Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 147 (2006) 135–161
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Typed Mobile Ambients in Maude[1]

Fernando Rosa-Velardo, Clara Segura, and Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Madrid, Spain
{fernandorosa,csegura,alberto}@sip.ucm.es

Abstract

Maude has revealed itself as a powerful tool for implementing different kinds of semantics so that quick prototypes are available for trying examples and proving properties. In this paper we show how to define in Maude two semantics for Cardelli and Gordon's Ambient Calculus. The first one is the operational (reduction) semantics which requires the definition of Maude strategies in order to avoid infinite loops. The second one is a type system defined by Cardelli and Gordon to avoid communication errors. The correctness of that system was not formally proved. We enrich the operational semantics with error rules and prove that well-typed processes do not produce such errors. The type system is highly non-deterministic. We show here one possible way of implementing such non-determinism in the rules.

Keywords: Ambient calculus, operational semantics, type systems, Maude.

1 Introduction

Maude, a high-level language and high-performance system supporting both equational and rewriting logic computation [8,7], has revealed itself as a powerful tool for representing different kinds of semantics [11,20,21]. Since Maude specifications are executable, what we get is an implementation of the language so that quick prototypes are available for trying examples and proving properties.

We use Maude as a *metalanguage* in which the syntax and semantics of particular languages can be formally defined. One of our aims is to maintain the representation distance as short as possible. There are several different

[1] Work partially supported by the MCyT Spanish project *MIDAS*, TIC 2003-01000.

doi:10.1016/j.entcs.2005.06.041

ways of mapping inference systems into rewriting logic. In the structural operational semantics case, judgements typically have the form of some kind of transition $P \to Q$ between states so that it makes sense to consider the possibility of mapping directly this transition relation between states to a rewriting relation between terms representing the states. When thinking this way, an inference rule like

$$\frac{P_1 \to Q_1 \quad \dots \quad P_n \to Q_n}{P_0 \to Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites. In this way the semantic rules become (conditional) rewrite rules, where the transition in the conclusion becomes the main rewrite of the rule, and the transitions in the premises become rewrite conditions.

In this paper we show how to integrate in the same framework two different semantics, namely operational and static, for the Ambient Calculus (AC). This will allow us to study properties involving them. First, we define the operational semantics given by Cardelli and Gordon as structural congruence and reduction rules. We show how to exploit the rewriting machinery in order to reduce the number of rules, and also the decisions we have taken in order to avoid infinite reductions. Rewrite rules need not be confluent or terminating. This theoretical generality needs some control when the specifications become executable, because the user needs to make sure that the rewriting process does not go in undesired directions. We have recently defined a strategy language for Maude, to control the rewriting process [12]. Strategy expressions can be defined to reduce the tree of rewritings of a given term. For example, replication in AC requires the definition of strategies to avoid infinite loops.

Cardelli and others defined several type systems for AC [5,3,4] in order to avoid different kinds of errors: basically communication errors, and violation of mobility and opening constraints. We study here how to implement the first type system defined by Cardelli and Gordon [5] to detect communication errors, and syntactically incorrect terms. There, they proved a subject reduction theorem in order to justify the correctness of the type system, but no formal definition of the meaning of types was given. Thus, there is no formal correctness relation between the type system and the operational semantics. It is an interesting exercise to formalize that relation in order to complete the correctness proof. We enrich the operational semantics with error reductions reflecting the kind of errors we want to avoid. Then we prove (by hand) that a well-typed process never causes such an error. We implement these error rules in Maude and also the type system.

The type rules are highly nondeterministic. We have developed two (equivalent) ways of managing nondeterminism to implement the rules but we only show here one of them (see [17] for the alternative implementation). Both implementations infer the process type as the result of the rewriting.

Additionally, as a consequence of the study of the type rules we have encountered that by adding a new rule, more processes that do not produce communication errors can be typed, and consequently we have slightly increased the power of the type system.

The representation in Maude of the AC operational semantics and type systems, in a way quite close to the original mathematical formulation, has provided us interpreters where these inference systems can be executed. Our final aim is to go one step further in the exploitation of using Maude and take advantage of the formal tools designed for this language. For example, automatic reasoning about specifications in Maude is supported by the experimental ITP tool [6], a rewriting-based theorem prover (implemented also in Maude) that can be used to prove inductive properties of equational specifications.

The rest of the paper is organized as follows. Section 2 presents a short description of Maude and the strategy language we use (in [8,12] you may find more details and illustrating examples). Section 3 reviews the ambient calculus briefly. Section 4 describes the implementation of the operational semantics of AC and uses it to execute a non-trivial example. Section 5 enriches the semantics with error rules and implements the type system for avoiding communication errors. Finally Section 6 gives conclusions and future work. Figures A.1 to A.4, showing the syntax and operational semantics of the Ambient Calculus and Cardelli and Gordon's type system, appear in Appendix A.

2 Maude in a Nutshell

In rewriting logic and Maude the data on the one hand and the state of a system on the other are both formally specified as an algebraic data type by means of an equational specification. Maude uses a very expressive version of equational logic, namely *membership equational logic* [1]. In this kind of specifications we can define new types (by means of keyword `sort(s)`); subtype relations (understood as inclusion relations) between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; equations (`eq`) that identify terms built with these operators; and memberships (`mb`) $t : s$ stating that the term t has sort s. Both equations and memberships can be conditional, with respective keywords `ceq` and `cmb`. Conditions are formed by a conjunction (written `/\`) of equations and memberships.

Equations are assumed to be confluent and terminating, that is, we can use the equations from left to right to reduce a term t to a unique, canonical form t' (modulo the operators attributes as associativity, commutativity, and identity) that is equivalent to t (they represent the same value).

The *dynamic* behavior of a system is specified by rewrite rules of the form

$$t \longrightarrow t' \ \ \textit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t and the conditions are fulfilled, it can be transformed into the corresponding instance of the pattern t'.

2.1 Strategies

Because system modules are rewrite theories that do not need to be either confluent or terminating, we need to have good ways of controlling the rewriting inference process—which in principle could not terminate or may go in many undesired directions—by means of adequate *strategies*. We have defined a strategy language for Maude that can be used to control how rules are applied to rewrite a term [12]. The simplest strategies are the constants `idle`, which always succeeds by doing nothing, and `fail`, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control by means of search expressions how the rewrite conditions are solved. An operation `top` to restrict the application of a rule just to the *top* of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (`;`), union (`|`), and iteration (`*` for 0 or more iterations, `+` for 1 or more, and `!` for a "repeat until the end" iteration). Another strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy. The language also provides an `orelse` combinator where the second strategy is applied only when the first one is unsuccessful, and a (`x`)`matchrew` combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten.

