n, h\} \mid \{\overline{\texttt{in}}\ n, k\} \xrightarrow[h':-]{-} \mathbf{0} \gg^0 h\{\texttt{go}\ k\} \parallel k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$	n is IM
[New-Locamb]	$h^i\colon m[n[P] \mid Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu} k)(k^0\colon n[P]_h)$	n is ST
[New-Locamb']	$h^i\colon m[n[P] \mid Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu} k, k')(k^0\colon n[P]_{k'} \parallel k' \rhd^\infty h)$	n is IM, $k, k' \notin FL(Q)$

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

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 rules COMPL-COIN and LOCAL-IN: 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 32 (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 an IM ambient inside an IM ambient without creating the persistent forwarder associated to the inner ambient: the side condition of rule* NEW-LOCAMB *can thus be relaxed to "n is ST or m is IM".*

In order to establish the correctness of this machine, we need to extend the definition of well-formedness as follows:

Definition 33 (Well-formedness extension)

1. *Property 7 is extended with the following requirement: "The target of a* `migrate` *completion message is a located, waiting, single threaded ambient."*
2. *We impose that every IM ambient is located under a persistent forwarder, and contains at most one* wait *prefix in its local process.*
3. *In property 12, we consider* $\mu'(h) = \mu(h) - \#\{\ k\{\texttt{OK}\overline{\texttt{in}}\ h\}\ /\ k\ IM\ \}$ *and require that* $i = \mu'(h)$ *whenever* $i < \infty$.

The first item above allows us to guarantee that only ST ambients get opened. The second item ensures that immobile locations do not escape their persistent forwarder, and do not flood unnecessarily their parent locations with $\overline{\texttt{in}}$ requests. Finally, we just ignore $\texttt{OK}\overline{\texttt{in}}$ messages whose target is IM, since such an ambient will not relocate itself upon reception of this completion.

The proof of Theorem 30 can then be adapted to systems that also include IM ambients:

Theorem 34 *For any well-typed (using ST and IM types) SA process* P, $[\![P]\!] \approx [\![P]\!]_{gc}$.

5 An Implementation of the GCPAN

5.1 Description of the Implementation

We now present an experimental implementation of the GCPAN[2] in OCaml [19]. In developing the implementation, we have tried to keep the code as close as possible to its formal description, while allowing ourselves a few optimisations (reported below).

5.1.1 Physical Distribution

In the PAN, the fact that each ambient is evaluated at its own location makes a completely distributed execution of the ambient tree a priori possible. It is however likely that some ambients should reside on the same site – computer, UNIX process – so that we are led to aggregate locations: each site is a multi-threaded program, where each thread corresponds to a location. Communications between locations are therefore implemented using shared memory when these locations reside on the same site, and with network sockets otherwise.

[2] available for download at `http://perso.ens-lyon.fr/damien.pous/gcpan/`.

5.1.2 GenComm: general primitives for first order channel communications

In order to abstract over these two kinds of communications, we wrote a library that handles in a transparent way local and network communications. This is rather general-purpose piece of code, and we comment below on the main ideas of this part of the development. This paragraph goes into some rather low-level and OCaml-oriented explanations, and can be skipped by the non interested reader.

The library for communications is implemented in the module `GenComm`. Its interface is given on Table 2.

```
module type CONV = sig
  type +'c t
  val conv: ('a -> 'b) -> 'a t -> 'b t
end

type host = string * int  (* hostname, port number *)

type 'a receiver =
| Receive of ('a -> 'a receiver)
| Close

module Make(Msg: CONV)(Cmd: CONV): sig
  type channel
  type message = channel Msg.t
  type command = channel Cmd.t

  val send: channel -> message -> unit
  val channel: message receiver -> Thread.t * channel

  val send_command: host -> command -> unit
  val master: (command -> unit) -> Thread.t
end
```

Table 2: (Simplified) `GenComm` interface

The functor `Make` allows one to define communication primitives for a given message type `'c Msg.t`. The type variable `'c` may be used for channel values. For example, to define communication primitives for the monadic (resp. polyadic) π-calculus, we would use the functor with `type 'c Msg.t = 'c` (resp. `'c list`).

Function `channel` is used to create a new channel, given a *message receiver*: a message receiver is a function that expects a message and either returns a new receiver, or closes the channel. The created channel is returned along with the thread identifier associated to the receiver. Note that no function to receive on a channel is provided: a channel is created on a site, with a given, unique listener, and the corresponding input capability cannot be shared or migrate on the network. Implementing this would probably involve some mechanism resembling the linear forwarders of [10].

