Electronic Notes in Theoretical Computer Science 66 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 15 pages

On the Expressiveness of Movement in Pure Mobile Ambients

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, I-40127, Bologna, Italy.
E-mail: busi,zavattar@cs.unibo.it

Abstract

Pure Mobile Ambients (i.e., Mobile Ambients without communication) provides three mobility primitives: *in* and *out* for ambient movement, and *open* to dissolve ambient boundaries. In this paper we consider the expressiveness of the primitives *in* and *out* for ambient movement; more precisely, we concentrate on the interplay between ambient movement and the ability to create new names (exploiting the restriction operator). To this aim, we consider a version of Pure Mobile Ambients (with explicit recursive definitions instead of replication) and we concentrate on the three fragments of the calculus that can be obtained removing either one or both between movement and the ability to create new names. The unique mobility primitive that we retain in all of the considered calculi is *open*. Te three fragments are denoted as follows: MA^{-mv} without ambient movement, $\mathrm{MA}_{-\nu}$ without restriction, and $\mathrm{MA}_{-\nu}^{-mv}$ without both movement and restriction. We prove that both the fragments MA^{-mv} and $\mathrm{MA}_{-\nu}$ are Turing-complete, while this is not the case for $\mathrm{MA}_{-\nu}^{-mv}$. Indeed, we prove that in this latter calculus the existence of an infinite computation turns to be a decidable property.

1 Introduction

Mobile Ambients (MA) [5] is a well known formalism for the description of distributed and mobile systems in terms of *ambients*. An ambient is a named collection of active processes and nested sub-ambients. In the pure version of MA, 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 primitives 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 of the process.

©2002 Published by Elsevier Science B. V.

Since its introduction, a considerable effort has been devoted to the analysis of the expressiveness of MA and its variants. For instance, in the paper introducing MA [5] it is shown that the calculus is Turing-complete. Many other papers are devoted to the analysis of fragments and/or variants of MA, see e.g. [2,15,13,8,6] just to mention some of them. Typically, these papers pursue two different goals.

On the one hand, the aim is to define significant fragments or variants of MA which maintain the same expressive power of the full original calculus. For example, Boxed Ambient [2], a variant of MA without the *open* primitive and with a limited form of parent-child communication, is shown to be at least as flexible as Mobile Ambients.

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 [6] a finite-control fragment of MA is presented which admits a decidable algorithm for model checking of the ambient logic [4].

Despite this considerable amount of work, little attention has been payed to investigate the relevance and expressiveness of the original *in* and *out* primitives, which represent the fundamental primitives for ambient movement. Differently from our approach, related work considers variants of the original movement primitives as in, e.g., [15] where *in* and *out* are investigated in the setting of Pure Safe Ambients [1], which is a variant of MA 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.

The goal of this paper is to initiate an investigation of the relevance and expressiveness of the original *in* and *out*. We consider the communication-free fragment (thus restricting to the pure version of MA). Moreover, we consider a more general form of recursion which is more adequate to the aims of this paper: namely, as already made in other papers such as [6,13], recursion is obtained by means of explicit recursive definitions instead of replication.

As a first step in our investigation, we wonder whether or not the expressiveness of the calculus is affected by the elimination of the primitives *in* and *out*. Quite surprisingly, we prove that the fragment MA^{-mv} without ambient movement is still Turing-complete. The proof is based on a simulation of Random Access Machines [14] (RAMs), a well known register based Turing-complete formalism. The encoding of RAMs that we present makes use of the restriction operator in order to dynamically create new ambient names. It is worth noting that our modeling of RAMs does not exploit the possibility to introduce inside an ambient an active process; indeed, we use only empty ambients such as $a[]$. This allows us to conclude that also the fragment of MA^{-mv} without nested ambients, more precisely containing only empty ambients of the form $n[\mathbf{0}]$, is Turing-complete.

