LABORATOIRE D'INFORMATIQUE, DE ROBOTIQUE
ET DE MICROÉLECTRONIQUE DE MONTPELLIER

Unité Mixte CNRS - Université Montpellier II C 09928

RAPPORT DE RECHERCHE

Operational Semantics of Multi-Agent Organizations

Jacques FERBER — `ferber@lirmm.fr`
Olivier GUTKNECHT — `gutkneco@lirmm.fr`

01/99
R.R.LIRMM 99003

161, Rue Ada - 34392 Montpellier Cedex 5 - France
Tél: (33) 67 41 85 85 - Fax: (33) 67 41 85 00

Abstract

This report introduces a formal description of the operational semantics of multiagent organizations expressed in the AALAADIN generic model. This formalization is based on the π-calculus and the Chemical Abstract Machine (CHAM).

By mapping an agent to a set of π-calculus processes and groups to CHAM solutions, we show that it is possible to associate a precise semantics for the definition and dynamics of agents, groups and roles, independently of any implementation.

Our show that formalization verifies the properties of AALAADIN: agents act in several groups simultaneously, communications are described through abstract roles interaction, and organization management is performed by agents.

Contents

1 Introduction **2**

2 Pi-calculus and the CHAM **2**
- 2.1 The Pi-Calculus 2
- 2.2 The Chemical Abstract Machine 4

3 How to represent multi-agent systems **4**
- 3.1 Basic representations and definitions 4
- 3.2 Example 5

4 The Aalaadin model of organizations **5**
- 4.1 Agent 6
- 4.2 Group 6
- 4.3 Role 6

5 An operational semantics of Aalaadin **6**
- 5.1 Multi-Agent Abstract Machine 6
- 5.2 Basic calculus 7
- 5.3 Agent representation 7
- 5.4 Group creation 8
- 5.5 Agent admission 9
- 5.6 Example 9

6 Conclusion **10**

A The Picol language **10**
- A.1 Syntax of Picol 10
- A.2 Semantics of Picol 11
- A.3 Primitives structures 12

1 Introduction

In [8], an organizational model of MAS based on groups, roles and agents has been proposed.

A general description of its concepts and its use has been described, but there was no clear description of how such a model could effectively work. For instance, crucial point such as what is actually performed when an agent wants to join a group or how a group is created has been left out of the presentation.

We propose in this report to bridge this gap and to show that it is possible to formally describe the operational semantics of organizations without resorting to any implementation details. Our formalization is based on the π-calculus and the CHAM which have been used extensively to capture the essence of concurrent programming and distributed computations.

Thus, this report is the first, to our knowledge, to give a formal account of what happens in multiagent organizations, i.e. to give an operational semantics of organized multiagent systems.

Section 2 gives a brief description of the main features of the π-calculus and the CHAM. Section 3 explains how "classical" multiagent systems (i.e. without groups and roles) may be represented into those formalisms. Section 4 gives an abstract of the main characteristics of the AALAADIN model of multiagent organizations. Section 6 presents the operational semantics of the AALAADIN.

2 Pi-calculus and the CHAM

2.1 The Pi-Calculus

The π-calculus is a process calculus whose aim is to provide a sound foundation to concurrent computations and be the "λ-calculus" of parallel processes. Technically, the π-calculus is an extension of CCS that deals with name passing: processes pass names to other processes through named channels. Milner has shown that the π-calculus is very expressive, because one can encode the λ-calculus into the π-calculus [10].

In the π-calculus, every expression denotes a process, a free-running computational entity which run in parallel with other processes, and may possibly contains several subprocesses. Communications between processes are performed by exchanging a message on a named channel. The following rules gives the syntax of π-calculus expressions. P and Q are metavariables over processes and x, y, z, v and u are metavariables over names.