Commands are used to effectively distribute the channels: the `channel` function creates only *local channels*, so that we cannot escape the local site. The `master` function is used to associate an *external listener* for the local site. This is a thread that reacts on commands sent over the network with `send_command` by invoking the given function. Typically, this can be used for instance by a site that asks another site to create a fresh channel and to send it back on a given channel.

Internally, channels are represented using polymorphic variants:

```
type gchannel = [ `Gen of host*int ]
type channel  = [ `Loc of Local.t*int | gchannel ]
```

where `Local` is the module defining shared memory channels (the integer is the index of the channel within a local table).

Before being sent on a network channel, a message has to be packaged in order to give meaning to the local names that are transmitted in the message (if any). This conversion is performed by the following function:

```
let gchannel_of_channel = function
  | #gchannel as g -> g
  | `Loc (_,id) -> `Gen ((localhost,port), id)
let message_conv = Msg.conv gchannel_of_channel
```

The type of `gchannel_of_channel` is `channel->gchannel`, `gchannel` being the type of 'sendable channels'. Symmetrically, when the distant host receives such a message, we perform the inverse conversion, and rely for this on *sub-typing*. Indeed, covariant types are associated to `Msg` and `Cmd` in the definition of `CONV` (see Table 2).

Since the size of transmitted messages is not bound, we use TCP for network communications. In order to reduce the number of connections, we cache them so that a site connects only once to an other given site. An additional improvement would be to able to close these connections once they are no longer used.

5.1.3 Optimisations w.r.t. the formal description

Using the previous primitives the code remains very close to the reductions rules of Section 2. We implemented two independent optimisations to improve the management of forwarders, that depart from the formal description.

- *Relocation:* in our rules, blocked forwarders are relocated by the broadcast of a $\mathtt{go}_{\triangleright}$ message. This may be very inefficient if the forwarders are located on a site that is different from the site of the *real parent* (that performs the broadcast). In order to avoid such losses in efficiency, we give a blocked forwarder the possibility to transmit the relocation message to the blocked forwarders running on the same site (and that were already registered in the request message when it crossed this forwarder). This way, messages will be broadcasted once for every distant site, at which place a further transmission takes place.

  This optimisation is described by the following reductions rules, where $h \lhd^i E$ denotes a blocked forwarder that has to relocate E (Fw-Send2 is applied when there is $e \in E$ such that e and h are on the same site):

  $$\begin{array}{lrcl}
  \text{[Fwd-Send1]} & h\{req/j, E\} \parallel h \rhd^{i+1} k & \longmapsto & h \lhd^i [] \parallel k\{req/j+1, h::E\} \\
  \text{[Fwd-Send2]} & h\{req/j, E\} \parallel h \rhd^{i+1} k & \longmapsto & h \lhd^i [e] \parallel k\{req/j+1, E\{h \backslash e\}\} \\
  \text{[Fw-Reloc]} & h \lhd^i E \parallel h\{\mathtt{go}_{\triangleright} k\} & \longmapsto & h \rhd^i k \parallel E\{\mathtt{go}_{\triangleright} k\} \\
  \text{[Loc-Rcv]} & h^{i+1}\colon n[P]_k \parallel h\{req/j, E\} & \longmapsto & h^{i+j}\colon n[P \mid \{req\}]_k \parallel E\{\mathtt{go}_{\triangleright}\ h\}
  \end{array}$$

- *Packaging of request messages:* when a forwarder is in blocked state, it has to filter received messages until it finds a $\mathtt{go}_{\triangleright}$ message (to move to a new location). In doing this, incoming request messages have to be enqueued in order to be transmitted upon forwarder relocation. Instead of transmitting these messages sequentially, as described by the original reduction rules, we can pack them, and send them all together:

  $$\begin{array}{lrcl}
  \text{[Fw-Reloc]} & h \lhd^i_S \parallel h\{\mathtt{go}_{\triangleright} k\} & \longmapsto & h \rhd^i_S k \\
  \text{[Fw-Pack]} & h\{R/j, E\} \parallel h \lhd^i_{(j', E', R')} & \longmapsto & h \lhd^i_{(j+j', E@E', R@R')} \\
  \text{[Fw-Send]} & h\{R/j, E\} \parallel h \rhd^{i+1}_{(0,[],[])} k & \longmapsto & h \lhd^i_{(0,[],[])} \parallel k\{R/j+1, h::E\} \\
  \text{[Fw-SendPack]} & h \rhd^{i+\#R}_{(j,E,R)} k & \longmapsto & h \lhd^i_{(0,[],[])} \parallel k\{R/j+1, h::E\}
  \end{array}$$

These improvements can be switched off using appropriate commands.

