An ASM-Based Approach for Mobile Systems

Marcelo de Almeida Maia[1] and Roberto da Silva Bigonha[2]

[1] Departamento de Computação, Universidade Federal de Ouro Preto
Campus Morro do Cruzeiro, 35400-000 Ouro Preto MG, Brazil
URL: www.dcc.ufmg.br/~marcmaia
email: marcmaia@dcc.ufmg.br
Phone: +55 31 499-5881 - Fax: +55 31 499-5858

[2] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais,
Av. Antônio Carlos, 6627, 31270-010 Belo Horizonte MG, Brazil
URL: www.dcc.ufmg.br/~bigonha

Abstract. In this work we present the use of a novel language based on Gurevich Abstract State Machines[7] to specify active mobile objects semantics. We focus on the mobility support based on the explicit interaction abstraction between units of specification. The mobility is expressed by changing the communication topology dynamically. We provide the formal semantics of the new approach and show how the proposed method provides an adequate way to reason about a specification based on mobile objects.

Keywords: mobility, formal semantics, reasoning, ASM

1 Introduction

The nineties have been marked by a great explosion in the use of the Internet. The Wide World Web is accessible everywhere in the world, and this situation causes a substantial impact on how people use computers. Tools for developing the corresponding new generation of programs have been evolving to address problems such as mobility, security, fault-tolerance and many others. But such class of tools still lacks solid mathematical foundations that would help assessing the final product behavior. Because of the inherent difficulty of these issues, i.e. mobility and concurrency, the use of mathematical formalism to help the systematic development of correct new generation applications is even more imperative than for the classical applications.

The methodology for producing ASM specifications provides a vertical abstraction mechanism, in the sense that the ground model (the model resulted from converting the informal specification into the method language) is successively refined until considered adequate. However, it still does lack some kind of horizontal abstraction for composing existent specifications. In the context of ASM, there is already some work in the direction of providing them with some

kind of horizontal abstraction. Glavan and Rosenzweig developed a theory of concurrency [6] that enables the encoding of some traditional calculus, such as the $\pi-$calculus[11] and the Chemical Abstract Machine [2]. However, Glavan's work does not explicitly address the problem of mobility and it does not support encapsulation and information hiding mechanisms, issues which will be directly treated in this work. May [10] has developed a work with similar aims as ours, and although it provides some form of encapsulation and information hiding, the usual modularization concepts must be further added to the model. The mobility issue is not considered either.

In order to address these issues, we have proposed in [9] new abstraction mechanisms in the context of ASM, namely the Interactive Abstract State Machines (IASM). Our approach provides special constructions for explicit definition of how different pieces of specification interact with each other, so as to free the user from the burden of providing the whole low-level specification of the communication topology and the message interchanging between different specifications. In [8] we have shown how the mechanisms proposed in [9] would address the mobility issue. The present work provides a revised semantics of the novel constructions in a style similar to the Lipari Guide[7], and we also provide a more detailed reasoning of our case study.

The remainder of this paper is structured as follows. In the Section 2, we set the context on which we will be interested. In Section 3, we give the semantics of our extended language of IASM. In Section 4, we show the adequacy of this new approach by means of a case study. And finally, in Section 5, we summarize our conclusions.

2 Active Mobile Objects

The World-Wide-Web has originated a new model of computation that previous methods of specification seems not to address satisfactorily. Here we will not try to define exactly what are all the necessary abstraction mechanisms to support this new model of computation. Instead, we propose an abstract framework on which important features of mobile systems may be reasoned about. This framework eventually may be extended to support other features related to mobile systems. We hope to demonstrate the ease of use provided by the new specification framework.

Perhaps, the main work that has influenced researches on formal semantics for mobile systems is π-calculus[11], where the channels of communication can be transmitted over other channels, so that a process can dynamically acquire new channels. The transmission of a channel over another channel gives the recipient the ability to communicate on that channel. It is becoming a common approach, the addition of discrete locations to a process calculus and consider failure of those locations[1][5]. The work described in [5] has added a notion of named locations, and a notion of distributed failure; locations form a tree, and subtrees can migrate from one part of the tree to another, therefore becoming subject to different observable failure patterns. Cardelli and Gordon, in a recent work[4],

argues for a more explicit notion of movement across boundaries, defining the ambient calculus.

Instead, our approach for mobility is based on the dynamic construction of distributed active daemons and on the dynamic configuration of the communication topology between them. Our solution is entirely distributed, in the sense that all global information may only be inferred by input/output interactions with the environment. The location issue is addressed by local state information. Barriers to mobility are addressed by pairwise commitment between interacting units. Another difference of our approach is that it is not based on process algebra. Instead, we use an operational approach to formalize our ideas.

3 Interactive Abstract State Machines

In this section we present the Interactive Abstract State Machines (IASM) language and its formal semantics. This extension is heavily based on the multi-agent ASM[1]. Instead of using the established terminology of ASM, such as, modules and agents, we prefer to introduce other related names, respectively, unit definitions and unit instances. All basic concepts such as static algebras, vocabularies, terms, universes, appropriate states, Fun notation, update sets, basic rules and import rules should be borrowed from [7].

