On the Expressive Power of Movement and Restriction in Pure Mobile Ambients

Nadia Busi and Gianluigi Zavattaro [2]

Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy

---

Abstract

Pure Mobile Ambients is a process calculus suitable to focus on issues related to mobility, abstracting away from aspects concerning process communication. However, it incorporates name *restriction* (i.e. the (νn) binder) and ambient *movement* (i.e. the *in* and *out* capabilities) that can be seen as characteristics adapted, or directly borrowed, from the tradition of communication-based process calculi. For this reason, we retain that it is worth to investigate whether or not these features can be removed from Pure Mobile Ambients without losing expressive power.

To this aim, we consider two variants of Pure Mobile Ambients which differ in the way infinite processes can be defined; the former exploits process replication, while the latter is more general and permits recursive process definition. We analyse whether or not the elimination of ambient movement and/or name restriction reduces the expressive power of these two calculi, using the decidability of process termination as a yardstick. We prove that name restriction can be removed from both calculi without reducing the expressive power. On the other hand, the elimination of both ambient movement and name restriction strictly reduces the expressive power of both calculi. As far as the elimination of only ambient movement is concerned, we prove an interesting discrimination result: process termination is undecidable under recursive process definition, while it turns out to be decidable under process replication.

---

1 Introduction

Mobile Ambients [8] is a well known formalism for the description of distributed and mobile systems in terms of *ambients*. An ambient is a named

[1] Revised and extended version of [5]

[2] Corresponding author. E-mail: zavattar@cs.unibo.it.

collection of active processes and nested sub-ambients. In the pure version of Mobile Ambients, the so-called Pure Mobile Ambients, only three mobility primitives are used to permit ambient and process interaction: *in* and *out* for ambient movement, and *open* to dissolve an ambient boundary.

More precisely, a process performs an *in* m primitive to instruct its surrounding ambient to move inside a sibling ambient named m, *out* m to instruct its surrounding ambient to exit its parent ambient named m, and *open* m to dissolve the boundary of an ambient named m located at the same level as the process.

The calculus of Pure Mobile Ambients has been introduced because it represents a simple and elegant calculus to focus on issues related to mobility, abstracting away from aspects concerning process communication. However, this calculus incorporates operators that can be seen as adapted, or directly borrowed, from the tradition of communication-based process calculi.

First of all, the name *restriction* operator (νn) is a feature that Pure Mobile Ambients directly borrow from the tradition of name passing calculi. See, e.g. the π–calculus of Milner, Parrow and Walker [16], where the same operator (νn) is used to permit the generation of new channel names.

Moreover, let us consider the *in* and *out* operations used as primitives for supporting ambient *movement*. This form of movement can be seen as the adaptation, to the setting of Mobile Ambients, of the form of mobility permitted by calculi with higher-order communication. For example, in calculi such as Thomsen's CHOCS [23], agents can be exchanged between processes exploiting higher-order channels. In absence of a channel topology, but in presence of a hierarchical structure of nested ambients, the *in* and *out* primitives similarly permit the movement of ambients inside the hierarchy of ambient nestings.

In light of these observations, we think it is worthwhile to investigate whether or not *movement* and/or *restriction* can be removed from Pure Mobile Ambients without losing expressive power.

To this aim, we consider two infinite variants of the Pure Mobile Ambients. The first one corresponds to the original calculus as defined in [8] where infinite processes can be obtained using the replication operator $!P$. The term $!P$ represents the parallel composition of an unbounded amount of instances of the process P. The second variant uses a more general mechanism, namely recursive process definition $rec\,X.P$. Clearly, recursive definition is more general than replication as $!P$ can be seen as the process $rec\,X.(P|X)$. The calculus with replication is denoted with $\mathrm{MA}_!$ while the variant with recursive definition is denoted with MA_r.

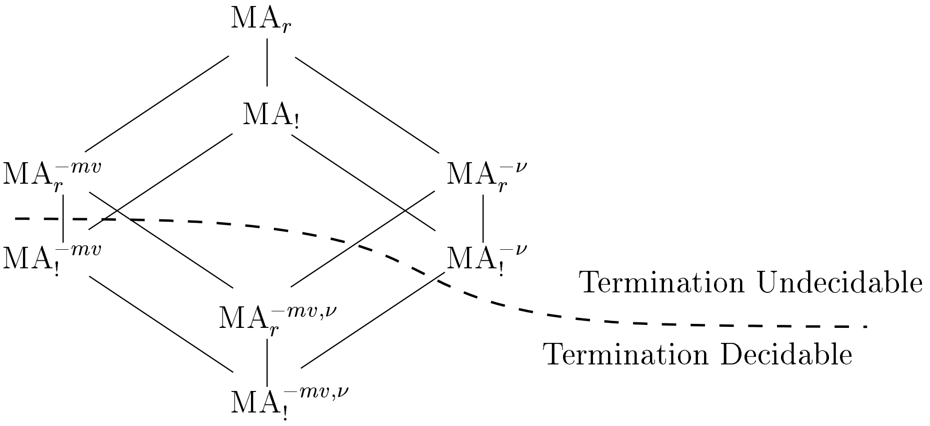

Fig. 1. Overview of the calculi.

We consider the decidability of process termination as a yardstick to evaluate the expressive power. In [8] the Turing completeness of Pure Mobile Ambients is discussed; guidelines are reported which indicate how to encode Turing machines in the calculus. A consequence of the argumentations in that paper is that process termination is undecidable in both $MA_!$ and in MA_r. In order to evaluate whether or not the elimination of movement and/or restriction reduces the expressive power, we consider the fragments of the two calculi obtained by removing movement, restriction, or both. We use the following notation: $MA_!^{-mv}$ (resp. MA_r^{-mv}) for the fragment of $MA_!$ (resp. MA_r) without movement, $MA_!^{-\nu}$ (resp. $MA_r^{-\nu}$) for the fragment of $MA_!$ (resp. MA_r) without name restriction, and $MA_!^{-mv,\nu}$ (resp. $MA_r^{-mv,\nu}$) for the fragment of $MA_!$ (resp. MA_r) without both movement and name restriction. We study whether or not process termination is decidable in each of these calculi.

In Figure 1 we report an overview of the considered calculi ordered according to a sub-calculus relation; even if the calculi with replication are not syntactically subsets of the corresponding calculi with recursive definition, we consider them as sub-calculi because (as discussed above) recursive definition is more general than replication. In the Figure we also report the overview of our results, separating the calculi in which process termination is decidable from those in which it is not.

The discussion about Turing completeness of Pure Mobile Ambients in [8] considers how to encode Turing Machines; the proposed encoding exploits name restriction. In this paper, we show that name restriction is not necessary; indeed, we show how to model in the calculus $MA_!^{-\nu}$ Random Access Machines (RAMs), another well-known register based Turing powerful formalism. As recursive definition is more general, this Turing completeness result holds also in $MA_r^{-\nu}$.

If we remove name restriction and ambient movement, both calculi turn out to be non Turing complete. This is a consequence of the fact that termination is decidable in $\mathrm{MA}_r^{-mv,\nu}$, thus also in the less general calculus $\mathrm{MA}_!^{-mv,\nu}$. Actually, the decidability result is proved for a slightly more general calculus, obtained by extending $\mathrm{MA}_!^{-mv}$ with a limited form of recursion that basically forbids the definition of infinite chains of restrictions.

As far as the elimination of only ambient movement is concerned, we prove an interesting discrimination result between replication and recursive definition; process termination is not decidable under recursive definition, while it turns to be decidable under replication. Indeed, we show how RAMs can be modelled also in MA_r^{-mv} , and we show that process termination is decidable in $\mathrm{MA}_!^{-mv}$.

1.1 Structure of the paper

The paper is structured as follows. Section 2 is devoted to the formal definition of the syntax and the semantics of the calculi. The proofs of the Turing completeness of MA_r^{-mv} and $\mathrm{MA}_!^{-\nu}$ are reported in Section 3. The proofs of the decidability of process termination in $\mathrm{MA}_!^{-mv}$ and $\mathrm{MA}_r^{-mv,\nu}$ can be found in Section 4. Finally, in Section 5 a comparison with related literature and some conclusive remarks are reported.

2 Syntax and Semantics of the Calculi

In this section we present the definition of the syntax and semantics of the fragments/variants of the Pure Mobile Ambients that we are interested in. We start by recalling the definition of the original calculus, that we denote with $\mathrm{MA}_!$, as defined in [8].

Definition 2.1 *Let Name, ranged over by n, m, ..., be a denumerable set of ambient names. The terms of $MA_!$ are defined by the following grammar:*

$$\begin{array}{rcl} P & ::= & \mathbf{0} \;\mid\; M.P \;\mid\; n[P] \;\mid\; P|P \;\mid\; (\nu n)P \;\mid\; !P \\ M & ::= & in\ n \;\mid\; out\ n \;\mid\; open\ n \end{array}$$

We use $\prod_k P$ to denote the parallel composition of k instances of the process P, while $\prod_{i\in 1\ldots k} P_k$ denotes the parallel composition of the indexed processes P_i.

The term $\mathbf{0}$ represents the inactive process (and it is usually omitted). $M.P$ is a process guarded by one of the three mobility primitives (already discussed in the Introduction): after the execution of the primitive the process behaves like P. The term $n[P]$ denotes an ambient named n containing process P. A process may be also the parallel composition $P|P$ of two subprocesses. The restriction operator $(\nu n)P$ is used to create a new name n which is bound in P. Finally, the replication operator $!P$ is used to put in parallel an unbounded amount of instances of the process P.

Definition 2.2 *The unique binder in the calculus is the restriction operator $(\nu n)P$ which binds the ambient name n to the process P. Given a process P, the set of the free names in P (denoted with $fn(P)$) is composed of those names n appearing outside any binder $(\nu n)Q$; the set of the other names in P (i.e. the bound names) is denoted with $bn(P)$. We use also $n(P)$ to denote the union of free and bound names.*

The operational semantics is defined in terms of a structural congruence plus a reduction relation.

Definition 2.3 *The structural congruence $\equiv$ is the smallest congruence relation satisfying:*

$$
\begin{array}{ll}
(\nu n)\mathbf{0} \equiv \mathbf{0} & (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \textit{if } n \notin fn(P) \\
(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & (\nu n)(m[P]) \equiv m[(\nu n)P] \quad \textit{if } n \neq m \\
P \mid \mathbf{0} \equiv P & P \mid Q \equiv Q \mid P \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & !P \equiv P \mid !P \\
\textit{if } P \equiv_\alpha Q \textit{ then } P \equiv Q &
\end{array}
$$

where $fn(P)$ denotes the free names in P, and $P \equiv_\alpha Q$ means that P and Q are α-convertible.

Definition 2.4 *The reduction relation is the smallest relation $\rightarrow$ satisfying* *the following axioms and rules:*

$$
\begin{array}{ll}
(1) & n[in\ m.P \mid Q] \mid m[R] \ \rightarrow\ m[n[P \mid Q] \mid R] \\
(2) & m[n[out\ m.P \mid Q] \mid R] \ \rightarrow\ n[P \mid Q] \mid m[R] \\
(3) & open\ n.P \mid n[Q] \ \rightarrow\ P \mid Q \\
(4) & \dfrac{P \ \rightarrow\ Q}{P \mid R \ \rightarrow\ Q \mid R} \\
(5) & \dfrac{P \ \rightarrow\ Q}{n[P] \ \rightarrow\ n[Q]} \\
(6) & \dfrac{P \ \rightarrow\ Q}{(\nu n)P \ \rightarrow\ (\nu n)Q} \\
(7) & \dfrac{P' \equiv P \quad P \ \rightarrow\ Q \quad Q' \equiv Q}{P' \ \rightarrow\ Q'}
\end{array}
$$

As usual, we use $\rightarrow^+$ to denote the transitive closure and $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. Moreover we use $P \rightarrow^k Q$, with $k \geq 1$ to denote that $P \rightarrow P_1 \ldots \rightarrow P_{k-1} \rightarrow Q$ (i.e. Q can be reached from P after the execution of k transitions).

Axioms (1), (2) and (3) describe the semantics of the three primitives *in*, *out* and *open*, respectively. A process inside an ambient n can perform an *in* m operation in presence of a sibling ambient m; if the operation is executed then the ambient n moves inside m. If inside an ambient m there is an ambient n with a process performing an *out* m action, this results in moving the ambient n outside the ambient m. Finally, a process performing an *open* n operation has the ability to remove the boundary of an ambient $n[Q]$ composed in parallel with it.

Rules (4), (5) and (6) are the contextual rules that respectively indicate that a process can move also when it is in parallel with another process, when it is inside an ambient, and when it is under a restriction. Finally, rule (7) is used to ensure that two structurally congruent terms have the same reductions.

In $\mathrm{MA}_!$ infinite processes can be obtained by exploiting the replication operator. Replication supports *in-width* recursion as exemplified by the axiom $!P \equiv P|!P$. On the other hand, other variants of Mobile Ambients, (see e.g. [9,19,22]) permit recursive definition of processes via the usual notation $rec\,X.P$, where X is a process variable possibly occurring free in P. Clearly, recursive definition is more general and permits both *in-width* (see, e.g., the process $rec\,X.(P|X)$ corresponding to $!P$) and *in-depth* recursion (see, e.g., the process $rec\,X.(\nu n)(P|X)$ corresponding to an infinite nesting of restrictions).

Definition 2.5 *We denote with MA_r the calculus obtained from $MA_!$ by replacing replication with recursive process definition. Formally, let Var, ranged over by X, Y, ..., be a set of process variables; the syntax of MA_r is as follows:*

$$P \ ::= \ \mathbf{0} \ \mid \ M.P \ \mid \ n[P] \ \mid \ P|P \ \mid \ (\nu n)P \ \mid \ X \mid \ rec\,X.P$$

where M is defined as in Definition 2.1. In the term $rec\,X.P$, the initial part $rec\,X.$ is a binder for the process variable X; we will consider only processes with no free process variables. Moreover, as done in [19], in order to avoid infinite ambient nesting we assume unboxed recursion, *i.e. given $rec\,X.P$ the variable X cannot appear inside any ambient of P.*
The operational semantics of MA_r is defined as for the calculus $MA_!$, where the axiom $!P \equiv P|!P$ is replaced by the following one permitting the unfolding of recursively defined processes

$$rec\,X.P \equiv P\{rec\,X.P/X\}$$

where $P\{rec\,X.P/X\}$ denotes the term obtained by replacing $rec\,X.P$ for any free occurrence of X in P.

As described in the Introduction, we are interested in fragments of these calculi obtained by removing ambient movement (i.e. the *in* and *out* capabilities) and/or name restriction (i.e. the (νn) operator).