5.1.4 Non-determinism

In the implementation, non-determinism is left unspecified: while we impose no fairness constraint at all in the communication of messages, a rather high amount of non-determinism is guaranteed by the fact that the machine runs in a multi-threaded and distributed setting.

5.2 Experimental Observations

5.2.1 An example session

is shown on Table 3, where lines beginning with - are the answers of the top-level. A server is first installed on the host A; the corresponding SA term is the following:

$$\begin{array}{ll}(\boldsymbol{\nu} \mathit{Server}) & (\ \mathit{Server}[!\overline{\texttt{in}}\ \mathit{Server} \mid !\overline{\texttt{out}}\ \mathit{Server} \mid \mathit{data}[\texttt{in}\ \mathit{user}.\overline{\texttt{open}}\ \mathit{data}.\texttt{print}\mathit{secret}]] \\ & \quad \mid !\texttt{open}\ \mathit{o}.\mathit{key}[\texttt{in}\ \mathit{user}.\overline{\texttt{open}}\ \mathit{key}.\quad (\ \mathit{enter}[\overline{\texttt{open}}\ \mathit{enter}.\texttt{in}\ \mathit{Server}.\texttt{open}\ \mathit{entered}] \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{leave}[\overline{\texttt{open}}\ \mathit{leave}.\texttt{out}\ \mathit{Server}.\texttt{open}\ \mathit{left}])]\end{array}$$

then we connect two clients with the server, one on the same host:

$$\begin{array}{l}\mathit{user}[\overline{\texttt{in}}\ \mathit{user}.\texttt{open}\ \mathit{key}.\texttt{open}\ \mathit{enter}.\mathit{entered}[\overline{\texttt{open}}\ \mathit{entered}.\texttt{print}\ \mathit{entered}.\texttt{pause}. \\ \qquad \texttt{open}\ \mathit{leave}.\mathit{left}[\overline{\texttt{open}}\ \mathit{left}.\texttt{print}\ \mathit{left}]]] \\ \mid \mathit{o}[\overline{\texttt{open}}\ \mathit{o}]\end{array}$$

and one on from a second host B, that is willing to get access to the data:

$$\begin{array}{l}\mathit{user}[\overline{\texttt{in}}\ \mathit{user}.\texttt{open}\ \mathit{key}.\texttt{open}\ \mathit{enter}.\mathit{entered}[\overline{\texttt{open}}\ \mathit{entered}.\texttt{print}\ \mathit{entered}.\texttt{in}\ \mathit{user}.\texttt{open}\ \mathit{data}. \\ \qquad \texttt{open}\ \mathit{leave}.\mathit{left}[\overline{\texttt{open}}\ \mathit{left}.\texttt{print}\ \mathit{left}]]] \\ \mid \mathit{o}[\overline{\texttt{open}}\ \mathit{o}]\end{array}$$

The machine makes it possible to execute Safe Ambients in a distributed fashion, thanks to the primitives for communication between the various concurrent runtimes. The user must enter well-typed processes (names of immobile ambients begin with a capital letter): we have started studying type inference, but we only have preliminary results for the moment. Some top-level commands (starting with a `#`), as well as some additional capabilities in the syntax of Ambients, allow the user to:

- pause and resume computation (`pause`, `#step`);
- inspect the state of the system and to display statistics about the number of local/network messages and of created/collected forwarders: (`print`, `#tree`, `#stats`);
- add a process to the ambient currently being executed (`#add`; additionally, writing `n@H[P]` delegates the execution of ambient `n` to the site `H`, while `#addto H n[P]` lets `n` be executed locally, but under the root ambient of the runtime at site `H`).

5.2.2 Comparison with the PAN.

One can easily adapt our code to obtain an implementation of the PAN: simply get rid of counters and additional decorations. This allows us to draw comparisons between the two abstract machines.