Remark 2.1 *In this report we will use the syntax of the* PICOL *language which is a syntactic variant and extension of the bare π-calculus. See Appendix A to get a description of its syntax and semantics.*

$$
\begin{array}{lcl}
\mathtt{P,\ Q} & ::= & \mathbf{0} \\
 & \mid & P \mid Q \\
 & \mid & P \ \mathtt{;} \ Q \\
 & \mid & \mathtt{x(z).P} \\
 & \mid & \uparrow\mathtt{x(y).P} \\
 & \mid & C\mathtt{?}P \\
 & \mid & \nu\mathtt{(v)P} \\
 & \mid & \mathtt{A}(\mathtt{y}_1, .., \mathtt{y}_\mathtt{n})
\end{array}
$$

1. **0** is the null process, i.e. the process which does nothing. It is often omitted. For instance,
   `x(y).↑y(a).0` is generally written `x(y).↑y(a)`
2. $P \mid Q$ is the parallel composition of two concurrent processes P and Q acting in parallel.
3. $P; Q$ means that the process behave either as P or as Q. Processes P and Q are generally guarded by a condition (see Appendice).
4. $x(z).P$ is the input prefix. x can be thought as an input channel and z as the parameter which holds the message sent to x. When the process has received the message z, the prefix is suppressed and the process acts as $P[y/z]$.
5. $\uparrow\mathtt{x}(y).P$ is the output prefix: x is the output port where the name y is sent. Once the name sent, $\uparrow\mathtt{x}(y).P$ reduces to P.
6. $C?P$ is a guarded process. if the condition C holds, then $C?P$ is reduced to P otherwise it is reduced to **0**. The condition is supposed to be a primitive. In the bare π-calculus, the condition is assumed to be an equality over names, for instance $(x = y)$. We will generalize here the condition to hold over any primitive operation (see Appendice A for details).
7. $\nu(x).P$ defines a new unique name x distinct from all names in P
8. $A(y_1, .., y_n)$ is a defining equation

Some expressions are "equivalent" in the sense that they may be substituted without changing anything in the computation. This equivalence is given through a structural congruence relation $\equiv$, which formalizes the intuition that the order of subexpressions does not matter with the operator '—' which is both commutative and associative.

$$\begin{aligned} P \mid Q &\equiv Q \mid P \\ (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \end{aligned}$$

The operational semantics of the π-calculus is given through a set of reduction rules as in the λ-calculus. The main rule describes how processes can communicate by sending messages through channels:

$$\uparrow\mathtt{x}(y).P \mid x(z).Q \rightarrow P \mid [y/z]Q$$

Other rules describes the parallelization of processes, i.e. the fact that reductions are not modified when processes are placed in parallel [11].

$$\begin{aligned} P \mid R &\rightarrow Q \mid R \text{ if } P \rightarrow Q \\ \nu(x)P &\rightarrow \nu(x)Q \text{ if } P \rightarrow Q \\ P &\rightarrow Q \text{ if } P \equiv P' \text{ and } Q \equiv Q' \text{ and } P' \rightarrow Q' \end{aligned}$$

We will use throughout the continuation passing style of programming, a style which is very well known in actor languages [2] and also advocated to describe multi-agent oriented computations [7]. We will assume that the local continuation is passed as the last argument of messages. For instance, the recursive factorial function using continuation passing style can be described as follows:

```
Fact(n,c) =def (n =  0)?(↑c(1));
               (n ≠  0)?(ν(c1)(Fact(n-1,c1)|
                                c1(r).↑c(n*r)))
```

2.2 The Chemical Abstract Machine

The Chemical Abstract Machine or CHAM for short, has been introduced by G. Berry and G. Boudol [4]. Since then, Boudol has shown that many theoretical constructs such as the π-calculus can easily be described in the CHAM [6].

The CHAM is an abstract machine based on a chemical metaphor originated in the Γ language of Banâtre and Le Metayer [3]. This machine has been introduced to provide a formal framework for describing the operational semantics of parallel languages and the description of parallel computation.

A configuration of the CHAM is described by a solution which takes the form of a finite multiset of elements (called molecules), denoted by $S = \{|m_1, ..., m_k|\}$ A CHAM is specified by defining molecules, which are built according to some abstract syntax, and a set of so-called “reaction rules” that describe transformations over solutions. A reaction rule is of the form:

$$\{|m_1, ..., m_k|\} \to \{|m'_1, ..., m'_n|\} \quad \text{if condition } (m_1, ..., m_k)$$

and it means that if the molecules of the left-hand side are present in a solution and if a condition over these molecules holds, then they are replaced by the molecules of the right-hand side. The behavior of the CHAM obeys some general laws. The main law is called the *chemical law* and formally expresses that reaction can be performed freely within any solution:

$$\frac{S \to S'}{S \uplus S'' \to S' \uplus S''}$$

The CHAM has been used to describe concurrent calculus such as CCS [9] (see [4]) or the aynchronous π-calculus (see [5]).