3 Ambient Calculus

In this section we will present the basic notions about the Ambient Calculus [2], a process algebra that focuses on the notions of locations, mobility (of agents and their environments) and authorizations (to move or interact). *Ambients* will be the main entities of this model. An ambient is a place limited by a boundary where computations take place. They are hierarchically structured, so that we do not abstract from the path needed to arrive at each destination. Agents are confined to ambients and ambients move under the control of agents, allowing the movement of nested environments, that also include data and live computation.

In Figure A.1 the syntax for AC with communication primitives is presented. We will consider two disjoint sets, $\mathcal{N} = \{m, n, \ldots\}$ for names and $\mathit{Var} = \{x, y, \ldots\}$ for variables, and a special symbol ϵ. We will denote $Id = \mathcal{N} \cup \mathit{Var}$ for identifiers and $\mathit{Cap} = \{\mathit{in}\ N, \mathit{out}\ N, \mathit{open}\ N \mid N \in Id\}$ for capabilities. We will write as A^* the set of paths (sequences) formed over elements of A.

Ambients are denoted as $n[P]$, where n is its name and P is its content, which is essentially a parallel composition of sequential processes and subambients. These sequential processes can be prefixed processes, $M.P$, meaning that it must consume M before behaving as P; polyadic inputs $(x_1 : W_1, \ldots, x_n : W_n)P$; and polyadic asynchronous outputs $\langle M_1, \ldots, M_n \rangle$. We will assume that the variables appearing in the input construction are pairwise distinct. Also, new names can be created (restriction) $(\nu n : W)P$ and processes may be replicated $!P$. There is a special process 0 that is inactive.

Notice that for simplicity, in the syntax definition ambient names and capabilities belong to the same syntactic category. As a consequence the syntax allows the construction of meaningless processes such as $n.P$ or $\mathit{in}\ n[P]$. Later these terms will be ruled out by the type system that we will discuss in Section 5.

The operational semantics of the language is defined by means of a structural congruence relation $\equiv$ and a reduction relation $\rightarrow$. The former basically identifies those processes that are equivalent up to some trivial syntactic reorganization. It is the least equivalence relation satisfying the rules in Figure A.2. For example, the inactive process 0 can be eliminated (or added) when in parallel with other processes.

In addition, processes are identified by α-conversion up to the renaming of bound names and variables:

$$(\nu n : W)P = (\nu m : W)P\{n := m\} \quad \textit{if } m \notin \textit{fn}(P)$$

$$(x_1, \ldots, x_n)P = (y_1, \ldots, y_n)P\{x_i := y_i\} \quad \textit{if } y_i \notin \textit{fv}(P)$$

A restricted name cannot be used outside its scope. However, α-conversion can be used to avoid name clashes, and in this way it is reflected the fact that the restricted name cannot be known, in principle, out of the restricted term. By means of the extrusion rule we can augment the scope of the restriction from a parallel component to the whole parallel composition, provided the restricted name does not appear in the other components:

$$P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) \quad \textit{if } n \notin \textit{fn}(P)$$

As said before, if process P above does have n as a free name and we want P and Q to interact we can always apply α-conversion. This can also be applied to ambients, as rule (Struct Res Amb) shows.

The reduction rules mainly present the axioms for mobility and communication. Ambients can move into their sibling ambients or out of their enclosing ambient, as said in rules (Red In) and (Red Out) respectively. They may also dissolve the boundary of their subambients, so that the processes contained in the opened ambient now belong to the opener ambient, as defined in rule (Red Open). Finally, communication may happen inside them (Red Comm). The rest of the rules state that reductions may occur inside some constructors, namely restriction, ambients, and parallel, but not inside inputs, prefixes, or replications. Finally, rule (Red $\equiv$) makes explicit the fact that we are working modulo structural equivalence.

As an illustrative example of the semantics, let us consider the example

$$n[a[\textit{out } n.\textit{in } m.\langle M\rangle]] \mid m[\textit{open } a.(x)Q]$$

This process can evolve in the following way:

$$
\begin{array}{ll}
(n[a[\mathit{out}\ n.\mathit{in}\ m.\langle M\rangle]] \mid m[\mathit{open}\ a.(x)Q]) & \equiv \\
(n[a[\mathit{out}\ n.\mathit{in}\ m.\langle M\rangle] \mid 0] \mid m[\mathit{open}\ a.(x)Q]) & \rightarrow \\
a[\mathit{in}\ m.\langle M\rangle] \mid n[0] \mid m[\mathit{open}\ a.(x)Q] & \equiv\ (1) \\
n[0] \mid a[\mathit{in}\ m.\langle M\rangle \mid 0] \mid m[\mathit{open}\ a.(x)Q] & \rightarrow\ (2) \\
n[0] \mid m[a[\langle M\rangle \mid 0] \mid \mathit{open}\ a.(x)Q] & \equiv \\
n[0] \mid m[\mathit{open}\ a.(x)Q \mid a[\langle M\rangle]] & \rightarrow \\
n[0] \mid m[(x)Q \mid \langle M\rangle] & \rightarrow \\
n[0] \mid m[Q\{x := M\}] &
\end{array}
$$

where, for instance, the equivalence (1) can be proved to hold using rules (Struct Par), (Struct Par Comm), (Struct Zero Par) and (Struct Amb), and step (2) can take place using the rules (Red Par) and (Red In).

4 An Implementation of Mobile Ambients in Maude

In this section we implement in Maude the operational semantics of AC. We have tried to be as faithful as possible to the way in which the calculus was originally described. First, we define the syntax, and then we implement the operational semantics through both equations and rewrite rules. Finally we define strategies that control the application of rewrite rules. All the code is available in Maude's site [16].

4.1 Syntax definition

We define here AC syntax. For the sake of readability we omit variable declarations and operators precedence in most of the source code.

Syntax definition has to consider how to deal with bound names and variables. In AC there are two binding operators: the creation of new names (νn) that binds names and the input action (x) that binds variables. We need de Bruijn's indexes in order to distinguish occurrences of the same name or variable that are bound by different binding operators [18]. In an indexed name n_i, i represents the number of intermediate n-bindings between the free occurrence and its binding occurrence.

Consequently we have to use indexed names and variables in AC syntax. We can use Maude's `Qid` to define both. For the sake of clarity, names are `Qids` beginning with letters 'a' to 't', and variables those beginning with letters 'u' to 'z', which can easily be defined using membership axioms. Indexed names and variables, which we call `Acid`, are defined as:

```
sorts Qidn Qidx .
subsorts Qidn Qidx < Qid .

var q : Qid .
cmb q : Qidn if first-char(q) < "u" .
cmb q : Qidx if first-char(q) >= "u" .

sorts Name Var Acid .
subsorts Name Var < Acid .

op _{_} : Qidx Nat -> Var .
op _{_} : Qidn Nat -> Name .
```

Additionally, we need functions to manage indexed names and variables [18]. Essentially, they increment indexes whenever necessary to avoid name clashes.