Table 4 shows an example session illustrating the improvements of the GCPAN over the PAN[3]. We start by defining some simple terms (note that process named `c` depends on a process `P`). These definitions are used to build two ambient trees having `c` ambients as internal nodes, the leafs consisting of `A` (resp. `b`) nodes. We then release an ambient `!open c` to erase all internal nodes, which generates forwarders (at each stage in the computation, a `pause` (resp. `step`) directive is used to suspend (resp. resume) the execution of the SA terms). Each `b` ambient is programmed to enter and exit an ambient `A`, so that request messages are transmitted along the forwarders, thus revealing the difference between the two machines (which can be seen by looking at the way forwarders appear in the resulting trees).

[3]The details of the system answers in the session we show may vary in the future along possible updates of the implementation. The overall behaviour, and the way SA processes are executed, shall however remain unchanged.

```
(* on a host named A *)
(nu Server)( Server[ !in_Server | !out_Server
                   | data[in user.open_data.print secret]]
           | !open o.key[in user.open_key.( enter[open_enter.in  Server.open entered]
                                          | leave[open_leave.out Server.open left])]);;
#add user[in_user.open key.
            open enter.entered[open_entered.print entered.pause.
              open leave.left[open_left.print left]]] | o[open_o];;
-- entered
#tree;;
-- Root[Server#0[data[]] | user[leave[]]]]  (* suffix #0 added to preserve uniqueness *)
#step;;
-- left
#tree;;
-- Root[Server#0[data[]] | user[]]

(* on a host named B *)
#addto A user[in_user.open key.
            open enter.entered[open_entered.print entered.in_user.open data.
              open leave.left[open_left.print left]]];;
#addto A o[open_o];;
-- entered
-- secret  (* this is printed on B since the process 'print secret' migrated *)
-- left
```

Table 3: An example session: the firewall

The statistics that are collected show the gain in terms of efficiency brought by the GcPan. `4+0+4-4` reads as follows: 4 forwarders have been created by an Open, 0 by an Out, 4 persistent forwarders have been introduced along the creation of immobile ambients, and 4 forwarders have been garbaged collected. We can observe that we obtain less forwarders in the GcPan. As a consequence, less messages are communicated in the run of the system, and that forwarder chains are much shorter in the GcPan. It has to be stressed that for the sake of clarity, we give an example with few processes and ambient interactions — the results scale to larger systems, with similar proportions between the two machines.

In the firewall example however, since all opened ambients are *empty*, there are no additional messages brought by the corresponding forwarders. In such a case, the GcPan reduces the number of created forwarders, but not the number of messages sent over the network.

Indeed, in terms of messages, the loss introduced by the acknowledgement with the grandparent for Out moves has to be compensated by the contraction of frequently used, long, forwarder chains. It is not yet clear whether non-artificial examples tend to generate and use such long chains or not. However, typical distributed systems, such as e.g. servers, are specified not to terminate, which means that even chains consisting of a few forwarders may lead to dramatic inefficiencies, that our setting allows us to avoid.

6 Conclusions

6.1 Related Work

The comparison with the PAN [11] 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 [9], a JoCaml implementation of an abstract machine for Mobile Ambients, named AtJ, is presented.

```
(* PAN session *)

let a()   = A[pause i.(!in_A | !out_A)];;
let b()   = b[pause i.in A.pause o.out A];;
let c('P) = c[open_c | P];;

c(c(c(a()|a()|a())|c(a()))) | c(c(c(b()|b()|b())|c(b()))));; #tree;;
-- R@@t[c[c[c[A[]] | c[A[] | A[] | A[]]]] | c[c[c[b[]] | c[b[] | b[] | b[]]]]]
#add !open c;; #tree;;
-- R@@t[<[<[<[A[]] | <[A[] | A[] | A[]]]] | <[<[<[b[]] | <[b[] | b[] | b[]]]]]
#step i;; #tree;;
-- R@@t[<[<[<[A[b[]]] | <[A[b[]] | A[b[]] | A[b[]]]]] | <[<[<[] | <[]]]]
#step o;; #tree;;
-- R@@t[<[<[<[A[] | b[]] | <[b[] | b[] | b[] | A[] | A[] | A[]]]]
         | <[<[<[] | <[]]]]

#stats;;
-- forwarders                      : 8
-- messages                        : 100
-- IN/OUT/OPEN                     : 4/4/8
-- average forwarder chains length : 1.50


(* GCPAN session *)

let a()   = A[pause i.(!in_A | !out_A)];;
let b()   = b[pause i.in A.pause o.out A];;
let c('P) = c[open_c | P];;

c(c(c(a()|a()|a())|c(a()))) | c(c(c(b()|b()|b())|c(b()))));; #tree;;
-- R@@t[c[c[c[<[A[]]] | c[<[A[]] | <[A[]] | <[A[]]]]]
         | c[c[c[b[]] | c[b[] | b[] | b[]]]]]
#add !open c;; #tree;;
-- R@@t[<[<[A[]]] | <[<[A[]] | <[A[]] | <[A[]]] | <[b[]] | <[b[] | b[] | b[]]]
#step i;; #tree;;
-- R@@t[<[A[b[] | b[] | b[]]] | <[A[]] | <[A[b[]]] | <[A[]]]
#step o;; #tree;;
-- R@@t[<[b[] | b[] | A[] | b[]] | <[A[]] | <[b[] | A[]] | <[A[]]]

#stats;;
-- forwarders                      : 4 + 4 - 4 = 4
-- messages                        : 78
-- IN/OUT/OPEN                     : 4/4/8
-- average forwarder lifetime      : 0.5
-- average forwarder chains length : 1.09
```