An *IASM specification* $\mathcal{SPEC}$ is a triple ($\mathcal{CUD}$, $\mathcal{V}$, $\mathcal{M}$), where:

- $\mathcal{CUD}$ is a collection of unit definitions $\mathcal{UD}$,
- $\mathcal{V}$ is the vocabulary of the whole specification.
- $\mathcal{M}$ is the main specification,

A *unit definition* $\mathcal{UD}$ is a quintuple ($\mathcal{N}^0$, $\mathcal{N}^1$, $\mathcal{IS}$, $\mathcal{IR}$, $\mathcal{IC}$) where:

- $\mathcal{N}^0$ is a static nullary function name corresponding to the name of the unit definition. Different unit definition names are interpreted as different elements, as suggested by multi-agent ASM semantics.
- $\mathcal{N}^1$ is a universe name, i.e., a unary relation name. The corresponding universe contains all unit instances derived from $\mathcal{UD}$. We adopt the convention of overloading this name, so we can use the same name for the unit definition name and for the corresponding universe.
- $\mathcal{IS}$ defines the internal vocabulary $Fun(\mathcal{UD})$ of $\mathcal{UD}$.
- $\mathcal{IR}$ is the declaration of the interaction rule which guides the interaction of the respective unit instances with their environment. This rule establishes how communication between unit instances can occur. It does not only define the information interchanging but also the synchronization restrictions imposed between the units.
- $\mathcal{IC}$ is the declaration of the internal computation rule which updates the local state of the instance. The internal rules of a unit definition have the same syntax as those of the pure ASM model. They can refer only to the respective instance view of the state, i.e., references to the vocabulary of other units are forbidden (as we will see in the definition of $\mathcal{V}$).

[1] Distributed Ealgebras in the Lipari Guide.

The *vocabulary* $\mathcal{V} = Fun(\mathcal{SPEC})$ of the whole specification is defined as:

$$(\bigcup_{i=1}^{|\mathcal{CUD}|} Fun(\mathcal{UD}_i)^{+Self}) \cup (\bigcup_{i=1}^{|\mathcal{CUD}|} Lab(\mathcal{UD}_i)^{+Self}) \cup$$
$$Fun(\mathcal{M}) \cup \_Messages \cup \_Connections \cup Aux + \{\texttt{Def}\} - \{\texttt{Self}\}$$

where:

- $Fun(\mathcal{UD}_i)^{+Self}$ is the internal vocabulary of the unit definition i, where each function declared in $\mathcal{UD}_i$ has its arity incremented by 1, and the extra argument is the function `Self`. This assures that each instance has its local state. Note that equal function names in different unit definitions are in fact the same name in $\mathcal{V}$, but can coexist securely because their interpretations for each unit instance are disjoint.
- $Lab(\mathcal{UD}_i)^{+Self}$ is the collection of function names defined by the labels occurring in $\mathcal{UD}_i$, such that $Fun(\mathcal{UD}_i)^{+Self} \cap Lab(\mathcal{UD}_i)^{+Self} = \emptyset$. There are two kinds of labels:
  1. *channel labels*: a channel label `c` occurs in input, output and connection rules. It defines a nullary static function name `c` which is interpreted as a quotation of itself, i.e., `"c"`. Obviously, we suppose `"c"`$\in X$, where X is the superuniverse.
  2. *counting labels*: a counting label `c` occurs in labeled interaction rules and defines a unary function name `c`, which is defined only in the argument `Self`. In the initial state of $\mathcal{SPEC}$, all these functions are interpreted as 0 in `Self` argument.
- $Fun(\mathcal{M})$ is the global vocabulary, i.e., the collection of nullary function names whose values are the static instances, and the collection of unit definition names and their respective universe names.
- $\_Messages$ is an auxiliary universe name which, in the same sense of the ASM universe $Reserve$, is forbidden to be mentioned by any rule. It will be manipulated by the input and output interaction rules.
- $\_Connections$ is another auxiliary universe name which is also forbidden to be mentioned by any rule. It will be manipulated by the connection interaction rule.
- Aux is the collection of auxiliary functions names used to control the interaction rules. We adopt the convention that the first character of all the private functions names in Aux is the underscore `"_"`, but, for the sake of conciseness, we do not explicitly show how these function names are used internally, since this could be straightforwardly induced. The public function names will be introduced as needed.
- `Def` is the function which maps each unit instance `u` to its corresponding unit definition name `U`, such that, `Def(u)=U`.
- `Self` has the same role as in pure ASM.

The *main specification* $\mathcal{M}$ defines the initial state of $\mathcal{SPEC}$, and is a pair $(\mathcal{CSUI}, \mathcal{CSC})$, where:

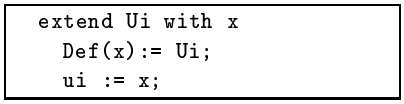

Fig. 1. Static instantiation rule

- $\mathcal{CSUI}$ is a collection of static unit instance declarations of the type `u:U`, where `u` is a nullary function name whose value is supposed to be an instance of the unit definition U.
  Formally, let G'_0 be the state resulted after firing the following update set on any $\mathcal{V}$-state G, with superuniverse X, such that, $G \models (\forall x \in X)$`U(x)=false`: $\bigcup_{i=1}^{|\mathcal{CSUI}|} Updates(SInst_i, G)$, where $SInst_i =$ `ui:Ui` has the same update set of the rule defined in Figure 1. Note that the first occurrence of `Ui` in rule refers to universe `Ui`, whereas the second occurrence refers to the unit definition name.
- $\mathcal{CSC}$ is a collection of static connections `u1 <-> u2`, where `u1` and `u2` are static unit instances names in $\mathcal{CSUI}$. Formally, if $\mathcal{CSC} = \emptyset$ then the initial state of $\mathcal{SPEC}$ is G'_0. Otherwise, let c_2 be a function name in $Fun($`u1`$)^{+Self}$ and c_1 be a function name in $Fun($`u2`$)^{+Self}$, such that, $\forall\ u_a$`<->`$u_b \in \mathcal{CSC}$, in the state G'_0, c_1 must be updated with the u_a, and c_2 must be updated with u_b. Note that the vocabulary of u_a and u_b must have the required function names in order to each u_a `<->` u_b be a valid connection. Besides this, we require that these function be uniquely identified by assigning to them their corresponding signatures in terms of universe names.

