Interaction Based Semantics for Mobile Objects

Marcelo de Almeida Maia
marcmaia@dcc.ufmg.br
Universidade Federal de Ouro Preto

Roberto da Silva Bigonha
bigonha@dcc.ufmg.br
Universidade Federal de Minas Gerais

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. The interaction specification part of a unit also provides usual constructions, such as sequential, parallel, and non-deterministic composition, used to describe the synchronization restrictions the units must preserve between themselves. We show how the proposed method provides a suitable way to reason about specifications.

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.

In order to address these issues, we have proposed new abstraction mechanisms[9] in the context of Abstract State Machines (ASM) formal method[7]. The methodology for producing ASM specifications provides a vertical abstraction mechanism, in the sense that the ground model (the model resulted from transcripting 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 address an explicit message passing mechanism 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 explicit message passing encoding is not considered too.

Instead of putting on the user the burden of providing the whole low-level specification of the communication topology and the message interchanging between different specifications, as it occurs with the pure ASM method, our approach provides special constructions to help the explicit specification of how different pieces of specification interact with each other. We will not try to defend, for the moment, the choice of this approach, since we hope the reader will be convinced when reading our case study.

In the next section, we set the context on which we will be interested. In Section 3 we review the ASM method. In Section 4 we review our extension to the original language of the method. In Section 5 we go into more details of the main body of our extension: the interaction specification. In Section 6 we illustrate the method by means of a case study. And finally, in Section 7 we conclude standing up for the suitability of the new constructions in the development of large scale concurrent mobile specifications.

2 Active Mobile Objects

The recent explosion of the Internet and the World-Wide-Web 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 will propose an abstract framework on which important features of mobile systems may be reasoned about. This framework eventually may be extended to support more features related to mobile systems. We hope to demonstrate the facility provided by the new specification framework.

Perhaps, the main work that has influenced the researches on the 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 latter adds 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, in a recent work[3], argues for a more explicit notion of movement across boundaries, and defines the ambient calculus[4].

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 interactions with the environment. The location issue is addressed by a 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 Abstract State Machines

Abstract state machines (ASM) [7] are transition systems whose states are first-order interpretations of functions symbols defined by a signature Υ over a non-empty set U called the *super-universe*. These states are also called static algebras. The transition relation is given by a finite set of transition rules describing the modifications of the interpretation of the function symbols from one state to another. This is the reason why ASM were formerly called Evolving Algebras. Before introducing the transition rules, let us define the auxiliary notions: locations, updates, update set. A *location* l of a state S is a pair $(f, \bar{x})$, where f is a function symbol, $\bar{x} \in U^n$ and n is the arity of f. An update α over the state S is a pair (l, t), where l is a location and t is a term in the sense of first-order logic. If $v \in U$ is the value for interpreting the term t on S, then firing $\alpha = ((f, \bar{x}), t)$ at state S transforms S into S' such that the result of interpreting $(f, \bar{x})$ is v and all other locations are not affected. An update set $Updates(R, S)$ is a set of updates over the state S, collected from the transition rule R. The update set is consistent if it does not contain any two updates α, α' such that $\alpha = (l, x)$ and $\alpha' = (l, y)$ and $x \neq y$. Otherwise, the update set is inconsistent. To fire an update set over a state S means to fire *simultaneously* all its updates and produce the corresponding state S'. Firing an inconsistent update set means to do nothing, i.e., means to produce a state $S' = S$.

The transition relation of ASM is defined by the following transition rules:

R ::= $f(t_1, ..., t_n)$:= t
| $R_1 ... R_k$
| `if` e `then` R_1 `else` R_2 `endif`
| `extend` U `with` v R_1 `endextend`
| `choose` v `in` U `satisfying` e R `endchoose`
| `var` v `ranges over` U R_1 `endvar`

where e is a boolean term, and v is a variable.

The first three kind of rules are called basic rules, respectively, the *update* rule, the *block* rule and the *conditional* rule. Their semantics are given by means of an update set $Updates(R, S)$, i.e., to fire R over a state S fire $Updates(R, S)$. This update set is inductively defined on the structure of R:

1. if $R \equiv f(t_1, ..., t_n) := t$ then $Updates(R, S) = \{(l, S(t))\}$, where $l = (f, (S(t_1), ..., S(t_n)))$, and $S(t)$ is the result of interpreting t on S;
2. if $R \equiv R_1 ... R_k$ then $Updates(R, S) = \cup_{i=1}^{k} Updates(R_i, S)$;
3. if $R \equiv$ `if` e `then` R_1 `else` R_2 `endif`, then $Updates(R, S)$ is defined as:

$$\begin{cases} Updates(R_1, S) & \text{if } S(e) \text{ holds} \\ Updates(R_2, S) & \text{otherwise.} \end{cases}$$