Definition 2.6 *We define the following calculi as fragments of $MA_!$ and MA_r:*

- *$MA_!^{-mv}$ (resp. MA_r^{-mv}) denotes the fragment of $MA_!$ (resp. MA_r) without movement,*
- *$MA_!^{-\nu}$ (resp. $MA_r^{-\nu}$) denotes the fragment of $MA_!$ (resp. MA_r) without name restriction,*
- *$MA_!^{-mv,\nu}$ (resp. $MA_r^{-mv,\nu}$) denotes the fragment of $MA_!$ (resp. MA_r) without both movement and name restriction.*

The following definition formally introduces the notion of process termination: informally, a process P is *terminated* if it has no outgoing reductions, while P *terminates* if all its computations terminate.

Definition 2.7 *We say that a process P is* terminated *if there exists no Q such that $P \rightarrow Q$; we say that P* terminates *if it cannot give rise to an infinite* *computation, i.e.*

$$P \text{ terminates} \quad \text{iff} \quad \begin{array}{l} \textit{there exists no } \{P_i\}_{i \in \mathbb{N}} \textit{ s.t.} \\ P_0 = P \textit{ and } P_j \rightarrow P_{j+1} \textit{ for any } j \in \mathbb{N} \end{array}$$

3 Undecidability Results

In this Section we prove that process termination is undecidable in MA_r^{-mv} and $\mathrm{MA}_!^{-\nu}$. We prove these results by showing how to model, in these two calculi, Random Access Machines (RAMs) [21], a well known Turing powerful formalism. We start by recalling what RAMs are.

3.1 Random Access Machines

RAMs are a computational model based on finite programs acting on a finite set of registers. More precisely, a RAM R is composed of the registers $r_1, \ldots, r_n$, that can hold arbitrary large natural numbers, and by a sequence of indexed instructions $(1 : I_1), \ldots, (m : I_m)$. In [17] it is shown that the following two instructions are sufficient to model every recursive function:

- $(i : Succ(r_j))$: adds 1 to the contents of register r_j and goes to the next instruction;
- $(i : DecJump(r_j, s))$: if the contents of the register r_j is not zero, then decreases it by 1 and goes to the next instruction, otherwise jumps to the instruction s.

The computation starts from the first instruction and it continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached.

A state of a RAM is modelled by $(i, c_1, \ldots, c_n)$, where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively. We use the notation $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ to indicate that the state of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$, as a consequence of the execution of the i-th instruction.

A state $(i, c_1, \ldots, c_n)$ is *terminated* if the program counter i is strictly greater than the number of instructions m. We say that a RAM R *terminates* if its computation reaches a terminated state. Observe that RAM computations are deterministic, thus it is not necessary to assume that all computations terminate.

3.2 Modelling RAMs in MA_r^{-mv}

In this section we show how to model RAMs in the calculus with recursive process definition, but without ambient movement. Actually, we show a stronger result presenting how to model RAMs in an asynchronous fragment of CCS [15] and proving that this calculus tightly corresponds to the fragment of MA_r^{-mv} in which ambients are always empty, i.e., for any term $n[P]$, P is the empty process $\mathbf{0}$.

The subsection is now divided in three parts respectively containing:

- the definition of the asynchronous version of CCS that we call aCCS and the proof that it corresponds to a fragment of MA_r^{-mv};
- the definition of aCCS+, a syntactic extension of aCCS which comprises also an input guarded choice operator and the definition of an encoding function which reduces terms of aCCS+ into terms of aCCS;
- the encoding of RAMs written using the extended syntax of aCCS+ and the proof that the corresponding terms of aCCS (obtained via the reduction defined in the previous item) faithfully model the considered RAM.

3.2.1 Asynchronous CCS

We call aCCS the asynchronous fragment of CCS that we use to encode RAMs. We use notations borrowed from the π–calculus [16], because it is more convenient for our purposes. For instance, differently from the original definition of CCS [15], we use the prefixed notation $(\nu n)P$ instead of the postfix one $P\backslash n$ for restriction, and we define the operational semantics directly in terms of a reduction relation $\rightarrow$ instead of a labelled semantics $\xrightarrow{\alpha}$.

In the definition of aCCS we overload some of the notations used for the Pure Mobile Ambients calculi of the previous section (e.g. the set $Name$ and and the congruence relation $\equiv$); the actual meaning of these operators is made clear by the context.

Definition 3.1 *In the definition of aCCS we use the set $Names$, ranged over by n, m, ..., introduced in Definition 2.1 and the set Var, ranged over by X, Y, ..., of Definition 2.5. The terms of aCCS are defined by the following grammar:*

$$P \ ::= \ \mathbf{0} \ | \ n.P \ | \ \overline{n} \ | \ P|P \ | \ (\nu n)P \ | \ X \ | \ recX.P$$

As usual we consider only closed terms, i.e. those processes in which each variable X appears under some binder $rec\,X$. Free names and bound names are defined as usual (the unique binder is restriction).
The operational semantics of aCCS is defined in terms of a structural congruence $\equiv$ and a reduction relation $\rightarrow$.
The structural congruence $\equiv$ is the smallest congruence relation satisfying:

$$
\begin{array}{ll}
P \mid \mathbf{0} \equiv P & P \mid Q \equiv Q \mid P \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \textit{if } P \equiv_\alpha Q \textit{ then } P \equiv Q \\
(\nu n)\mathbf{0} \equiv \mathbf{0} & (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \textit{if } n \notin fn(P) \\
(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & rec\,X.P \equiv P\{rec\,X.P/X\}
\end{array}
$$

where $fn(P)$ denotes the free names in P, and $P \equiv_\alpha Q$ means that P and Q are α-convertible.

The reduction relation is the smallest relation $\rightarrow$ satisfying the following axioms and rules:

$$
\begin{array}{ll}
(i) & \overline{n} \mid n.P \ \rightarrow\ P \\
(ii) & \dfrac{P \ \rightarrow\ Q}{P \mid R \ \rightarrow\ Q \mid R} \\
(iii) & \dfrac{P \ \rightarrow\ Q}{(\nu n)P \ \rightarrow\ (\nu n)Q} \\
(iv) & \dfrac{P' \equiv P \quad P \ \rightarrow\ Q \quad Q' \equiv Q}{P' \ \rightarrow\ Q'}
\end{array}
$$

In aCCS the notion of process termination is defined as in Definition 2.7.

It is trivial to see that aCCS tightly corresponds to the fragment of MA_r^{-mv} in which we have only empty ambients: the output particle $\overline{n}$ corresponds to the empty ambient $n[\mathbf{0}]$ and the input prefix $n.P$ corresponds to the open capability *open* $n.P$.

Proposition 3.2 *Let $\{\!\!\{\ \}\!\!\}$ be the function mapping processes of aCCS into* *terms of MA_r^{-mv} inductively defined as follows:*

$$
\begin{array}{lll}
\{\!\{\mathbf{0}\}\!\} = \mathbf{0} & \{\!\{n.P\}\!\} = \mathit{open}\ n.\{\!\{P\}\!\} & \{\!\{\overline{n}\}\!\} = n[\mathbf{0}] \\
\{\!\{P \mid Q\}\!\} = \{\!\{P\}\!\} \mid \{\!\{Q\}\!\} & \{\!\{(\nu n)P\}\!\} = (\nu n)\{\!\{P\}\!\} & \\
\{\!\{X\}\!\} = X & \{\!\{\mathit{rec}\,X.P\}\!\} = \mathit{rec}\,X.\{\!\{P\}\!\} &
\end{array}
$$

Observe that the function $\{\!\{\ \}\!\}$ has as codomain the fragment of MA_r^{-mv} in which ambients have always an empty content $\mathbf{0}$*.*
Let P and Q be processes of aCCS; we have that

- *$P \equiv Q$ if and only if $\{\!\{P\}\!\} \equiv \{\!\{Q\}\!\}$;*
- *$P \to Q$ if and only if $\{\!\{P\}\!\} \to \{\!\{Q\}\!\}$.*

Proof:
The first item is proved by induction on the proof of $P \equiv Q$, while the second by induction on the length of the derivation of $P \to Q$. ■

This proves the first result of this subsection, i.e., that aCCS is essentially a fragment of MA_r^{-mv}. The remainder of this section is devoted to prove that termination is undecidable in aCCS, thus the same holds also in MA_r^{-mv} as a consequence of the following Corollary.

Corollary 3.3 *Let P be a process of aCCS, then P terminates if and only if $\{\!\{P\}\!\}$ terminates.*

Proof:
Direct consequence of the second item of Proposition 3.2. ■

3.2.2 Input guarded choice

In our modelling of RAMs we need to define terms that can have two different alternative continuations. For instance, a *DecJump* instruction may either activate the following instruction or jump. To model this behaviour, we use an extended syntax that we call aCCS+ which comprises also a choice operator guarded on inputs, and we define an encoding function that reduces terms of the extended syntax into processes of the original aCCS calculus.

Definition 3.4 *Consider the processes defined by the grammar of Definition 3.1 extended by adding a new form of process:*

$$P \ ::= \ \ldots \mid n.P + n.P$$

To each process P of aCCS+ we associate a process $[\![P]\!]$ of aCCS where $[\![P]\!]$

is defined inductively on the structure of P *as an homomorphism for all the operators (e.g.* $[\![\mathbf{0}]\!] = \mathbf{0}$ *and* $[\![P|Q]\!] = [\![P]\!] \mid [\![Q]\!]$*) but for the new operator* $n.P + m.Q$*:*

$$\begin{aligned}[\![n.P + m.Q]\!] \;=\; & (\nu ok)(\nu ko_n)(\nu ko_m)(\nu ack) \;\; (\; \overline{ok} \mid \\ & \qquad n.(\; ok.(\overline{m} \mid \overline{ko_m} \mid ack.[\![P]\!]) \mid ko_n.\overline{ack} \;) \mid \\ & \qquad m.(\; ok.(\overline{n} \mid \overline{ko_n} \mid ack.[\![Q]\!]) \mid ko_m.\overline{ack} \;) \;)\end{aligned}$$

The idea behind our encoding of the choice operator is similar to previous proposals by Nestmann and Pierce [18]: the two alternative input operations n and m are activated concurrently and, in order to avoid the undesired activation of both subsequent continuations (i.e. the processes $[\![P]\!]$ and $[\![Q]\!]$), we impose mutual exclusion exploiting a shared output particle $\overline{ok}$ that must be consumed before activating any of the two continuations.

The main difference between our approach and that of [18] is that our encoding works fine only in case there are no concurrent processes willing to consume the same inputs tested by the choice operator as, e.g., in the process $n.P + m.Q \mid m.R$. This is not restrictive for our purposes because, as we will discuss in the following, our modelling of RAMs satisfies this assumption.

Our encoding, moreover, provides us the possibility to collect as garbage the pending branch of an already resolved choice before activating the selected continuation. The de-activation is achieved by producing the output particle $\overline{m}$ (resp. $\overline{n}$) before activating the continuation $[\![P]\!]$ (resp. $[\![Q]\!]$). The assumption about the absence of concurrent processes willing to consume $\overline{m}$ (resp. $\overline{n}$) ensures that the output particle is consumed by the pending branch.

Example 3.5 *In order to better understand how our encoding of the choice operator works, let us consider the following term of aCCS+:* $P = \overline{a} \mid a.\overline{c} + b.\overline{d}$*. We have that only the first alternative of the choice operator could succeed. Each computation of the corresponding process* $[\![P]\!]$ *of aCCS blocks reaching a term* Q *such that*

$$Q \equiv \overline{c} \mid (\nu ok)(\nu ko_a)(\nu ko_b)(\nu ack)(ko_a.\overline{ack} \mid ok.(\overline{a} \mid \overline{ko_a} \mid ack.\overline{d}))$$

Observe that the term guarded by the input prefix ok. is blocked, and for this reason it could be considered as garbage*. Indeed, it is composed by processes which are blocked on input operations executed on the restricted names* ko_a *and* ok*.*

In order to formally state the properties of our encoding of choice, we need to deal with the form of garbage described in the Example 3.5. To this aim we introduce an auxiliary congruence relation $\equiv_g$.

Definition 3.6 *Let $\equiv_g$ be the minimal congruence on the calculus aCCS satisfying the following laws:*

$$\textit{if } P \equiv Q \textit{ then } P \equiv_g Q \qquad (\nu n_1)\ldots(\nu n_l) \prod_{i\in 1\ldots l} n_i.P_i \equiv_g \mathbf{0}$$

Observe that the garbage described in the Example 3.5 is congruent to the empty process $\mathbf{0}$ w.r.t. the congruence relation $\equiv_g$. The following Proposition permits to abstract away from garbage as it states that two processes P and Q are indistinguishable in case $P \equiv_g Q$.

Proposition 3.7 *Let P and Q be processes of aCCS such that $P \equiv_g Q$. If $P \to P'$ then there exists Q' such that $Q \to Q'$ and $P' \equiv_g Q'$.*

Proof:
By induction on the derivation of the relation $P \equiv_g Q$. ■

We are now ready to formally state the correctness of our encoding.

Proposition 3.8 *Let P, Q and R be processes of aCCS+ such that the processes $[\![P \mid n.Q + m.R]\!]$, $[\![\overline{n} \mid P]\!]$ and $[\![\overline{m} \mid Q]\!]$ are terminated (i.e. they have no outgoing transitions). The following holds:*

- *for each S such that $[\![\overline{n} \mid P \mid n.Q + m.R]\!] \to^5 S$ then $S \equiv_g [\![P \mid Q]\!]$ and*
- *for each S such that $[\![\overline{m} \mid P \mid n.Q + m.R]\!] \to^5 S$ then $S \equiv_g [\![P \mid R]\!]$.*

Proof:
By observing that the process $[\![P]\!]$ cannot interfere with the other processes during the five transitions required to resolve the choice operator. ■

3.2.3 Encoding RAMs

The modelling of RAMs is based on an encoding function, which transforms instructions and registers independently. The main difficulty is in the modelling of the actual content of the registers: to represent the natural number n we use a nesting of n restrictions. To achieve this we exploit restriction inside recursively defined terms.

We write the encoding using the extended syntax of aCCS+, but we will prove the formal properties of the encoding taking into account the corresponding terms of aCCS. In the previous subsection, given a term P of aCCS+, we have used $[\![P]\!]$ to denote the corresponding term of aCCS. To lighten the notation, in this subsection we omit the double square brackets $[\![\,]\!]$; in other words we write the processes using the aCCS+ syntax while we implicitly mean the corresponding aCCS terms.