The $\mathcal{V}$-state G is a state of $\mathcal{SPEC}$ if it satisfies the restrictions imposed by $\mathcal{UD}$, $\mathcal{V}$ and $\mathcal{M}$. An element u is a unit instance at G, if there is a unit definition name U, such that, $G \models$ `Def(u)=U` and `U(u)=true`. Let $\mathcal{IR}_U$ and $\mathcal{IC}_U$ be the interaction and internal computation rules of the unit definition U, respectively, and `++` be the concatenation of rules. The corresponding program of `u` is $Prog(u) = \mathcal{IR}_U$ `++` $\mathcal{IC}_U$. The vocabulary $Fun(u)$ of u is $Fun(\mathcal{UD}_U^{+Self})$. $View_u(G)$ is the reduct of the $\mathcal{V}$-state G of to vocabulary $Fun(u)^{+Self} \cup Fun(\mathcal{M}) \cup \_Messages \cup \_Connections \cup Aux +$ {`Def`} $-$ {`Self`} expanded with `Self`, which is interpreted as u.

The execution of the whole specification is based on the execution of each unit instance, which is defined by their interaction and computation rules. A unit instance u can *make a move* at G by firing $Prog(u)$ at $View_u(G)$, and changing G accordingly. Even if $Fun(u_1)^{+Self} \cap Fun(u_2)^{+Self} \neq \emptyset$, neither u_1 or u_2 cannot modify the state defined by the other, since this restriction is imposed by construction of $\mathcal{V}$ (recall the introduction of an extra argument in $Fun(\mathcal{UD}_\rangle), 1 \leq i \leq n$). In order to perform a move of a instance u, fire:

$$Updates(u, G) = Updates(Prog(u), View_u(G)) =$$
$$Updates(\mathcal{IR}_U, View_u(G)) \cup Updates(\mathcal{IC}_U, View_u(G)).$$

```
i ::= c=t -> u
  |   f(tt) <- u.c
  |   connect c=u:U.s in i
  |   new u:U | destroy u
  |   i | i   |   i ;; i   |   i : l
  |   waiting(name)
  |   if guard  then   i
```

Fig. 2. Interaction Rule

A *run* of $\mathcal{SPEC}$ is defined exactly in the same way of partially ordered runs of multi-agent ASM, so we will omit the definition here. We can reuse the same definition because the difference between multi-agent ASM and Interactive ASM runs is in the definition of move, and the definition of a partially ordered run does not rely on how a move is defined.

In order to complete the semantic description of Interactive ASM we need to define how the update set of an instance move is computed. In the following sections we define how to compute the update set of the interaction and internal rules. We have adopted an indented writing style, avoiding the use of the delimiters `endif`, `endconnection`. Interaction rules are defined as in Figure 2.

3.1 I/O Rules

Let Out = `c=t -> u` be *output* rule, where `t` is a term, `c` is the label of the output channel, and `u` is a term representing the target unit instance. If `t` is a nullary function name we may omit `c` and assume `c=t`. The update set $Updates(Out, G)$ is defined as $Updates(R_{out}, G)$ where R_{out} is defined in Figure 3.

Let In = `f(tt) <- u.c` be an *input* rule , where `f` is a function name, `tt` is a tuple of terms whose length equals the arity of `f`, `u` is a term representing the source unit instance and `c` is the label of the channel in the source unit instance. The update set $Updates(In, G)$ is defined as $Updates(R_{in}, G)$ where R_{in} is defined in Figure 4. Note that both input and output rules update the auxiliary universe $\_Messages$ in order to proceed with the interaction. An element of this universe has three attributes: the target instance (`target`) which will receive the message, the unit instance which has sent the message (`source`), a label indicating the source channel from where the message has been sent (`label`), and the contents of the message (`value`).

3.2 Topology Configuration Rules

Let $Conn$ = `connect c=u:U.s in i` be a *connection* rule, where `c` is a label defining the point of the connection inside the current instance, `u` is a term representing the instance to be connected, `U` is the unit definition name from whose definition `u` was derived, `s` is a function name such that, `s` $\in Fun(u)$, and when

```
extend _Messages with x
  target(x):= u;
  source(x):= Self;
  label(x):= c;
  value(x):=t;
```

Fig. 3. Output Rule R_{out} for `t:c -> u`

```
if (∃ m ∈ _Messages)
       target(m) = Self ∧ source(m) = u ∧ label(m)= c then
  choose m in _Messages satisfying
         target(m) = Self ∧ source(m) = u ∧ label(m)= c
    f(tt) := value(m);
    _Messages(m) := false;
```

Fig. 4. Input Rule R_{in} for `f(tt) <- u.c`

the connection is completely performed `s` is supposed to be updated with the current instance. If `u` is a nullary function name, we may omit `c` and assume `c=u`. We may also omit `U` and `s`, and assume that their corresponding value is `undef`. The connection rule mainly updates the auxiliary universe *_Connections*, whose elements are unordered[2] pairs of quintuples: $((u_1, c_1, U_1, o_1, a_1), (u_2, c_2, U_2, o_2, a_2))$, where u_i are the unit instances of the connection, c_i are the labels defining the point of connection inside the current instance, U_i are the corresponding unit definition names, o_i are labels defining the point of connection in the other instance, such that, when the connection is performed $c_1 = o_2$ and $c_2 = o_1$, and a_i represent each instance awareness of the connection, for $i \in \{1, 2\}$. The connection rule is performed in three stages, what means that it takes at least three moves to be completely performed, since the stages need to be performed in sequence.

1. In the first stage a connection with the other instance must be established. There are three cases to be considered here:
   (a) The connection rule defines which is the unit instance `u` to be connected to, and that instance `u` had already attempted the connection. Formally, this means that `u` is defined, and `_Connections`$((party_1, party_2))$= `true`, where $party_1$ = (`u`, `c`, `U`, `f`, `true`), $party_2$ =(v, k, V, x, `false`), v = `Self` or v = `undef`, k = `f` or k = `undef`, V = `Def(Self)` or V = `undef`, and x = `c` or x = `undef`. If any of the values v, k, V, and x are interpreted as `undef`, this means that the instance `u` does not require the corresponding restriction connected, i.e., any value for v, k, V, and x should satisfy. In this first case $Updates(Conn, G)$ for the first stage is the same as the one defined by rule (a) in Figure 5.