At this point, we wonder whether the restriction operator is strictly necessary in order to model RAMs in the absence of ambient movement; we prove that this is indeed the case. In fact, the fragment $\mathrm{MA}_{-\nu}^{-mv}$ of the calculus without both ambient movement and restriction is not Turing-complete (more precisely, we prove that the existence of an infinite computation turns to be a decidable property in $\mathrm{MA}_{-\nu}^{-mv}$).

Finally, we conclude our investigation wondering whether the re-introduction of ambient movement permits to to model RAMs also without exploiting the ability to create new names. We prove that indeed this is the case: more precisely, we show that the fragment $\mathrm{MA}_{-\nu}$ without the restriction operator is Turing complete. This result has been recently and independently proved by Hirschkoff, Lozes, and Sangiorgi [8]. Their proof, however, exploits Turing Machines which, in our opinion, reveal more complex to be modeled with respect to RAMs.

The paper is structured as follows: in Section 2 we present the syntax and semantics of the full calculus. In Section 3 we prove that the fragment of the calculus without *in* and *out* is Turing-complete; in Section 4 we show that if we remove also the restriction operator the existence of a divergent computation turns to be a decidable property; in Section 5 we prove that the calculus without restriction, but with ambient movement, is Turing-complete. Section 6 reports some conclusive remarks.

2 Pure Mobile Ambients with Explicit Recursion

In this section we recall Pure Mobile Ambients. The unique difference with respect to the original syntax of [5] is that we use, as already done in [6,13], an explicit recursive definition instead of replication. In the following we refer to this calculus with MA.

Let $Name$, ranged over by n, m, $\ldots$, be a set of ambient names, and let Var, ranged over by X, Y, $\ldots$, be a set of term variables. The terms of MA are defined by the following grammar:

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

where we consider only closed terms, i.e, terms in which each variable X appears inside a term $rec\,X.P$. Moreover, as done in [13], in order to avoid infinite ambient nesting we assume *unboxed recursion*, i.e., that the variable X cannot appear inside any ambient of P.

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 firing of the primitive the process becomes P. The term $n[P]$ denotes an ambient named n containing process P; a process, may be also the parallel composition $P|Q$ of two subprocesses. The restriction operator $(\nu n)P$ is used to create a new name n which is bound in P. As usual (see, e.g.,[9]) the terms X and $rec\,X.P$ are used for the recursive definition of processes.

The operational semantics is defined in terms of a structural congruence plus a reduction relation. The structural congruence $\equiv$ is the smallest congruence relation satisfying:

$$
\begin{array}{ll}
(\nu n)\mathbf{0} \equiv \mathbf{0} & (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \\
(\nu n)(P|Q) \equiv P|(\nu n)Q \ \text{ if } n \notin fn(P) & (\nu n)(m[P]) \equiv m[(\nu n)P] \\
P|\mathbf{0} \equiv P & P|Q \equiv Q|P \\
P|(Q|R) \equiv (P|Q)|R & rec\,X.P \equiv P\{rec\,X.P/X\}
\end{array}
$$

where $fn(P)$ denotes the free names in P and $P\{rec\,X.P/X\}$ denotes the term obtained by substituting $rec\,X.P$ for any occurrence of X occurring in P not inside any subterm $rec\,X.Q$.

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

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

3 MA^{-mv}: the Fragment without Movement

In this section we consider the fragment of MA without the primitives *in m* and *out m* for ambient movement. Quite surprisingly, we prove that this fragment comprising the unique mobility primitive *open* is expressive enough to model all recursive functions. More precisely, we how to model in MA^{-mv} Random Access Machines (RAMs) [14], a well known Turing-complete formalism.

A Random Access Machine is composed of a finite set of registers, that can hold arbitrary large natural numbers, and by a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the content of registers) or conditional jumps.

To perform a computation, the inputs are provided in registers $r_1, \dots, r_m$; if other registers $r_{m+1}, \dots, r_l$ are used in the program, they are supposed to contain the value 0 at the beginning of the computation. The program is composed by the sequence of instructions $I_1 \dots I_k$. The execution begins with the first instruction I_1 and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than k is reached. If the program terminates, the result of the computation is the contents of the registers.