Notice that in a system defined by a user that wants to execute an example, every name and variable has a 0 index, as indexes different from 0 only arise through communication or α-conversion. So we have defined a decoration function `dec` that fills the system defined by the user with the appropriate 0 indexes. We do not give here its definition (see [16] for the complete code).

Having defined names and variables, we can define straightforwardly messages and processes. Messages can be (indexed) names and variables, basic values (such as integers), capabilities, and paths:

```
sorts Message Capability Path .
subsorts Int Acid Capability Path < Message .

op in[_]   : Message -> Capability .
op out[_]  : Message -> Capability .
op open[_] : Message -> Capability .

op eps : -> Path .
op _._ : Message Message -> Path [assoc] .
```

In order to define processes we need to define first input and output sequences so that multiple communication can take place. Input sequences should rule out multiple occurrences of the same variable (this is done by defining the concatenation operator `_,_` for input sequences as partial `~>`, and giving a conditional membership that states when the concatenation is meaningful). Input sequences include type annotations (sort `AType`) for each input variable; this is because later we will define a type system for AC (for the moment they can be ignored).

```
sort InputSeq .
```

```
op _:_ : Qidx AType -> InputSeq .
op _,_ : InputSeq InputSeq ~> InputSeq [assoc] .

op bel : Qidx InputSeq -> Bool .
eq bel(x, y : T) = x == y .
eq bel(x, (I1, I2)) = bel(x, I1) or bel(x, I2) .

cmb (( x : T ), IS) : InputSeq if not bel(x, IS) .

sort OutputSeq .
subsorts Message < OutputSeq .
op _,_ : OutputSeq OutputSeq -> OutputSeq [assoc] .
```

Processes are defined as follows

```
sorts NSProcess Process .
subsort NSProcess < Process .

op stop : -> Process .  *** 0 process
op _._ : Message Process -> NSProcess .
op _|_ : Process Process -> Process [assoc comm id: stop] .
op _|_ : NSProcess NSProcess -> NSProcess [assoc comm id: stop] .
op !_ : Process -> Process .
op !_ : NSProcess -> NSProcess .
op _`[_`] : Message Process -> NSProcess .
op <_> : OutputSeq -> NSProcess .
op `(_`)_ : InputSeq Process -> NSProcess .
op new`[_:_`]_ : Qidn AType Process -> Process .
op new`[_:_`]_ : Qidn AType NSProcess -> NSProcess .
```

where we use two different sorts: `NSProcess` for processes that are different from stop, and `Process` for every process. We will see the advantages of this approach in the next section. From now on, we will use `P`, `Q`, `R` as variables of sort `Process`, and `NSP`, `NSQ`, `NSR` as variables of sort `NSProcess`.

Using this syntax definition we can write the following firewall example shown in [2]:

```
op firewall : Process Process -> Process .
eq firewall(P,Q) = new ['k : Amb[Shh]] ('n [open['k] . P]
 | new['m : Amb[Shh]] ('m ['k [ out['m] . in['n] . in['m] . stop] | Q])) .
```

where the type annotations can be ignored by now. Ambient `'m` can be regarded as a firewall that an agent `'n` wants to cross. The above mechanism can be used to guarantee authentication, to ensure freshness of messages by means of nonces or to model shared-key cryptography.

4.2 Operational semantics

The operational semantics for AC consists of a set of structural congruence rules and a set of reduction rules. Happily, Maude gives us some congruence rules for free. In particular:

- Rules (Struct Res) to (Struct Action) and (Struct Input), which define the congruence with respect to each process constructor, do not need to be defined due to equational congruence in Maude.
- Rules (Struct Par Assoc), (Struct Par Comm) and (Struct Zero Par) are obtained by indicating in the declaration of the parallel operator the associativity, commutativity, and identity attributes.

Rules (Struct ϵ), (Struct Path), (Struct Zero Res) and (Struct Zero Repl) are defined through Maude equations and consequently will be applied only from left to right. We write them looking for a *normal form* so that confluence holds:

```
eq eps . P = P .
eq (M . N) . P = M . (N . P) .

eq ! stop = stop .
eq new[n : T] stop = stop .
```

Extrusion rules (Struct Res Res), (Struct Res Par), and (Struct Res Amb) are written as three equations for α-conversion and (alphabetic) reordering of bound names:

```
ceq new[k : T1] new[l : T2] P = new[l : T2] new[k : T1] P
 if string(l) < string(k) .
eq ((new[n : T] NSP) | NSQ) = new[n : T](NSP | ([shiftup n] NSQ)) .
eq M [new[n : T] P] = new[n : T](([shiftup n] M)[P]) .
```

where `shiftup` increments indexes adequately.

Notice that for the first rule to be really confluent we should define a (merely syntatic) order between the types in the (unfrequent) case the same name with different types is used. In that case, the reordering should rearrange the indexes of the involved names.

Notice that we use in the second (α-conversion) rule variables `NSP` and `NSQ` of sort `NSProcess` so that it is only applicable when processes are different from `stop`. If we used variables of sort `Process`, the identity attribute of `|` could produce an infinite loop by generating once and again `| stop` in order to match with the equation. We will use the same idea in similar situations below. This approach allows us to avoid conditional equations that would increase the execution time.

We do not lose any power by writing the previous congruence rules as equations as they only reorder terms so that the subsequent reduction rules can be applied. For this to be true the parallel operator attributes and equational congruence are fundamental. The application of the equations produces a *normal form* where:

- `stop` only appears after a prefix (capability or input action) or inside an ambient;

- eps does not appear anywhere;
- sequences of capabilities associate to the right; and
- new operators are extruded as far as possible (so that interactions can take place) and are ordered alphabetically.

Rule (Struct Repl Par) will be discussed later. Finally we have the reduction rules as rewrite rules in Maude, some of which are conditional rewrite rules:

```
rl [RedIn]   : n[in[m] . P | Q] | m[R] => m[n[P | Q] | R] .
rl [RedOut]  : m[n[out[m] . P | Q] | R] => n[P | Q] | m[R] .
rl [RedOpen] : open[n] . P | n[Q] => P | Q .
rl [RedComm] : ((I)P) | < O > => bound(I,O) P .
crl [RedRes] : new[k : T] P => new[k : T] Q if P => Q .
crl [RedAmb] : n[P] => n[Q] if P => Q .
crl [RedPar] : NSP | NSR => Q | NSR if NSP => Q .
```

The function bound(I,O) generates a substitution as a result of the communication that is then applied to the process.

Notice that we have the interleaving of congruence and reduction rules (Red $\equiv$) for free as Maude itself interleaves the application of equations with rewrite rules.