3 How to represent multi-agent systems

3.1 Basic representations and definitions

In a simple multi-agent system an agent can be represented as a set of processes which bear the same name, i.e. a set of processes that have the same input port. Thus the behaviour of a simple agent can be represented as process which is identified by its name, i.e. a unique symbol. The behaviour of an agent is thus represented by a set of equations of the form:

$$A_j(id, v_{1j}, .., v_{kj}) = P$$

where P is a process (as in the π-calculus) and where id is a constant which represent the name of the agent and where the v_{ij} are the parameters of the agent (its internal variables and attributes).

3.2 Example

Let us describe a very simple example where an agent `x` of class `A` asks an agent `y` of class `B` and known by `x` to perform a specific task `t` (`x` and `y` are variables that stands over names of agents of class `A` and `B`). At the outset, `x` will perform Q_1 if `y` accepts and Q_2 otherwise. We will assume that `y` accepts or refuse to carry out the task `t` depending of a condition `P(t)` whose application will be considered here as being primitive. Here is the description of the two classes of agents:

```
A(x,t,y) =def ν(c)(↑y(request,t,c).
                    c(m).((m = agree)?Q1;
                          (m = refuse)?Q2))

B(x) =def x(m,z,k).(m = request)?
                   (P(z)?↑k(agree);↑k(refuse))
```

Let us now show and example of execution. We will consider an initial situation with two agents `a` and `b` respectively of class `A` and `B`. We will assume that the condition `P` applied to task `t` is true.

```
(1)    {|A(a,t,b), B(b) |}
(2)    {|ν(c)(↑b(request,t,c).c(m)..  ,
            b(m,t,k).(m = request).. |}
(3)    {|c(m)((m = accept)?Q1; (m = refuse)?Q2)),
            (request = request)?
              (P(t)?↑c(accept);↑c(refuse))  |}
(4)    {|c(m).((m = accept)?Q1; (m = refuse)?Q2)),
            (P(t)?↑c(accept);↑c(refuse))|}
(5)    {|c(m).((m = accept)?Q1; (m = refuse)?Q2)),
            ↑c(accept) |}
(6)    {|((accept = accept)?Q1;
           (accept = refuse)?Q2) |}
(7)    {|Q1 |}
```

The name `c` is used as a kind of "customer" in the terminology of actor languages [1] which receives the reply of agent `b` and carry out the continuation of the behavior of `a`. Due to the restriction operation `ν(c)` the name `c` is unique and thus, the answer cannot be confused with any other response received by an agent `x` of class `A`, because `accept` and `refuse` messages are not handled directly by `x` but by a process whose input port `c` is unique.

Remark 3.1 *In this example, agents disappear once they have executed their behavior. In a more realistic case, an agent behavior will be defined as a recursive equation.*

4 The Aalaadin model of organizations

[8] advocates that considering organizational concepts, such as groups, roles, structures, dependencies, etc. as first class citizens might be a key issue for building large scale, heterogeneous systems.

In their definition (so called the AALAADIN model), an organization is viewed as a framework for activity and interaction through the definition of groups, roles and their relationships. But, by avoiding an agent-oriented viewpoint, an organization is regarded as a structural relationship between a collection of agents. Thus, an organization can be described solely on the basis of its structure, i.e. by the way groups and roles are arranged to form a whole, without being concerned with the way agents actually behave, and multi-agent systems will be analyzed from the "outside", as a set of interaction modes. Thus, the specific architecture of agents is purposely not addressed.

4.1 Agent

The model places no constraints on the internal architecture of agents. An *agent* is only specified as an active communicating entity which plays *roles* within *groups*. This agent definition is intentionally general to allow agent designers to adopt the most accurate definition of agenthood relative to their application. The agent designer is responsible for choosing the most appropriate agent model as internal architecture.

4.2 Group

Groups are defined as atomic sets of agent aggregation. Eachagent is part of one or more groups. In its most basic form, the group is only a way to tag a set of agents. In a more developed form,in conjunction with the role definition, it may represent any usual multi-agent system. An agent can be a member of n groups at the same time. A major point of AALAADIN groups is that they can freely overlap. A group can be founded by any agent.

4.3 Role

The role is an abstract representation of an agent function, service or identification within a group. Each agent can handle multiple roles, and each role handled by an agent is local to a group. Handling a role in a group must be requested by the candidate agent, and is not necessarily awarded. Abstract communication schemes are thus defined from roles.