In [10] it is shown that the following two instructions are sufficient to model every recursive function:

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

The RAM encoding we present is inspired by the encoding in an asynchronous version of CCS [9] that we proposed in [3].

The translation of the RAMs is based on two different encodings, one for program instructions, and one for registers. The terms representing instructions, and those modeling registers, interact via asynchronous communication. This is achieved exploiting empty ambients which are produced by senders and opened by receivers. More precisely, if a sender wants to emit a message a, it simply spawns the ambient $a[]$; when the receiver wants to receive the message, it simply performs an *open a* primitive which consumes the corresponding message $a[]$.

Moreover, we want to permit to a receiver to choose among two possible available messages. More precisely, we want to use an extra-term $(open\ a.P) \oplus (open\ b.Q)$ which has the ability to open either ambient/message $a[]$ or $b[]$, and then activates the continuation P, in the former case, or Q, in the latter. This term can be mapped in the original calculus following an approach that has been already exploited in [11] to encode choice in the asynchronous π–calculus:

$$
\begin{array}{l}
(open\ a.P) \oplus (open\ b.Q) \ = \\
(\nu ok)(\nu ko_a)(\nu ko_b) \ \ (\ ok[] \mid \\
\quad open\ a.(\ open\ ok.(P \mid ko_b[]) \mid open\ ko_a.a[]\ ) \mid \\
\quad open\ b.(\ open\ ok.(Q \mid ko_a[]) \mid open\ ko_b.b[]\ ) \\
)
\end{array}
$$

It is worth noting that this encoding is not a general encoding for nondeterministic choice between *open* operations, but it assumes that the subambients to be open should be empty; in our case this is ensured by the fact that we use empty ambients as representation of asynchronously exchanged messages.

The idea behind the above encoding is to activate concurrently both the two alternative *open* a and *open* b primitives. In order to avoid the undesired activation of both the continuations P and Q, mutual exclusion is achieved by means of a shared unique ambient $ok[]$ which must be opened before activating any of the two continuations. The process which succeeds in opening the ambient $ok[]$, produces another ambient which is used to communicate to the concurrent branch that it should fail, that is, in the case the concurrent branch opens the corresponding ambient, then it must reproduce it. Observe that the auxiliary names ok, ko_a, and ko_b are new names in order to avoid undesired name collisions.

We are now ready to define the encoding of program instructions.

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

The instruction at position i has a corresponding "program counter ambient" $p_i[]$. Each instruction is modeled by a recursively defined process which first consumes its program counter ambient, then modifies or test the content of the registers, and finally produces the program counter ambient for the next instruction to be executed.

A *Succ* instruction on register r_j produces the ambient $inc_j[]$, representing a request for the increment of register r_j, and then waits for the acknowledgement (the ambient $ack_j[]$) indicating that the increment has been successfully executed.

An instruction $DecJump(r_j, s)$ produces the ambient $test_j[]$, representing a request for testing register r_j and decrementing it if its content is greater than zero; if the contents of r_j is zero, then the ambient $zero_j[]$ is produced, otherwise the ambient $dec_j[]$ is spawn. For this reason, the process modeling the *DecJump* instruction, must exploit the choice operator described above in order to be able to react to the two possible alternative results. In the case the *open* $zero_j$ primitives succeeds, the program counter ambient $p_s[]$ is produced, otherwise the ambient $p_{i+1}[]$ is spawn.