[2] i.e., $((u_1, c_1, U_1, o_1, a_1), (u_2, c_2, U_2, o_2, a_2)) \equiv ((u_2, c_2, U_2, o_2, a_2), (u_1, c_1, U_1, o_1, a_1))$

(b) The connection rule defines which is the unit instance to be connected, and that unit had not attempted the connection yet. Formally, this means that `u` is defined, and `_Connections((`$party_1$`,`$party_2$`))=false`, where $party1$ and $party_2$ are the same as above. In this case, there are two stages to be performed in sequence, so these two stages takes at least two moves to be performed:
   i. The current unit instance has to warn the other instance that it is attempting a connection. Formally, this means that the update set $Updates(Conn, G)$ for the first sub-stage of the first stage is the same as the one defined by rule (b) in Figure 5.
   ii. The current unit instance waits until the other instance notifies that it is aware of the connection. Formally, the current instance checks if the other instance had updated $\_Connections$, either with the indication that it is aware of the connection, just like the rule (a) or (c) in Figure 5, or with rule (b) in Figure 5, and in this case it is necessary to fire rule (a) in Figure 5 to indicate that the current instance is aware of the connect. When the above has been performed, the private control functions in the sub-vocabulary Aux are properly updated to release the sequence.

(c) The connection rule does not define a specific unit instance to be connected. In this case the only action the current instance `Self` can perform is to check if there is any instance trying a connection such that `Self` could satisfy. Formally, the current instance checks whether there is an element in `_Connections`, such that, either $party_1$ or $party_2$ equals (a, b, A, o, `false`), where a=`Self` or a=`undef`, b=`s` or b=`undef`, A=`Def(Self)` or A=`undef`, o=`u` or o=`undef`. If there is such element, let the other connect point be $k = v : V.f$ in a rule where the current instance must notify the other instance, that it has properly accepted the connection. Formally, this means that the update set used to establish the connection is the same as defined by rule (c) in Figure 5. Note that the undefined `u` is properly updated with the other instance.

2. In the second stage the interaction i is performed. By induction, the update set on this stage is $Updates(i, G)$.
3. In the last stage the current unit instance must disconnect from the other instance. Formally, this means that the element x that represented the connection must be discarded, and if the function `u` was undefined before the connection, it must continue undefined after the connection. Note that if x has already been discarded by the other instance, `_Connections` is not updated. So the update set in this stage equals $(\{Update((\_Connections(x), false), G)\} \uplus \emptyset) \cup (\{Update((\mathtt{u}, undef), G)\} \uplus \emptyset)$, where $s_1 \uplus s_2$ means either s_1 or s_2, but not both.

Let *Alloc* = `new u:U` be an *allocation* rule, where `u` is the identification of the new instance, `U` is the unit definition from where `u` is derived. Formally, the update set *Updates(Alloc, G) = Updates(*R_{alloc}*, G)*, where R_{alloc} rule is the

```
choose x in _Connections satisfying
            value(x) = (party1, party2) with
      _Connections(x) := false;
extend _Connections with x
      value(x):=((u,c,U,f,true),(Self,f,Def(Self),c,true));
                                        (a)
extend _Connections with x
      value(x):=((u,c,U,f,false),(Self,f,Def(Self),c,true));
                                        (b)
choose x in _Connections satisfying
            value(x) = ((v,k,V,f,false),(Self,f,Def(Self),k,true)) with
      _Connections(x) := false;
extend _Connections with x
      value(x):=((v,k,V,f,true),(Self,f,Def(Self),k,true));
u := v
                                        (c)
```

Fig. 5. Connection Rule

same as the rule in Figure 1, substituting `u` and `U` for `ui` and `Ui`, respectively. This rule provides the dynamic instantiation of a unit.

Let *Dealloc* = `destroy u` be a *deallocation* rule, where `u` is the identification of instance to be discarded. Formally, the update set *Updates(Dealloc, G)* = *Updates(*$R_{dealloc}$*, G)*, where $R_{dealloc}$ rule is `U(u) := false; Def(u) := undef`.

3.3 Control Rules

Let $Par = i_1 \mid i_2$ be a *parallel* rule, where i_1, i_2 are interaction rules. By induction on i_1 and i_2, we define $Updates(Par, G) = Updates(i_1, G) \cup Updates(i_2, G)$. The operator $\mid$ may be omitted, if desired.

Let $Seq = i_1 \ ;; \ i_2$ be a *sequential* rule, where i_1, i_2 are interaction rules. By induction on i_1 and i_2, let $Updates(Seq, G) = Updates(i_1, G) \uplus Updates(i_2, G)$. There are some more semantics embedded in the sequential rule. Let u be any unit instance. Let $SM = m_1, \ldots, m_r$, for some finite r, be a initial valid sequence of moves performed by unit u. Let $SSM = m_{seq1}, \ldots, m_{seqr}$ be a subsequence of SM, such that, the rule $Seq = i_1 \ ;; \ i_2$ is ready to be fired on, and only on, each m_{seqi} $(1 \leq i \leq seqr)$. Suppose $m_i = m_{seqm}$, and $m_j = m_{seqn}$, for some $i, j \in \{1, \ldots, r\}$, and for some $seqm$, $seqn \in \{seq1, \ldots, seqr\}$. If $i < j$ then $seqm < seqn$, but not necessarily $m_{i+1} = m_{seqm+1}$. By induction on $seq1, \ldots, seqr$, the update set $Updates(Seq, G)$ is calculated as $Updates(i_1, G)$ on m_{seq1}. Suppose on move m_{seqi} $(1 \leq i \leq seqr)$, $Updates(Seq, G)$ is calculated as $Updates(i_1, G)$. Then, on move m_{seqi+1}, $Updates(Seq, G)$ is calculated as:

$$\begin{cases} Updates(i_2, G) \text{ if } i_1 \text{ has been } \textit{completely performed} \\ Updates(i_1, G) \text{ otherwise.} \end{cases}$$