However, the reduction relation of the calculus is not a congruence for all the operators, but only for the restriction operator (Red Res), ambient construction (Red Amb), and parallel operator (Red Par). This means that we cannot freely use the rewrite rules we have written, as Maude would apply them anywhere in a term; and we do not want them to be applied after prefixes, inputs, and replication. This is one of the reasons why the definition of a strategy that controls the application of these rules is necessary.

We study now what happens with replication. Replication behavior is described in AC through a congruence rule (Struct Repl Par). We cannot write it as an equation as the other ones because none of the orientations is convenient. If we apply it from left to right we get an infinite loop as we can infinitely unroll the replication. If we apply it from right to left we cannot see how the system evolves when new copies of the replicated process interact with other parts of the system or even with other copies of itself. As an example, for the process $!n[in\ n.0]$ to evolve it is necessary to unroll replication twice:

$$\begin{array}{ll} !n[in\ n.0] & \equiv \\ n[in\ n.0]\ |!n[in\ n.0] & \equiv \\ n[in\ n.0]\ |\ n[in\ n.0]\ |!n[in\ n.0] & \rightarrow \\ n[in\ n.0|n[0]]\ |!n[in\ n.0] & \rightarrow \\ \ldots & \end{array}$$

This has led us to write this congruence rule as two rewrite rules:

```
rl [Rep] : ! P => P | ! P .
rl [UnRep] : P | ! P => ! P .
```

Still we have the same problem, so we have to define strategies to control the application of these rules. We want to apply rule `Rep` only when it is necessary for subsequent interactions, and rule `UnRep` to delete isolated unnecessary copies of the replicated process.

4.3 Strategies for evaluation

We need strategies to control the application of the rewrite rules defined before. Rules for movement and communication can be applied anywhere in the term but under prefixes and replication. So we first define a strategy to control the application of these rules called `norep` (no replication). As we have written rules for reducing inside ambients, in parallel processes, and under name restriction, we just have to apply all the rewrite rules at the top level. This means that the strategy will be applied recursively but that it will stop when a prefix or a replication is encountered.

Rules `RedRes`, `RedAmb` and `RedPar` are conditional rewrite rules so the strategy needs to know which strategy to apply in the rewrite condition and how to search in the resulting rewrite tree. In this case we want the same strategy to be (recursively) applied and a depth first search is enough for our purposes (strategies are defined in a `seq` declaration):

```
seq norep = top(RedIn) | top(RedOut) | top(RedOpen) | top(RedComm) |
            top(RedAmb{dfs(norep)}) |
            top(RedPar{dfs(norep)}) |
            top(RedRes{dfs(norep)}) .
```

Now we combine this strategy with a new one to control replication. We would like to unroll replication only when *necessary*, that is, when as a consequence of the unrolling, a movement or a communication takes place. However we have to be careful because two unrollings could be *necessary* for the movement or the communication to take place, as happened in process $!n[in\ n.0]$. Additionally, even when one unrolling is enough to make a reduction step, we could lose rewrites if we force such reduction immediately. For example, if our strategy applied `norep` after each unrolling to process $n[0]\ |!n[in\ n.0]$, we would obtain the following rewriting:

$$\begin{array}{ll} n[0] \mid !n[in\ n.0] & \equiv \\ n[0] \mid n[in\ n.0] \mid !n[in\ n.0] & \rightarrow \\ n[n[0]] \mid !n[in\ n.0] & \equiv \\ n[n[0]] \mid n[in\ n.0] \mid !n[in\ n.0] & \rightarrow \\ n[n[0] \mid n[0]] \mid !n[in\ n.0] & \ldots \end{array}$$

so that only processes like $n[n[0] \mid \ldots \mid n[0]] \mid !n[in\ n.0]$ could be obtained, losing (among others) the following possible rewriting:

$$\begin{array}{ll} n[0] \mid !n[in\ n.0] & \equiv \\ n[0] \mid n[in\ n.0] \mid !n[in\ n.0] & \equiv \\ n[0] \mid n[in\ n.0] \mid n[in\ n.0] \mid !n[in\ n.0] & \rightarrow \\ n[0] \mid n[in\ n.0 \mid n[0]] \mid !n[in\ n.0] & \rightarrow \\ n[n[n[0]] \mid !n[in\ n.0] & \rightarrow \\ \ldots & \end{array}$$

We claim that two unrollings are enough to obtain all the solutions, meaning by solutions those processes to which no reduction rule can be applied. This can be easily proved by inspection of the rewriting trees for $S \mid P \mid P \mid !P$ and $S \mid P \mid P \mid P \mid !P$. The only difference is the level where we find the solutions.

Considering the two previous observations we define a new rule that allows us to unroll twice any replication appearing in the process but after prefixes and under replication (for the same reasons as `norep`)

```
rl [Rep2] : P => rep(P) .
```

being `rep` defined as

```
op rep : Process -> Process .
eq rep(! P) = P | P | ! P .
eq rep(M[P]) = M[ rep(P) ] .
eq rep(NSP | NSQ) = rep(NSP) | rep(NSQ) .
```

```
eq rep(new[n : T] P) = new[n : T] rep(P) .
eq rep(P) = P [owise] .
```

As we want the unrolling to affect the whole process, this rule should be applied also at the top level

```
seq unroll-rep = top(Rep2) .
```

Of course, it can happen that unrolling does not help to the evolution of the process and just generates idle copies. In this case we apply rule `UnRep` to absorb those garbage copies. As an example, by unrolling twice and then communicating, process $\langle n\rangle \mid !(x)x[0]$ would rewrite to $n[0] \mid (x)x[0] \mid !(x)x[0]$ and then by applying rule `UnRep` we would obtain $n[0] \mid !(x)x[0]$.

Additionally, in order to avoid infinite computations when processes are nonterminating the user should tell the strategy how many semantic reduction steps he wants to execute. Consequently, the strategy applies replication unrolling (if there is any) and immediately applies one more movement and/or communication step (if it is possible and we are not finished). When we are finished we eliminate every idle copy.

```
seq cg(0) = UnRep ! .
seq cg(s(n:Nat)) = (unroll-rep ; norep ; cg(n:Nat))
                   orelse (UnRep !) .
```

For example, when rewriting

```
!('n{0} [in['n{0}] . stop] | 'n{0} [in['n{0}] . stop])
```

by using strategy `cg(1)` we obtain the following solution:

```
!('n{0}[in['n{0}]. stop]| 'n{0}[in['n{0}]. stop]) |
  'n{0}[in['n{0}]. stop | 'n{0}[stop]]
```

where one copy of the replicated process has evolved. By applying `cg(2)` we obtain the following three solutions:

```
Solution 1 :
!('n{0}[in['n{0}]. stop] | 'n{0}[in['n{0}]. stop]) |
  'n{0}[in['n{0}]. stop] | 'n{0}[in['n{0}]. stop | 'n{0}[stop]| 'n{0}[stop]]
Solution 2 :
!('n{0}[in['n{0}]. stop] | 'n{0}[in['n{0}]. stop]) |
  'n{0}[in['n{0}]. stop] | 'n{0}[in['n{0}]. stop | 'n{0}['n{0}[stop]]]
Solution 3 :
!('n{0}[in['n{0}]. stop] | 'n{0}[in['n{0}]. stop])|
  'n{0}[in['n{0}]. stop | 'n{0}[stop]] | 'n{0}[in['n{0}]. stop | 'n{0}[stop]]
```

obtained by only two movements and one final application of `UnRep`. The three possibilities can be easily obtained by writing process $n[in\ n.0] \mid n[in\ n.0] \mid n[in\ n.0] \mid n[in\ n.0]$ and all the possible ways of making only two movements.

