Encoding Distributed Process Calculi into LMNtal

Kazunori UEDA

Dept. of Computer Science and Engineering, Waseda University
ueda@ueda.info.waseda.ac.jp

October 2007

Abstract

Towards a unifying model of concurrency, we have designed and implemented LMNtal (pronounced "elemental"), a model and language based on hierarchical graph rewriting that uses logical variables to represent connectivity and membranes to represent hierarchy. Diverse computational models including the π-calculus and the λ-calculus have been encoded into LMNtal and tested on our LMNtal system. This paper describes the encoding of the ambient calculus with running examples. The technical crux is the distributed management of names in the presence of locality an mobility. We have designed and implemented a self-adjusting management technique of names in which the name management and mobility operations are performed concurrently.

1 Introduction

LMNtal [3][7] is a language model based on hierarchical graph rewriting that uses logical variables to represent connectivity and membranes to represent hierarchy. Its objectives are (i) to serve as a computational model that encompasses diverse formalisms related to multiset rewriting, concurrency, and mobility, and (ii) to provide a practical programming language based on hierarchical graph rewriting and its implementation[1]. LMNtal is an outcome of the attempt to unify constraint-based concurrency [6] and Constraint Handling Rules (CHR) [2], the two notable extensions to concurrent logic programming. The main feature of LMNtal is its ability to deal with both connectivity and hierarchy in a simple setting. Although born from a very different background, LMNtal shares many of its motivations with Bigraphical Reactive Systems [5]. Connectivity and hierarchy are the two major structuring mechanisms found in many fields ranging from society to biology, not to mention the world of computing.

In order to demonstrate the expressive power of LMNtal as a unifying model, we have encoded various computational models including the λ-calculus and the π-calculus into LMNtal [4]. One of the goals of LMNtal is to represent wide-area distributed computing. It is for this purpose that LMNtal allows both graph nodes and rewrite rules to be enclosed in a membrane to delimit the scope of the rewrite rules. In an attempt to relate existing models of distributed computing and LMNtal, this paper studies the encoding of the ambient calculus [1], one of the best-known distributed process calculi.

Distributed process calculi are process calculi with some (named or unnamed) notion of locations. Membranes, a construct often used to represent locations in distributed

[1] http://www.ueda.info.waseda.ac.jp/lmntal/

process calculi, represent locality by enclosing a multiset of atomic processes. Ambients are essentially membranes that may form hierarchies, and are in common with LMNtal membranes in that both are used to represent administrative domains. LMNtal and the ambient calculus exhibit similarities also in that processes in remote locations do not interact directly ignoring the membrane topology; they interact only by succession of proximity interaction. On the other hand, the ambient calculus allows the reconfiguration of the ambient structure, which accordingly causes the reconfiguration of the scope of names. The main motivation of the present work is to study how to encode the dynamic reconfiguration of the scope of names in a hierarchical setting.

2 The Ambient Calculus

The ambient calculus [1] is a model of concurrency in which ambients move around in a hierarchical ambient structure based on authentication. The full ambient calculus features both mobility and communication, but this paper is concerned with the mobility aspect, namely the *pure mobility calculus*.

Expressions of the ambient calculus are defined as follows, where the syntactic category n representing names is presupposed:

$$\begin{array}{rrcl} \text{(processes)} & P & ::= & (\nu n)P \mid \mathbf{0} \mid P \mid P \mid !P \mid n[P] \mid M.P \\ \text{(capabilities)} & M & ::= & \mathsf{in}\ n \mid \mathsf{out}\ n \mid \mathsf{open}\ n \end{array}$$

Here, $(\nu n)P$ represents hiding (or the creation of fresh local names), $\mathbf{0}$ represents an inert process, $P \mid P$ represents parallel composition, $!P$ represents repetition of P, $n[P]$ represents an ambient with the name n, and $M.P$ represents a process that performs M and then becomes P.

The operational semantics of the ambient calculus consists of a structural congruence and a reduction relation. We remind the readers of the reduction rules for $\mathsf{in}\ m$, $\mathsf{out}\ m$, $\mathsf{open}\ m$:

$$\begin{array}{rcl} n[\mathsf{in}\ m.P \mid Q] \mid m[R] & \rightarrow & m[n[P \mid Q] \mid R] \\ m[n[\mathsf{out}\ m.P \mid Q] \mid R] & \rightarrow & n[P \mid Q] \mid m[R] \\ \mathsf{open}\ m.P \mid m[Q] & \rightarrow & P \mid Q \end{array}$$

The rules of the structural congruence and the structural rules of the reduction relation are standard and omitted.