An agent a can enter a group g to play a role r associated with a boolean acceptation evaluation function $f_{g,r}$, if and only if, $f_{g,r}(a)$ evaluates to *true*. Note that the particular mechanism for role access within a group is not defined (systematic acceptance or refusal, admission conditioned by skills or by an admission dialog, relation to a group metrics, ...).

5 An operational semantics of Aalaadin

We will give an operational semantics of AALAADIN in terms of a calculus based on the π-calculus using the CHAM style of description. This calculus uses the basic definitions of the π-calculus with two additional operators: the *new g* operator which creates a new group, and the $in\, g.P$ operator which starts a new process P in a group g, within a specialized CHAM.

5.1 Multi-Agent Abstract Machine

A Multi-Agent Abstract Machine of MAAM for short is represented as a kind of multicham CHAM, i.e. as a set of labeled solutions $\{S_0, .., S_r\}$ where each solution is labeled by a name. We will note

$$\{|m_1, .., m_k|\}_{\mathtt{a}}$$

a solution of name a. Thus the state M of a MAAM is given by the state of all the solutions. The solution S_0 is called the *Origin group* (or $Orig$ for short) is supposed to be accessible by all agents and plays a specific role. We will call $gn(M)$ the set of all group names for a given state M of a MAAM. We assume that all group names are different.

5.2 Basic calculus

The basic calculus is simply the π-calculus with some syntactic extensions (see Appendix A) and with two specific operators: *new* which create an empty group and *in* which insert a process into a group. Here is their semantics:

$$\begin{array}{rcl} \{|in\,\mathrm{g}.P, Q|\}_{\mathtt{a}}, \{|R|\}_{\mathtt{g}} & \rightarrow & \{|Q|\}_{\mathtt{a}}, \{|P, R|\}_{\mathtt{g}} \\ M = \{|new\,g.P, Q|\}_{\mathtt{a}} & \rightarrow & M' = \{|P, Q|\}_{\mathtt{a}}, \{|\,|\}_{\mathtt{g}} \\ & & \text{if } g \notin gn(M) \\ & \rightarrow & M' = \{|P, Q|\}_{\mathtt{a}} \text{ if } g \in gn(M) \end{array}$$

But, by itself, the basic calculus cannot give a real account of group creation and agent admission. For example, in AALAADIN, group creation is controlled by a specific agent called the *GroupServer*, and group admission is managed by specific group managers. Thus, to give a precise description of the operational nature of AALAADIN, it is necessary to add a set of constructs and contraints to this basic calculus.

We will now formalize the main characteristics of AALAADIN, i.e. agent representation considering the set of roles an agent can handle.

5.3 Agent representation

In the AALAADIN model of MAS organizations an agent may be seen as the set of its role behaviors.

Each role behavior is represented as a process, thus as a set of process equations. Consequently, the behavior of an agent can be represented as a n-tuple $\langle A^0, .., A^n \rangle$ where each A^i is represented by a set of process equations of the form $A^i_j(id, v^i_{1j}, .., v^i_{kj}) = P^i$ where P^i is a process and where id is a constant which refers to the agent identity (a unique name that federates all its behaviors).

One of these role behavior, A^0 has a special meaning: it is the main behavior of the agent, the behavior that "glues" together all the roles played by one agent. This is where global information, i.e. information that do not refer to specific roles, is stored. This behavior is called the *personal behavior* of the agent.

There are some restrictions on the way agent behaviors can be described. Except for the role behavior A^0, the following applies:

Constraint 5.1 *All equations A^i_j of a role i behavior of an agent should not refer to any other role behavior other than itself and the global behavior. Thus, for any role behavior such that $A^i_j(id, v^i_{1j}, .., v^i_{nj}) =_{def} P^i$ and for any definition A^r_k that are present in P^i, $r = i$ or $r = 0$.*

This constraint means that a role cannot directly invoke another role directly. All roles are opaque to one another. All connections between roles are performed by the personal behavior A^0.

The difficulty of this modeling lies in the way the different role behaviors of an agent, which are situated in different CHAM solutions, can communicate together in a secure way: an agent x should not be allowed to communicate with the personal behavior of an agent y.

The general way to solve this problem in our framework is to associate a private group, represented as a labeled solution, with each agent and to put its personal behavior in this group when the agent is created.