As a final example, when rewriting `firewall(P,Q)` using `cg(4)` we obtain:

```
new['k : Amb[Shh]]new['m : Amb[Shh]]('m{0}[Q | 'n{0}[[shiftup 'm]P]])
```

as expected. By using process variables `P` and `Q` we are able to universally quantify the execution of the firewall example and give a general result for any two processes. However, there are some operations that cannot be applied, like `shiftup`, and are left as such.

4.4 An Example: Electoral Systems

In [13] the problem of coding pure ambient calculus in π-calculus is studied. In particular, it is shown that symmetric electoral systems of arbitrary size exist for pure ambient calculus (AC with no communication), which implies that AC is not encodable in the π-calculus with separate choice. The authors of [13] claim that the following process is a symmetric electoral system:

$$Net_k = P_0 \mid \ldots \mid P_{k-1}$$

$$P_i = n_i[\prod_{j \in S_i^k} in\ n_j.0 \mid \prod_{s \in T_i^k} m_i[in(s).out(s^-).out\ n_i.0]]$$

where $\prod$ denotes parallel composition, S_i^k is the set of all natural numbers less than k excluding i, T_i^k is the set of all strings of length $k-1$ using the members of S_i^k exactly once each, s^- is the string s in reverse order and $in(s)$ is the sequence of $in\ n_j$ for each successive $j \in s$ (analogously, $out(s)$).

For a symmetric net as the one above to be an electoral system it must be the case that all of its maximal computations produce exactly one *observable*, being all of them different. In this case, the observables are the ambients with names in $\{m_1, \ldots, m_{k-1}\}$ at the top level. We have implemented the example above in our representation of AC:

```
op Net : Nat -> Process .
eq Net(k) = elect(0, k) .

op elect : Nat Nat -> Process .
ceq elect(i, k) = Pr(i, k) | elect(i + 1, k) if i < k .
eq elect(k, k) = stop .
```

where `Pr(i, k)` implements P_i (see details in [17]).

We can now take profit from our implementation of ambients to check that if we rewrite `Net(2)` by using strategy `cg(400)` we obtain: [2]

```
Solution 1 :
  'm{1}[stop] | 'n{1}[in['n{0}]. stop |
                      'n{0}['m{0}[in['n{1}]. out['n{1}]. out['n{0}].stop]]]
Solution 2 :
  'm{0}[stop] | 'n{0}[in['n{1}]. stop |
```

[2] We use 400 as a limit for the number of reduction steps in the strategy to make sure we obtain the maximal rewritings. This does not affect the efficiency.

```
                         'n{1}['m{1}[in['n{0}]. out['n{0}]. out['n{1}].stop]]]
No more solutions.
```

Indeed, there are only two possible (maximal) rewritings: solution 1 corresponds to observable `'m{1}` (n_1 wins) and solution 2 to observable `'m{0}` (n_0 wins). The same can be done with nets of size bigger than 2, getting analogous results.

5 A Type System for Mobile Ambients

In this section we first present Cardelli and Gordon's type system for detecting communication errors. Then we define error reductions that precisely describe such errors and prove (by hand) that a well-typed process does not produce these communication errors along its execution (for more details see [15]). We implement the error reductions and define a strategy that allows us to know if a communication error occurs along the execution of a process. Then we implement the type system inferring the type of an annotated process as a result of the rewriting. Additionally, as a consequence of the study of the typing rules we have encountered that by adding a new rule, more processes that do not produce communication errors can be typed, and consequently we have slightly increased the power of the type system.

5.1 Types for the Ambient Calculus

In [5] the first type system for the Ambient Calculus is presented. Its main purpose is to avoid meaningless processes. Such processes may arise after some undesired communication interactions. For instance, the process $(x)x[P] \mid \langle n\rangle \mid (y)y.Q \mid \langle open\ n\rangle$ may evolve to $n[P] \mid open\ n.Q$ but also to $(open\ n)[P] \mid n.Q$.

One way to avoid these meaningless terms[3] is to restrict the type of communications within each ambient, thus defining the *exchange types*. These types will not only specify whether ambients or capabilities are exchanged, but also what kind of ambients (what kind of information can be exchanged inside them) or what kind of capabilities (what kind of messages they unleash).

There are two kinds of exchange types: one for no exchange, *Shh*, and another one for tuple exchange, where each component will be an ambient type or a capability type, as shown in Figure A.3.

The judgments of the type system, $\Gamma \vdash P : T$ and $\Gamma \vdash M : W$, are derived with respect to a type environment, as usually. Now we comment some of the

[3] In fact they are only meaningless at the intuitive level. Formally they just include useless blocked subterms.

typing rules shown in Figure A.4:

- (Zero) Process 0 does not produce any communication action. Thus, its natural type should be *Shh*. However, it can be understood that it has any type, so that if it is in parallel with any other process, it does not interfere with its communication behavior. Alternatively, it would be possible to introduce a subtype relation among types, giving 0 the minimal type, together with a new subsumption rule, as done in [22].
- (Amb) In order to type an ambient $M[P]$ one must check that its name is indeed an ambient name, $M : Amb[T]$. As process 0, it can be typed with any type.
- (In/Out) Movement capabilities do not unleash any exchange and, therefore, they can produce any capability type.
- (Open) If $M : Amb[T]$ then M is an ambient that contains processes of type T and, therefore, *open* M is a capability that may unleash exchanges of type T.
- (Prefix) This rule forces P and $M.P$ to have the same type, which is the type determined by the prefix M when M is a capability *open* or a path containing one.
- (Parallel) Every sequential process within the same ambient must have the same type. This will only be a restriction for those processes that are responsible for communications.
- (Input) The residual of the input must be typeable with the same type that determines the input. Therefore, the communication type will be the same along the execution of the process.
- The rest of the rules are standard.

Rules (In/Out) and (Open), together with rule (Prefix) causes the opening capabilities to be the only ones that contribute to the type of a path. For example, if $\Gamma(n) = Amb[T]$ and $\Gamma(m) = Amb[S]$ then it holds that $\Gamma \vdash in\ n.open\ m : Cap[S]$.