The in operation transforms a sibling relation between ambients into a parent-child relation. Conversely, the out operation transforms a parent-child relation into a sibling relation. The open operation removes an ambient's membrane and makes its contents belong to the parent ambient. All these capabilities are suspended if there is no ambient with the name m.

3 LMNtal

This section briefly describes the syntax and the semantics of LMNtal. For details omitted from here, the readers are referred to [3].

The syntax of LMNtal is shown in Fig. 1, where the two syntactic categories, X for *link names* and p for *atom names* are presupposed. We reserve the atom name "=" for connectors described below.

(process) P	$::=$	$\mathbf{0}$	(null)
	$\mid$	$p(X_1, \ldots, X_m)$ $\quad (m \geq 0)$	(atom)
	$\mid$	P, P	(molecule)
	$\mid$	$\{P\}$	(cell)
	$\mid$	$T :- T$	(rule)
(process template) T	$::=$	$\mathbf{0}$	(null)
	$\mid$	$p(X_1, \ldots, X_m)$ $\quad (m \geq 0)$	(atom)
	$\mid$	T, T	(molecule)
	$\mid$	$\{T\}$	(cell)
	$\mid$	$T :- T$	(rule)
	$\mid$	$@p$	(rule context)
	$\mid$	$\$p[X_1, \ldots, X_m \mid A]$	(process context)
	$\mid$	$p(*X_1, \ldots, *X_m)$ $\quad (m > 0)$	(aggregate)
(residual) A	$::=$	$[]$	(empty)
	$\mid$	$*X$	(bundle)

Figure 1: Syntax of LMNtal

Each link name occurring in a process P can occur at most twice (the *Link Condition*). A link name occurring exactly once in P represents a *free link* of P, while a link name occurring exactly twice in P is considered bound and represents a *local link* of P.

Intuitively, $\mathbf{0}$ is an empty process, $p(X_1, \ldots, X_m)$ $(m \geq 0)$ is an m-ary *atom*, P, P is parallel composition, $\{P\}$ is a *cell* formed by wrapping a process P with a *membrane* `{ }`, and T `:-` T is a rewrite rule of processes. An atom $X = Y$, called a *connector*, interconnects the link X and the link Y.

Process templates on the both sides of rewrite rules allow additional constructs as explained below. Rule contexts and process contexts represent "the rest of the processes" inside a membrane. A *rule context* $@p$ matches a possibly empty ruleset (multiset of rules) inside a membrane, while a *process context* $\$p[X_1, \ldots, X_m \mid A]$ $(m \geq 0)$ matches a process (not containing rules) inside a membrane. The argument of a process context specifies what links may or must occur free. When the *residual* A is $[]$, the argument is abbreviated to $[X_1, \ldots, X_m]$ and means that the set of free links of $\$p$ must be exactly $\{X_1, \ldots, X_m\}$. When A is of the form $*X$ (called a *bundle*), it represents zero or more free links of the context that may occur in addition to the "must-occur" links $X_1, \ldots, X_m$. The final construct, $p(*X_1, \ldots, *X_n)$ $(n > 0)$, stands for an aggregate of n-ary atoms with the same name; see [3] for technical details. Rewrite rules must observe several additional syntactic conditions in order (i) to ensure the well-formedness of processes obtained by expanding process/rule contexts and aggregates, (ii) to ensure that the processes represented by process/rule contexts and aggregates can be uniquely determined, and (iii) to ensure that the application of rewrite rules preserves the well-formedness of processes.

The operational semantics of LMNtal consists of the structural congruence defined by (E1)–(E10) (Fig. 2) and the reduction relation defined by (R1)–(R6) (Fig. 3). (E1)–

(E1)	$\mathbf{0}\,,\,P$	$\equiv$	P	
(E2)	$P\,,\,Q$	$\equiv$	$Q\,,\,P$	
(E3)	$P\,,\,(Q\,,\,R)$	$\equiv$	$(P\,,\,Q)\,,\,R$	
(E4)	P	$\equiv$	$P[Y/X]$	if X occurs bound in P
(E5)	$P \equiv P'$	$\Rightarrow$	$P\,,\,Q \equiv P'\,,\,Q$	
(E6)	$P \equiv P'$	$\Rightarrow$	$\{P\} \equiv \{P'\}$	
(E7)	$X = X$	$\equiv$	$\mathbf{0}$	
(E8)	$X = Y$	$\equiv$	$Y = X$	
(E9)	$X = Y\,,\,P$	$\equiv$	$P[Y/X]$	if P is an atom and X occurs free in P
(E10)	$\{X = Y\,,\,P\}$	$\equiv$	$X = Y\,,\,\{P\}$	if exactly one of X and Y occurs free in P