Thus the creation of an agent with personal role $A(a, v_1, .., v_n)$ is done as follows:

```
NewAgent(A) =$_{def}$
   $in$ Orig.$\nu$(a).($new$ a.$in$ a.A(a,$v_1$,..,$v_n$))
```

One can note that we have used the same name `a` for both the agent and its personal solution, for simplicity sake.

Communications between the personal behavior and the various roles of an agent are done using the in primitive which allow for a transfer of processes between groups. Thus, for a role behavior of agent `a` in group `g`, sending a request `req` with arguments $a_1, .., a_n$ and continuation `c`, can be described by the following code:

```
SendSelf(a,g,req($e_1$,..,$e_n$),c) =$_{def}$
  $\nu$(c1)($in$ a.$\uparrow$a(req($e_1$,..,$e_n$),c1).c1(r).$in$ g.$\uparrow$c(r))
```

The message `req` is wrapped into a process which is transferred to the personal group (whose name is also `a`). When a result is sent to the continuation `c1`, the result is transferred back to the group `g` and the result is sent to the overall continuation `c`.

5.4 Group creation

As we have seen above, at the basic level, there is a primitive in the calculus that adds a new group to a MAAM. At a higher level there exists a specific, primitive agent called the *groupServer* that stands in the Origin group and is the only one that can use the group creation primitive. All requests for creating a group have to be asked to the group server which registers all the created groups.

When the group server is requested to create a new group with name `g` and group manager `gm`, and if it does agree, a group is created (a solution is created with name `g`), the group manager `gm` is registered by the group server, and the groupmanager is started in the new group `g`.

Here is the code that define the behavior of the group server. The group names are represented with a dictionary (see Appendix):

```
GroupServer(gnames) =$_{def}$
  groupServer(msg)
    match(msg)(
      createGroup(g,$A^{gm}$,c) $\rightarrow$
          $\nu$(c1)($\uparrow$gnames(hasKey(g,c1)).
          c1(r)(r = true)?
             ($new$ g|c(OK)|$\uparrow$gnames(atput(g,true)).
                (GroupServer(gnames)|$in$ g.$A^{gm}$)
             :c(fail)|GroupServer(gnames));
      groupNames(c)$\rightarrow$ $\uparrow$gnames(keys(c)
```

```
                    |GroupServer(gnames);
      isGroup(g,c)→ ↑gnames(at(g,c))
                   |GroupServer(gnames);
```

```
CreateGroup(g,gm,c) =$_{def}$
  in Orig.↑groupServer(createGroup(g,gm,c))
```

The group manager is the agent that manages a group, i.e. the agent which authorize or refuse admission to an agent, and which knows all agents of a group and their roles.

5.5 Agent admission

In each group there is one agent that has the role of managing the group, i.e. of authorizing or refusing admission to an agent, and of knowing all members of the group with their roles. This agent is called the *group manager*. But, despite its name, this is not necessarily a specific agent that plays that role: any agent can play this role if it has the capability of doing so.

The group manager can be accessed from the Origin solution, i.e. the solution where all global behaviors reside. Here is the description of the process of joining a group.

To enter a group `g`, an agent `a` asks the `groupServer` the name of the agent manager, `gm` of `g`. Then, it asks `gm` if it can enter the group playing the role `r`. If the answer is positive, the agent `a` can actually enter the group and start acting in this group using its role behavior $\mathtt{A}^{\mathtt{r}}_{\mathtt{start}}$`(a)`.

```
JoinGroup(a,g,r,$A^r_0$,c) =$_{def}$ in Orig.$\nu$(c1)(
  ↑groupServer(isGroup(g,c1)).
      c1(b).(b = true)?
        $\nu$(c2)((EnterGroup(a,g,r,$A^r_0$,c2)).
          c2(r).in a.c(r)):
        in a.c(nogroup))
```

```
EnterGroup(a,g,r,$A^r_0$,c) =$_{def}$ in Orig.$\nu$(c1)(
  $\nu$(c1) in g.↑groupManager(check(a,r,..,c1)).c1(r)
    match(r)
      refuse() → c(refused)
      agree(s) → in s.$A^r_0$ | c(succeed))
```

5.6 Example

In this section we will give a simple example of group admission and creation. Let us suppose that two agents want to play ping-pong, transmitting a message back and forth five times. To achieve this goal, the PingPong agent tries to locate a `ppg` group. If this group exists, it joins with the role of `p` (player) and wait for the first message, and if not, it creates the group, joins it, and sends the first message.