The modeling of register r_j, that initially contains the value 0, is given by the term Z_j defined as follows:

$$\begin{aligned}
Z_j &= rec\,X.\ (open\ test_j.(zero_j[] \mid X)) \oplus \\
&\qquad (open\ inc_j.(ack_j[] \mid (\nu a)(O_j \mid open\ a.X))) \\
O_j &= rec\,Y.\ (open\ test_j.(dec_j[] \mid a[])) \oplus \\
&\qquad (open\ inc_j.(ack_j[] \mid (\nu b)(E_j \mid open\ b.Y))) \\
E_j &= rec\,V.\ (open\ test_j.(dec_j[] \mid b[])) \oplus \\
&\qquad (open\ inc_j.(ack_j[] \mid (\nu a)(Y \mid open\ a.V)))
\end{aligned}$$

If the term Z_j receives a $test_j$ request, then the ambient $zero_j[]$ is produced as an answer to communicate that the register is empty. If, on the other hand, an inc_j is received, then Z_j produces the corresponding $ack_j[]$ and it becomes the term $(\nu a)(O_j|open\ a.Z_j)$. The term $open\ a.Z_j$ is blocked by the primitive $open\ a$ until the term O_j creates an ambient $a[]$. Observe that a is a new name known only to the terms O_j and $open\ a.Z_j$. After the first increment, the register r_j should contain the value 1; then, if a request $test_j$ is produced, then the term O_j should generate the ambient $dec_j[]$ as answer, and then activates the term Z_j by producing the ambient $a[]$. Otherwise, if a request inc_j is produced, then O_j should produce the corresponding $ack_j[]$ ambient, and becomes $(\nu b)(E_j|open\ b.O_j)$. In this case the term $open\ b.O_j$ is guarded by the $open\ b$, and it waits to be activated by the term E_j. We have used a different new name in order to avoid that the term E_j will incorrectly activate the other term $open\ a.Z_j$. The term E_j is defined in the same way as O_j, with the unique difference that the name b is used instead of a, and vice versa. The restriction operator (νa) is used to generate a new instance of name a which does not interfere with the name a used in the term Z_j. In this way, the agent O_j is used to represent the register r_j when it contains odd values, while E_j is used for even values.

Let consider the program $I_1, \ldots, I_k$ with inputs $n_1, \ldots, n_m$ that uses the registers $r_1, \ldots, r_l$. In order to execute it, first we have to introduce every input n_i in the corresponding register r_i. This is done by the following agent B that performs the *bootstrap* of the system by initializing the registers before emitting the program counter ambient $p_1[]$:

$$\begin{aligned}
B = {} & \underbrace{inc_1[]|\ldots|inc_1[]}_{n_1 \text{ times}}|\ldots|\underbrace{inc_m[]|\ldots|inc_m[]}_{n_m \text{ times}}| \\
& \underbrace{open\ ack_1.\ldots.open\ ack_1}_{n_1 \text{ times}}\ldots\underbrace{open\ ack_m.\ldots.open\ ack_m}_{n_m \text{ times}}.p_1[]
\end{aligned}$$

The above program is then modeled by:

$$Z_1|\ldots|Z_l|B|[\![I_1]\!]|\ldots|[\![I_k]\!]$$

It is worth noting that our modeling of RAMs does not exploit the possibility to introduce inside an ambient an active process; indeed, we use only empty ambients such as $a[]$. This allows us to conclude that also the fragment of MA^{-mv}, containing only ambients of the form $n[\mathbf{0}]$, is Turing-complete.

4 MA$^{-mv}_{-\nu}$: the Fragment without Movement and Restriction

In the RAM encoding provided in the previous section, the contents of registers is modeled through a sequence of nested restrictions. In this section we investigate if restriction is really necessary to obtain Turing-completeness in the movement-free fragment of the calculus.

We show that this is the case by proving the decidability of the existence of a diverging computation for the fragment of MA without movement and restriction. The decidability result is based on the theory of well-structured transition systems [7]: given a process in MA$^{-mv}_{-\nu}$, we define a corresponding transition system, with the same behaviour w.r.t. divergence, as an intermediate model; then, by exploiting the theory developed in [7], we show that divergence is decidable for the class of transition systems corresponding to MA$^{-mv}_{-\nu}$ processes.

We start recalling some basic definitions and results of [7], concerning well-structured transition systems, that will be used in the following.