Figure 2: Structural congruence on LMNtal processes

$$\text{(R1)}\ \frac{P \longrightarrow P'}{P\,,\,Q \longrightarrow P'\,,\,Q} \qquad \text{(R2)}\ \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}}$$

$$\text{(R3)}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)}\quad \{X = Y\,,\,P\} \longrightarrow X = Y\,,\,\{P\} \quad \text{if } X \text{ and } Y \text{ occur free in } \{X = Y\,,\,P\}$$

$$\text{(R5)}\quad X = Y\,,\,\{P\} \longrightarrow \{X = Y\,,\,P\} \quad \text{if } X \text{ and } Y \text{ occur free in } P$$

$$\text{(R6)}\quad T\theta\,,\,(T :\!\!-\, U) \longrightarrow U\theta\,,\,(T :\!\!-\, U)$$

Figure 3: Reduction relation on LMNtal processes

(E3) are the characterization of processes as multisets. (E4) stands for α-conversion of local link names. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7) says that a self-absorbed loop is equivalent to $\mathbf{0}$, while (E8) expresses the symmetry of connectors. (E9) and (E10) stand for the absorption/emission rules of connectors for atoms and cells, respectively.

Computation proceeds by rewriting processes using rules collocated in the same "place" of the nested membrane structure. (R1)–(R3) are standard structural rules, and (R4)–(R5) are the mobility rules for connectors. The central rule of LMNtal is (R6). The substitution θ is to map process contexts, rule contexts, and aggregates into specific processes, rules, and atoms, respectively. The major challenge in the design of the operational semantics has been the proper treatment of interplay between graph structures formed by links and hierarchical structures formed by membranes that may be crossed by links.

4 Issues in Encoding the Ambient Calculus

The most prominent similarity between the ambient calculus and LMNtal is that both feature membranes that can be nested. This suggests that any natural encoding from the ambient calculus to LMNtal should map ambients into LMNtal cells, and we regard

this as the boundary condition in designing our encoding. Henceforth cells representing ambients are referred to as *ambient cells*.

The major issue should now be how to encode names of the ambient calculus.

4.1 Representing Names

As in the π-calculus, names play a crucial role in the ambient calculus. The basic operations on names are

(a) to create a fresh local name,

(b) to pass it to anther process using communication primitives,

(c) to name ambients, and

(d) to form capabilities.

The last two are closely related; local ambient names are used as secret keys for entering, exiting from, and removing ambients.

Two possible ways of representing names of the ambient calculus in LMNtal are (i) to map them into LMNtal atom names and (ii) to map them into graph structures formed by cells and links. We take the latter approach and map

- names into cells (which we call *name cells*) that represents identity and
- name *occurrences* into links entering the name cells.

The use of graphs rather than atom names is motivated by the following:

1. to demonstrate the expressive power of graph rewriting in LMNtal,
2. to express the topology of name references and its changes explicitly,
3. to be able to express local names introduced by ν, and
4. to limit the use of LMNtal atom names to the encoding of the fixed set of primitives of the ambient calculus.

Compared with ambient cells that form administrative domains and contain rewrite rules, name cells are lightweight; they simply hold name references together with auxiliary information for name management. In other words, cells are used as records that allow duplication of field names.

We have already encoded the π-calculus under the same policy [4], but it turns out that the encoding of the ambient calculus is significantly more complicated because of the membranes representing ambient boundaries.

Names in the ambient calculus are referenced from various "places" of an ambient hierarchy. Accordingly, the identity of names must be represented and managed globally as a whole, and at the same time the identity must be checkable locally inside each ambient in order to allow local computation to proceed. Global, monolithic management of names is inadequate for handling the both requirements; we need to let each ambient hold one *proxy* for each name referenced from inside the ambient. This results in representing a name (in the ambient calculus) in terms of a tree structure comprising *root cells* and *proxy cells*. We call this tree structure a *name tree*. There must be exactly one

name tree for each global name. Henceforth both root cells and proxy cells are called *name cells* generically.

A root cell and a proxy cell have the form

$$\begin{aligned} \texttt{\{id, name(}n\texttt{), +}L_1\texttt{, ..., +}L_m\texttt{\}} &\quad \text{(root of a global name)} \\ \texttt{\{id, +}L_1\texttt{, ..., +}L_m\texttt{\}} &\quad \text{(root of a local name)} \\ \texttt{\{id, -}L_0\texttt{, +}L_1\texttt{, ..., +}L_m\texttt{\}} &\quad \text{(proxy)} \end{aligned}$$

respectively, where $m > 0$, L_0 is connected to its parent name cell, and L_k $(1 \leq k \leq m)$ is connected either to an occurrence of the name inside the ambient or to a proxy cell held by a child ambient. The "+" and "−" signs are unary atom names written as prefix operators. `name(`n`)` is an abbreviation of `name(L),` n`(L)`.