5.2 Communication errors

In this section we formalize the meaning of types in the previous type system. These types are intended to capture syntactic errors arising from the use of two different kind of entities (names and capabilities) in the same syntactic category. In order to relate the type system with the operational semantics, we define in Figure 1 an error relation err_1. An error is found, for instance, whenever a name is prefixing a process (instead of a capability). The definition

$$\frac{M \notin (Var \cup Cap)^*}{M.P \to err_1} \qquad \frac{M \notin Id}{M[P] \to err_1} \qquad \frac{M_i \notin Id \cup (Cap \cup Var)^*}{\langle \tilde{M} \rangle \to err_1}$$

$$\frac{P \to err_1}{N[P] \to err_1} \qquad \frac{P \to err_1}{N.P \to err_1} \qquad \frac{P \to err_1}{(\tilde{x} : \tilde{W})P \to err_1}$$

$$\frac{P \to err_1}{(\nu n : W)P \to err_1} \qquad \frac{P \to err_1}{P \mid Q \to err_1} \qquad \frac{Q \to err_1}{P \mid Q \to err_1} \qquad \frac{P \to err_1}{!P \to err_1}$$

Fig. 1. Rules for syntactic errors

of err_1 attempts to detect the error as soon as possible, in the sense that it looks in every subcomponent of the process, without considering variables, since we do not know what they will be replaced by.

If we suppose that $P \not\to err_1$ and $Q \not\to err_1$ then we can easily verify that:

$$err_1 \not\leftarrow (x : W)x[P] \mid \langle n \rangle \mid (y : W')y.Q \mid \langle open\ n \rangle \begin{cases} \nearrow \ n[P] \mid open\ n.Q \not\to err_1 \\ \searrow \ (open\ n)[P] \mid n.Q \longrightarrow err_1 \end{cases}$$

Notice that this error relation describes a dynamic behavior, while the type system tries to statically capture it. Frequently, the type system is not complete and non-typeable terms would have to be executed in order to know whether they produce an error. Consequently, the implementation of the error relation is useful to effectively know if a term produces such kind of error. Moreover, it would also be needed in order to mechanically prove the correctness property using Maude (see Section 6).

Here, we have proved by hand that typed processes do not cause such error. First we need an easy to prove lemma:

Lemma 5.1

(i) *If* $\Gamma \vdash M : Amb[T]$ *then* $M \in Id$.

(ii) *If* $\Gamma \vdash M : Cap[T]$ *then* $M \in (Cap \cup Var)^*$.

(iii) *If* $\Gamma \vdash M : W$ *then* $M \in Id \cup (Cap \cup Var)^*$.

and then we can prove the main theorem

Theorem 5.2 *If* $\Gamma \vdash P : T$ *then* $P \not\to err_1$.

Proof. [sketch] This result can be proved by induction on the rules used to derive $\Gamma \vdash P : T$ and using the previous lemma. Basically, it holds because processes that cause an error are those containing a subterm of the form

$(cp\ N)[P]$, $\langle cp\ (cp'\ N)\rangle$, $n.P$, or $cp\ (cp'\ N).P$ (with $cp, cp' \in \{in, out, open\}$). These processes are not typeable, nor any process that contains them (in the type system every subterm must be typed in order to type the whole term).□

A subject reduction theorem for exchange types is proved in [5]. Using it we get our safety theorem:

Theorem 5.3 *If* $\Gamma \vdash P : T$ *and* $P \rightarrow^* Q$ *then* $Q \not\rightarrow err_1$.

It is straightforward to implement the error relation in Maude. We consider err_1 as a constant process `err1` and introduce rewritings from erroneous processes (according to the conditions stated in Figure 1) to `err1`.

```
op err1 : -> Process .
crl [errPref] : M . P => err1 if not isCap(M) .
crl [errAmb]  : M[P]  => err1 if not isAmb(M) .
crl [errMsg]  : < O > => err1 if not isMsg(O) .
```

The fact that errors are transmitted to the rest of the process is defined by the following equations, stating that any process containing an erroneous subterm is erroneous:

```
eq M[err1] = err1 .
eq err1 | NSP = err1 .
eq M . err1 = err1 .
eq ! err1 = err1 .
eq (I) err1 = err1 .
eq new[ n : T ] err1 = err1 .
```

Therefore, an error occurs whenever one of the three error rules above can be applied. The strategy `error1` tries to apply one of those rules. Then, `errcg` is a slight variation of the strategy `cg` in Section 4.3. It restricts normal steps to happen only when no error can be produced:

```
seq error1 = errPref | errAmb | errMsg .
seq errcg(0) = error1 orelse cg(0) .
seq errcg(s(n:Nat)) = error1 orelse (cg(1) ; errcg(n:Nat)) .
```

This strategy allows us to know if a given process produces sometime along its execution an `err1`. For example, the previous example written in Maude

```
eq fail = ((’x : Amb[Shh]) (’x [P])) | < ’n > |
          ((’y : Cap[Shh]) (’y . Q)) | < open[’n] > .
```

rewritten with `errcg(3)` produces `P | Q` but also `err1`.

5.3 Implementation of the type system

In order to implement the type system we first define the syntax for types. We have sorts `EType` representing exchange types and `MType` representing message types. We also need `TMType` to represent tuples of message types. We have included a basic type for the integers `bint`.

```
sorts EType MType TMType .   subsorts MType < TMType < EType .

op bint : -> MType .
op Shh : -> EType .
op _x_ : TMType TMType -> TMType [assoc] .
op Amb[_] : EType -> MType .
op Cap[_] : EType -> MType .
```

Types decorate restricted names and input variables. When we defined ambients syntax, identifiers were decorated with (still not defined there) annotation types `AType`. As several type systems can be defined over the same syntax, we have decided to use `AType` as a supertype of any type that could annotate identifiers in a given type system. So when using a specific type system we have to say which types are used to annotate:

```
subsort MType < AType .
```

Typing environments assign types to (indexed) names and variables. We have defined them over `AType` so that they can be used in other type systems. Their treatment is standard (see [17] for more details).

We now have to define the rules of the type system (Figure A.4). We have written type judgements like $\Gamma \vdash P : T$ as rewrite rules $(\Gamma \vdash P) \longrightarrow T$ where a typing environment and a process are rewritten to the type of the process. In this way we infer the type as a result of the rewriting.