4.1 Well-Structured Transition System

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

Definition 4.1 A *well-quasi-ordering* 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 $x_i \leq x_j$.

Definition 4.2 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.

We write $\rightarrow^+$ (resp, $\rightarrow^*$) for the transitive (resp. the reflexive and transitive) closure of $\rightarrow$.

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

Definition 4.3 A *well-structured transition system* is a transition system $TS = (S, \rightarrow)$, equipped with a quasi-ordering $\leq$ on S, such that the two following conditions hold:

(i) **well-quasi-ordering**: $\leq$ is a well-quasi-ordering, and

(ii) **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 sequence $t_1 \rightarrow^* t_2$ such that $t_1 \leq t_2$.

$$
\begin{aligned}
dec(\mathbf{0}) &= \emptyset \\
dec(open\ n.P) &= \{open\ n.P\} \\
dec(n[P]) &= \{n[dec(P)]\} \\
dec(P|Q) &= dec(P) \oplus dec(Q) \\
dec(rec\,X.P) &= \{rec\,X.P\}
\end{aligned}
$$

Table 1
Decomposition function.

A well-structured transition system has *transitive compatibility* if for all $s_1 \leq t_1$ and transition $s_1 \rightarrow s_2$ there exists a sequence $t_1 \rightarrow^+ t_2$ such that $t_1 \leq t_2$.

Theorem 4.4 *Let $TS = (S, \rightarrow)$ be a well-structured transition system with transitive compatibility, decidable $\leq$ and computable Succ. The existence of an infinite computation starting from a state $s \in S$ is decidable.*

4.2 Divergence is decidable in $MA_{-\nu}^{-mv}$

In this section we define a transition system corresponding to a $\mathrm{MA}_{-\nu}^{-mv}$ process P, whose states represent processes reachable from P and each state exhibits the same behaviour of the corresponding process w.r.t. divergence.

A state is basically a multiset containing sequential processes (i.e., processes of kind *open* $n.P$ or $rec\,X.P$) or a representation of an ambient (i.e., items with the form $n[m]$, where m is a multiset). Note that, because of the restricted form of unboxed recursion we adopt, there exists an upper bound to the level of nesting of ambients in the processes reachable from a given process P. Moreover, as the no new name generation mechanism is present in the fragment $\mathrm{MA}_{-\nu}^{-mv}$, the multisets corresponding to processes reachable from P have a finite domain. To lighten the notation, in the following we consider only processes containing a single variable for recursion.

Definition 4.5 Given a set S, a *finite multiset* over S is a function $m : S \rightarrow \mathbb{N}$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The *multiplicity* of an element s in m is given by the natural number $m(s)$. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. A multiset m such that $dom(m) = \emptyset$ is called *empty* .

Given the multiset m and m', we write $m \oplus m'$ to denote *multiset union*: $m \oplus m'(s) = m(s) + m'(s)$ for all $s \in S$.

Definition 4.6 Let $P \in \mathrm{MA}_{-\nu}^{-mv}$. The transition system $TS(P) = (S, \rightarrow)$ is defined as follows.

The set S is the least set satisfying the following equation:

$$open\ n.Q \oplus n[m] \ \rightarrow \ dec(Q) \oplus m$$

if $dec(Q\{rec\,X.Q/X\}) = open\ n.Q' \oplus m'$ then

$$rec\,X.Q \oplus n[m] \ \rightarrow \ dec(Q') \oplus m' \oplus m$$

if $dec(R\{rec\,X.R/X\}) = n[m'] \oplus m''$ then

$$open\ n.Q \oplus rec\,X.R \ \rightarrow \ dec(Q) \oplus m' \oplus m''$$

if $dec(Q\{rec\,X.Q/X\}) = open\ n.Q' \oplus m'$ and
$dec(R\{rec\,X.R/X\}) = n[m''] \oplus m'''$ then

$$rec\,X.Q \oplus rec\,X.R \ \rightarrow \ dec(Q') \oplus m' \oplus m'' \oplus m'''$$

$$\frac{m \ \rightarrow \ m'}{n[m] \ \rightarrow \ n[m']}$$

$$\frac{m \ \rightarrow \ m'}{m \oplus m'' \ \rightarrow \ m' \oplus m''}$$

Table 2
Transitions specification.