Next, we define the *normal form* of a name tree. Intuitively, a name tree in a normal form should accord with the underlying ambient hierarchy. Let us make this more precise. First, note that an ambient hierarchy forms a tree structure that will be referred to as an *ambient tree*. A name tree in a normal form must have a root cell at the uppermost node of some connected subgraph of the ambient tree[2], and a proxy cell at each of other nodes of the above connected subgraph.

From this condition and the condition $m > 0$, it follows that

1. an ambient cell containing a leaf node of a name tree must have a reference to that name, and that
2. a link interconnecting two (root or proxy) nodes of a name tree crosses an ambient boundary exactly once.

Since the above condition does not mention where to place the root of a name tree, we add two more conditions:

- The root of the name tree of a *global* name is placed at the *top level* of the ambient hierarchy.
- The root of the name tree of a *local* name is placed at the *innermost* ambient containing all the ambients referring to that local name (minimality).

Figure 4 shows an ambient structure that has five normal-form name trees representing two global names (`a` and `b`) and three (anonymous) local names. Ellipses stand for ambients, squares stand for root cells, and filled circles stand for proxy cells.

4.2 The Encoding Rules

We define the encoding from the ambient calculus into LMNtal by means of the function $[\![\cdot]\!]$ defined as follows. The encoding of the repetition $!P$ will be discussed in the next section.

$$\begin{aligned} [\![\mathbf{0}]\!] &\stackrel{\text{def}}{=} \mathbf{0} \\ [\![P \mid Q]\!] &\stackrel{\text{def}}{=} ([\![P]\!], [\![Q]\!])\downarrow \end{aligned}$$

[2] A connected subgraph S of a tree T is not necessarily a subtree of T; indeed, S is obtained from some subtree S_0 of T by deleting zero or more subtrees of S_0.

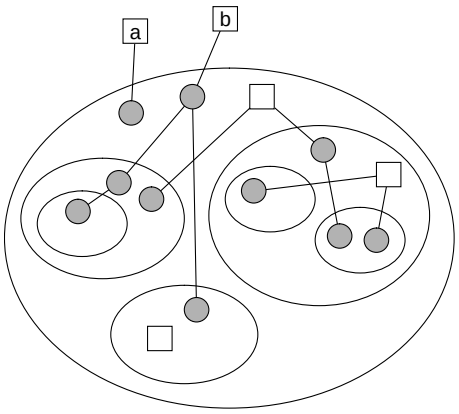

Figure 4: An ambient structure with name trees

$$
\begin{array}{rcl}
[\![(\nu n)P]\!] & \stackrel{\text{def}}{=} & (\text{hide}_n([\![P]\!]\downarrow))\downarrow \\
[\![n[P]]\!] & \stackrel{\text{def}}{=} & \texttt{\{}@amb\texttt{, amb(}L\texttt{), }[\![n]\!]\texttt{(L), }[\![P]\!]\texttt{\}}\downarrow \\
[\![M.P]\!] & \stackrel{\text{def}}{=} & ([\![M]\!]([\![P]\!]))\downarrow \\
[\![op\ n]\!] & \stackrel{\text{def}}{=} & [\![op]\!]([\![n]\!]) \quad (op \in \{\mathsf{in}, \mathsf{out}, \mathsf{open}\}) \\
[\![op]\!] & \stackrel{\text{def}}{=} & \lambda f\,.\,\lambda p\,.\,\texttt{(}op\texttt{(L,M), \{+}M\texttt{, }p\texttt{\}, }f\texttt{(L))} \quad (op \in \{\mathsf{in}, \mathsf{out}, \mathsf{open}\}) \\
[\![n]\!] & \stackrel{\text{def}}{=} & \lambda l\,.\,\texttt{\{id, name(}n\texttt{), +}l\texttt{\}}
\end{array}
$$

Here, $\downarrow$ is a normalization function that transforms name trees in an encoded graph structure into their normal forms by preserving their connectivity as graphs and adjusting the placement and the numbers of root cells and proxy cells. The details will be shown as the LMNtal code in the next section. It should be legitimate to regard the hiding construct (νn) as *syntax* rather than *operations*, and the following auxiliary function, hide_n, is used to remove global name information:

$$\text{hide}_n(\texttt{(\{id, name(}n\texttt{), }P\texttt{\}, }Q\texttt{)}) \quad = \quad \texttt{(\{id, }P\texttt{\}, }Q\texttt{\})}$$

The notation *@amb* in the definition of $[\![= ambnP]\!]$ stands for the encoded ruleset shown in the next section. For computation inside each ambient to proceed, the encoded ruleset must take effect within each ambient cell. Since LMNtal's reduction rules act only on the process at the same place in the membrane hierarchy, rulesets for the ambient calculus must be brought into each ambient membrane.