Definition 3.9 *Consider a RAM R with instructions $(1 : I_1)$, …, $(m : I_m)$ and registers r_1, …, r_n; we encode a state $(i, c_1, \ldots, c_n)$ as follows:*

$$\begin{aligned}\llbracket (i, c_1, \ldots, c_n) \rrbracket_R = \overline{p_i} \mid \llbracket (1 : I_1) \rrbracket_R \mid \ldots \mid \llbracket (m : I_m) \rrbracket_R \mid \\ \llbracket r_1 = c_1 \rrbracket_R \mid \ldots \mid \llbracket r_n = c_n \rrbracket_R\end{aligned}$$

The encoding for the instruction $(i : I_i)$ is as follows:

$$\begin{aligned}
\llbracket (i : Succ(r_j)) \rrbracket_R &= rec\, X.p_i.(\ \overline{inc_j} \mid ack_j.(\overline{p_{i+1}} \mid X)\ ) \\
\llbracket (i : DecJump(r_j, s)) \rrbracket_R &= rec\, X.p_i.(\ \overline{test_j} \mid \\
&\qquad\qquad zero_j.(\overline{p_s} \mid X) \ + \ dec_j.(\overline{p_{i+1}} \mid X)\ )
\end{aligned}$$

The modelling of the register r_j with content l is defined as follows:

$$\llbracket r_j = l \rrbracket_R \ = \ \begin{cases} Z_j & \textit{if } l = 0 \\ (\nu a)(R^j_{a,l-1} \mid a.(\overline{dec_j} \mid Z_j)) & \textit{otherwise} \end{cases}$$

where Z_j is a shorthand notation defined as follows:

$$Z_j = rec\, X.(\ test_j.(\overline{zero_j} \mid X) \ + \ inc_j.(\overline{ack_j} \mid (\nu a)(\ R^j_{a,0} \mid a.(\overline{dec_j} \mid X)))\ )$$

while $R^j_{a,l}$ is as follows:

$$\begin{aligned}
R^j_{a,0} &= A_j \\
R^j_{a,l} &= (\nu b)(S^j_{b,l-1} \mid b.(\overline{dec_j} \mid A_j)) \\
S^j_{b,0} &= B_j \\
S^j_{b,l} &= (\nu a)(R^j_{a,l-1} \mid a.(\overline{dec_j} \mid B_j))
\end{aligned}$$

where A_j and B_j are shorthand notations defined as follows:

$$\begin{aligned}
A_j &= rec\, Y.(\ test_j.\overline{a} \ + \ inc_j.(\overline{ack_j} \mid (\nu b)(B'_j \mid b.(\overline{dec_j} \mid Y)))\ ) \\
B'_j &= rec\, V.(\ test_j.\overline{b} \ + \ inc_j.(\overline{ack_j} \mid (\nu a)(Y \mid a.(\overline{dec_j} \mid V)))\ ) \\
B_j &= rec\, V.(\ test_j.\overline{b} \ + \ inc_j.(\overline{ack_j} \mid (\nu a)(A_j \mid a.(\overline{dec_j} \mid V)))\ )
\end{aligned}$$

Observe that B_j and B'_j differ only in the fact that B'_j has a free process variable Y, while B_j has the term A_j instead. This difference follows from the fact that B'_j is part of the definition of A_j which contains the binder for the variable Y; in this way the recursive definition of A_j is closed, thus correct. On the contrary, B_j is a process defined in terms of the (closed) process A_j (thus it is correct as well).

The encoding function produces the parallel composition of the message $\overline{p_i}$ modelling the program counter (it indicates that the next instruction to execute is I_i), plus a process for each instruction, and a process for each register.

The encoding for the instruction $(i : I_i)$ depends on the kind of instruction. Nevertheless, in both cases the considered process initially consumes the program counter $\overline{p_i}$, then it produces a message with the aim to update a register, it waits for an answer indicating that the register has been updated, and finally it produces the subsequent program counter.

Namely, the encoding of a $Succ(r_j)$ instruction produces the message $\overline{inc_j}$ representing a request for the increment of the register r_j; then it waits for an acknowledgement (the message $\overline{ack_j}$) indicating that the increment has been successfully executed.

On the other hand, an instruction $DecJump(r_j, s)$ produces the message $\overline{test_j}$, which is sent to the process modelling the register r_j. The register r_j reacts as follows: if the current content of r_j is zero, then the message $\overline{zero_j}$ is produced, otherwise the content of the register is decremented and the message $\overline{dec_j}$ is produced. The process modelling the $DecJump$ instruction exploits the choice operator in order to properly react to the two possible alternative results. In case the $zero_j$ input succeeds, the new program counter is $\overline{p_s}$, otherwise it is $\overline{p_{i+1}}$.

In the modelling of the register r_j the process Z_j is used to represent the register when it is empty. The process Z_j produces a message $\overline{zero_j}$ when it receives a request for decrement; on the other hand, in case of increment it produces the acknowledgement $\overline{ack_j}$ and becomes $[\![r_j = 1]\!]_R$.

The notation $R^j_{a,l-1}$ is used in the representation of the register r_j when it is not empty. This process is responsible for reacting to the increment and decrement requests sent to the register. Moreover, it has to reactivate Z_j when the register becomes empty again, e.g. as consequence of the execution of $l-1$ decrement operations. As a trigger to reactivate Z_j, the process $R^j_{a,l-1}$ exploits the output particle $\overline{a}$.

In the definition of $R^j_{a,l-1}$ we use several auxiliary notations: A_j which is responsible for the production of the trigger $\overline{a}$ when the register becomes empty again; $S^j_{b,l-1}$ which is symmetric w.r.t. $R^j_{a,l-1}$ because it uses the name

a instead of b and vice versa; B_j which is the symmetric w.r.t. A_j because produces b instead of a.

In order to better describe how the modelling of registers works we report an illustrative example.

Example 3.10 *Let us consider the following encoding:*

$$[\![r_j = 3]\!]_R \; = \; (\nu a) \Big((\nu b) \Big((\nu a) \Big(A_j \mid a.(\overline{dec_j} | B_j) \Big) \mid b.(\overline{dec_j} | A_j) \Big) \mid a.(\overline{dec_j} | Z_j) \Big)$$

Observe that we encode the number contained in the register using a corresponding number of nested restrictions. The unique active process is A_j inside the inner restriction; all other processes are blocked by input operations performed either on the name a or b. These processes will be activated in case the register will be decremented. The inner process A_j produces a message $\overline{a}$ when it receives a decrement request. The nesting of restrictions ensures that, on production of this message $\overline{a}$, the answer $\overline{dec}$ is produced and the process B_j is activated.

We are now ready to formally prove the correctness of our RAM modelling. We use two Propositions; the former states that at each computation step of a RAM, moving from state s to state s', there exists a corresponding non-empty sequence of reductions moving from the encoding of s to the encoding of s'.

Proposition 3.11 *Let R be a RAM with program $(1 : I_1), \ldots, (m : I_m)$ and state $(i, c_1, \ldots, c_n)$. If $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ then there exists P such that $[\![(i, c_1, \ldots, c_n)]\!]_R \rightarrow^+ P$ and $P \equiv_g [\![(i', c'_1, \ldots, c'_n)]\!]_R$.*

Proof:
We proceed by case analysis. There are four cases obtained as combination of two pairs of possible alternatives: the instruction I_i is either $Succ(r_j)$ or $DecJump(r_j, s)$, and the register r_j is either empty or non-empty.

We report only on the most significant cases.

- Assume that the instruction I_i is $DecJump(r_j, s)$ and that the register r_j is not empty. Let us consider the following computation of the process $[\![(i, c_1, \ldots, c_n)]\!]_R$.
  The program counter $\overline{p_i}$ is consumed by the process $[\![(i : DecJump(r_j, s)]\!]_R$. Thus two processes are spawned: the request $\overline{test_j}$ and a second process which waits for the answer that will be produced by the process $[\![r_j = c_j]\!]_R$. We have assumed that r_j is not empty, thus $c_j > 0$ and we have that

  $$[\![r_j = c_j]\!]_R = (\nu a)(R^j_{a,c_j-1} \mid a.(\overline{dec_j} \mid Z_j))$$

  We consider the inner term of the inductive definition of R^j_{a,c_j-1}: it could be either $R^j_{a,1}$ or $S^j_{b,1}$. We consider the first case (the second case is treated symmetrically):

$$R^j_{a,1} = (\nu b)(B_j \mid b.(\overline{dec_j} \mid A_j))$$

The term B_j consumes the message $\overline{test_j}$ and subsequently produces $\overline{b}$ (plus some garbage due to the choice operator). This message is consumed by the process $b.(\overline{dec_j}|A_j)$. At this point, the considered inner process becomes congruent, w.r.t. $\equiv_g$, to the term

$$\overline{dec_j} \mid (\nu b)A_j$$

It is necessary to consider the congruence relation $\equiv_g$ in order to abstract away from the garbage. The message $\overline{dec_j}$ is consumed by the second process that has been spawned by the process $[\![(i : DecJump(r_j, s)]\!]_R$. Before analysing the effect of this message consumption, consider the other process $(\nu b)A_j$; as b is not free in A_j we have that $(\nu b)A_j \equiv A_j$. Moreover, we have that A_j corresponds to $R^j_{a,0}$. Thus the inner term is no more $R^j_{a,1}$ but $R^j_{a,0}$, which implies that the whole process modelling the register r_j has become congruent, w.r.t. $\equiv_g$, to:

$$(\nu a)(R^j_{a,c_j-2} \mid a.(\overline{dec_j} \mid Z_j)) = [\![r_j = c_j - 1]\!]_R$$

We still have to consider the reaction to the consumption of the message $\overline{dec_j}$. The process consuming this message produces three processes: (i) the program counter $\overline{p_{i+1}}$, (ii) a new instance of the recursively defined term $[\![(i : DecJump(r_j, s))]\!]$, (iii) and some garbage due to the choice operator. At this point of the computation the overall process is structurally congruent, w.r.t. $\equiv_g$, to the encoding of the new state of the RAM R; indeed, the new program counter is $\overline{p_{i+1}}$ and the new process modelling the register r_j is $[\![r_j = c_j - 1]\!]_R$. ■

We now consider the second Proposition which states that if a process P representing a particular state of a RAM has an outgoing transition, i.e. $P \rightarrow P'$, then the RAM is not blocked and the process P' continues its computation eventually reaching a state that corresponds (up to the congruence relation $\equiv_g$) to the subsequent configuration of the RAM.

Proposition 3.12 *Let R be a RAM with program $(1 : I_1), \ldots, (m : I_m)$ and state $(i, c_1, \ldots, c_n)$. If the process $P = [\![(i, c_1, \ldots, c_n)]\!]_R$ has a reduction step $P \rightarrow P_1$, then for each maximal computation of P_1 (i.e. a sequence of transitions starting from P_1 that cannot be extended) there exists an intermediary term P_l of the computation, i.e. $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_l \rightarrow \ldots$, such that $P_l \equiv_g [\![(i', c'_1, \ldots, c'_n)]\!]_R$ and $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$.*

Proof:
Consider the first step $P \rightarrow P_1$. Analysing the structure of the term $P =$

$\llbracket(i, c_1, \ldots, c_n)\rrbracket_R$, we see that the first step corresponds to the consumption of the program counter $\overline{p_i}$. This implies that in the program of the RAM R there is an instruction $(i : I_i)$ that can execute in the state $(i, c_1, \ldots, c_n)$, thus there exist $i', c'_1, \ldots, c'_n$ such that $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$.

We proceed by case analysis considering the four cases obtained as combination of the possible instruction (I_i is either $Succ(r_j)$ or $DecJump(r_j, s)$) and the possible state of the register (r_j is either empty or non-empty). In all cases, it is possible to show that the computation proceeds deterministically (up to the structural congruence $\equiv$) until a new program counter (say $\overline{p_{i'}}$) is produced. Moreover, the term of the computation (say P') in which $\overline{p_{i'}}$ appears for the first time is congruent to $\llbracket(i', c'_1, \ldots, c'_n)\rrbracket_R$ w.r.t. the congruence $\equiv_g$; thus, P' satisfies the conditions imposed to P_l in the statement of the Proposition.

We report only the analysis of the case in which the instruction is $Succ(r_j)$ and r_j is empty. The other cases are treated similarly.

- As the i-th instruction is $(i : Succ(r_j))$ we have that

  $$\llbracket(i : Succ(r_j))\rrbracket_R = rec\, X.p_i.(\overline{inc_j} \mid ack_j.(\overline{p_{i+1}} \mid X))$$

  This process is the only one able to consume the program counter p_i; thus, we have that in P_1 this process becomes structurally congruent to $\overline{inc_j} \mid ack_j.(\overline{p_{i+1}} \mid \llbracket(i : Succ(r_j))\rrbracket_R)$.
  Looking at the structure of P_1, we see that every reduction $P_1 \rightarrow P_2$ involves the consumption of the message $\overline{inc_j}$. The message is consumed by the process

  $$\llbracket r_j = 0\rrbracket_R = Z_j$$

  The process Z_j contains a choice operator; thus, we have that the subsequent 4 steps of the computation ($P_2 \rightarrow \ldots \rightarrow P_6$) are the internal steps performed in order to simulate the choice operator. We have that in P_6, the process modelling the register r_j becomes structurally congruent to

  $$\begin{array}{l} \overline{ack_j} \mid (\nu a).(A_j \mid a.(\overline{dec_j} \mid Z_j)) \mid \\ (\nu ok)(\nu ko_{test_j})(\nu ko_{inc_j})(\nu ack)(ack.(\overline{zero_j} \mid Z_j) \mid ko_{inc_j}.\overline{ack}) \end{array}$$

  This term is composed of three processes. The first process is the acknowledgement $\overline{ack_j}$; the second process corresponds to $\llbracket r_j = 1\rrbracket$; the third one is garbage, i.e. it is congruent, w.r.t. $\equiv_g$, to the null process $\mathbf{0}$.
  By observing the structure of P_6, we see that in any possible subsequent reduction $P_6 \rightarrow P_7$ the message $\overline{ack_j}$ is consumed by the term that was produced by $\llbracket(i : Succ(r_j))\rrbracket_R$. As discussed above, this process is structurally congruent to $ack_j.(\overline{p_{i+1}} | \llbracket(i : Succ(r_j))\rrbracket_R)$, and after the consumption of $\overline{ack_j}$

it becomes (structurally congruent to) $\overline{p_{i+1}} | [\![(i : Succ(r_j))]\!]_R$.
At this point, it is enough to observe that P_7 is congruent, w.r.t. $\equiv_g$, to the process $[\![(i+1, c_1, \ldots, c_{j-1}, 1, c_{j+1}, \ldots, c_n)]\!]_R$. ■

The following is a direct corollary of Propositions 3.11 and 3.12.

Corollary 3.13 *Let R be a RAM with program $(1 : I_1), \ldots, (m : I_m)$. We have that a state $(i, c_1, \ldots, c_n)$ of R is terminated if and only if $[\![(i, c_1, \ldots, c_n)]\!]_R$ is terminated.*

We can now conclude with the Theorem which states that our modelling of RAMs preserves termination.