$S = \mathcal{M}_{fin}(\{open\,n.Q, rec\,X.Q \mid n$ is a name occurring in P and $open\,n.Q$, $rec\,X.Q$ are subprograms of $P\} \cup$
$\{n[m] \mid n$ is a name occurring in P and $m \in S\})$

The function dec, associating to each process $Q \in \mathrm{MA}^{-mv}_{-\nu}$ the corresponding multiset in S, is defined in Table 1.

The set $\rightarrow$ is the least set satisfying the axioms and the rules in Table 2.

The following definition introduces a quasi-ordering relation on states of the transition system associated to a process. The underlying idea is the following: a marking m_1 is related to m_2 if, for each sequential process, the number of its occurrences in m_1 is lesser that the number of its occurrences in m_2; moreover, for each occurrence of ambient $n[m'_1]$ in m_1 there exists an ambient $n[m'_2]$ in m_2 such that m'_1 is lesser than m'_2.

Definition 4.7 Let $P \in \mathrm{MA}^{-mv}_{-\nu}$ and $TS(P) = (S, \rightarrow)$. The relation $\preceq$ on S is defined as follows.

Let $m_1, m_2 \in S$.

We have that $m_1 \preceq m_2$ iff one of the following conditions holds:

- there exists $\bar{m}$ such that $m_2 = m_1 \oplus \bar{m}$, or
- there exist $m'_1, m''_1, m'_2, m''_2, n$ such that $m_1 = n[m'_1] \oplus m''_1$, $m_2 = n[m'_2] \oplus m''_2$, $m'_1 \preceq m'_2$ and $m''_1 \preceq m''_2$.

The relation $\preceq$ is a quasi-ordering, making $TS(P)$ a well-structured transition system with transitive compatibility.

Proposition 4.8 *Let $P \in MA_{-\nu}^{-mv}$ and $TS(P) = (S, \rightarrow)$. The relation $\preceq$ is a quasi-ordering.*

Lemma 4.9 *Let $P \in MA_{-\nu}^{-mv}$ and $TS(P) = (S, \rightarrow)$.*

The transition system $TS(P)$ equipped with the quasi-ordering $\preceq$ is a well-structured transition system with transitive compatibility.

As a consequence of Theorem 4.4, we have that divergence is decidable for $TS(P)$, hence also for the MA process P.

Corollary 4.10 *Let $P \in MA_{-\nu}^{-mv}$ and $TS(P) = (S, \rightarrow)$.*

The existence of an infinite computation starting from dec*(P) is decidable.*

5 MA$_{-\nu}$: the Fragment without Restriction

In the previous sections we first noted that MA is Turing-complete even without the ability to move ambients, and we have subsequently proved that this result holds only if we consider the ability to create new names via the restriction operator. In this section, we wonder whether the restriction operator is strictly necessary to model every recursive function even if you re-introduce in the calculus the *in* and *out* primitives for ambient movement; we prove that this is not the case. This is proved by showing how to encode RAMs in the fragment MA$_{-\nu}$.

The encoding of RAMs that we present uses a simpler form of recursively defined processes which corresponds to the replication operator $!P = rec\, X.(X|P)$. In this way, we can conclude that the result proved in this section applies also to the standard Pure Mobile Ambients with replication instead of explicit recursive definition.