5 Representing the Encoding in LMNtal

Following the encoding policy described in the previous section, we have built a running LMNtal program realizing the ambient calculus (Fig. 5). Some remarks on the syntactic convention of LMNtal are appropriate here to read the LMNtal code:

- Rule are prefixed by rule names using the extended syntax of LMNtal [7].
- A nullary process context `$p` is an abbreviation of `$p[|*X]`; it is a process context with no constraints on the occurrences of free links.

- A newly created ambient must be given an atom `amb.use` inside it. The LMNtal system comes with a module system, where the notation m`.use` is used as the standard idiom for expanding the module's ruleset into that place by mentioning the module name m. Since the atom `amb.use` itself is unnecessary for the computation, it is removed by the rule `gc_amb` in the expanded ruleset.

Note that the atom `amb.use` does not occur in the rules `in`, `out`, `open` of Fig. 5. It is unnecessary because the ruleset for local computation is received by the rule contexts (`@p`, `@q`, `@r`) on the left-hand side and is passed to the right-hand side.

As one can see, each of `in`, `out`, `open` is taken care of by a single LMNtal rule. However, these rules alone are not sufficient because they migrate name cells together with ambient cells and do not necessarily preserve the normal form condition of name trees. If name trees are not in a normal form, subsequent reductions may not be able to recognize the identity of names (though it never happens that occurrences of different names are wrongly recognized as occurrences of the same name). One may wonder if we can reinforce the first three rules of Fig. 5 to ensure the normal form property, but this seems difficult because the migration of an ambient moves *all* free names of the ambient that may not be specified explicitly in the rewrite rules.

5.1 Name Management

The above consideration motivated us to give a set of rules for name management independently of the rules for ambient mobility.

Rules whose names start with `proxy_` are to reestablish the normal-form conditions of a name tree structure when the set of names referred to in each ambient is changed by mobility primitives. `proxy_enter` enables two references to the same name within an ambient to be recognized within that ambient. `proxy_resolve` merges two serial proxies for the same name in the same ambient. `proxy_insert_middle` is activated when `in` moves two directly connected name cells (root or proxy) to remote places not in a parent-child relation, and inserts a new proxy in between. `proxy_insert_outer` is activated when `out` exports some child proxy out of an ambient and inverts the parent-child relationship between proxies, and creates a new proxy at the parent level. `proxy_merge_outer` is activated when the second parent of a proxy is created (by `proxy_insert_outer`) and merges the two parents.

`local_name_in` and `global_name_out` are to normalize the location of root cells; `local_name_in` moves a root cell into the innermost possible level, while `global_name_out` moves a root cell towards the top level of the ambient hierarchy. `root_merge` merges two root cells with the same global name.

Rules whose names start with `gc_` are used to garbage-collect unused names. `gc_local_name` and `gc_global_name` remove unreferenced root cells, while `gc_proxy` removes unreferenced proxy cells.

Rules in Fig. 5 preserve the following invariant properties about the name trees:

1. Every occurrence of a name in the encoded process is linked to some name cell (root or proxy) of the name.
2. If we regard the root cells of the same global names as interconnected, then names are in one-to-one correspondence with connected components of name cells.
3. A link interconnecting two name cells is terminated with a "`-`" atom on one end and with a "`+`" atom on the other end.

```
{ module(amb).

/* n[in m.P | Q] | m[R] --> m[n[P|Q] | R] */
in@@
{amb(N0), {id,+N0,$n}, {id,+M0,-M1,$m0}, in(M0,{$p}), $q,@q},
{amb(M2), {id,+M2,-M3,$m1}, $r,@r},
{id,+M1,+M3,$m2} :-
   {amb(M4), {id,+M4,+M5,-M,$m1},
      {amb(N2), {id,+N2,$n}, {id,-M5,$m0}, $p,$q,@q},
   $r,@r},
   {id,+M,$m2}.

/* m[n[out m.P | Q] | R] --> n[P|Q] | m[R] */
out@@
{amb(M0), {id,+M0,+M2,$m1}, {id,+N1,$n2},
  {amb(N0), {id,+M1,-M2,$m0}, {id,+N0,-N1,$n}, out(M1,{$p}), $q,@q},
  $r,@r} :-
    {amb(N2), {id,-M3,$m0}, {id,+N2,-N3,$n}, $p,$q,@q},
    {amb(M4), {id,+M3,+M4,$m1}, {id,+N3,$n2}, $r,@r}.

/* open m.P | m[Q] --> P|Q */
open@@
open(M,{$p}), {amb(M1), {id,+M1,-M2,$mm}, $q,@q}, {id,+M,+M2,$m} :-
   $p, $q, {id,$m,$mm}.

proxy_enter@@
{$p[M0,M1|*P],@p}, {id,+M0,+M1,$m} :-
   {$p[M0,M1|*P],@p, {id,+M0,+M1,-M}}, {id,+M,$m}.

proxy_resolve@@
{id,-M,$m0}, {id,+M,$m1} :- {id,$m0,$m1}.

proxy_insert_middle@@
{{{id,-M,$m},$p,@p},$q,@q} :- {{id,+M0,-M}, {{id,-M0,$m},$p,@p},$q,@q}.

proxy_insert_outer@@
{{id,+M0,$m0},$p,@p} :- {{id,-M,$m0},$p,@p}, {id,+M0,+M}.

proxy_merge_outer@@
{id,+M0,$m0}, {id,+M1,$m1}, {{id,-M0,-M1,$m2},$p,@p} :-
     {id,+M,$m0,$m1}, {{id,-M,$m2},$p,@p}.

local_name_in@@
{$p[M|*P],@p}, {id,+M} :- {{id,+M}, $p[M|*P],@p}.

global_name_out@@
{{id,name($n),+M0},{$p[M0|*M],@p},$q,@q} :- unary($n) |
    {{id,+M0,-M},{$p[M0|*M],@p},$q,@q}, {id,name($n),+M}.

root_merge@@
{id,name($n0),$m0}, {id,name($n1), $m1} :- unary($n0), unary($n1), $n0=$n1 |
  {id,name($n0),$m0,$m1}.

gc_local_name@@ {id} :- .
gc_global_name@@ {id,name($n)} :- unary($n) | .
gc_proxy@@ {id,+X,$m}, {{id,-X}, $p,@p} :- {id,$m}, {$p,@p}.
gc_amb@@ amb.use :- .

}.
```

Figure 5: LMNtal code of the ambient calculus

4. A link interconnecting a name cell and a name occurrence does not cross ambient membranes.

The normalization rules turn name trees into their normal forms preserving the above invariants. It is easy to see that a name tree is in a normal form if no normalization rule

```
/* !(open m.P) | m[Q] --> P | Q | !(open m.P) */
open_repl@@  /* special case of !open */
open_repl(M,{$p}), {amb(M1), {id,+M1,-M2,$mm}, $q,@q}, {id,+M,+M2,$m} :-
   nlmem.copy({$p},cp,Copies), copies(Copies,P),
   $q, {id,+M3,$m,$mm}, open_repl(M3,P).
open_repl_aux@@
copies(cp(C1,C2),P), {+C1,$p1} :- $p1, P=C2.
```

Figure 6: Encoding of repetition

is applicable and that a normal form is uniquely determined.

It should be noted that name normalization and ambient operations may run concurrently. This means that in, out and open rules may be applied even when name trees are not in their normal forms. However, the three rules also preserve the above-mentioned invariants, and we can allow name tree normalization to proceed asynchronously with ambient operations.

5.2 Encoding Repetition

Like many other models of concurrency, the ambient calculus features the repetition construct $!P$. The semantics of $!P$ is given by the relation $!P \equiv P \mid !P$, and its purpose is to spawn an instance of P on demand rather than to create infinitely many P's. Indeed, one will notice that the uses of ! are rather limited in each model of concurrency; in the case of the ambient calculus it is almost always used in the form !(open $n.P$). This is regarded as the encoding of procedure calls, and the creation of an ambient n triggers the execution of the procedure body P. Readers may recall that ! in the π-calculus is mostly used for the encoding of procedures, too. So it makes sense not to allow ! in its general form but give an encoding of the specialized form !(open $n.P$) instead (Fig. 6).

One issue that arises in the encoding of !(open $n.P$) is that the duplication of P creates new references to the free names of P. Duplication of $[\![P]\!]$ with free names can be expressed using aggregates. Aggregates are the only construct not yet supported in our current implementation, but the LMNtal system instead supports an `nlmem` (nonlinear membrane) API which does the necessary job for our purpose. `nlmem.copy({`P`},`a`,`R`)`, which is an abbreviated form of `(nlmem.copy(`R_0`,`a`,`R`),{+`R_0`,`P`})`, creates two copies of the cell `{+`R_0`,`P`}` with all its free links renamed, and connects R and the two fresh copies of R_0 using a ternary atom with the name a. Furthermore, for each free link L except R_0 of the original cell `{+`R_0`,`P`}`, `nlmem.copy` connects the two fresh copies of L and the original L via the ternary atom a. The semantics of `nlmem.copy` can be given by the following rule scheme:

```
nlmem.copy({$p[|*X]},a,R) :-
    {+R1, $p[|*Y]}, {+R2, $p[|*Z]},
    a(*Y,*Z,*X), a(R1,R2,R).
```