So we first define the lefthand sides (for processes and messages) of the typing rewrite rules:

```
sorts JudgeP JudgeM .
op _|-_ : Env Process -> JudgeP .
op _|-_ : Env OutputSeq -> JudgeM .
```

But in order to be able to rewrite terms of sort `JudgeP` to terms of sort `EType` these sorts have to belong to the same connected component (in the Maude subsort relation):

```
subsort EType < JudgeP .  subsort TMType < JudgeM .
```

As we have previously seen, typing rules are highly nondeterministic. We would like to get inference by writing the rules as literally as possible. For example, some rules for typing messages and processes can be written as:

```
rl [Exp]    : E |- a => E[a] .
crl [Tup]   : E |- M, O => W x TW if E |- M => W /\ E |- O => TW .
crl [Open] : E |- open[M] => Cap[T] if E |- M => Amb[T] .

crl [Repl]   : E |- ! P => T if E |- P => T .
crl [Output] : E |- < O > => TW if E |- O => TW .
crl [Res]    : E |- new[n : Amb[T]] P => S if E[n -> Amb[T]] |- P => S .
```

Nondeterministic rules like (Zero) cannot be literally written as a rewrite rule, as we cannot rewrite to a partially undefined term. The same happens with rules (Empty), (In), and (Out) for typing messages. In fact, when we conclude that 0 has type T, we are saying that such T could be any type. Following this idea we define a new type constant `X` which means *any process type*:

```
op X : -> EType .
```

so that now we can write the following rules for messages

```
rl [Empty] : E |- eps => Cap[X] .
crl [In]   : E |- in[M] => Cap[X] if E |- M => Amb[T] .
crl [Out]  : E |- out[M] => Cap[X] if E |- M => Amb[T] .
```

and for 0 process

```
rl [Zero] : E |- stop => X .
```

We still have to write rules for (Path), (Prefix), (Amb), (Par), and (Input). Let us study rule (Par), the rest of them are similar (see [17]). Rule (Par) requires that the processes in parallel have the same type, so we could write:

```
crl [Par] : E |- NSP | NSQ => T
 if E |- NSP => T /\ E |- NSQ => T .
```

but now we have a new type `X` that is any type and consequently that is compatible with any other one, so we need to add a new rule saying this:

```
crl [Par2] : E |- NSP | NSQ => T
 if E |- NSP => T /\ E |- NSQ => X .
```

If any of the processes (or both) has type `X`, then they are compatible and the process can be typed. Due to commutativity we do not need to write a third rule.

While writing these rules we have noticed that rule (Open) is more restrictive than needed. If we try to type process $(\nu n : Amb[Shh])(open\ n.\langle n\rangle) \mid n[0])$, rule (Open) would give type $Cap[Shh]$ to *open* n and consequently, rule (Prefix) could not be applied as $\langle n\rangle$ has type $Amb[Shh]$. However, when n is opened no communication error happens and the process just evolves to $(\nu n : Amb[Shh])\langle n\rangle$ with type $Amb[Shh]$.

The problem is that rule (Open) has not distinguished the case when the opened ambient has a silent type, like in the example. So we replace the previous rule `Open` by the following ones:

```
crl [OpenShh] : E |- open[M] => Cap[X] if E |- M => Amb[Shh] .
crl [Open]    : E |- open[M] => Cap[TW] if E |- M => Amb[TW] .
```

where the first one can only be applied to silent ambients and the other one to non-silent ambients.

The advantage of this form of implementation is that the rules are simple translations of the original rules being its disadvantage that in some cases they have to be duplicated. However, such duplication can be easily avoided by defining a *partial* function that computes the resulting type covering the different possibilities arising in the premises. For example, the rules Par and Par2 would merge into the following rule

```
crl [Par] : E |- NSP | NSQ => T''
 if E |- NSP => T /\ E |- NSQ => T' /\ T'' := compare(T,T') .
```

where the operation compare is defined as

```
op compare : EType EType ~> EType [comm] .
eq compare(T, T) = T .
eq compare(T, X) = T .
```

and the matching equation (:=) binds T'' only when compare(T,T') is defined.

As an example, let us see the firewall we saw in Section 4. In order to type it we need to give particular processes P and Q. If they were stop then given the following environment

```
op E : -> Env .
eq E = ('n{0}, Amb[Shh]) ('m{0}, Amb[Shh]) ('k{0}, Amb[Shh]) .
```

the rewriting of E |- firewall(stop,stop) returns X, so it is well-typed and has any type.

The example shown in Section 4.4 does not engage in any communication and therefore, if no erroneous term appears at the beginning, neither will it appear after any number of steps. Indeed, if we define the environment giving every ambient silent type:

```
op EnvElec : Nat -> Env .
eq EnvElec(0) = empty .
eq EnvElec(s k) = ('n{k}, Amb[Shh]) ('m{k}, Amb[Shh]) EnvElec(k) .
```

then we can try to type Net(k) under environment EnvElec(k). For instance, EnvElec(3) |- Net(3) rewrites only to X.

6 Conclusions and future work

We have exploited many features of the high-level language Maude in order to implement different semantics, both operational and static ones, for the Ambient Calculus. First, we have implemented the operational semantics given by Cardelli and Gordon. Although we follow the approach used in [21] of mapping reduction rules to rewrite rules, due to the particularities of the Ambient Calculus we have used the recently designed strategy language for Maude [12] in order to control the application of the rewrite rules. As far as we know this is the first time that this language is used to implement a calculus with mobility.

The treatment we have done of the replication operator by means of rules controlled by strategies is different from the approach used in [19] for the π-calculus. In the π-calculus the reduction rules are somewhat more compositional, allowing the recursive definition of the replication operator. On the contrary, in the standard Ambient Calculus semantics there cannot be a compositional reduction rule for replication as there is a control flow from inner processes to the outside. For example, the evolution of $n[!in\ m.0] \mid m[0]$ is not defined in terms of the evolution of $!in\ m.0$, but instead making $!in\ m.0$ congruent to $in\ m.0 \mid !in\ m.0$, so that the movement can take place.

We have also implemented a type system for the Ambient Calculus defined in [5] to detect communication errors. There, only an intuitive meaning of the types was given. So first we have formally defined the errors intended to be captured by the type system and proved that well-typed processes do not produce such errors. From this result, together with the subject reduction result, we can conclude that the type system is sound (more details in [15]). Then we have implemented the typing rules. These are highly nondeterministic. Usually nondeterminism in typing rules [14] arises due to the existence of several applicable rules to the same term or because different premises can be chosen in order to type the term. Such nondeterminism is treated to get type inference algorithms by modifying the rules or by applying the nondeterministic ones following a strategy (only at certain points of the type derivation). The nondeterminism arising in this system is quite different as, even though the rules are completely syntax-directed and in this sense deterministic, the conclusions of the rules are not uniquely determined. We have shown one way of treating this nondeterminism. In [17] we present an alternative way by exploiting a flat subtype relation that is implicit in the type system. Such implementation returns the (in some sense) minimum type. Additionally, we have added a new typing rule that slightly strengthens the power of the type system.