Theorem 3.14 *Let R be a RAM with program $(1 : I_1), \ldots, (m : I_m)$ and initial state $(i, c_1, \ldots, c_n)$. Then we have that the RAM R terminates if and only if the process $[\![(i, c_1, \ldots, c_n)]\!]_R$ terminates.*

Proof:
Let us suppose that the RAM R terminates. Thus, there exists the computation $(i, c_1, \ldots, c_n) \rightarrow_R \ldots \rightarrow_R (i', c'_1, \ldots, c'_n)$ where $(i', c'_1, \ldots, c'_n)$ is a terminated state. Assume that the number of steps $\rightarrow_R$ of the computation is k. We proceed by contraposition assuming that $[\![(i, c_1, \ldots, c_n)]\!]_R$ does not terminate, thus we have an infinite computation $[\![(i, c_1, \ldots, c_n)]\!]_R \rightarrow P_1 \rightarrow \ldots$. Applying k times Proposition 3.11 (and the Proposition 3.7) we see that for each of these infinite computations there exists an intermediary state $P_{k'}$ such that $P_{k'} \equiv_g [\![(i', c'_1, \ldots, c'_n)]\!]_R$. The process $P_{k'}$ cannot be terminated as it is an intermediary process of an infinite computation. By the Corollary 3.13 (and the Proposition 3.7), then also $[\![(i', c'_1, \ldots, c'_n)]\!]$ is terminated, but this contrasts the hypothesis.

Now, suppose that $[\![(i, c_1, \ldots, c_n)]\!]_R$ terminates. We proceed by contraposition also in this case, assuming that the RAM R does not terminate, i.e. it has an infinite computation. By applying the Proposition 3.11 (and the Proposition 3.7) we see that also $[\![(i, c_1, \ldots, c_n)]\!]_R$ has an infinite computation; but this clearly contrasts the hypothesis. ■

The undecidability of process termination in the calculus MA_r^{-mv} is a direct consequence of Theorem 3.14 and Corollary 3.3.

3.3 Modelling RAMs in $MA_!^{-\nu}$

In the modelling of RAMs presented in the previous subsection, the key idea was to model natural numbers with a corresponding nesting of restrictions. Here, we consider a fragment without restriction, and we present an alternative modelling of the RAMs that exploits nested ambients instead. Moreover, the new modelling exploits process replication instead of recursive process definition.

Definition 3.15 *Consider a RAM R with instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. We define the encoding as follows:*

$$\begin{aligned}[\![(i, c_1, \ldots, c_n)]\!]^R = {} & \mathit{open}\ p_i \mid [\![(1 : I_1)]\!]^R \mid \ldots \mid [\![(m : I_m)]\!]^R \mid \\ & [\![r_1 = c_1]\!]^R \mid \ldots \mid [\![r_n = c_n]\!]^R \mid !\mathit{open}\ \mathit{msg} \mid \mathit{garbage}[]\end{aligned}$$

The encoding for registers $[\![r_j = l]\!]^R$ is defined inductively:

$$\begin{aligned}[\![r_j = 0]\!]^R = {} & \mathit{zero}_j[\ !\mathit{open}\ \mathit{increq}0_j.(\mathit{msg}[\ \mathit{out}\ \mathit{zero}_j.s_j[\ \mathit{REG}_j\ ]\ ] \mid \\ & \qquad \mathit{in}\ s_j.\mathit{incack}_j[\ \mathit{out}\ \mathit{zero}_j.!\mathit{out}\ s_j]\ ) \mid \\ & !\mathit{open}\ \mathit{zeroreq}_j.\mathit{okzero}_j[\ \mathit{out}\ \mathit{zero}_j.\mathit{in}\ \mathit{dj}_j\ ]\ ] \\ [\![r_j = l+1]\!]^R = {} & s_j[\ \mathit{REG}_j \mid [\![r_j = l]\!]^R\ ]\end{aligned}$$

where REG_j is a shorthand notation defined as follows:

$$\begin{aligned}\mathit{REG}_j = {} & \mathit{open}\ \mathit{decreq}_j.\mathit{okdec}_j[\ \mathit{out}\ s_j.\mathit{in}\ \mathit{dj}_j\ ] \mid \\ & !\mathit{open}\ \mathit{msg}\end{aligned}$$

If the i-th instruction is $\mathit{Succ}(r_j)$, its encoding is:

$$\begin{aligned}[\![(i : \mathit{Succ}(r_j))]\!]^R = {} & !p_i[\ \mathit{increq}_j[\ !\mathit{in}\ s_j \mid \mathit{in}\ \mathit{zero}_j.\mathit{increq}0_j[\mathit{out}\ \mathit{increq}_j\ ]\ ] \mid \\ & \mathit{open}\ \mathit{incack}_j.\mathit{open}\ p_{i+1}\ ]\end{aligned}$$

On the other hand, if the i-th instruction is $\mathit{DecJump}(r_j, s)$ the encoding is as follows:

$$\begin{aligned}[\![(i : \mathit{DecJump}(r_j, s))]\!]^R = {} & !p_i[\ \mathit{zeroreq}_j[\ \mathit{in}\ \mathit{zero}_j\ ] \mid \mathit{decreq}_j[\ \mathit{in}\ s_j\ ] \mid \\ & \mathit{dj}_j[\ \mathit{DJ}_{ijs}\ ]\ ]\end{aligned}$$

where

$$DJ_{ijs} = open\ okzero_j.in\ garbage.msg[\ out\ dj_j.out\ garbage.$$
$$open\ decreq_j.open\ p_s\ ]\ |$$
$$open\ okdec_j.in\ garbage.msg[\ out\ dj_j.out\ garbage.open\ zeroreq_j.$$
$$open\ s_j.open\ p_{i+1}\ ]$$

We model each process instruction $(i : I_i)$ with a replicated ambient $!p_i[P_i]$ where P_i is a process modelling the instruction I_i. The ambient is replicated in order to permit an unbounded amount of activations of the process P_i. Each activation is obtained simply by dissolving one of the boundaries p_i. For this reason, the program counter is modelled by a process *open* p_i.

The register r_j is modelled either with an ambient named $zero_j$ (when it is empty) or with an ambient named s_j (when not empty). The idea is to use a nesting of n ambients named s_j when the actual contents is n. As an example, consider the modelling of the register r_j with content 2:

$$[\![r_j = 2]\!]^R \ = \ s_j[\ REG_j \mid s_j[\ REG_j \mid [\![r_j = 0]\!]^R\ ]\ ]$$

Observe that we use a nesting of two ambients s_j; in the inner ambient (see the definition of $[\![r_j = 0]\!]^R$) we can find another ambient named $zero_j$. We will discuss the behaviour of the term REG_j, and of the processes inside the ambient $zero_j$, after having discussed the encoding function for the instructions.

If the i-th instruction is $Succ(r_j)$ the ambient p_i contains two processes. The first process is the ambient $increq_j$ that represents a request for increment of the register r_j. The second process blocks waiting for an acknowledgement that will be produced by the process modelling the register r_j; after the acknowledgement is received, this process increments the program counter spawning *open* p_{i+1}.

The ambient $increq_j$ has the ability to enter the boundary of the ambient modelling the register r_j, to move through the nesting of ambients, and finally to enter the inner ambient $zero_j$. After that, a new ambient $increq0_j$ exits the ambient $increq_j$ becoming in parallel with the processes of the ambient $zero_j$. One of these processes detects the arrival of the new ambient and reacts by producing an ambient $s_j[REG_j]$: the ambient $zero_j$ then moves inside this new ambient. In this way the nesting of ambients s_j is incremented by one. Moreover, the acknowledgement is produced in terms of an ambient named $incack_j$ that moves outside the register boundary.

On the other hand, if the i-th instruction is $DecJump(r_j, s)$ the ambient p_i contains three processes. The first process is an ambient named $zeroreq_j$ which represents a request for a test for zero of the register r_j; the second process is an ambient named $decreq_j$ representing a request for decrement of the register r_j; the third process is an ambient named dj_j which is in charge to manage the acknowledgement that the process modelling the register r_j will produce. The acknowledgement indicates whether the decrement, or the test for zero request, has succeeded.

Let us consider the test for zero request. The ambient $zeroreq_j[in\ zero_j]$ has the ability to move inside the ambient $zero_j$. This can occur only if the register r_j is currently empty; in fact, if r_j is not empty, the ambient $zero_j$ is not at the outer level. If the request enters the register boundary, the processes inside the ambient $zero_j$ react by producing an acknowledgement. The acknowledgement is modelled by an ambient named $okzero_j$ which moves inside the ambient dj_j.

Now, consider the request for decrement. The ambient $decreq_j[in\ s_j]$ has the ability to enter the boundary of the process modelling the register r_j; this can occur only if the register is not empty (otherwise there is no ambient s_j). Inside the ambient s_j, the process REG_j reacts by producing an acknowledgement. The acknowledgement is modelled by an ambient named $okdec_j$ which moves inside the ambient dj_j.

The process DJ_{ijs} inside the ambient dj_j has the ability to detect which kind of acknowledgement has been produced, and reacts accordingly. In case of $okzero_j$, the reaction is to move the ambient dj_j inside the ambient $garbage$, and to dissolve the boundary of the outer ambient $decreq_j$. This is necessary to remove the decrement request that has failed. In case of $okdec_j$, the process also removes one of the boundaries s_j, in order to actually decrement the register. In both cases, the program counter is finally updated by either jumping to instruction s, or by activating the next instruction $i+1$, respectively.

Example 3.16 *In this example, we sometime substitute ... for processes which are not relevant for the reported discussion. Let us consider the RAM R with the program* $(1 : Succ(r_1))$, $(2 : DecJump(r_1, 1))$. *We have that*

$$
\begin{aligned}
[\![(1,0)]\!]^R = {} & open\ p_1 \mid !open\ msg \mid garbage[] \mid \\
& \quad !p_1[\ increq_1[\ !in\ s_1 \mid in\ zero_1.increq0_1[out\ increq_1\ ]\ ] \mid \\
& \quad\quad open\ incack_1.open\ p_2\ ] \mid \\
& \quad !p_2[\ \ldots\ ] \mid \\
& \quad zero_1[\ !open\ increq0_1.(msg[\ out\ zero_1.s_1[\ REG_1\ ]\ ] \mid \\
& \qquad\qquad\qquad in\ s_1.incack_1[\ out\ zero_1.!out\ s_1]\ ) \mid \\
& \quad\quad \ldots\ ]
\end{aligned}
$$

The execution of the first instruction requires 9 reduction steps, after which the process becomes (structurally congruent to):

$open\ p_2 \mid !open\ msg \mid garbage[] \mid !out\ s_1 \mid$

$!p_1[\ \dots\ ] \mid$

$!p_2[\ zeroreq_1[\ in\ zero_1\ ] \mid decreq_1[\ in\ s_1\ ] \mid dj_1[\ DJ_{211}\ ]\ ] \mid$

$s_1[\ REG_1 \mid zero_1[\ \dots \mid$

$increq_1[!in\ s_1]\ ]\ ]$

Observe the processes $!out\ s_1$ *(at the outer level) and* $increq_1[!in\ s_1]$ *(inside the ambient* $zero_1$*) which are produced during the computation. These processes will not be involved in any possible subsequent computation step; for this reason, we refer to them as* garbage. *At this point of the computation, the second instruction can execute; this requires 12 steps, after which the process becomes (structurally congruent to):*

$open\ p_3 \mid !open\ msg \mid !out\ s_1 \mid in\ zero_1 \mid !open\ msg \mid$

$garbage[\ dj_1[\ \dots\ ]\ ] \mid$

$!p_1[\ \dots\ ] \mid !p_2[\ \dots\ ] \mid$

$zero_1[\ \dots \mid increq_1[!in\ s_1]\ ]$

At this point the computation blocks. Observe that other garbage is produced, namely the processes in $zero_1$ *and* $!open\ msg$ *at the outer level, and the ambient* dj_1 *inside the ambient* $garbage$*.*

In order to formally prove the correctness of our RAM modelling, we need to introduce some further notation used to manage the garbage, i.e. processes which are left by executed instructions that do not influence the subsequent computation. There are several forms of garbage which are left:

- each increment operation leaves an ambient $increq_j[!in\ s_j]$ inside the ambient $zero_j$, plus the process $!out\ s_j$ at the outer level;
- each decrement operation leaves an ambient dj_j inside the ambient $garbage$, plus the two processes $in\ zero_j$ and $!open\ msg$ at the outer level;
- each test for zero operation leaves an ambient dj_j inside the ambient $garbage$, plus the process $in\ s_j$ at the outer level.

Definition 3.17 *We use the notation* $[\![(i, c_1, \dots, c_n)]\!]^R_{i_1,\dots,i_n,d_1,\dots,d_n,z_1,\dots,z_n}$ *to denote the encoding of the state* $(i, c_1, \dots, c_n)$*, assuming that this state is reached after that the RAM computation has performed, on each register* r_j*,* i_j *increments,* d_j *decrements, and* z_j *test for zero operations.*

This new notation is defined as follows:

$$
\begin{array}{l}
[\![(i,c_1,\ldots,c_n)]\!]^R_{i_1,\ldots,i_n,d_1,\ldots,d_n,z_1,\ldots,z_n} \;\; = \\
\mathit{open}\ p_i \mid [\![(1:I_1)]\!]^R \mid \ \ldots\ \mid [\![(m:I_m)]\!]^R \mid [\![r_1=c_1]\!]^R_{i_1} \mid \ \ldots\ \mid [\![r_n=c_n]\!]^R_{i_n} \mid \\
!\mathit{open}\ \mathit{msg} \mid \prod_{j\in 1\ldots n}\prod_{i_j} !\mathit{out}\ s_j \mid \prod_{j\in 1\ldots n}\prod_{d_j}(\mathit{in}\ \mathit{zero}_j \mid !\mathit{open}\ \mathit{msg}) \mid \\
\prod_{j\in 1\ldots n}\prod_{z_j} \mathit{in}\ s_j \mid \\
\mathit{garbage}[\ \prod_{j\in 1\ldots n}\prod_{d_j} dj_j[\ \mathit{open}\ \mathit{okzero}_j.\mathit{in}\ \mathit{garbage}. \\
\qquad\qquad\qquad\qquad \mathit{msg}[\ \mathit{out}\ dj_j.\mathit{out}\ \mathit{garbage}.\mathit{open}\ \mathit{decreq}_j.\mathit{open}\ p_s\ ]\ ] \mid \\
\qquad\quad \prod_{j\in 1\ldots n}\prod_{z_j} dj_j[\ \mathit{open}\ \mathit{okdec}_j.\mathit{in}\ \mathit{garbage}. \\
\qquad\qquad\qquad\qquad \mathit{msg}[\ \mathit{out}\ dj_j.\mathit{out}\ \mathit{garbage}.\mathit{open}\ \mathit{zeroreq}_j. \\
\qquad\qquad\qquad\qquad\quad \mathit{open}\ s_j.\mathit{open}\ p_{i+1}\ ]\ ]\ ]
\end{array}
$$

The above definition uses a new encoding for the registers; the new encoding considers the garbage left inside the ambients $zero_j$:

$$
\begin{array}{l}
[\![r_j=0]\!]^R_{i_j} = \mathit{zero}_j[\ !\mathit{open}\ \mathit{increq}0_j.(\mathit{msg}[\ \mathit{out}\ \mathit{zero}_j.s_i[\ \mathit{REG}_j\ ]\ ] \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ s_j.\mathit{incack}_j[\ \mathit{out}\ \mathit{zero}_j.!\mathit{out}\ s_j\ ]) \mid \\
\qquad\qquad\qquad !\mathit{open}\ \mathit{zeroreq}_j.\mathit{okzero}_j[\ \mathit{out}\ \mathit{zero}_j.\mathit{in}\ dj_j\ ] \mid \\
\qquad\qquad\qquad \prod_{i_j} \mathit{increq}[!\mathit{in}\ s_j]\ ] \\
[\![r_j=l+1]\!]^R_{i_j} = s_j[\ \mathit{REG}_j \mid [\![r_j=l]\!]^R_{i_j}\ ]
\end{array}
$$

It is worth to observe that $[\![(i,c_1,\ldots,c_n)]\!]^R = [\![(i,c_1,\ldots,c_n)]\!]^R_{0,\ldots,0}$; in fact, the two encodings differ only in the garbage and, at the beginning of the computation, no garbage is present.