6 Examples

We have encoded most of the examples in [1] into LMNtal and run them successfully on our LMNtal system. Let us give two examples.

```
// Firewall Access
//   Firewall =def (new w) w[k[out w.in kk.in w] | open kk.open kkk.P]
//   Agent    =def kk[open k.kkk[Q]]

{id,name(k),+K9,+K3}, {id,name(kk),+L3,+L9}, {id,name(kkk),+M9,+M3}, {id,+W9},
{amb.use. amb(W0),
  {id,+W0,+W8,-W9}, {id,+K8,-K9}, {id,+L1,+L8,-L9}, {id,+M0,-M9},
  {a.use. amb(K0), {id,+K0,-K8}, {id,+W1,+W2,-W8}, {id,+L0,-L8},
     out(W1,{in(L0,{in(W2,{})})})},
  open(L1,{open(M0,{pp})})
},
{amb.use. amb(L2), {id,+L2,-L3}, {id,+K2,-K3}, {id,+M2,-M3},
  open(K2,{{amb(M1), {id,+M1,-M2}, qq}})
}.
```

Figure 7: Example: Firewall Access

The first example is the encoding of firewall access. Figure 7 is the result of expanding the parallel composition of *Firewall* and *Agent* defined as follows:

$$\begin{aligned} \mathit{Firewall} &\stackrel{\text{def}}{=} (\nu w) w[k[\mathsf{out}\ w\,.\,\mathsf{in}\ kk\,.\,\mathsf{in}\ w] \mid \mathsf{open}\ kk\,.\,\mathsf{open}\ kkk\,.\,P] \\ \mathit{Agent} &\stackrel{\text{def}}{=} kk[\mathsf{open}\ k\,.\,kkk[Q]] \end{aligned}$$

The above definition expresses a protocol with which an *Agent* holding keys k, kk, kkk can enter a *Firewall* ambient with a private ambient name w. The basic idea here is to let an ambient k inside w go outside and bring the *Agent* back. In Fig. 7, `pp` and `qq` represent the processes P and Q to be executed after the *Agent*'s entrance. They can be regarded as free names representing processes and could be replaced by encoded processes by adding rewrite rules for them.

The result of execution (under the `--hiderule` option) is:

```
{pp, qq, amb(L749), {id, +L749}}.
```

This represents $(\nu w) w[P \mid Q]$, which means P was allowed to enter an ambient with a private name.

Second, we show the encoding of objective moves in terms of the ambient calculus's subjective moves (Fig. 8). Objective moves allow processes not protected by ambient membranes to enter or exit from ambients. This is very different from the migration of computation protected by ambient membranes in that the permissions of the target ambients are crucial for security.

$$\begin{aligned} \mathsf{allow}\ n &\stackrel{\text{def}}{=} !(\mathsf{open}\ n) \\ \mathsf{mv\ in}\ n.P &\stackrel{\text{def}}{=} (\nu k) k[\mathsf{in}\ n.\mathit{enter}[\mathsf{out}\ k.\mathsf{open}\ k.P]] \\ \mathsf{mv\ out}\ n.P &\stackrel{\text{def}}{=} (\nu k) k[\mathsf{out}\ n.\mathit{exit}[\mathsf{out}\ k.\mathsf{open}\ k.P]] \\ n^{\Updownarrow}[P] &\stackrel{\text{def}}{=} n[P \mid \mathsf{allow}\ \mathit{enter}] \mid \mathsf{allow}\ \mathit{exit} \end{aligned}$$

mv in and mv out are the objective versions of in and out, respectively. $n^{\Updownarrow}[P]$ is an ambient that allows itself to be the target of mv in and mv out.