Alternatively, if on m_{seqi}, $Updates(Seq, G)$ is calculated as $Updates(i_2, G)$, i.e.:

$$\begin{cases} Updates(i_1, G) \text{ if } i_2 \text{ has been } \textit{completely performed} \\ Updates(i_2, G) \text{ otherwise.} \end{cases}$$

An interaction i will be *completely performed* in a move m, if and only if:

- i is an input rule and there is a message in $\_Messages$ satisfying the input condition, and thus enabling the update of the target function on move m.
- i is a connection rule the disconnection occurs on move m.
- i is a synchronization rule and the corresponding `waiting` function is updated to *false* by the internal computation rules on move m.
- i is any other rule to be performed on move m.

Note: The necessity or not of the sequential rule may be a point for controversy, because pure ASM avoid this notation in favor of more logical transparency. We argue for the need of the sequential rule because it makes the interaction rule of a unit definition clear, in the sense that in the interaction rule it is forbidden explicit update rules. Otherwise, the conceptual transparency of the interaction rule would be lost, if updates were allowed.

Let $Cond$ = `if` g `then` i_1 `else` i_2 be a *conditional rule*, where i_1, i_2 are interaction rules. By induction on i_1 and i_2, we define $Updates(Cond, G) = Updates(i_1, G) \uplus Updates(i_2, G)$. If the guard g is evaluated to *true* on state G the update set is $Updates(i_1, G)$, otherwise it is $Updates(i_2, G)$.

Let $Sync$ = `waiting(name)` be a *synchronization* rule, where `name` is a nullary function name, and `waiting` is a unary public function name in Aux. The function `name` must be used only as argument for the `waiting` function. Different function names used as arguments for the waiting function are interpreted as different elements. When the `waiting` rule is performed, the `waiting` function is updated to *true* and the `waiting` rule is completely performed when the internal computation rules update the `waiting` function to *false*.

Let Lab = i `:` `label` be a *labeled* rule, where i is a interaction rule and `label` is a nullary public function name in Aux. By induction, the update set $Updates(Lab, G)$ is $Updates(i, G) \cup (\{Update((\texttt{label}, \texttt{label} + 1), G)\} \uplus \emptyset)$. The update that increments the value of label has to be fired in the same move that i is completely performed.

3.4 Internal Computation Rules

The *internal computation* rules have the same syntax and the same meaning of pure ASM rules. The only restriction is that rules can mention only function names in $\mathcal{V}_{local}$, where $\mathcal{V}_{local}$ is a subset of $\mathcal{V}$ equals $Fun(u)^{+Self} \cup Aux_{public} + \{\texttt{Def}\} + \{\texttt{Self}\}$, where Aux_{public} is the subset of Aux which contains the public functions names.

4 A Case Study: The Program Committee Problem

Let us give a problem to be specified. This will be useful when comparing this approach with others and for illustrating its reasoning capability. Following the

A conference is announced, and an electronic submission form is publicized. Each author fetches the form and activates it. Each author fills an instance of the form with the required data and attaches a paper. The form checks that none of the required fields are left blank and sends the data and the paper to the program chair. The program chair collects the submission forms and assigns the submissions to the committee members, by instructing each submission form to generate a review form for each assigned member. Each assigned member is a reviewer, and may decide to review the paper directly or to send it to another reviewer. The review form keeps track of the chain of reviewers. Eventually, a review is filled and it finds its way back to the program chair. The program chair collects all review forms. The chair merges all review forms for each paper in a paper report form. Then the chair declares each report form an accepted paper report form, or a rejected paper review form, and finally returns this form to each author. All accepted paper report forms are required to generate final version forms on which the author attaches the final version of the paper and sends it back to the program chair.

Fig. 6. Problem Specification

traditional ASM philosophy we do not plan to introduce a logical calculus to prove properties about the specification. Instead, we make rigorous, mathematical reasoning but without an associated formal system.

The Figure 6 reproduces verbatim a simplified and modified version of the problem consisting in managing a virtual program committee meeting for a conference. The problem was presented as a challenge in [3].

Now we will give a partial Interactive ASM semantics for this problem. We will focus on the *interaction section* of the specification and will intentionally omit the declaration of the internal state and the the internal rules of each unit, for the sake of conciseness and to show how the relative independence of the interaction rules from the internal computation rules provides an adequate mechanism for modular reasoning.

In Figure 7, we present the unit definition *Author* which models the behavior an author must have in order to correctly participate of a call for papers. It has the following three parallel actions:

1. Whenever there is a paper ready to be submitted, the nullary function *wants_to_submit* (supposed to be initiated with `false`) will be set to `true` by an internal computation rule and the first action will be enabled. This action requires a connection with any instance derived from *Submission*, and then: *i*) performs an output of the paper and respective data, which are supposed to be consistently updated by the internal computation rules; *ii*) and performs an input from the *Submission* instance indicating if the paper was properly received.
2. Whenever an author gets a response from a *Submission* instance, it internally either decides to re-send the paper, or to stay waiting the result of the submitted, or even to send another paper. All these actions should be

```
unit Author
...
    interaction
        if (wants_to_submit) then
            connect s : Submission.a in
                data_paper -> s;;
                response <- s.response : responses
        if (waiting_result) then
            connect the_chair: Chair.author_result
                result <- the_chair.result
        if (ready_final) then
            connect the_chair: Chair.author_final
                final -> the_chair
    ...
```

Fig. 7. Unit Author

```
unit Submission
...
    interaction
        connect a: Author.s in
            data_paper <- a.data_paper;;
            waiting(checking_data);;
            if (data_ok) then
                response="ok" -> a
                connect the_chair: Chair.s in
                    data_paper -> the_chair;;
            else
                response="fail" -> a
...
```

Fig. 8. Unit Submission

specified by the internal rules. In the case of waiting the result of the submission, all we know is that the function `waiting_result` should be set to `true` by the internal computation rules. Whenever this occurs, the second action will be waiting a connection from the unit `the_chair`, which is a static instantiated one. Note that the function `the_chair` declared inside the unit *Author* is presumably initiated with the static instance `the_chair` declared in the main specification defined in the sequel. Whenever the connection is performed, the action will be waiting the result to be sent from `the_chair`.