We are now ready to prove the correctness of our RAM modelling. As for the previous subsection, we consider two separated Propositions. We do not report the proofs because they are similar to the proofs of the corresponding Propositions in the previous subsection.

Proposition 3.18 *Let R be a RAM with program $(1:I_1),\ldots,(m:I_m)$ and state $(i,c_1,\ldots,c_n)$. If $(i,c_1,\ldots,c_n)\rightarrow_R(i',c'_1,\ldots,c'_n)$ then for any $P=[\![(i,c_1,\ldots,c_n)]\!]^R_{i_1,\ldots,i_n,d_1,\ldots,d_n,z_1,\ldots,z_n}$ there exist Q and $i'_1,\ldots,i'_n,d'_1,\ldots,d'_n,z'_1,\ldots,z'_n$ such that $P\rightarrow^+Q$ and $Q\equiv[\![(i',c'_1,\ldots,c'_n)]\!]^R_{i'_1,\ldots,i'_n,d'_1,\ldots,d'_n,z'_1,\ldots,z'_n}$.*

Proposition 3.19 *Let R be a RAM with program $(1:I_1),\ldots,(m:I_m)$ and state $(i,c_1,\ldots,c_n)$. If the process $P=[\![(i,c_1,\ldots,c_n)]\!]^R_{i_1,\ldots,i_n,d_1,\ldots,d_n,z_1,\ldots,z_n}$ has a reduction step $P\rightarrow P_1$, then for each maximal computation of P_1 there exist $i'_1,\ldots,i'_n,d'_1,\ldots,d'_n,z'_1,\ldots,z'_n$ and an intermediary term P_l of the computa-*

tion, i.e. $P \rightarrow P_1 \rightarrow \ldots \rightarrow P_l \rightarrow \ldots$*, s.t.* $P_l \equiv [\![(i', c'_1, \ldots, c'_n)]\!]^R_{i'_1,\ldots,i'_n,d'_1,\ldots,d'_n,z'_1,\ldots,z'_n}$ *and* $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$*.*

In the same way as in the previous subsection, the following consequences of the Propositions 3.18 and 3.19 can be proved.

Corollary 3.20 *Let* R *be a RAM with program* $(1 : I_1), \ldots, (m : I_m)$*. We have that a state* $(i, c_1, \ldots, c_n)$ *of* R *is terminated if and only if* $[\![(i, c_1, \ldots, c_n)]\!]^R_{i_1,\ldots,i_n,d_1,\ldots,d_n,z_1,\ldots,z_n}$ *is terminated for any* $i_1, \ldots, i_n, d_1, \ldots, d_n, z_1, \ldots, z_n$*.*

Theorem 3.21 *Let* R *be a RAM with program* $(1 : I_1), \ldots, (m : I_m)$ *and initial state* $(i, c_1, \ldots, c_n)$*. Then we have that the RAM* R *terminates if and only if the process* $[\![(i, c_1, \ldots, c_n)]\!]^R$ *terminates.*

The undecidability of process termination in the calculus $\mathrm{MA}_!^{-\nu}$ is a direct consequence of the Theorem 3.21.

4 Decidability Results

In this section we show that termination is decidable for the fragments $\mathrm{MA}_r^{-mv,\nu}$ and $\mathrm{MA}_!^{-mv}$.

These results are based on the theory of well-structured transition systems [10]: the existence of an infinite computation starting from a given state is decidable for finitely branching transition systems, provided that the set of states can be equipped with a well-quasi-ordering, i.e., a quasi-ordering relation which is compatible with the transition relation and such that each infinite sequence of states admits an increasing subsequence.

We start by providing some basic definitions and results on well-structured transition systems and on well-quasi-ordering on sequences of elements belonging to a well-quasi-ordered set, that will be used in the following parts of this Section.

Then, we prove the decidability of termination for the calculus $\mathrm{MA}_!^{-mv}$: to this aim, we first provide an alternative labelled semantics that is equivalent w.r.t. termination to the one presented in Section 2, but which is based on a finitely branching transition system and permits to define a well-quasi-ordering on the derivatives of a given process (i.e., the set of processes reachable from a given initial process). Then, by exploiting the theory developed in [10], we show that termination is decidable for $\mathrm{MA}_!^{-mv}$ processes.

The decidability of termination for $\mathrm{MA}_r^{-mv,\nu}$ is obtained adapting the previous proof to a slightly more general calculus, obtained by extending $\mathrm{MA}_!^{-mv}$ with a limited form of recursion. If we consider the proof of Turing completeness for MA_r^{-mv} presented in Subsection 3.2, we note that the natural numbers contained in registers are modelled through sequences of nested restrictions. We can prevent the possibility to produce an unbounded number of nested restrictions by adopting *unrestricted recursion* defined as follows.

Definition 4.1 *Given a term P containing some recursively defined processes $rec\, X.Q$, we say that P adopts* unrestricted recursion *if for each subprocess $rec\, X.Q$ no free occurrence of X appears inside a subprocess of Q of the form $(\nu n)R$. We say that P adopts* guarded recursion *if for each subprocess $rec\, X.Q$ each free occurrence of X inside Q appears in prefixed processes of the form $M.P$.*
Let $MA_{!,uur}^{-mv}$ be the calculus obtained by extending $MA_!^{-mv}$ with unboxed and unrestricted recursion and let $MA_{!,uugr}^{-mv}$ be the calculus obtained by extending $MA_!^{-mv}$ with unboxed, unrestricted and guarded recursion.

The proof of decidability of termination for the calculus obtained by extending $\mathrm{MA}_!^{-mv}$ with unboxed and unrestricted recursion (called henceforth $\mathrm{MA}_{!,uur}^{-mv}$) is split in two parts. We show that the proof of decidability for $\mathrm{MA}_!^{-mv}$ can be adapted to $\mathrm{MA}_{!,uugr}^{-mv}$. Then, decidability of termination in $\mathrm{MA}_{!,uur}^{-mv}$ is obtained by providing a termination preserving transformation of unguarded recursion in a combination of guarded recursion and replication.

4.1 Well-Structured Transition System

We start by recalling some basic definitions and results from [10], concerning well-structured transition systems, that will be used in the following.

A *quasi-ordering* (qo) is a reflexive and transitive relation.

Definition 4.2 *A* well-quasi-ordering *(wqo) is a quasi-ordering $\leq$ over a set X such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leq x_j$.*

Note that, if $\leq$ is a wqo, then any infinite sequence $x_0, x_1, x_2, \ldots$ contains an infinite increasing subsequence $x_{i_0}, x_{i_1}, x_{i_2}, \ldots$ (with $i_0 < i_1 < i_2 < \ldots$).

Transition systems can be formally defined as follows.

Definition 4.3 *A* transition system *is a structure $TS = (S, \rightarrow)$, where S is a set of* states *and $\rightarrow \subseteq S \times S$ is a set of* transitions.
We write $Succ(s)$ to denote the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of S.

TS *is* finitely branching *if all* $Succ(s)$ *are finite. We restrict to finitely branching transition systems.*

Well-structured transition systems, defined as follows, provide the key tool to decide properties of computations.

Definition 4.4 *A* well-structured transition system (with strong compatibility) *is a transition system* $TS = (S, \rightarrow)$*, equipped with a quasi-ordering* $\leq$ *on* S*, also written* $TS = (S, \rightarrow, \leq)$*, such that the two following conditions hold:*

(1) **well-quasi-ordering***:* $\leq$ *is a well-quasi-ordering, and*
(2) **strong compatibility***:* $\leq$ *is (upward) compatible with* $\rightarrow$*, i.e., for all* $s_1 \leq t_1$ *and all transitions* $s_1 \rightarrow s_2$*, there exists a state* t_2 *such that* $t_1 \rightarrow t_2$ *and* $s_2 \leq t_2$*.*

The following theorem (a special case of a result in [10]) will be used to obtain our decidability result.

Theorem 4.5 *Let* $TS = (S, \rightarrow, \leq)$ *be a finitely branching, well-structured transition system with decidable* $\leq$ *and computable* $Succ$*. The existence of an infinite computation starting from a state* $s \in S$ *is decidable.*

4.2 Higman's Lemma

To prove that the quasi-ordering relation we will define on $\mathrm{MA}_!^{-mv}$ processes is a well-quasi-ordering we need the following result, due to Higman [11] and stating that the set of the finite sequences over a set equipped with a wqo is well-quasi-ordered.

Given a set S, with S^* we denote the set of finite sequences of elements in S.

Definition 4.6 *Let* S *be a set and* $\leq$ *a wqo over* S*. The relation* $\leq_*$ *over* S^* *is defined as follows. Let* $t, u \in S^*$*, with* $t = t_1 t_2 \ldots t_m$ *and* $u = u_1 u_2 \ldots u_n$*. We have that* $t \leq_* u$ *iff there exists an injection* f *from* $\{1, 2, \ldots, m\}$ *to* $\{1, 2, \ldots, n\}$ *such that* $t_i \leq u_{f(i)}$ *and* $i \leq f(i)$ *for* $i = 1, \ldots, m$*.*

Note that relation $\leq_*$ is a preorder over S^*.

Theorem 4.7 *[Higman] Let* S *be a set and* $\leq$ *a wqo over* S*. Then, the relation* $\leq_*$ *is a wqo over* S^**.*

Also the following lemma will be used to prove that the relation on processes is a well-quasi-ordering:

Lemma 4.8 *Let* S *be a finite set. Then the equality is a wqo over* S*.*

4.3 Termination is decidable for $MA_!^{-mv}$

In this section we show that the existence of an infinite computation is decidable in $\mathrm{MA}_!^{-mv}$. To apply the Theorem 4.5 on decidability of termination to $\mathrm{MA}_!^{-mv}$ processes, we need a semantics that gives rise to a finitely branching transition system and for which it is possible to define a well-quasi-ordering on the set of processes reachable from a given process.

The idea is to define a wqo on processes by exploiting the results presented in Section 4.2. To this end, we note that each process P can be regarded as a finite sequence of objects of the following kind:

- elementary items, represented by sequential and replicated subprocesses, i.e., subprocesses of kind *open* $n.Q$ or $!Q$;
- composed items, represented by ambients and restricted processes, i.e., items of the form $n[R]$ or $(\nu n)R$, where R is any process.

If we consider the level of nesting of ambients and restrictions, we have that each process R contained in a composed item of P is simpler than P, in the sense that the nesting level of ambients and restrictions in R is strictly smaller than the nesting level in P.

As far as ambients are concerned, we note that, because of the absence of primitives for movement in $\mathrm{MA}_!^{-mv}$, the level of nesting of ambients cannot increase during the computation; hence, there is an upper bound to the nesting level of ambients in the set of the derivatives of a process. Because of the structural congruence rules, the sentence above is not true for restriction.

In fact, if we consider process

$$P = m[] \mid !(\nu n)(\mathit{open}\ m.(m[] \mid Q))$$

by application of the structural congruence rules it is possible to obtain a structurally equivalent process containing any number of nested restrictions

$$\begin{aligned} P &\equiv (\nu n)(m[] \mid \mathit{open}\ m.(m[] \mid Q)) \mid !(\nu n)(\mathit{open}\ m.(m[] \mid Q)) \\ &\equiv (\nu n)((\nu n)(m[] \mid \mathit{open}\ m.(m[] \mid Q)) \mid \mathit{open}\ m.(m[] \mid Q)) \mid \\ &\quad\ !(\nu n)(\mathit{open}\ m.(m[] \mid Q)) \\ &\equiv \ldots \end{aligned}$$

Indeed, the structural congruence rules

$$(\nu n)(P|Q) \equiv P|(\nu n)Q \qquad \text{if } n \notin fn(P)$$

and

$$!P \equiv P \mid !P$$

are necessary to permit the following reductions to take place:

$$\begin{aligned} P &\to (\nu n)(m[] \mid Q) \mid !(\nu n)(open\ m.(m[] \mid Q)) \\ &\to (\nu n)((\nu n)(Q) \mid m[] \mid Q) \mid !(\nu n)(open\ m.(m[] \mid Q)) \\ &\to \ldots \end{aligned}$$

However, if we move to a semantics based on a labelled transition system, the above rules are no longer necessary and we can obtain an upper bound also for the nesting level of restrictions in the set of derivatives of a process.

Summarising, each derivative of a process P is a sequence of elementary and composed items, and for each composed item $n[Q]$ (or $(\nu n)Q$) appearing in the sequence, the nesting level of restrictions and ambients in Q is strictly smaller than the nesting level in P.

Moreover, the adoption of a semantics based on labelled transition systems also permits to get rid of the structural congruence rule for alpha conversion. In this way, we have that the set of ambient names appearing in the derivatives of a process P is finite. As a consequence, also the set of elementary items that are subprocesses of some derivative of P is finite. The two facts above will be used to obtain the wqo on processes.

Then, we introduce a quasi-ordering $\preceq$ on processes, and we show that $\preceq$ is a wqo by using Higman's Lemma.

4.3.1 A finitely branching transition system for $MA_!^{-mv}$

We define a semantics for $\mathrm{MA}_!^{-mv}$ based on a labelled transition system.