We start our description of RAMs taking into account how to model registers. The fact that register r_i contains value k is represented by the process $[\![r_i = k]\!]$, defined as follows:

$$\begin{aligned}
[\![r_i = 0]\!] = zero_i[\ & !open\ increq0_i.(msg[\ out\ zero_i\ .s_i[\ SCONT_i\ ]\ ]\ | \\
& \qquad\qquad in\ s_i.incack_i[\ out\ zero_i.!out\ s_i\ ])\ | \\
& !open\ zeroreq_i.okzero_i[\ out\ zero_i.in\ dj_i\ ]\ ]
\end{aligned}$$

$$[\![r_i = n+1]\!] = s_i[\ SCONT_i \mid [\![r_i = n]\!]\ ]$$

where

$$SCONT_i = open\ decreq_i.okdec_i[\ out\ s_i.in\ dj_i\ ] \mid$$
$$!open\ msg$$

When register r_i is empty, it is modeled by an ambient named $zero_i$, when it is not empty, by an ambient named s_i. The requests of increment, test for zero, or decrement of the register r_i are sent to the register by means ambients named respectively $increq_i$, $zeroreq_i$, and $decreq_i$ which enter the register boundary.

When an increment request is received, the register modifies its structure by creating a new ambient s_i and moving itself inside this new boundary. In this way, when the register contains value n then register is formed by n nested ambients named s_i and a inner ambient named $zero_i$. Besides changing its nesting structure, the register replies to the request with an acknowledgement, modeled by an ambient named $incack_i$.

On the other hand, when a $zeroreq_i$ enters the ambient named $zeroreq_i$, there is no modification of the structure of the register, but simply a reply is produced represented by an ambient name $okzero_i$.

Finally, in the case a request $decreq_i$ enters an ambient s_i, a reply is produced, which is represented by an ambient named $okdec_i$ (this behaviour is given by the term $SCONT_i$ which is present at any level of nesting of the ambients named n_i). As described below, the reply will be managed by the instruction that performed the decrement operation, which is responsible to dissolve the outer s_i boundary in order to update the nesting structure of the register.

We are now ready to describe the encoding of instructions. The i-th instruction is modeled by (the replication of) an ambient named p_i, which contains processes defined according to the kind of instruction. An instruction is activated by dissolving the boundary of one of the corresponding ambients (replication ensures the possibility to execute each instruction for an unbounded amount of times).

If the i-th instruction is an increment of register r_j, its encoding is the process

$$[\![i : Succ(r_j)]\!] = !p_i[\ increq_j[\ !in\ s_j \mid in\ zero_j.increq0_j[out\ increq_j\ ]\ ] \mid$$
$$open\ incack_j.open\ p_{i+1}\ ]$$

The ambient p_i contains two processes. The first process models the increment request: it is an ambient named $increq_i$ which has the ability to enter register r_i, move through all its nested ambients s_i, and finally enter the inner ambient $zero_i$. The second process waits for the acknowledgement of the increment instruction, and then activates the subsequent instruction.

On the other hand, if the i-th instruction is a decrement of r_j or jump to

s, its encoding is

$$\begin{aligned}[\![i : DecJump(r_j, s)]\!] = !p_i[\, & decreq_j[\ in\ s_j\ ] \mid \\ & zeroreq_j[\ in\ zero_j\ ] \mid \\ & dj_j[\ DJCONT_{ijs}\ ]\ ]\end{aligned}$$

where

$$\begin{aligned}& DJCONT_{ijs} = \\ & \quad open\ okdec_j.in\ garbage.msg[\ out\ dj_i.out\ garbage.open\ zeroreq_j. \\ & \qquad\qquad open\ s_j.open\ p_{i+1}\ ] \mid \\ & \quad open\ okzero_j.in\ garbage.msg[\ out\ dj_i.out\ garbage. \\ & \qquad\qquad open\ decreq_j.open\ p_s\ ]\end{aligned}$$

In this case, the ambient p_i contains three processes. The first two processes are ambients which respectively represent the request for decrement or test for zero of register r_j, while the third one, named dj_j is an ambient which is used to model the reaction to the answer that will be provided by the register r_j.

One and only one of the two requests for decrement and test for zero will succeed. Indeed, the former requires to enter an ambient s_i while the latter considers ambient $zero_j$. Due to the modeling of registers describe above, it is ensured that either an ambient s_j or $zero_j$ is available, but not both. In the first case, the answer of the register r_j will be an ambient named $okdec_j$, in the second one an ambient named $okzero_j$. In both cases the reply will move inside the third ambient named dj_j cited above.