Table 4: PAN vs. GCPAN

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 Pan and GcPan only `open` induces physical movement.

[17] presents a distributed abstract machine for the Channel Ambients calculus, a variant of Boxed Ambients [2]. 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 [7], 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. For instance, this should be possible for Controlled Ambients [22], 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 [15] – where the parent ambient (b) is not involved in an Out move – or the Push and Pull Ambient Calculus [18] – 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 started our study with Ambient-based calculi because they are concise formalisms with a well-studied theory. 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 further simplify the correctness proof for the machine. Although being structured, 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 further 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] P. Bidinger and J.-B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. of FMOODS 2003*, volume 2884 of *LNCS*, pages 109–123. Springer Verlag, 2003.

[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] 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.

[7] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, 1998.

[8] 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.

[9] 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.

[10] P. Gardner, C. Laneve, and L. Wischik. Linear Forwarders. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*, pages 408–422. Springer Verlag, 2003.

[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] F. Le Fessant. *JoCaml: conception et implémentation d'un langage à agents mobiles*. PhD thesis, École Polytechnique, 2001.

[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] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients (extended abstract). In *Proc. of POPL '02*, pages 71–80, 2002.

[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] 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.

[19] INRIA projet Cristal. The OCaml programming language. Available at `http://caml.inria.fr`.

[20] D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer Verlag, 2001.

[21] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.

[22] 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.

[23] A. Unyapoth and P. Sewell. Nomadic pict: correct communication infrastructure for mobile computation. In *Proc. of 28th POPL*, pages 116–127. ACM Press, 2001.

A A non-expansion result

We present a *non-expansion result*, that shows why the technique of weak bisimulation up to expansion cannot be used in the proof of Section 3. We omit counters when writing GcPan nets for the sake of readability.

Lemma 35 *In the* GcPan*, there exists a net A such that:*

- $h \rhd k \parallel k : n[P]_{k'} \parallel A$ *and* $k : n[P]_{k'} \parallel A\{k/h\}$ *are well-formed nets, and*
- $\underbrace{h \rhd k \parallel k : n[P]_{k'} \parallel A}_{T} \succeq \underbrace{k : n[P]_{k'} \parallel A\{k/h\}}_{U}$ *does not hold.*

Proof: Let

$$\begin{cases} P &= \overline{\texttt{open}}\ n \\ A &= k' : m[\texttt{open}\ n.Q]_l \parallel l : o[] \parallel B \end{cases}$$

Suppose $T \succeq U$. By performing the Open move on both sides, we deduce

$$\underbrace{h \rhd k \parallel k \rhd l \parallel l : o[Q] \parallel B}_{T'} \succeq \underbrace{k \rhd l \parallel l : o[Q] \parallel B\{k/h\}}_{U'} .$$

Then, let

$$\begin{cases} Q &= \overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o \\ B &= h\{\texttt{out}\ o, a\} \parallel h\{\texttt{out}\ o, b\} \parallel k\{\texttt{out}\ o, c\} \parallel C \end{cases}$$

$$\begin{array}{rcl} T' & \equiv & h\{a\} \parallel h\{b\} \parallel h \rhd k \parallel k\{c\} \parallel k \rhd l \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \parallel C \\ U' & \equiv & k\{a\} \parallel k\{b\} \parallel \quad k\{c\} \parallel k \rhd l \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \parallel C\{k/h\} \end{array}$$