There are three kinds of actions: with *open* n we denote the ability of a process to open an ambient with name n, whereas the coaction $\overline{open\ n}$ can be performed by a process containing an ambient named n that can be opened; finally, τ denotes an internal action. In the following we use α to range over *open* n, $\overline{open\ n}$, τ. With $n(\alpha)$ we denote the set of names appearing in a label α: $n(open\ m) = n(\overline{open\ m}) = \{m\}$ and $n(\tau) = \emptyset$.

The labelled transition relation is the least relation satisfying the following axioms and rules (we omit the symmetric versions of rules (4L) and (8L)):

$$(1L)\ open\ n.P \overset{open\ n}{\mapsto} P$$

$$(2L)\ n[P] \overset{\overline{open\ n}}{\mapsto} P$$

$$(3L)\ \frac{P \overset{\alpha}{\mapsto} P'}{P \mid Q \overset{\alpha}{\mapsto} P' \mid Q}$$

$$(4L)\ \frac{P \overset{open\ n}{\mapsto} P' \qquad Q \overset{\overline{open\ n}}{\mapsto} Q'}{P \mid Q \overset{\tau}{\mapsto} P' \mid Q'}$$

$$(5L)\ \frac{P \overset{\tau}{\mapsto} P'}{n[P] \overset{\tau}{\mapsto} n[P']}$$

$$(6L)\ \frac{P \overset{\alpha}{\mapsto} P'}{(\nu m)P \overset{\alpha}{\mapsto} (\nu m)P'} \qquad m \notin n(\alpha)$$

$$(7L)\ \frac{P \overset{\alpha}{\mapsto} P'}{!P \overset{\alpha}{\mapsto} P' \mid !P}$$

$$(8L)\ \frac{P \overset{open\ n}{\mapsto} P' \qquad P \overset{\overline{open\ n}}{\mapsto} P''}{!P \overset{\tau}{\mapsto} P' \mid P'' \mid !P}$$

The semantics of replication is defined by the rules $(7L)$ and $(8L)$. We do not adopt the standard rule

$$\frac{P \mid !P \overset{\alpha}{\mapsto} P'}{!P \overset{\alpha}{\mapsto} P'}$$

because we need to obtain a finitely branching transition system (For a deeper discussion of this issue in the π-calculus, see [20]).

The labelled transition system is finitely branching:

Proposition 4.9 *Let $P \in MA_!^{-mv}$. Then the set $\{Q \mid \exists \alpha : P \overset{\alpha}{\mapsto} Q\}$ is finite.*

Proof:
By induction on the structure of P. ■

The labelled semantics is equivalent to the reduction semantics presented in Section 2. Here we report only the final results. For a detailed version of the proof of a similar result, see e.g. the equivalence of labelled and reduction semantics in the π-calculus reported in [20].

Proposition 4.10 *Let $P \in MA_!^{-mv}$. If $P \overset{\tau}{\mapsto} P'$ then $P \to P'$.*

Proposition 4.11 *Let $P, Q \in MA_!^{-mv}$. If $P \equiv Q$ and $P \overset{\alpha}{\mapsto} P'$ then there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P' \equiv Q'$.*

Proposition 4.12 *Let $P \in MA_!^{-mv}$. If $P \to P'$ then there exists P'' such that $P \overset{\tau}{\mapsto} P''$ and $P'' \equiv P'$.*

As a consequence of Propositions 4.10 and 4.12, we have that a process P admits an infinite computation according to the reduction semantics if and only if it admits an infinite sequence of τ moves according to the labelled semantics. We say that P is terminating according to the labelled semantics if there exists no infinite sequence of τ moves starting from P.

Proposition 4.13 *Let $P \in MA_!^{-mv}$. Then P terminates according to the reduction semantics iff P terminates according to the labelled semantics.*

4.3.2 Decidability of termination for $(MA_!^{-mv}, \overset{\tau}{\mapsto})$

In this section we equip the transition system $(\mathrm{MA}_!^{-mv}, \overset{\tau}{\mapsto})$ with a preorder $\preceq$ on processes which turns out to be a well-quasi-ordering compatible with $\overset{\tau}{\mapsto}$. Thus, exploiting the results presented in subsection 4.1, we show that termination is decidable.

We introduce a quasi-order $\preceq$ such that $P \preceq Q$ if for each occurrence of an elementary item in P there exists a corresponding occurrence of the same elementary item in Q, whereas for each occurrence of a composed item $n[P']$ (resp. $(\nu n)P'$) in P there exists a corresponding item $n[Q']$ (resp. $(\nu n)Q'$) in Q such that $P' \preceq Q'$.

We note that the elementary items of any process reachable from a given process P are contained in the (clearly finite) set of subprocesses of the kind *open* $n.Q$ and $!Q$ of P. Thus, if we consider processes containing elementary items only, by result 4.8 we obtain that $\preceq$ is a wqo. For processes containing also composed items, the proof that $\preceq$ is a wqo proceeds by induction on the number of nested restrictions and ambients and by using Higman's Lemma.

With $Deriv(P)$ we denote the set of derivatives of a process P.

Definition 4.14 *Let $P \in MA_!^{-mv}$. With $Deriv(P)$ we denote the set of processes reachable from P with a sequence of τ steps:*

$$Deriv(P) = \{Q \mid P \overset{\tau}{\mapsto}^{*} Q\}$$

(where $\stackrel{\tau}{\mapsto}^*$ denotes the reflexive and transitive closure of $\stackrel{\tau}{\mapsto}$).

To define the qo relation on processes, and to show that this relation turns out to be a wqo, we will use the following equivalence, stating that $\mathrm{MA}_!^{-mv}$ with parallel composition is a commutative monoid, and that turns out to be compatible with $\stackrel{\alpha}{\mapsto}$.

Definition 4.15 *We define $=_|$ as the least equivalence relation satisfying the following axioms:*

$$P|Q =_| Q|P$$
$$P|(Q|R) =_| (P|Q)|R$$
$$P|0 =_| P$$

Proposition 4.16 *Let $P, Q \in MA_!^{-mv}$. If $P =_| Q$ and $Q \stackrel{\alpha}{\mapsto} Q'$ then there exists P' such that $P \stackrel{\alpha}{\mapsto} P'$ and $P' =_| Q'$.*

Definition 4.17 *Let $P, Q \in MA_!^{-mv}$. $P \preceq Q$ iff one of the following conditions holds:*

- *there exists R such that $Q =_| P \mid R$, or*
- *there exist n, P_1, P_2, Q_1, Q_2 such that $P_i \preceq Q_i$ for $i = 1, 2$ and one of the following holds:*
  · *either $P =_| P_1 \mid n[P_2]$ and $Q =_| Q_1 \mid n[Q_2]$*
  · *or $P =_| P_1 \mid (\nu n)P_2$ and $Q =_| Q_1 \mid (\nu n)Q_2$*

We introduce some auxiliary definitions that will be useful to prove that $\preceq$ is a wqo:

Definition 4.18 *Let $P \in MA_!^{-mv}$.*

With $d_a(P)$ we denote the maximum number of nested ambients and nested restrictions in process P:

$$d_a(\mathbf{0}) = 0$$
$$d_a(open\ n.P) = d_a(P)$$
$$d_a(P|Q) = max(\{d_a(P), d_a(Q)\})$$
$$d_a(n[P]) = 1 + d_a(P)$$
$$d_a((\nu n)P) = 1 + d_a(P)$$
$$d_a(!P) = d_a(P)$$

The set of sequential and replicated subprocesses of P is defined as:

$$
\begin{aligned}
&Sub(\mathbf{0}) = \emptyset \\
&Sub(open\ n.P) = \{open\ n.P\} \cup Sub(P) \\
&Sub(P|Q) = Sub(P) \cup Sub(Q) \\
&Sub(n[P]) = Sub(P) \\
&Sub((\nu n)P) = Sub(P) \\
&Sub(!P) = \{!P\} \cup Sub(P)
\end{aligned}
$$

The following definition characterizes a superset of the set of the derivatives of a process P. The set $\mathcal{P}_{P,k}$ will denote the set of $\text{MA}_!^{-mv}$ processes whose sequential and replicated subprocesses are contained in the corresponding elements of P. Moreover, the ambient names used in processes in $\mathcal{P}_{P,k}$ are a subset of the ambient names of P, and the nesting level of ambients and restrictions is not greater than k:

Definition 4.19 *Let $P \in MA_!^{-mv}$ and $k \geq 0$.*

$$\mathcal{P}_{P,k} = \{Q \in MA_!^{-mv} \mid Sub(Q) \subseteq Sub(P) \wedge n(Q) \subseteq n(P) \wedge d_a(Q) \leq k\}$$

We start showing that the relation $\preceq$ (defined in 4.17) is strongly compatible with $\overset{\alpha}{\mapsto}$. To this aim, we need the following auxiliary proposition:

Proposition 4.20 *Let $P_1, P_2, Q_1, Q_2, \in MA_!^{-mv}$. If $P_i \preceq Q_i$ for $i = 1, 2$ then $P_1|P_2 \preceq Q_1|Q_2$.*

Proof:
By induction on the sum of the number of parallel compositions in P_1 and in P_2. ■

Theorem 4.21 *Let $P, Q, P' \in MA_!^{-mv}$. If $P \overset{\alpha}{\mapsto} P'$ and $P \preceq Q$ then there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P' \preceq Q'$.*

Proof:
The proof is by induction on $d_a(P)$, and then by case analysis on the definition of $\preceq$. ■

Clearly the above result implies that $\preceq$ is also strongly compatible with $\overset{\tau}{\mapsto}$.

Note that $\preceq$ is a qo; in order to prove that $\preceq$ is also a well-quasi-ordering we need the following preliminary results on properties of the set of derivatives of a process.

Proposition 4.22 *Given a process $P \in MA_!^{-mv}$, the set $Sub(P)$ is finite.*

Proof:
By induction on the structure of P. ■

Proposition 4.23 *Let $P \in MA_!^{-mv}$ and $Q \in \mathcal{P}_{P,k}$. If $Q \overset{\alpha}{\mapsto} Q'$ then $Q' \in \mathcal{P}_{P,k}$.*

Proof:
By induction on the derivation of the transition $Q \overset{\alpha}{\mapsto} Q'$. ■

An easy consequence of the proposition above is that all the derivatives of P belong to $\mathcal{P}_{P,d_a(P)}$:

Corollary 4.24 *Let $P \in MA_!^{-mv}$. We have that $Deriv(P) \subseteq \mathcal{P}_{P,d_a(P)}$.*

The following lemma shows that all processes in $\mathcal{P}_{P,n}$ can be written in a sort of normal form up to $=_|$.

Lemma 4.25 *Let $P \in MA_!^{-mv}$, $n \leq d_a(P)$ and $Q \in \mathcal{P}_{P,n}$. Suppose that $|n(P)| = m$ and $n(P) = \{n_1, \ldots, n_m\}$.*

Then there exist l, $k_1, \ldots, k_m$ such that

$$Q =_| \prod_{i=1}^{l} Q_i \ \mid \ \prod_{j=1}^{m} \prod_{h=1}^{k_j} n_j[R_{j,h}] \ \mid \ \prod_{j=1}^{m} \prod_{h=1}^{k'_j} (\nu n_j) R'_{j,h}$$

for some

$Q_i \in Sub(P)$ *for* $i = 1, \ldots, l$

$R_{j,h} \in \mathcal{P}_{P,n-1}$ *for* $j = 1, \ldots, m$ *and* $h = 1, \ldots, k_j$

$R'_{j,h} \in \mathcal{P}_{P,n-1}$ *for* $j = 1, \ldots, m$ *and* $h = 1, \ldots, k'_j$

Proof:
By induction on the structure of Q. ■

Now we are ready to prove that $\preceq$ is a wqo. The key idea is the following: we use Lemma 4.25 to transform each derivative of P in $1+2m$ (finite) sequences, where m is the cardinality of $n(P)$. The first sequence is over $Sub(P)$ which is a finite set, whereas the other sequences are over processes that are "simpler" than P, in the sense that the nesting level of ambients and restrictions in those processes is strictly smaller than $d_a(P)$. The result is proved by proceeding by induction on the nesting level of ambients and restrictions and using Higman's lemma. Given an infinite sequence of processes expressed in the normal form provided by Lemma 4.25, we start extracting a subsequence of processes whose first part (i.e., the sequence of processes over $Sub(P)$) is increasing. Then, from this new sequence we extract a subsequence of processes whose second part (i.e., the sequence of processes of kind $n_1[R_{1,h}]$) is increasing, and so on.

The following two propositions will be used to show that if the sequence of the subsequences corresponding (via Lemma 4.25) to an infinite sequence of processes are increasing, then the sequence of processes is itself increasing.

Proposition 4.26 *Let $P \in MA_!^{-mv}$ and $P_1, P_2, \ldots, P_m, Q_1, Q_2, \ldots, Q_n \in Sub(P)$. If $P_1P_2\ldots P_m =_* Q_1Q_2\ldots Q_n$ then $\prod_{i=1}^m P_i \preceq \prod_{i=1}^n Q_i$.*

Proof:
If $P_1P_2\ldots P_m =_* Q_1Q_2\ldots Q_n$ then there exists an injection $f : \{1,\ldots,m\} \to \{1,\ldots,n\}$ such that $P_i = Q_{f(i)}$. Denoting by $\underline{cod}(f)$ the set $\{i \mid \forall j : 1 \leq j \leq m \Rightarrow i \neq f(j)\}$, we have that $\prod_{i=1}^n Q_i =_| \prod_{i=1}^m Q_f(i) \mid \prod_{i\in\underline{cod}(f)} Q_i$, hence $\prod_{i=1}^n Q_i =_| \prod_{i=1}^m P_i \mid \prod_{i\in\underline{cod}(f)} Q_i$, thus also $\prod_{i=1}^m P_i \preceq \prod_{i=1}^n Q_i$. ∎

Proposition 4.27 *Let $P_1, P_2, \ldots, P_m, Q_1, Q_2, \ldots, Q_k \in MA_!^{-mv}$ and n be an ambient name. If $P_1P_2\ldots P_m \preceq_* Q_1Q_2\ldots Q_k$ then:*

- $\prod_{i=1}^m n[P_i] \preceq \prod_{i=1}^k n[Q_i]$, *and*
- $\prod_{i=1}^m (\nu n)P_i \preceq \prod_{i=1}^k (\nu n)Q_i$.

Proof:
The proof is similar to the proof of Proposition 4.26.

Now we are ready to show that $\preceq$ is a well-quasi-ordering:

Theorem 4.28 *Let $P \in MA_!^{-mv}$ and $n \geq 0$. The relation $\preceq$ is a wqo over $\mathcal{P}_{P,n}$.*

Proof:
The proof is by induction on n.