The last three kind of rules introduce variables, respectively, the `extend` rule which produces new fresh elements that are added to the extended universe, the `choose` rule which performs non-deterministic choices, and the `var` rule, which allows a simple form of synchronous parallelism. Note that `choose` and `var` resemble, respectively, the existential and universal quantifiers from first-order logic. Variables must be bound to some value of a universe belonging to the super-universe. So, the definition of the update set is extended with an environment ρ which binds the variables to values, and a choice function ξ which determines the variable bindings for `extend` and `choose` rules. The function ξ maps the bound variables to elements of a special universe called *Reserve*, which is used to produce new elements. The update set $Updates(R, S, \rho, \xi)$ can be defined inductively as well.

Given an initial state S_0, a *run* is a sequence of states S_0, S_1,... such that the state S_{i+1} is obtained as the result of firing the transition relation (represented by the update set of all rules) at S_i.

Finally, let us define a *multi-agent ASM,* which contains several computational *agents,* which execute concurrently a number of programs (called modules). A distributed ASM consists of a finite indexed set of programs π_v (modules) and finitely many agents a such that, for some module name v, $Mod(a) = v$, where the function name *Mod* represents the relation between modules names and agents. Each module has a corresponding enumerating universe of agents, which can be extended or contracted as necessary. There is also a nullary function name `self` that allows the self-identification of agents: `self` is interpreted as a by each agent a.

An agent a makes a *move* from a given state S if the corresponding update set of a is fired at S resulting a new state S'. Thus, a move can be represented as a pair (S, S'). Building upon this basic concept of move, a partially ordered notion of *run* for distributed ASM can be defined as a triple (M, A, σ), where:

1. M is a poset (partially ordered set) where its elements are agent moves.
2. A is a function that, given a move from M, returns the agent performing that move. It is used to impose that the set $\{m : A(m) = a\}$ is linearly ordered.
3. σ is a function that given an initial segment of M (possibly empty) assigns to it the corresponding state S. An initial segment of a poset P is a substructure X of P such that if $x \in X$ and $y < x$ in P then $y \in X$. Since X is a substructure, $y < x$ in X if and only if $y < x$ in P whenever $x, y \in X$.
4. The coherence condition: If x is a maximal element in a finite initial segment X of M and $Y = X - x$, then $A(x)$ is an agent in $\sigma(Y)$, and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $A(x)$ at $\sigma(Y)$.

4 Interacting Abstract State Machines

Now we present our proposed extensions to the method ASM. For the sake of understandability, we will present the new language in an informal way. The interested reader may find the formal translation of the Interactive Abstract State Machines (IASM) language to the pure ASM notation in [8].


```
i ::= function_name -> u_name
    | term:label -> u_name
    | buffered_var <-- u_name.pub_name
    | function_name <- u_name.pub_name
    | connect unit : U.s in i endconnect
    | new unit : U
    | destroy unit : U
    | i | i  |  i ;; i  |  i : l
    | waiting(name)
    | if guard then  i endif
```

Figure 1: Interaction Rule

A specification is defined as a set of unit definitions and unit instances. The intention of describing a system as a set of units is to encapsulate smaller pieces of specification. Unit definitions and unit instances are based on the concept of pure ASM modules and agents, respectively. Each unit definition corresponds to an ASM module, and each unit instance is an agent derived from the ASM module corresponding to the respective unit definition.

A unit definition has several sections:

1. *Internal State*
   The internal state of a unit is represented by the interpretation of the private r-ary function names. The state can be altered only by local unit rules and explicit interactions, which will be described soon. The only way the internal state of a unit can be transmitted to another unit is by an explicit interaction between them. The purpose is to achieve a high degree of information hiding and a rigid notion of distributed behavior.
2. *Internal Rules*
   The internal rules of a unit definition have the same syntax as those of the ASM model. They can refer only to the internal state, i.e., references to the state of other units are forbidden. A section designated as `rules` defines all the internal rules of a unit.
3. *Interaction Rules*
   The interaction section establishes how communication between units can occur. It does not only define the information interchanging but also the synchronization restrictions imposed between the units.

5 Unit Interactions

In this section, we will present the several rules used to specify the interaction pattern of a unit. In Figure 1 we show the possible interaction rules.