Inside this ambient a process $DJCONT_{ijs}$ is present which is responsible for managing the reply. In the case of a $okdec_j$ reply, the process is responsible for removing the request of test for zero which has failed, to dissolve an ambient boundary s_j in order to actually decrement the register r_j, and then activating the subsequent instruction, by dissolving an ambient p_{i+1}. On the other hand, if a $okzero_j$ is received, the process will remove the decrement request which has failed, before activating the s-th instruction.

Finally, the encoding of a RAM formed by instructions $I_1 \dots I_k$, starting the computation with values $v_1 \dots v_l$ in registers $r_1 \dots r_l$, is the following process:

$$\begin{aligned}& [\![I_1]\!] \mid \ \dots \ \mid [\![I_k]\!] \mid \\ & [\![r_1 = v_1]\!] \mid \ \dots \ \mid [\![r_n = v_n]\!] \mid \\ & open\ p_1 \mid !open\ msg \mid garbage[]\end{aligned}$$

Observe that an extra ambient named $garbage$ is used in order to introduce in it all those empty ambients named dj_j which are created but not consumed by the $DecJump$ instruction. Moreover, observe that at the outer level, as well as in each level of the nested ambients s_i, there is a process $!open\ msg$ which opens all those ambients named msg containing processes which enter that specific ambient level in order to perform actions in that ambient.

6 Conclusion and Future Work

In this paper we started an investigation on the expressiveness of some fragments of Mobile Ambients. More precisely, we study the impact of movement capabilities and new name generation on the Turing-completeness of the communication-free fragment of the calculus, or, in other words, on the decidability of properties such as divergence.

The first result shows that the calculus remains Turing-complete even if we remove movement capabilities, and also if we restrict to the subcalculus with empty ambients. As this result exploits nesting of restrictions to model the contents of registers, we wonder if it is possible to get rid of restriction, while retaining Turing-completeness. The answer is negative, because we show that the existence of an infinite computation is decidable for the movement free and restriction free fragment of the calculus. This second result yields to asking whether restriction is an unavoidable ingredient to obtain Turing-completeness of Mobile Ambients: we show that it is possible to get rid of restriction, at the price of reintroducing the movement capabilities.

A lot of interesting problems remain to be investigated. For example, it could be interesting to study what happens if the open capability, instead of the movement capabilities, is dropped. An interesting starting point for such an investigation is [2], where *boxed ambients*, a variant of mobile ambients obtained by removing the open capability and by adding new primitives for parent-children communication, are introduced.

Finally, we would like to point out that the Turing-completeness result of MA^{-mv} relies on the ability of representing natural numbers by sequences of nested restrictions, which can easily be modeled in the variant of MA with recursive definitions we adopted in this paper. Consider the variant of MA^{-mv} with replication instead of recursive definitions. While we claim that the addition of communication makes this calculus Turing-complete, it is not clear if communication is really necessary to reach Turing-completeness. Hence, it could be worthwhile to investigate the interchangeability of replication and recursive definitions in (fragments of) Mobile Ambients, as already done in [12] for Temporal Concurrent Constraint Programming Languages.

Acknowledgement: We would like to thank Luca Cardelli for his insightful comments and suggestions.

References

[1] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of POPL'01*, pages 222–235. ACM Press, 2001.

[2] 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.

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

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

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

[6] W. Charatonik, A.D. Gordon, and J. Talbot. Finite-control mobile ambients. In *Proc. of ESOP'02*, volume to appear of *Lecture Notes in Theoretical Computer Science*. Springer-Verlag, Berlin, 2002.

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

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

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

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

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

[12] M. Nielsen, C. Palamidessi, and F. D. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *Proc. of PPDP'02*. ACM Press, 2002.

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

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

[15] 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, 2002.