Let $n = 0$.
Take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathcal{P}_{P,0}$ for $i > 0$.
By Lemma 4.25, for any i we have that $P_i =_| \prod_{j=1}^{n_i} P_{i,j}$, with $P_{i,j} \in Sub(P)$.
Hence, we have an infinite sequence of elements of $Sub(P)^*$; as $Sub(P)$ is finite (by Proposition 4.22), by Lemma 4.8 and Higman's Lemma (4.7) we have that $=_*$ is a wqo over $Sub(P)^*$.
Hence, there exist i, k such that $P_{i,1}P_{i,2}\ldots P_{i,n_i} =_* P_{k,1}P_{k,2}\ldots P_{k,n_k}$. By Proposition 4.26 we have $P_i \preceq P_k$.

For the inductive step, let $n > 0$ and take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathcal{P}_{P,n}$ for $i > 0$.
By Lemma 4.25, there exists m such that, for any i:

$P_i =_{|} \prod_{j=1}^{h_i} P_{i,j} | \prod_{j=1}^{m} \prod_{h=1}^{k_{i,j}} n_j[R_{i,j,h}] | \prod_{j=1}^{m} \prod_{h=1}^{k'_{i,j}} (\nu n_j) R'_{i,j,h}$
with $P_{i,j} \in Sub(P)$ and $R_{i,j,h}, R'_{i,j,h} \in \mathcal{P}_{P,n-1}$.
Hence, each P_i can be seen as composed of $1 + 2m$ finite sequences:
$P_{i,1} \dots P_{i,n_i}$, $R_{i,1,1} \dots R_{i,1,k_{i,1}}$, $\dots$, $R_{i,m,1} \dots R_{i,m,k_{i,m}}$ and
$R'_{i,1,1} \dots R'_{i,1,k'_{i,1}}$, $\dots$, $R'_{i,m,1} \dots R'_{i,m,k'_{i,m}}$.
In the following, we use the following notation for the above sequences:
$\sigma(P_i) = P_{i,1} \dots P_{i,n_i}$,
$\sigma_j^{amb}(P_i) = R_{i,j,1} \dots R_{i,j,k_{i,j}}$, and
$\sigma_j^{restr}(P_i) = R'_{i,j,1} \dots R'_{i,j,k_{i,j}}$ for $j = 1, \dots, m$. We note that the first sequence, $\sigma(P_i)$, is composed of elements from the finite set $Sub(P)$, whereas the other $2m$ sequences are composed of elements in $\mathcal{P}_{P,n-1}$. We know from the base case that $=_*$ is a wqo over $Sub(P)^*$.
By inductive hypothesis, we have that $\preceq$ is a wqo on $\mathcal{P}_{P,n-1}$; hence, by Higman's Lemma we have that $\preceq_*$ is a wqo on $\mathcal{P}^*_{P,n-1}$.
As $=_*$ is a wqo over $Sub(P)^*$, we can extract an infinite subsequence from $P_1 \dots P_i \dots$ making the finite sequences $\sigma(P_i)$ increasing w.r.t. $=_*$. In other words, there exists a strictly increasing function f over the natural numbers such that $\sigma(P_{f(i)}) =_* \sigma(P_{f(l)})$ for any $i < l$. By Proposition 4.26 this ensures that $\prod_{j=1}^{h_{f(i)}} P_{f(i),j} \preceq \prod_{j=1}^{h_{f(l)}} P_{f(l),j}$ for any $i < l$.

As $\preceq_*$ is a wqo on $\mathcal{P}^*_{P,n-1}$, we can extract an infinite subsequence from the subsequence $P_{f(1)} \dots P_{f(i)} \dots$ obtained in the previous step, that makes the finite sequences $\sigma_1^{amb}(P_{f(i)})$ increasing w.r.t. $\preceq_*$. This means that there exists a strictly increasing function g_1 over the natural numbers such that
$\sigma_1^{amb}(P_{f(g_1(i))}) \preceq_* \sigma_1^{amb}(P_{f(g_1(l))})$ for any $i < l$. By Proposition 4.27 this ensures that $\prod_{h=1}^{k_{f(g_1(i)),1}} n_1[R_{f(g_1(i)),1,h}] \preceq \prod_{h=1}^{k_{f(g_1(l)),1}} n_1[R_{f(g_1(l)),1,h}]$ for any $i < l$. By proposition 4.20 we have also that
$\prod_{j=1}^{h_{f(g_1(i))}} P_{f(g_1(i)),j} \mid \prod_{h=1}^{k_{f(g_1(i)),1}} n_1[R_{f(g_1(i)),1,h}] \preceq$
$\prod_{j=1}^{h_{f(g_1(l))}} P_{f(g_1(l)),j} \mid \prod_{h=1}^{k_{f(g_1(l)),1}} n_1[R_{f(g_1(l)),1,h}]$ for any $i < l$.
Proceeding in this way, after $2m + 1$ steps we obtain a subsequence of the original sequence $P_1 \dots P_i \dots$ that is increasing w.r.t. $\preceq$. ■

The following theorem shows that the hypothesis of Theorem 4.5 are satisfied.

Theorem 4.29 *Let* $P \in MA_!^{-mv}$.
Then the transition system $(Deriv(P), \overset{\tau}{\mapsto}, \preceq)$ *is a well-structured transition system with decidable* $\preceq$ *and computable* $Succ$.

Proof:
Strong compatibility has been proved in Theorem 4.21.

The fact that $\preceq$ is a well-quasi-order on $Deriv(P)$ is a consequence of Corollary 4.24 and Theorem 4.28 (taking $n = d_a(P)$).

Given $Q, R \in Deriv(P)$, the decidability of $Q \preceq R$ is a consequence of Lemma 4.25 and proceeds by induction on $d_a(Q)$.

Take $n = d_a(Q)$. Suppose that $Q' \preceq R'$ is decidable for any Q', R' with $d_a(Q') < n$.

By Lemma 4.25 there exist l^Q, $k_1^Q, \ldots, k_m^Q$, $k_1'^Q, \ldots, k_m'^Q$ such that

$$Q =_{|} \prod_{i=1}^{l^Q} Q_i \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j^Q} n_j[Q_{j,h}] \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j'^Q} (\nu n_j) Q'_{j,h}$$

for some

$Q_i \in Sub(P)$ for $i = 1, \ldots, l^Q$

$Q_{j,h} \in \mathcal{P}_{P,n-1}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j^Q$

$Q'_{j,h} \in \mathcal{P}_{P,n-1}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j'^Q$

As $n = d_a(Q)$, by definition of $\mathcal{P}_{P,n-1}$ we have that $d_a(Q_{j,h}) < d_a(Q)$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j^Q$ and $d_a(Q'_{j,h}) < d_a(Q)$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j'^Q$.

By Lemma 4.25 there exist l^R, $k_1^Q, \ldots, k_m^Q$, $k_1'^Q, \ldots, k_m'^Q$ such that

$$R =_{|} \prod_{i=1}^{l^R} R_i \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j^R} n_j[R_{j,h}] \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j'^R} (\nu n_j) R'_{j,h}$$

for some

$R_i \in Sub(P)$ for $i = 1, \ldots, l^R$

$R_{j,h}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j^R$

$R'_{j,h}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j'^R$

Deciding $Q \preceq R$ amounts to find a family of injections

$f : \{1, \ldots, l^Q\} \to \{1, \ldots, l^R\}$

$f_j : \{1, \ldots, k_j^Q\} \to \{1, \ldots, k_j^R\}$ for $j = 1, \ldots, m$

$f'_j : \{1, \ldots, k_j'^Q\} \to \{1, \ldots, k_j'^R\}$ for $j = 1, \ldots, m$

such that

$Q_i = R_{f(i)}$ for $i = 1, \ldots, l^Q$

$Q_{j,h} \preceq R_{j,f_j(h)}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j^Q$

$Q'_{j,h} \preceq R'_{j,f'_j(h)}$ for $j = 1, \ldots, m$ and $h = 1, \ldots, k_j'^Q$

The syntactic equality checking required by the first family of relations is clearly decidable.

The decidability of the last two families of relations follows by inductive hypothesis, as $d_a(Q_{j,h}) < d_a(Q)$ for $j = 1, \ldots, m$, $h = 1, \ldots, k_j^Q$ and $d_a(Q'_{j,h}) < d_a(Q)$ for $j = 1, \ldots, m$, $h = 1, \ldots, k_j'^Q$. ■

Corollary 4.30 *Let $P \in MA_!^{-mv}$. The termination of process P is decidable.*

4.4 Termination for guarded recursion

Let $\mathrm{MA}_{!,uugr}^{-mv}$ denote the calculus obtained by extending $\mathrm{MA}_!^{-mv}$ with guarded (unboxed, unrestricted) recursion. We show that the decidability result presented in the previous Subsection for $\mathrm{MA}_!^{-mv}$ continues to hold also for $\mathrm{MA}_{!,uugr}^{-mv}$.

To cope with recursion, the labelled semantics is extended with the following rule:

$$(9L)\ \frac{P\{rec\,X.P/X\} \stackrel{\alpha}{\mapsto} P'}{rec\,X.P \stackrel{\alpha}{\mapsto} P'}$$

As we are dealing with a guarded version of recursion, we can show that the resulting transition system is finitely branching by proceeding inductively on the following complexity measure for processes, denoted by $co(P)$.

Definition 4.31 *Let $P \in MA_{!,uugr}^{-mv}$.*

We define the complexity of P, denoted $co(P)$ in the following way:

$$
\begin{aligned}
&co(\mathbf{0}) = 1\\
&co(open\ n.P) = 1\\
&co(P|Q) = co(P) + co(Q)\\
&co(n[P]) = 1 + co(P)\\
&co((\nu n)P) = 1 + co(P)\\
&co(!P) = 1 + co(P)\\
&co(X) = 1\\
&co(rec\,X.P) = 1 + co(P)
\end{aligned}
$$

Proposition 4.32 *Let $P \in MA^{-mv}_{!,uugr}$. Then the set $\{Q \mid \exists \alpha : P \overset{\alpha}{\mapsto} Q\}$ is finite.*

Proof:
By induction on the complexity of P (note that, as recursion is guarded in $\mathrm{MA}^{-mv}_{!,uugr}$, $co(P\{rec\,X.P/X\}) = co(P) < 1 + co(P) = co(rec\,X.P)$). ■

The results presented in the previous section continue to hold also for $\mathrm{MA}^{-mv}_{!,uugr}$, provided that we include recursive subprocesses (i.e., processes of the kind $rec\,X.Q$) in the set of elementary items.

The definitions of $d_a(P)$ and $Sub(P)$ are extended to $\mathrm{MA}^{-mv}_{!,uugr}$ processes in the following way:

$$
\begin{aligned}
&d_a(X) = 0\\
&d_a(rec\,X.P) = d_a(P)
\end{aligned}
$$

$$Sub(rec\,X.P) = \{rec\,X.P\} \cup (Sub(P))\{rec\,X.P/X\}$$

where with $(Sub(P))\{rec\,X.P/X\}$ we denote the set $\{Q\{rec\,X.P/X\} \mid Q \in Sub(P)\}$.

The intermediate results presented in the previous section for $\mathrm{MA}_!^{-mv}$ hold also for $\mathrm{MA}^{-mv}_{!,uugr}$; hence, termination is decidable also for $\mathrm{MA}^{-mv}_{!,uugr}$ processes:

Corollary 4.33 *Let $P \in MA^{-mv}_{!,uugr}$. The termination of process P is decidable.*

In this section we move to the calculus $\mathrm{MA}^{-mv}_{!,uur}$ thus considering unboxed and unrestricted recursion, and we show that (unguarded) recursion can be transformed in a combination of guarded recursion and replication. The basic idea is the following: given a process $rec\,X.P$, if no occurrence of variable X appears in P inside an ambient or a restriction, either each occurrence of X is guarded, or P is equivalent to a process containing an occurrence of X at top level, i.e., $P \equiv X \mid P'$. In the last case, recursion can be replaced by replication; more precisely, we will show that process $rec\,X.(X \mid P')$ is equivalent to process $!P'\{\mathbf{0}/X\}$.

To prove the result outlined above, we exploit the theory of bisimulation [15]. The proofs of the results listed below are omitted because they turn out to be an easy adaptation of the proofs of the same results for CCS, presented in [15].

Definition 4.34 *A relation $\mathcal{R} \subseteq MA^{-mv}_{!,uur} \times MA^{-mv}_{!,uur}$ is a* bisimulation *if, for each pair $(P,Q) \in \mathcal{R}$:*

- *if $P \overset{\alpha}{\mapsto} P'$ then there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $(P',Q') \in \mathcal{R}$;*
- *if $Q \overset{\alpha}{\mapsto} Q'$ then there exists P' such that $P \overset{\alpha}{\mapsto} P'$ and $(P',Q') \in \mathcal{R}$*

We say that two processes P and Q are bisimilar (denoted by $P \sim Q$) if there exists a bisimulation $\mathcal{R}$ containing the pair (P,Q).
Let P,Q be two terms whose free process variables are contained in $\{X_1,\ldots,X_n\}$. Then P and Q are bisimilar ($P \sim Q$) if and only if $P\{R_i/X_i, i=1,\ldots,n\} \sim Q\{R_i/X_i, i=1,\ldots,n\}$ for any sequence of $MA^{-mv}_{!,uur}$ processes $R_1,\ldots,R_n$.

Note that Proposition 4.11 also holds for $\mathrm{MA}^{-mv}_{!,uur}$ processes; hence, $\equiv$ is a bisimulation.