3. The third action is similar to the previous one: whenever an author gets the result it decides internally to send the final version, and so on.

The unit definition *Submission* acts as an intermediary between an author and the chair. It models an agent that interacts with the author of a paper, getting all necessary information with an attached paper. The internal rules should make all the necessary checking, including the one that prevents papers being submitted after the deadline. An agent instantiated from *Submission* performs the following action:

1. It requests a connection with an author. Note that if there is not any author wanting a connection, nothing is done. Then *i*) it receives the paper

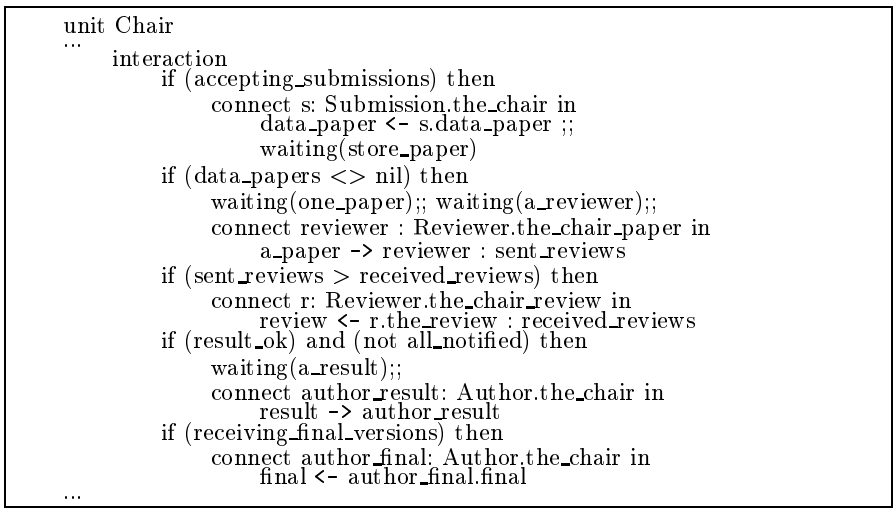

Fig. 9. Unit Chair

and respective data from the author, *ii*) waits until the data is checked, and *iii*) sends a response to the respective author. If the data is valid, the agent requests a connection with the chair. Then, *i*) it sends the paper and respective data to the chair.

The unit *Chair* has five parallel actions, each one represented by a guarded rule evaluated depending only on its internal state.

1. The first action is enabled by an internal function which is presumably initiated with `true`. It requires a connection with a *Submission* instance, possibly attempting a connection, and if there is such one, it receives a paper and the respective data, and then it waits until the internal computation rules store the received paper and its respective data into an internal function `data_papers` which represents a buffer that contains all papers that still need to be sent to reviewers.
2. The second action checks if the buffer for papers is not empty and then waits an internal selection of a pair paper/reviewer and dispatches the respective paper. Note that the channel that will receive the paper in the unit *Review* is `the_chair_paper`.
3. The third action action checks if still there are any papers that were not sent back by the reviewers. This checking is made easily, because we have labels counting the sent and received papers, respectively, `sent_reviews` and `received_reviews`. Note that the channel that will send the paper in the unit *Review* is `the_chair_review`.
4. The fourth action waits an internal processing of the results (represented by the guard `result_ok`) and will be enabled only if there is any author that was not notified (represented by the `not all_notified`).

5. Finally the last action connects to authors wanting to send the final version of the paper and receives the respective version.

The unit Reviewer has an elaborated scheme for creating the dynamic communication between reviewers. An agent instantiated from the unit Reviewer has basically two parallel interaction tasks:

1. It gets a paper from the chair and, either directly reviews the paper, or sends it to another reviewer together with a blank review form.
   Note that it is opened two connections to the same *Chair* instance. This is necessary because `the_chair` can be connected in the same step with two different reviewers: one that receives a paper for review and other that returns a reviewed paper. In order to solve this situation `the_chair` connects to different function names in unit *Reviewer*, namely `the_chair_paper` and `the_chair_review`. Whenever a paper is sent from a reviewer to another, it is managed a list of reviewers which forwarded the paper and the respective review. The static function `firsthistory` is internally defined to be a list with only two elements: `[self, the_chair]`. Note how the use of labels that makes the input/output matching possible.

2. The agent receives a paper from another reviewer and if the paper is already reviewed, the agent either passes the paper back to the chair, or passes it to the appropriate reviewer. If the paper is not reviewed, the agent either directly reviews the paper and passes it to the appropriate reviewer or just sends the paper to another reviewer. Note that when passing a paper forward, the agent adds its identification to the history of the reviewers for that paper. When passing a paper backward in the history, it removes a reviewer from the list being sent backward with the review.
   One of the new features in this action is the use of terms in the *connect* and *output* rules. Note that the function name `other` is crucially used by other `connect` rules when passing a paper forward or backward. After receiving a paper from another reviewer, the current one redirects the paper and respective review through three possible rules. Note that while redirecting the paper, the history is also being handled.

At last, in Figure 11, we specify the startup specification creating some initial unit instances. Other instances must have to be created dynamically by some *Authorization* unit, intentionally not specified. Now, we are going to present some propositions about the previous example and show their validity to illustrate the reasoning mechanism.