The program in Fig. 8 expands the module `b` containing the four auxiliary definitions, and executes $\mathsf{mv\ in}\ n\,.\,P \mid n^{\Updownarrow}[Q]$. The result obtained is:

```
// Objective Moves
//   allow n    =def !open n
//   mv in n.P  =def (new k) k[in n.enter[out k.open k.P]]
//   mv out n.P =def (new k) k[out n.exit[out k.open k.P]]
//   n_dnup[P]  =def n[P | allow enter] | allow exit

{ module(b).
  allow(N) :- open_repl(N,{}).
  mv_in(N,{$p}) :- {id,+K}, {id,name(enter),+E},
    {amb.use. amb(K0), {id,+K0,+K9,-K}, {id,+N0,-N}, {id,+E1,-E},
      in(N0,
         {{amb(E0), {id,+E0,-E1}, {id,+K1,+K2,-K9}, out(K1,{open(K2,{$p})})}})}.
  mv_out(N,{$p}) :- {id,+K}, {id,name(exit),+E},
    {amb.use. amb(K0), {id,+K0,+K9,-K}, {id,+N0,-N}, {id,+E1,-E},
      out(N0,
          {{amb(E0), {id,+E0,-E1}, {id,+K1,+K2,-K9}, out(K1,{open(K2,{$p})})}})}.
  n_dnup(N,{$p}) :- {id,name(enter),+E}, {id,name(exit),+Ex0},
    {amb.use. b.use. amb(N0), {id,+N0,-N}, {id,+E0,-E}, $p, allow(E0)},
    allow(Ex0).

  b.use :- .
  {module(b), @b} :- .
}.

b.use.
{id,name(n),+N0,+N1}, mv_in(N0,{pp}), n_dnup(N1,{qq}).
```

Figure 8: Example: Objective Moves

```
open_repl(L270,L535),
{pp, qq, amb(L324), open_repl(L591,L601),
  {+L601},
  {id, -L557, +L591},
  {id, -L338, +L324}},
{id, n(name), +L338},
{id, exit(name), +L270},
{id, enter(name), +L557},
{+L535},
```

which represents $n[P \mid Q \mid \mathsf{allow}\ \mathit{enter}] \mid \mathsf{allow}\ \mathit{exit}$, that is, $n^{\updownarrow}[P \mid Q]$.

7 Discussions and Conclusion

Computation in the ambient calculus changes the hierarchy of name references with the reconfiguration of ambient hierarchy. Ambient name occurrences should be considered as virtually interconnected *resources* because they represent access rights to the administrative domains defined by the ambients, but previous formulations handled the migration of these resources implicitly with ambient migration.

The main contribution of this paper is that our encoding in LMNtal makes this important operation explicit by representing the topology of name references by means of name trees and its reconfiguration algorithm by means of a set of rewrite rules that works autonomously and asynchronously.

Our encoding consists of fifteen rules, of which three are the direct translation of

the three primitives of the ambient calculus, eight for name management, and the rest for garbage collection. An important feature of LMNtal is that it allows diagrammatic interpretation of computation, and each encoded rule can be understood graphically.

The encoding of names into graphs (name trees) was useful for manifesting and understanding the behavior of names in the ambient calculus. Name management using proxy cells reflects the implementation of real distributed systems. Proxy cells were not necessary for the encoding of the π-calculus. This suggests that the the role of names in the ambient calculus is significantly more complex than that of the π-calculus.

Concurrency can be viewed as multiset rewriting of processes. LMNtal, which is a multiset rewriting language augmented with links and membranes, allows concise encoding of the operational semantics of various models of concurrency. The ambient calculus has several variants including boxed ambients, safe ambients, and bioambients. By changing some of the rules in Fig. 5, one should be able to readily obtain implementations of these variants. We plan to encode more computational models to demonstrate the value of LMNtal as a unifying model.

Acknowledgments. The author would like to thank the current and former members of the LMNtal development team who jointly built our publicly available LMNtal implementation. This work is partially supported by Grant-In-Aid for Scientific Research ((B)(2) 16300009; Priority Areas (C)(2)13324050, (B)(2)14085205 and 04560009), MEXT and JSPS.

References

[1] Cardelli, L. and Gordon, A. D., Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer-Verlag, 1998, 140–155.

[2] Frühwirth, T., Theory and Practice of Constraint Handling Rules, *J. Logic Programming*, **37** (1998), 95–138.

[3] Ueda, K. and Kato, N., LMNtal: a Language Model with Links and Membranes, in *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer-Verlag, 2005, 110–125.

[4] Inui, A., Kudo, S., Hara, K., Mizuno, K. and Ueda, K., A Unifying Programming Language LMNtal Based on Hierarchical Graph Rewriting, to appear in *Computer Software*, JSSST, 2008.

[5] Milner, R., Bigraphical Reactive Systems, in *Proc. 12th Int. Conf. on Concurrency Theory (CONCUR 2001)*, LNCS 2154, Springer-Verlag, 2001, 16–35.

[6] Ueda, K., Resource-Passing Concurrent Programming, in *Proc. 4th Int. Symp. on Theoretical Aspects of Computer Software (TACS 2001)*, LNCS 2215, Springer-Verlag, 2001, 95–126.

[7] Ueda, K., Kato, N, Hara, K. and Mizuno, K., LMNtal as a Unifying Declarative Language, in *Proc. Third workshop on Constraint Handling Rules (CHR 2006)*, 1–15.