Definition 4.35 *Let $\mathcal{S}$ be a bisimulation. A relation $\mathcal{R} \subseteq MA^{-mv}_{!,uur} \times MA^{-mv}_{!,uur}$ is a* bisimulation up to $\mathcal{S}$ *if, for each pair $(P,Q) \in \mathcal{R}$:*

- *if $P \overset{\alpha}{\mapsto} P'$ then there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P'\mathcal{S}\mathcal{R}\mathcal{S}Q'$;*
- *if $Q \overset{\alpha}{\mapsto} Q'$ then there exists P' such that $P \overset{\alpha}{\mapsto} P'$ and $P'\mathcal{S}\mathcal{R}\mathcal{S}Q'$.*

Proposition 4.36 *Let $\mathcal{S}$ be a bisimulation and $\mathcal{R}$ a bisimulation up to $\mathcal{S}$. If $(P,Q) \in \mathcal{R}$ then $P \sim Q$.*

Lemma 4.37 *Let P,Q be two terms whose free process variables are contained in $\{X\}$. If $P \sim Q$ then $R\{rec\,X.P/X\} \sim R\{rec\,X.Q/X\}$.*

Theorem 4.38 *$\sim$ is a congruence.*

Proposition 4.39 *Let $P, Q \in MA^{-mv}_{!,uugr}$, with $P \sim Q$. Then P terminates iff Q terminates.*

We are now ready to define the encoding of $\mathrm{MA}^{-mv}_{!,uur}$ processes in $\mathrm{MA}^{-mv}_{!,uugr}$ processes:

Definition 4.40 *Let $P \in MA^{-mv}_{!,uur}$. The function $f : MA^{-mv}_{!,uur} \rightarrow MA^{-mv}_{!,uugr}$ is defined as follows:*

$$
\begin{aligned}
& f(\mathbf{0}) = \mathbf{0} \\
& f(open\ n.P) = open\ n.f(P) \\
& f(P|Q) = f(P)|f(Q) \\
& f(n[P]) = n[f(P)] \\
& f((\nu n)P) = (\nu n)(f(P)) \\
& f(!P) = !(f(P)) \\
& f(X) = X \\
& f(rec\,X.P) = \begin{cases} !f(P)\{\mathbf{0}/X\} & \textit{if } X \textit{ occurs unguarded in } P \\ rec\,X.f(P) & \textit{otherwise} \end{cases}
\end{aligned}
$$

To prove the result we will make use of the following extension of the relation $=_{|}$:

Definition 4.41 *We define $\equiv_!$ as the least congruence relation satisfying the following axioms and rules:*

$$
\begin{aligned}
& \textit{if } P =_{|} Q \textit{ then } P \equiv_! Q \\
& !P \equiv_! P \mid !P \\
& !P \equiv_! !P \mid !P
\end{aligned}
$$

We note that the last axiom in the definition above does not belong to the set of classical structural congruence rules defined in Section 2. This axiom has been introduced in [12] to obtain a decidable version of structural congruence for the π-calculus, and can be considered as the limit of an infinite number of applications of the classical structural congruence axiom for replication, that is the second axiom in the definition above.

Note that $\equiv_!$ is a bisimulation:

Proposition 4.42 *Let $P, Q \in MA^{-mv}_{!,uur}$. If $P \equiv_! Q$ and $P \overset{\alpha}{\mapsto} P'$ then there* *exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P' \equiv_! Q'$.*

In the remaining part of the section, we show that each process is bisimilar to its encoding. As the encoding behaves homomorphically for all operators but rec, the proof of the above sentence is a consequence of the fact that $\sim$ is a congruence over $\mathrm{MA}^{-mv}_{!,uugr}$ and of the fact that the encoding behaves correctly for recursive processes with unguarded recursion.

The following proposition shows that, if recursion is unguarded in process $rec\,X.P$, then this process can be rewritten, modulo structural congruence, as $rec\,X.(X|P')$.

Proposition 4.43 *Let $rec\,X.P \in MA^{-mv}_{!,uur}$. If X occurs unguarded in P then there exists $P' \in MA^{-mv}_{!,uur}$ such that $P \equiv X \mid P'$.*

Proof:
By induction on the structure of P. ■

To show that process $Rec = rec\,X.(X|P')$ is bisimilar to its encoding $f(Rec) = !f(P')\{\mathbf{0}/X\}$ we need some preliminary results.

The following lemma states that the moves performed by a term, whose free occurrences of variable X are replaced by Rec, can be mimicked by the parallel composition of $f(Rec)$ with the process obtained from the term by replacing each occurrence of X with the inactive process $\mathbf{0}$:

Lemma 4.44 *Let $P \in MA^{-mv}_{!,uur}$ and $Q \in MA^{-mv}_{!,uugr}$. If $Q\{recX.(X \mid P)/X\} \overset{\alpha}{\mapsto} Q'$ then there exist R, R' such that $Q' = R\{recX.(X \mid P)/X\}$ and $Q\{\mathbf{0}/X\} \mid !P\{\mathbf{0}/X\} \overset{\alpha}{\mapsto} R'$ and $R' \equiv_! R\{\mathbf{0}/X\} \mid !P\{\mathbf{0}/X\}$.*

Proof:
By induction on the length of the derivation of $Q\{recX.(X \mid P)/X\} \overset{\alpha}{\mapsto} Q'$. ■

Each move performed by the process obtained by substituting each free occurrence of X in a term can be mimicked by the process obtained by replacing each free occurrence of X with Rec:

Proposition 4.45 *Let $P \in MA^{-mv}_{!,uur}$ and $Q \in MA^{-mv}_{!,uugr}$. If $Q\{\mathbf{0}/X\} \overset{\alpha}{\mapsto} Q'$ then there exists R such that $Q' = R\{\mathbf{0}/X\}$ and $Q\{rec\,X.(X \mid P)/X\} \overset{\alpha}{\mapsto} R\{recX.(X \mid P)/X\}$.*

Proof:
By induction on the length of the derivation of $Q\{\mathbf{0}/X\} \overset{\alpha}{\mapsto} Q'$. ■

The following proposition is useful in the proof of the next theorem:

Proposition 4.46 *Let $P \in MA^{-mv}_{!,uur}$. If $rec\,X.(X|P) \overset{\alpha}{\mapsto} P'$ then there exists*

P'' *such that* $P' = rec\,X.(X|P)|P''$.

Proof:
By induction on the length of the derivation of $rec\,X.(X|P) \overset{\alpha}{\mapsto} P'$. ∎

The parallel composition of *Rec* with the process obtained by replacing the occurrences of X with *Rec* in a term is bisimilar to the parallel composition of $f(Rec)$ with the process obtained by replacing the occurrences of X with the inactive process in the same term:

Theorem 4.47 *Let* $P \in MA^{-mv}_{!,uur}$ *and* $Q \in MA^{-mv}_{!,uugr}$.
Then $Q\{rec\,X.(X \mid P)/X\}|rec\,X.(X|P) \sim Q\{\mathbf{0}/X\} \mid !P\{\mathbf{0}/X\}$

Proof:
The theorem follows from the fact that
$\{(Q\{rec\,X.(X \mid P)/X\}|rec\,X.(X|P)\ ,\ Q\{\mathbf{0}/X\} \mid !P\{\mathbf{0}/X\}) \mid Q \in \mathrm{MA}^{-mv}_{!,uugr}\}$
is a bisimulation up to $\equiv_!$. ∎

Corollary 4.48 *Let* $P \in MA^{-mv}_{!,uur}$. *Then* $rec\,X.(X \mid P) \sim !P\{\mathbf{0}/X\}$.

Theorem 4.49 *Let* $P \in MA^{-mv}_{!,uur}$. *Then we have that* $P \sim f(P)$.

Proof:
By induction on the structure of P, using Theorem 4.38, Corollary 4.48 and Proposition 4.43. ∎

Note that, thanks to Proposition 4.39, the presented encoding preserves the existence of an infinite computation. Hence, decidability of termination in $\mathrm{MA}^{-mv}_{!,uur}$ has been reduced to decidability of termination in $\mathrm{MA}^{-mv}_{!,uugr}$.

Corollary 4.50 *Let* $P \in MA^{-mv}_{!,uur}$. *The termination of process* P *is decidable.*

Proof:
A consequence of Corollary 4.33, Theorem 4.49 and Proposition 4.39. ∎

5 Related Work and Conclusion

We have analysed the expressive power of *movement* and *restriction* in the context of the Pure Mobile Ambients. By movement we mean the possibility for an ambient to move inside a sibling ambient, or outside a parent ambient, exploiting the *in* and *out* capabilities, respectively. Restriction, on the other hand, is the mechanism used for ambient name generation.

We have considered two infinite variants of the Pure Mobile Ambients, one with process replication and another one with recursive process definition. For each of these calculi we have considered the fragments obtained by removing movement, restriction, or both. We have used the decidability of process termination as a yardstick to evaluate the expressive power of each of these calculi; the Figure 1 in the Introduction reports all the considered calculi, showing those in which process termination is decidable and those in which it is not the case.

We note that the undecidability of termination is closely connected to the ability to dynamically create chains of unbounded length: in Mobile Ambients we can construct such chains by exploiting either nested restrictions or nested ambients. Nested chains of ambients can be constructed in any fragment containing at least replication and the possibility to move ambients, whereas for constructing nested chains of restrictions it is necessary to use the restriction operator inside recursively defined terms.

Two of the results that we have proved for the Pure Mobile Ambients are particularly interesting in comparison with the form of mobility supported by its ancestor, namely, the π–calculus [16].

The first observation is about the discrimination result that we have proved for the fragment without movement; this fragment, indeed, appears to be Turing complete under recursive process definition, while this is not the case under process replication. On the contrary, in the π–calculus name mobility permits to encode recursive definitions in terms of replication. This discrepancy can be interpreted as follows: the form of mobility available in Pure Mobile Ambients without movement is not as powerful as name mobility, because it does not permit to obtain such an encoding. An interesting open problem is related to the lifting of this expressiveness gap to the full Pure Mobile Ambients; even if this calculus is Turing complete under both recursion and replication, the intuition is that the dynamic spatial structure of Mobile Ambients represents an obstacle for encoding recursion in terms of replication.

The second observation concerns the expressive power of name restriction. We have proved that this operator is not necessary for the Pure Mobile Ambients in order to be Turing complete, while this is not the case for the π–calculus. Indeed, the fragment of π–calculus without name restriction is not Turing complete (see [2]). This difference follows from the fact that in the π–calculus the unique way for dynamically creating nestings of arbitrary length is to use nested restrictions, while in the Pure Mobile Ambients we can instead use nested ambients.

We now consider related literature. As discussed in the Introduction, the Turing completeness of the Pure Mobile Ambients has been discussed in [8]. One of the results that we prove in this paper, the Turing completeness of $\mathrm{MA}_!^{-\nu}$, has been recently and independently proved also by Hirschkoff, Lozes, and Sangiorgi [13]. Their proof, however, exploits Turing Machines which, in our opinion, turn out to be more complex to be modelled with respect to register based machines.

Another recent work that exploits register based machines is [3], where the expressiveness of Pure Mobile Ambients without restriction and *open* is investigated. The main result is that reachability is undecidable for this fragment, whereas it turns out to be decidable under a different reduction semantics. The new semantics is obtained by replacing the structural congruence rule for replication ($!P \equiv P|!P$) with a reduction move ($!P \rightarrow P|!P$). It could be worthwhile to investigate the impact of this alternative reduction semantics on our results: we conjecture that termination becomes decidable also in the full Pure Mobile Ambients. This is due to the fact that any replicated process $!P$ has the ability to perform an infinite computation.

On the contrary, if we replace the structural congruence rule for recursion ($rec\,X.P \equiv P\{rec\,X.P/X\}$) with a reduction move ($rec\,X.P \rightarrow P\{rec\,X.P/X\}$) (as done, e.g., in [22]) termination remains undecidable also in the fragment MA_r^{-mv}. This is due to the fact that the RAM encoding that we have presented for the calculus MA_r^{-mv} works fine also under the alternative semantics for recursion.

Other papers are devoted to the analysis of the expressive power of fragments and/or variants of Mobile Ambients, see e.g. [4,24,9] just to mention some of them. Typically, these papers pursue two different goals.

On the one hand, the aim is to define variants of Mobile Ambients (or Pure Mobile Ambients) that support different forms of ambient interaction while remaining sufficiently expressive to model a significant class of mobile systems. For example, Boxed Ambients [4] (a variant of Mobile Ambients without the *open* primitive and with a limited form of parent-child communication) and Pure Safe Ambients [14,24] (in which each mobility primitive is enriched with a corresponding co-action that must be performed inside the target ambient in order to permit the execution of any of the mobility primitives) are shown to be sufficiently expressive to support the form of communication of the π-calculus.

On the other hand, the aim is the identification of fragments, or variants of the calculus, in which significant properties become decidable. For example, in [9] a finite-control fragment of Mobile Ambients is presented which admits a decidable algorithm for model checking the ambient logic [7].

Acknowledgement: We would like to thank Luca Cardelli, Daniel Hirschkoff, Étienne Lozes and Davide Sangiorgi for their insightful comments and the anonymous referees for their useful suggestions.

References

[1] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291-324, 1998.

[2] R. Amadio and C. Meyssonnier. On decidability of the control reachability problem in the asynchronous pi-calculus. *Nordic Journal of Computing*, 9(2):70-101, 2002.

[3] I. Boneva and J.-M. Talbot. When Ambients Cannot be Opened. In *Proc. of Sixth International Conference on Foundations of Software Science and Computation Structures*, FOSSACS 2003, volume 2620 of *Lecture Notes in Theoretical Computer Science*, pages 169-184. Springer-Verlag, Berlin, 2003.

[4] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proc. of TACS'01*, volume 2215 of *Lecture Notes in Theoretical Computer Science*, pages 38–63. Springer-Verlag, Berlin, 2001.

[5] N. Busi and G. Zavattaro. On the Expressiveness of Movement in Pure Mobile Ambients In *Proc. of Foundations of Wide Area Network Computing (FWAN'02)*, Electronic Notes in Theoretical Computer Science 66(3), Elsevier, 2002.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[7] L. Cardelli and A.D. Gordon. Anytime, anywhere, modal logics for mobile ambients. In *Proc. of POPL'00*, pages 365–377. ACM Press, 2000.

[8] L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[9] W. Charatonik, A.D. Gordon, and J. Talbot. Finite-control mobile ambients. In *Proc. of European Symposiom on Programming (ESOP 2002)*, volume 2305 of *Lecture Notes in Theoretical Computer Science*, pages 295-313. Springer-Verlag, Berlin, 2002.

[10] A. Finkel and Ph. Schnoebelen. Well-Structured Transition Systems Everywhere! *Theoretical Computer Science*, 256:63–92, 2001.

[11] G. Higman. Ordering by divisibility in abstract algebras. In *Proc. London Math. Soc.*, vol. 2, pages 236–366, 1952.

[12] D. Hirschkoff. *Mise en œvre de preuves de bisimulation.* PhD Thesis, ENPC, Champs sur Marne, France, 1999.

[13] D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, expressiveness, and decidability in the Ambient Logic. In *Proc. of LICS'02*, 2002.

[14] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proc. of POPL'00.* ACM Press, 2000.

[15] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[16] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.

[17] M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.

[18] U. Nestmann and B.C. Pierce. Decoding Choice Encodings. *Information and Computation*, 163:1–59, 2000.

[19] D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proc. of POPL'01*, pages 4–17. ACM Press, 2001.

[20] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

[21] J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

[22] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proc. of CONCUR'02*, volume 2421 of *Lecture Notes in Theoretical Computer Science*, pages 288-303. Springer-Verlag, Berlin, 2002.

[23] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD Thesis, Department of Computing, Imperial College, 1990.

[24] P. Zimmer. On the expressiveness of pure safe ambients. Technical Report RR-4350, INRIA Research, to appear in a special issue of Mathematical Structures of Computer Science, Cambridge University Press, 2003.