The personal behavior would be:

```
PingPong$^0$(a) =$_{def}$
  $\nu$(c1).JoinGroup(a,ppg,p,PingPong$^p_{init}$(a),c1).
    c1(r1).(r1= nogroup)?
      $\nu$(c2).CreateGroup(ppg,PingPong$^{gm}$(a),c2).
      c2(r2).(r2= success)?
        JoinGroup(a,ppg,p,PingPong$^p_{play}$(a),a)
```

The two process definitions for the player behavior are:

```
PingPong$^{c}_{init}$(a) =$_{def}$
   $\nu$(b).↑groupmanager(getAgentsWithRole(p,b)).
      ↑b(5,a).PingPong$^{p}_{play}$(a)
```

```
PingPong$^{p}_{play}$(a) =$_{def}$
   a(m,b).(m $\neq$ 0)?(↑b(m-1,a).PingPong$^{p}_{play}$(a))
```

For simplicity sake, we suppose in this exemple that the role is always awarded.

```
PingPong$^{gm}$(a) =$_{def}$
   groupManager(msg)
      match(msg)
         check(a,g,r,c) $\rightarrow$
            ↑c(agree(g))| ↑roles(atput(r,a))
         getRoles $\rightarrow$ ↑roles(keys(roles));
         getAgentsWithRole(r)$\rightarrow$
            ↑roles(at(r,c))
```

6 Conclusion

We have shown in this report that it is possible to give a precise and clear operational semantics of organizations in multi-agent systems using the AALAADIN generic model of organizations. This operational semantics has been given using π-calculus and the CHAM, considering groups as solutions and agents as set of role behaviors. We have also described the constraints that applies to the description of such organizations.

Some specific issues (security, model heterogeneity, reflection, ...) have been left out and will be described in a forthcoming report.

A The Picol language

The PICOL language (which stands for PI-Calculus Oriented Language) is used throughout this report to describe multi-agent behaviours. The PICOL language is a syntactic extension of the π-calculus which, while conserving its main features, facilitates the description of processes.

A.1 Syntax of Picol

Here is the syntax of the PICOL language. The following metavariables are used: P and Q denote processes, E denotes expressions, C denotes conditions, x and v denote names and variables, F denotes functions that are considered as primitives.

```
$P, Q$    ::=   nil
                $P \mid Q$
                $x$($v_1$,..,$v_n$).$P$
                ↑x($E_1$,..,$E_n$).$P$
                $C$?$P$
```

$$\begin{array}{lll}
 & & C\,\texttt{?}P\,\texttt{:}Q \\
 & & C_1\,\texttt{?}P_1\texttt{;..;}\,C_n\,\texttt{?}P_n \\
 & & \nu(v)P \\
 & & \texttt{!}P \\
 & & \texttt{match(x)(}f_1 \rightarrow P_1;..;f_k \rightarrow P_k\texttt{)} \\
 & & A(E_1,..,E_n) \\
C & ::= & \texttt{(}E_1\ \textit{rel-op}\ E_2\texttt{)} \\
 & & \texttt{(}C_1\ \texttt{or}\ C_2\texttt{)} \\
 & & \texttt{(}C_1\ \texttt{and}\ C_2\texttt{)} \\
 & & \texttt{not}\ C \\
E & ::= & \texttt{(}E_1\ \textit{op}\ E_2\texttt{)} \\
 & & F(E) \\
 & & x \\
 & & \text{numerical constant} \\
 & & \texttt{"}\text{string}\texttt{"} \\
\textit{op} & ::= & +,*,-,/,.. \\
\textit{rel-op} & ::= & =,\neq,<,>,\leq,\geq,\in,\notin
\end{array}$$

A.2 Semantics of Picol

Here is the operational semantics of PICOL in terms of reduction relations.