(where $x\{y\}$ stands for $x\{\texttt{out}\ o, y\}$)

Suppose $T' \Longmapsto T''$, $U' \Longmapsto U''$ and $T'' \succeq U''$. We can make the following three observations:

1. since the agents located at a, b and c may have completely different behaviours, T'' and U'' must have done exactly the same Out moves.
2. if the process of the agent located at l in T'' contains the message $\{x\}$ then, also the process of l in U'': otherwise, T'' can do the Out move x in one step (rule Local-Out), while U'' cannot.
3. if there is a message $l\{x\}$ in T'', then either it is also in U'' or the process of l in U'' contains the message $\{x\}$: otherwise T'' can consume the message using rule Loc-Rcv and reach the previous state, while U'' needs several steps in order to do so.

Now we show that we cannot play the expansion game from $T' \succeq U'$.

- Using rule Fw-Send,

$$T' \longmapsto \underbrace{h\{b\} \parallel h \lhd \parallel k\{a/[h]\} \parallel k\{c\} \parallel k \rhd l \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o]}_{T_1}$$

U' cannot move: if it brings a message up ($k\{a\}$ for instance):

$$U' \underbrace{\longmapsto k\{b\} \parallel k\{c\} \parallel k \lhd \parallel l\{a/[k,h]\} \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o]}_{U''},$$

it blocks its forwarder, and T_1 can choose to bring up another message ($k\{a\}$ or $k\{c\}$), so that U'' will not be able to satisfy remark 3 in one step.

Thus we need $T_1 \succeq U'$.

- Using rule Fw-Send,

$$U' \longmapsto \underbrace{k\{a\} \parallel k\{c\} \parallel k \lhd \parallel l\{b/[k]\} \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o]}_{U_1}$$

T_1 cannot move: if $k\{a\}$ or $k\{c\}$ goes up, we contradict remark 3. Thus $T_1 \succeq U_1$.

- Using rule Loc-Rcv,

$$U_1 \longmapsto \underbrace{k\{a\} \parallel k\{c\} \parallel k \lhd \parallel k\{\texttt{go}_{\rhd}\ l\} \parallel l : o[\{b\}|\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o]}_{U_2}$$

As previously, T_1 cannot move and $T_1 \succeq U_2$.

- Using rule Local-Out, U_2 can then emit the request corresponding to the b Out move:

$$U_2 \longmapsto \underbrace{k\{a\} \parallel k\{c\} \parallel k \lhd \parallel k\{\texttt{go}_{\rhd}\ l\} \parallel l : o[\overline{\texttt{out}}\ o] \parallel OUT_b}_{U_3}$$

(where OUT_b contains the handling of the b Out request)

From remark 1, T_1 has to do the same Out request:

$$\begin{array}{rcl}
T_1 & \longmapsto & h\{b\} \parallel h \lhd \parallel k\{c\} \parallel k \lhd \parallel l\{a/[k,h]\} \parallel l : o[\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \\
& \longmapsto & h\{b\} \parallel h \lhd \parallel k\{c\} \parallel k \lhd \parallel l : o[\{a\}|\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \parallel k\{\texttt{go}_{\rhd} l\} \parallel h\{\texttt{go}_{\rhd} l\} \\
& \Longmapsto & h\{b\} \parallel h \rhd l \parallel k\{c\} \parallel k \rhd l \parallel l : o[\{a\}|\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \\
& \Longmapsto & h \lhd \parallel k\{c\} \parallel k \lhd \parallel l : o[\{a\}|\{b\}|\overline{\texttt{out}}\ o.\overline{\texttt{out}}\ o] \parallel k\{\texttt{go}_{\rhd} l\} \parallel h\{\texttt{go}_{\rhd} l\} \\
& \Longmapsto & \underbrace{h \lhd \parallel k\{c\} \parallel k \lhd \parallel l : o[\{a\}|\overline{\texttt{out}}\ o] \parallel OUT_b}_{T_2} \parallel k\{\texttt{go}_{\rhd} l\} \parallel h\{\texttt{go}_{\rhd} l\}
\end{array}$$

But in order to do it we had to bring the message $h\{a\}$ into l, and this contradicts remark 2: $T_2 \succeq U_3$ does not hold.

$\diamondsuit$