We are extending the work presented here to more sophisticated type systems like those defined in [3,4]. We want to study if the same techniques can be applied to other calculi with mobility (AC variants) like for example Safe Ambients [10]. We also want to compare our results with existing inference algorithms like the one presented in [22], and with other formalisms like logic programming where unification in the inference process comes for free [9]. In this sense it is our aim to go further and get type reconstruction, i.e. to infer also the type annotations needed (if any) to type an initially non-annotated process. For this purpose we will have to introduce type variables in the type system and unifying mechanisms. We should also formalize some claims along the paper that have only been proved informally; for example the fact that two unrollings of a replicated process are enough to get all the possible rewritings.

Having implemented both the operational semantics and the type system in the same framework, this allows us to study properties that involve both of them. In particular, we are studying how to extend the ITP tool, the inductive theorem prover for Maude, to allow proofs by induction on the rewrite rules. In its current state, the ITP allows to work with Maude equational specifications, proving properties by induction on terms. Induction on rules would allow us to prove, in a (semi)automatic way, properties like that the two implementations of the type system are equivalent, or that typed processes do not produce errors.

References

[1] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, LNCS 1387, pages 140–155. Springer, 1998.

[3] L. Cardelli, G. Ghelli and A. D. Gordon. Mobility types for mobile ambients. In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, LNCS 1644, pages 230–239. Springer, 1999.

[4] L. Cardelli, G. Ghelli and A. D. Gordon. Ambient groups and mobility types. In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS'00*, LNCS 1872, pages 333–347. Springer, 2000.

[5] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL'99*, pages 79–92. ACM Press, 1999.

[6] M. Clavel. The ITP tool. http://maude.sip.ucm.es/itp, 2004.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, March 2004. http://maude.cs.uiuc.edu/manual.

[9] E. Giovannetti. Type Inference for Mobile Ambients in Prolog. In M. Atkinson, editor, *Proceedings of Computing: The Australasian Theory Symposium (CATS) 2004*, ENTCS 91, pages 96–115. Elsevier, 2004.

[10] F. Levi and D. Sangiorgi. Mobile Safe Ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[11] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.

[12] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004*, ENTCS 117, pages 417–441. Elsevier, 2004.

[13] I. Phillips and M. G. Vigliotti. Electoral systems in ambient calculi. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS'04*, LNCS 2987, pages 408–422. Springer, 2004.

[14] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[15] F. Rosa-Velardo. Typing techniques for security in mobile agent systems. Master's Thesis, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[16] F. Rosa-Velardo, C. Segura, and A. Verdejo. Ambients in Maude Web Page. http://maude.sip.ucm.es/ambients, 2005.

[17] F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed Mobile Ambients in Maude. Technical report 1-05, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2005. http://maude.sip.ucm.es/ambients.

[18] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000*, ENTCS 36, pages 71–92. Elsevier, 2000.

[19] P. Thati, K. Sen and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71, pages 217–237. Elsevier, 2002.

[20] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71, pages 239–257. Elsevier, 2002.

[21] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 2005. To appear.

[22] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Foundations of Software Science and Computation Structures, 3rd International Conference, FOSSACS'00*, LNCS 1784, pages 375–389. Springer, 2000.

A Ambient Calculus and Type System

$P, Q ::=$	processes
$(\nu n : W)P$	restriction
0	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	capability action
$(x_1 : W_1, \ldots, x_n : W_n)P$	input action
$\langle M_1, \ldots, M_n \rangle$	asynchronous output action
$M ::=$	messages
x	variable
n	name
$in\ M$	can enter into M
$out\ M$	can exit out of M
$open\ M$	can open M
ϵ	null
$M.M'$	path

Fig. A.1. Syntax of the Ambient Calculus

Structural congruence	
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P \mid 0 \equiv P$	(Struct Zero Par)
$(\nu n)0 \equiv 0$	(Struct Zero Res)
$!0 \equiv 0$	(Struct Zero Repl)
$P \equiv Q \Rightarrow (x_1, \ldots, x_n)P \equiv (x_1, \ldots, x_n)Q$	(Struct Input)
$\epsilon.P \equiv P$	(Struct ϵ)
$(M.N).P \equiv M.(N.P)$	(Struct Path)
Reduction	
$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$(x_1, \ldots, x_n)P \mid \langle M_1, \ldots, M_n \rangle \rightarrow P\{x_i := M_i\}_{i=1}^{n}$	(Red Comm)
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	(Red Res)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red Amb)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red $\equiv$)

Fig. A.2. Operational Semantics of the Ambient Calculus

Exchange type	
$T ::= Shh$	no exchange
$W_1 \times \ldots \times W_k$	tuple exchange
Message type	
$W ::= Amb[T]$	ambients that may contain exchanges of type T
$Cap[T]$	capabilities that may unleash exchanges of type T

Fig. A.3. Exchange Types

(Exp n)
$$\frac{\Gamma(n) = W}{\Gamma \vdash n : W}$$

(Path)
$$\frac{\Gamma \vdash M_1 : Cap[T] \Gamma \vdash M_2 : Cap[T]}{\Gamma \vdash M_1.M_2 : Cap[T]}$$

(Empty)
$$\frac{}{\Gamma \vdash \epsilon : Cap[T]}$$

(In/Out)
$$\frac{\Gamma \vdash M : Amb[T]}{\Gamma \vdash in/out\ M : Cap[S]}$$

(Open)
$$\frac{\Gamma \vdash M : Amb[T]}{\Gamma \vdash open\ M : Cap[T]}$$

(Prefix)
$$\frac{\Gamma \vdash M : Cap[T] \quad \Gamma \vdash P : T}{\Gamma \vdash M.P : T}$$

(Amb)
$$\frac{\Gamma \vdash M : Amb[T] \quad \Gamma \vdash P : T}{\Gamma \vdash M[P] : S}$$

(Res)
$$\frac{\Gamma, n : Amb[T] \vdash P : S}{\Gamma \vdash (\nu n : Amb[T])P : S}$$

(Zero)
$$\frac{}{\Gamma \vdash 0 : T}$$

(Par)
$$\frac{\Gamma \vdash P_1 : T \quad \Gamma \vdash P_2 : T}{\Gamma \vdash P_1 \mid P_2 : T}$$

(Repl)
$$\frac{\Gamma \vdash P : T}{\Gamma \vdash !P : T}$$

(Input)
$$\frac{\Gamma, x_1 : W_1, \ldots, x_k : W_k \vdash P : W_1 \times \ldots \times W_k}{\Gamma \vdash (x_1 : W_1, \ldots, x_k : W_k)P : W_1 \times \ldots \times W_k}$$

(Output)
$$\frac{\Gamma \vdash M_1 : W_1 \quad \ldots \quad \Gamma \vdash M_k : W_k}{\Gamma \vdash \langle M_1, \ldots, M_k \rangle : W_1 \times \ldots \times W_k}$$

Fig. A.4. Typing rules for Exchange Types