5.1 Input and Output Rules

The basic operators "->" and "<-" are used, respectively, to send a value to a unit (output) and to receive a value from a unit into a variable (input). The semantics is

```
unit sender
...
    interaction
        msgsend:channel -> R
...
unit receiver
...
    interaction
        msgrec <- S.channel
...
```

Figure 2: Example of Input and Output Interaction

similar to the usual asynchronous message exchanging model, such that, the sending of a value is non-blocking, while the reception is blocking. In Figure 2 we show the input/output interaction between two units. The output rule is defined by a term which denotes the value to be sent (`msgsend`), a label which specifies the output channel (`channel`), and a target unit (`R`) which will receive the corresponding value. When the term being sent is a function representing a location, the label may be omitted, and it will be assumed equals to the function name. The input rule is defined by a location which will receive the value (`msgrec`) and a identification of the source of that value. This identification is expressed by the name of the source unit (`S`) and the same label of the corresponding output rule (`channel`). The specification must obey the condition that there can not be generated two or more different outputs for one corresponding input in the same step. If this case eventually occurs, then it should be used a buffered input operator "<--".

5.2 Unit Instantiation

The instantiation of a unit causes the creation of an agent which executes the respective unit definition. There are two ways of instantiating units:

1. statically: a specification is defined by a set of unit definitions and some initial unit instances. The declaration of a unit in the startup definition actually creates statically a unit instance that will live the whole time the specification is being executed.

2. dynamically: the declaration of a dynamic unit as part of the internal state of a unit definition does not create any instance. Instead, the declaration defines a function that will be updated with a unit instance, either a statically or a dynamically created one. A dynamic unit may be created and destroyed with the interaction rules `new` and `destroy`, respectively.

5.3 Unit Connections

As stated before, a unit declaration inside a unit definition only defines a function. Our intention is that this function should be updated with a unit instance, whose definition contains an internally declared function which is also expected to be updated with the former unit instance. This situation indicates an agreement between the two instances, and it is performed with the rule `connect` $u : U.s$ `in` i `endconnect`. Consider two unit instances u_1 and u_2. In order to establish a connection between these two units, inside each unit there must have a `connect` rule, such

that, when both rules are executed, they update the proper internal functions in u_1 and u_2, with u_2 and u_1, respectively, thus completing the connection.

The arguments for this rule are: 1) a function name u corresponding to a unit instance, 2) the name U corresponding to the unit definiton from which the instance u was derived, and 3) a function name s declared inside U that we expect to be bound to the current unit instance.

The `connect` rules may be used in several circumstances:

- the current interaction rule will be waiting for a non-identified unit instance asking for connection, i.e., the function u is undefined. If the `connect` rule omits the function s, it is expected to exist just one possible connection in U. If the `connect` rule omits the name of the other unit definition, then any unit instance may perform the connection, otherwise only unit instances derived from the proper unit definition may perform the connection. Inside the `connect` rule, the function u denotes the connected instance.
- the `connect` rule specifies exactly which unit instance has to be connected. In this case the outer interaction rule will be waiting until that unit instance agrees with the connection. Just like the previous case, if the `connect` rule omits the function s, it is expected to exist just one possible connection in U.

It is necessary a successful connection in order to execute the interaction inside the `connect` rule.

5.4 Interaction Composition and Runs

The notion of run for interactive ASMs is the same as that of pure ASMs, i.e., a sequence of states, where each state is given by the union of all instance states. But, compared with the pure ASMs, the interaction portion of the specification has a some different rules, such as, the *connect* and *input* which may be not executed in just one step of the run. In order to fit the semantics of these intructions, as much as possible, in the context of the pure ASM we can view these rules as conditional rules. Suppose we need to evaluate the update set of a unit instance in order to proceed with the step, and we have a *connect* or an *input* rule. Before performing these rules we first check if it is possible to complete them. If not, then in the next step they will be checked again.

Definition 1 *Parallelism: The interaction section is a block of interaction rules* i_1 | ... | i_n, *where each of them is enabled to execute just like a block of pure ASM rules. The operator* | *may be omitted.*

Definition 2 *Sequence: If there is an enabled sequence interaction with the following form:* $(i_1;; i_2;; \cdots ;; i_n)$ *then one and only one interaction* i_k $(1 \leq k \leq n)$ *is enabled at each time and all the sequence is executed in the cycle. Note that we use the symbol ";;" to avoid confusion with the symbol ";" already used in pure ASM.*

Definition 3 *Blocking connection: Suppose there is an enabled interaction with the following form: (*`connect` δ ;; i*), where* δ *is the body of the* `connect` *rule and* i *is an interaction rule. Then, the respective* `connect` *blocks* i*, until the connection effectively occurs.*

Definition 4 *Blocking input: Suppose there is an enabled interaction with the following form:* (*a* <- *u.b* ;; *i*), *where a is a function name, and u.b is an incoming value from the function name b of the unit u. Then, the respective input blocks i, until the input effectively occurs.*

5.5 Synchronization of Internal Rules and Interactions

Unit internal rules are just like ASM rules and their semantics is exactly the same. The interaction and internal rules are *fired* simultaneously by the same agent daemon. In order to guarantee appropriated synchronization when executing these rules, the IASM method provides:

1. A `waiting` rule used in the interaction section. This rule defines a boolean function `waiting`. The rule, whenever executed, updates the function with `true` and freezes the execution of the current node in a interaction cycle until the function is updated with `false`.