Consider the following assumptions about the initial state and the internal computation rules:

– (Author$_1$): The function `wants_to_submit` equals *true* whenever there is a paper ready to be submitted.

```
unit Reviewer
...
    interaction
        if (receiving_from_the_chair) then
            connect the_chair_paper: Chair.reviewer in
                a_paper <- the_chair_paper.a_paper endconnect ;;
            if (directly_review) then
                waiting(the_review);;
                connect the_chair_review: Chair.r in
                    the_review -> the_chair_review
            else
                waiting(a_reviewer1);;
                connect r=r1: Reviewer.other in
                    history=firsthistory -> r |
                    a_paper -> r |
                    review=blank -> r
        if (receiving_from_others) then
            connect other: Reviewer.r in
                history2 <- other.history |
                a_paper2 <- other.a_paper |
                review2 <- other.review endconnect ;;
            if (review2 <> blank) and
               head(history2) = the_chair_review) then
                    connect the_chair_review: Chair.r in
                        the_review=review2 -> the_chair
            elseif (review2 <> blank or directly_review) then
                if (directly_review) then
                    wait(the_review2) ;;
                connect r=head(history2) : Review.other in
                    history=tail(history2) -> head(history2) |
                    a_paper=a_paper2 -> head(history2) |
                    review=review2 -> head(history2)
            else
                waiting(a_reviewer2);;
                connect r=r2: Reviewer.other in
                    history=cons(Self, history2) -> r |
                    a_paper=a_paper2 -> r |
                    review=review2 -> r
...
```

Fig. 10. Unit Reviewer

```
main specification
    the_chair: Chair;
    committee_member1: Reviewer;
    ...
    committee_memberN: Reviewer;
    secretary1: Submission;
    ...
    secretaryN: Submission;
end specification
```

Fig. 11. Startup Specification

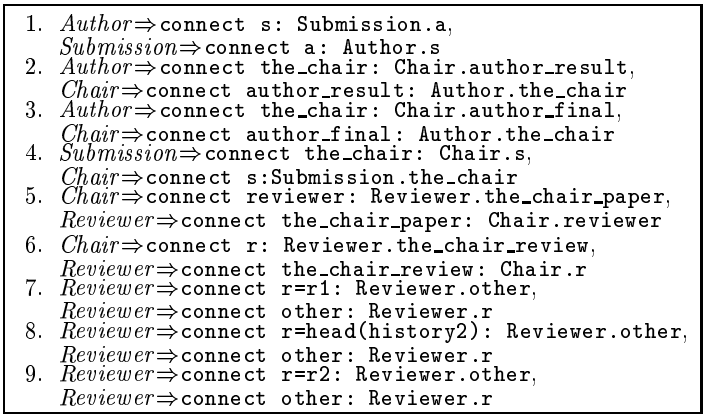

Fig. 12. Valid Connections

- (Author$_2$): Initial state: `the_chair(self) = the_chair` and `s(self) = x`, such that `x`$\in$ `Submission`.
- (Submission$_1$): Initial state: `the_chair(self) = the_chair` and `a(self) = undef`.
- (Chair$_1$): Initial state: `r(self) = undef`, `author_final(self) = undef`, `s(self) = undef`.
- (Chair$_2$): The internal computation rules update adequately the functions occurring in the guards, so as to correctly perform the conditional rules.
- (Chair$_3$): The internal computation rules update adequately: *i*) the function `a_paper` with a paper to be sent to `reviewer`, *ii*) `reviewer` with the selected reviewer of a paper and *iii*) the function `author_result` with the author which has to receive a result.
- (Chair$_4$): The same result is not sent back to the author more than once, by assuming the internal computation rules produce just one result per submission.
- (Reviewer$_1$): Initial state: `other(self) = undef`, `the_chair_paper(self) = the_chair`, `the_chair_review(self) = the_chair`.
- (Reviewer$_2$): The internal computation rules update adequately the functions occurring in the guards, so as to correctly perform the conditional rules.
- (Reviewer$_3$): The internal computation rules update adequately the functions `r1` and `r2` with other selected reviewers for a paper.

Proposition 1. *All connections in Figure 12 will be completely performed when necessary.*

Proof Sketch:

1. Let x be an instance of *Author*, and y be an instance of *Submission*. By Author$_1$, the `connect` rule of *Author* will be executed when necessary. The case for the `connect` rule in the first stage is case (b) because,

by Author_2, the function `s` is defined and, by Submission_1, no instance of *Submission* could have updated `_Connect`, since function `a` is undefined. The `connect` rule of y is always readily to be performed since it is a top-level rule. By Submission_1, the case for the `connect` rule of y for the first stage is (c) since the function `a` is undefined. So, on a first step x extends `_Connections` with `((`y`,"s", Submission, "a", false), (x, "a", Author, "s", true))`, and stays waiting for the awareness of y. On a second step, y fires the update set (c) of Figure 5, because there is above element insert by x is the required element. Finally, on a third step x is realesed and the two connection are aware and the inner interaction rule can be performed.

2. Let x be an author which is selected by `the_chair` to receive the result. In this connection, `the_chair` is defined in x, and `author_result` is defined in `the_chair`, by Author_2 and Chair_3, respectively. So, the cases for both connection rules are either cases (a) or (b) of the first stage. There are two possibilities:
   (a) Both instances update `_Connections` with rule (b) of Figure 5 in the same move. Note that in this case there will be two equal elements in `Connections`, but there is no problem since when disconnecting, each instance removes an element. On a second step, both instances will fire rule (a) to indicate that they are aware of the connection.
   (b) One instance updates `_Connections` in advance with rule (b). On a second step, the other instance enters the case (a) of the connect semantics and fires rule (a) of Figure 5. On a third step, the instance which was waiting completes the connection.
3. Let x be an instance of *Author*. Not that `the_chair` is the only instance of *Chair*. The case for the connection rule of x is case (b) because, by $\textit{Author}_2$, the function `the_chair` is defined and, by Chair_1, no instance of *Chair* could have updated `_Connect`, since function `author_final` is undefined. By Chair_2, the `connect` rule of y will be to be performed. By Chair_1, the case for the `connect` rule of `the_chair` for the first stage is (c) since the function `author_result` is undefined. The steps are the same as item 1.
4. Similar to item 3.
5. Similar to item 2.
6. Similar to item 3.
7. Similar to item 3. Note that `r` is defined and `other` is undefined.
8. Similar to item 7.
9. Similar to item 7.□