Concurrency

$$\begin{array}{rcl}
\{\!|(P \mid Q)\,|\!\}_{\mathtt{g}} & \rightleftharpoons & \{\!|P, Q\,|\!\}_{\mathtt{g}} \\
\{\!|\ \uparrow\!\mathtt{x}(e_1,..,e_n).P, & & \\
x(v_1,..,v_n).Q|\!\}_{\mathtt{g}} & \rightarrow & \{\!|P, Q[e'_1/v_1,..,e'_n/v_n]|\!\}_{\mathtt{g}} \\
 & & \text{if } \forall i\ e_i \rightarrow_p e'_i \\
\{\!|\nu(v)P|\!\}_{\mathtt{g}} & \rightarrow & \{\!|P\,|\!\}_{\mathtt{g}} \\
 & & \text{if } v \notin \text{free variables of } P
\end{array}$$

Conditionals

$$\begin{array}{rcl}
\{\!|(C?P)\,|\!\}_{\mathtt{g}} & \rightarrow & \{\!|P\,|\!\}_{\mathtt{g}} \text{ if } C \rightarrow_p \text{True} \\
 & \rightarrow & \{\!|\mathtt{nil}\,|\!\}_{\mathtt{g}} \text{ if } C \rightarrow_p \text{False} \\
\{\!|(C?P{:}Q)\,|\!\}_{\mathtt{g}} & \rightarrow & \{\!|P\,|\!\}_{\mathtt{g}} \text{ if } C \rightarrow_p \text{True} \\
 & \rightarrow & \{\!|Q\,|\!\}_{\mathtt{g}} \text{ if } C \rightarrow_p \text{False} \\
\{\!|C_1?P_1;..;C_n?P_n\,|\!\}_{\mathtt{g}} & \rightarrow & \{\!|P_i\,|\!\}_{\mathtt{g}} \text{ if } C_i \rightarrow_p \text{True}
\end{array}$$

Primitives

$$\begin{array}{rcl}
(e_1 \text{ op } e_2) & \rightarrow_p & \mathit{Apply}(op, e'_1, e'_2) \\
 & & \text{if } e_1 \rightarrow_p e'_1 \text{ and } e_2 \rightarrow_p e'_2 \\
F(e) & \rightarrow_p & \mathit{Apply}(F, e') \text{ if } e \rightarrow_p e' \\
(e_1 \text{ and } e_2) & \rightarrow_p & e'_1 \wedge e'_2 \text{ if } e_1 \rightarrow_p e'_1 \\
 & & \text{and } e_2 \rightarrow_p e'_2 \\
(e_1 \text{ or } e_2) & \rightarrow_p & e'_1 \vee e'_2 \text{ if } e_1 \rightarrow_p e'_1 \\
 & & \text{and } e_2 \rightarrow_p e'_2 \\
\text{not } e & \rightarrow_p & \neg e' \text{ if } e \rightarrow_p e'
\end{array}$$

A.3 Primitives structures

In the report we use some constructs such as dictionaries and lists. To ease the reading we have assumed that list structure (as in SCHEME or LISP) are primitives. Thus, primitives functions such as `cons`, `car`, `cdr`, `assoc`, `member` and `map` have the same meaning than in SCHEME. To see how lists can be defined using bare π-calculus, refer to [10].

A dictionary is simply made of an association list. The methods `at`, `atput`, `values`, `keys`, and `hasKeys` have the same meaning than their counterpart in SMALLTALK. One can note that in the `hasKeys` methods the dictionary sends a message `keys` to itself and process the list of keys in a continuation:

```
Dictionary(id,vals) =def
  id(msg).
    match(msg)(
      at(key,c) → ↑c(tl(assoc(key,vals))),
      atput(key,val) →
        Table(id,(cons(cons(key,val),vals))),
      atputlist(key,val) →
        ν(c1).(↑id(at(key,c1)).c1(lst).
          ↑id(atput(key,cons(val,lst)))
      values(c) → ↑c(map(cdr,vals)),
      keys(c) → ↑c(map(car,vals)),
      hasKeys(key,c) →
        ν(c1)(↑id(keys(c1)).c1(lst).
        ↑c(member(key,lst))))
```

References

[1] Gul Agha and Carl Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, Cambridge, MA, 1987.

[2] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] J-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Programming*, 15(1):55–77, 1990.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[5] G. Boudol. Some Chemical Abstract Machines. volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer-Verlag, Berlin, 1993.

[6] Gerard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, Inria, 1992.

[7] Jacques Ferber and Patrice Carle. Actors and agents as reflective concurrent objects: A mering IV perspective. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1420–1436, November-December 1991.

[8] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *3d International Conference on Multi-Agent Systems (ICMAS'98) Proceedings*. IEEE, 1998.

[9] R. Milner. *Lectures on a Calculus for Communicating Systems*, volume 197 of *LNCS*. Springer-Verlag, New York, NY, 1984.

[10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100(1):1–77, September 1992.

[11] Benjamin C. Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*, chapter 139. CRC Press, 1996.