2. All interactions may be labeled, for example:

   msgrec <- S.msgsend : nb_rcvd_msgs

   The corresponding label denotes an integer value which corresponds to how many times an interaction has been completed. This value can be used by the internal rules.

6 Reasoning Capability

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 pure 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 3 reproduces verbatim a simplified and eventually 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. For the sake of conciseness, we will focus on the *interaction section* of the specification and will *omit* the declaration of the internal state and the the internal rules of each unit. We will also omit the delimiters `endif`, `endconnection` in favour of an indented writing style.

In Figure 4, 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 location *wants_to_submit* (supposed to be initiated with `false`) will be set to `true` 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

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.

Figure 3: Problem Specification

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

Figure 4: Unit Author

data, which are supposed to be consistently updated by the internal 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 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`. 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.

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

Figure 5: Unit Submission

```
unit Chair
...
    interaction
        if (accepting_submissions) then
            connect s: Submission in
                data_papers <-- s.data_paper
        if (data_papers <> nil) then
            waiting(one_paper);; waiting(a_reviewer);;
            connect reviewer : Reviewer.the_chair_paper in
                a_paper -> reviewer : sent_review
        if (sent_review > received_review) then
            connect r: Reviewer.the_chair_review in
                review <- r : received_review
        if (result_ok) and (not all_notified) then
            waiting(a_result);;
            connect author_result in
                result -> author_result
        if (receiving_final_versions) then
            connect author_final in
                final <- author_final.final
...
```

Figure 6: Unit Chair

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, the sequence remains blocked until so. Then *i*) it receives the paper 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 ok, the agent requests a connection with the chair, and 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 into a buffered function.


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

Figure 7: Unit Reviewer

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 *Review* unit is `the_chair_paper`.
3. The third action action checks if still there are papers that were not sent back by the reviewers. This checking is made easily, because we have labels counting the sent and received papers, `sent_review` and `received_review`, respectively. Note that the channel that will send the paper in the *Review* unit 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 the authors that wants 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 the reviewers. An agent instantiated from the unit Reviewer has basically two 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.

There are some new features in this action. *i*) it is opened two connections to the same unit 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 function `firsthistory` is internally defined to be a list with only one element: `the_chair`. Note 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, we may 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 a proposition about the previous example and show its validity to illustrate the reasoning mechanism.

Proposition 1 *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:

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. 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 parallel with all the others, it will be executed without being blocked. We suppose that the internal rules are correct, so it will be selected a pair paper/reviewer. The connection with the appropriate reviewer in the channel `the_chair_paper` is guaranteed by assuming that the internal rules of the unit Review properly sets `receiving_from_the_chair` to `true`.

   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`). The connection to the chair is guaranteed because


```
specification
    the_chair: unit Chair;
    committee_member1: unit Reviewer;
    ...
    committee_memberN: unit Reviewer;
end specification
```

Figure 8: Startup Specification

there is an independent guarded rule in the unit `Chair` that will be enabled whenever there is a paper that has not came back. Other important, and easily provable, lemma to guarantee the enabling of this rule is that the same paper is not sent back to the author more than one time. Since the connection is guaranteed 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 previou 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, if the connection is successfully established. Assuming the guard `waiting_result` in the unit Author will be enabled, the connection between the author and the chair will successfully occur.

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.□

7 Conclusions

The language presented is suitable for producing clear mobile ASM specifications. The idea of explicitly isolating the interaction between computing units with different purposes make clearer their interdependencies, and provides a independent mechanism for reasoning about the agent interactions.

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 for this specification is much higher than for totally informal ones.

There are still some aspects that must be further studied:

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

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. Position Paper, http://www.luca.demon.co.uk/Papers.html, May 1998.

[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. The Formal Specification of the Interactive Abstract State Machine Language. Technical Report 005/98, Universidade Federal de Minas Gerais, Brazil, 1998. http://www.dcc.ufmg.br/~marcmaia/iasmformal.ps.gz.

[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.