Proposition 2. *All papers received by the chair are sent to a reviewer.*

Proof Sketch: This is guaranteed by the second guard of the unit definition *Chair*, that dispatches any submission that has not been sent to any reviewer. Since this guard is executed in parallel with all the others, it will be executed without being blocked. By Chair_3, it will be selected a pair paper/reviewer. The connection with the appropriate reviewer in the channel `the_chair_paper` is guaranteed by Proposition 1.□

Proposition 3. *The chair waits for the devolution of all papers sent to reviewers.*

Proof Sketch: This can be easily checked with the counters for the number of sent papers and the number of received papers. Whenever `sent_reviews` be greater than `received_reviews`, `the_chair` will be waiting a connection with a reviewer.□

Proposition 4. *All reviewed papers certainly reaches back the chair.*

Proof Sketch: Let `the_chair`, $r_1, \ldots, r_n$ be the instances which have received a paper p and let r_n be the reviewer of p. When r_1 receives p from `the_chair`, it sends the history $[r_1$`, the_chair`$]$ to r_2. By induction on i, when r_i receives p the history is $[r_{i-1}, \ldots, r_1,$ `the_chair`$]$. If $i = n$ then r_n reviews p and sends the history $[r_{i-2}, \ldots, r_1,$ `the_chair`$]$ to r_{i-1}. Again, by induction on i, the review will reach `the_chair`. Since, by Proposition 1, all connections will be performed, then all input/output rules will also be performed to enable the flow of the review.□

Proposition 5. *All submissions received by the chair generate a report form to the author if, and only if, there is a reviewer that directly reviews the paper.*

Proof Sketch:

1. If, for each paper, there is a direct reviewer for it, then all submissions generates a report form. We have that all the papers received by the chair are sent to a reviewer by Proposition 2.
   If the paper is already reviewed there are two possibilities: *i*) The chair is the head of the history list (`the_chair_review` is assumed to be initiated with the static instance `the_chair`). Since the connection is guaranteed by Proposition 1 the review reaches the chair. *ii*) The chair is not the head of the history list. It is assumed that since a reviewer instance has the function `receiving_from_others` set to true, it will not change this function, and so the head of the history list will forever be enabled to be connected. Since the connection is guaranteed, the history list is guaranteed to be consumed until head evaluates to `the_chair`, because by definition first element of the history is `the_chair`.
   If the paper is not reviewed there are two possibilities: *i*) The reviewer directly reviews the paper and send it back through the history list. The review will certainly reaches the chair, and this is guaranteed in the same way of the previous item. *ii*) The reviewer does not review the paper. But, by assumption, there will be a reviewer that directly reviews the paper. So, by the previous item, the paper will certainly be sent back through its history when reviewed.
   Finally, assuming that the internal behavior of `the_chair` produces the final result when all the reviews of the paper had come back, then by the fourth guard of the unit Chair, we can guarantee that all results are sent back to the authors, since the connection will be eventually established.

2. All submissions would have back the result report, only if there is a reviewer that directly reviews each paper.
   Suppose if, for a certainly review, there is not a reviewer that directly reviews it. Then, that review will never be sent to the chair, and consequently, will not be sent to the author.
   Depending on the internal behavior of the chair when preparing the result, it may happen that none of the authors receives the review, but this situation has to be addressed elsewhere.□

There are many issues that have not been addressed in the above specification, such as, timing constraints and failures. But this is just a problem of refining the specification. In [9] we have modeled an unreliable channel using additional unit definitions.

5 Conclusions

The simple, yet powerful language presented is suitable for producing clear mobile ASM specifications. The idea of explicitly isolating the interaction between computing units with different purposes makes clear their interdependencies, and provides an independent mechanism to reason about the agent interactions. This can be seen in this paper since we have not specified the internal computation rules in our case study, and we still have reasoned about the specification, making independent assumptions about the internal rules.

The explicit interaction for connecting agents, even with a simple meaning, has several usages: binds internal names dynamically, provides dynamic communication topology useful for specifying a wide spectrum of concurrent programs, and resembles Web connections. The application migration is not modeled by a code movement, but only by an update on the connection.

We have shown the reasoning capability of the method for a simplified example, but the high modularization degree and the emphasis given to the interaction part make us believe that the method will scale up to bigger specifications. Since the proof has not an associated mechanized deduction, it is subject to errors, but the chances of correctness of this specification is much higher than totally informal ones.

There are still some aspects that must be further studied:

1. from the software engineering point of view, issues such as refinement and reutilization;
2. from the Web usage point of view, failure and security issues.

We expect the above item to be modeled without any major adaptation in the present approach.

References

1. R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

2. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
3. L. Cardelli. Abstractions for mobile computation. to appear as a chapter in Secure Internet Programming, available from http://www.luca.demon.co.uk, 1999.
4. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Verlag, 1998.
5. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. 7th International Conference on Concurrency Theory CONCUR 96*, pages 406–421, 1996.
6. P. Glavan and D. Rosenzweig. Communicating Evolving Algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer, 1993.
7. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
8. M. Maia and R. Bigonha. Interaction-based semantics for mobile objects. Technical Report LP 002/99, Universidade Federal de Minas Gerais, Programming Languages Laboratory, Brazil, 1999. submitted to III Brazilian Symposium on Programming Languages.
9. M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*, 1998.
10. W. May. Specifying Complex and Structured Systems with Evolving Algebras. In *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, pages 535–549. Springer, 1997.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.