Electronic Notes in Theoretical Computer Science 71 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume71.html 17 pages

A Conference Reviewing System in Mobile Maude ⋆

Francisco Durán [a] and Alberto Verdejo [b]

[a] *ETSI Informática, Universidad de Málaga, Spain.* duran@lcc.uma.es

[b] *Facultad de Informática, Universidad Complutense, Madrid, Spain.* alberto@sip.ucm.es

Abstract

A useful way of presenting a new language is by means of complete examples that show the language features *in action*. In this paper we do so for the Mobile Maude language, an extension of Maude that supports mobile computation. We implement an ambitious wide area application, namely a conference reviewing system, an example described by Cardelli as a challenge for any wide area language to demonstrate its usability.

1 Introduction

The popularity of the Internet has brought much attention to the world of distributed applications development. Now, more than ever, the network is being viewed as a platform for the development of cost-effective, mission-critical applications. Mobile code and mobile agents [12,11] are emerging technologies that promise to make it very much easier to design, implement, and maintain distributed systems. Mobile agents may reduce the network traffic, provide an effective means of overcoming network latency, and, perhaps more important, help us to construct more robust and fault-tolerant systems, thanks to their ability to operate asynchronously and autonomously of the process that created them.

While distributed applications represent a great potential future, the task of writing distributed applications which run over the Internet is flooded with problems of scalability, reliability, and security. The scale of applications now being considered for network environments will require higher levels of security and reliability. In order for agent systems to present a realistic development alternative for such applications, these systems must evolve to provide the

⋆ Research supported by projects TIC2000-0701-C02-01 and TIC2001-2705-C03-02.

highly secure and reliable environments needed. Intrinsically concurrent formalisms with a precise semantics seem to be unavoidable for such a task, and declarative approaches seem quite promising.

The flexibility of rewriting logic for representing very different styles of communication, either synchronous or asynchronous, its facility for supporting distributed, concurrent object-oriented systems, and its reflective capabilities for supporting metaprogramming and dynamic reconfiguration, make it a very suitable formalism for the specification of distributed systems based on mobile agents, on which the proof of properties about security, correctness, and performance, can be based.

Recently, an extension of the multiparadigm, declarative language Maude [5] has been proposed with mechanisms for supporting mobile computation. It is known as Mobile Maude [7], and its main features, that make it a language appropriate for the specification and prototyping of distributed systems based on mobile agents, are: (1) it is based on rewriting logic, a simple logic which is easy to use for the specification of concurrent, distributed, and open systems, including aspects as data, events, system evolution, security and real time; (2) it is object-oriented, supporting distributed objects and their communication, either synchronous or asynchronous, with a formal, precise semantics; (3) it is reflective, with formal support for metaprogramming at any level of reflection; (4) it is wide spectrum, allowing transformations and refinements of specifications; (5) it is modular, with an advanced module system; (6) it is mobile, supporting the movement of data, states, and programs; and (7) it has a formal basis for the development of security models, and the verification of properties for such models.

The aim of this work is to contribute to the understanding of Mobile Maude, illustrating the main features of the language in the specification of an application of certain complexity, namely, the reviewing system for a conference, going from its announcement to the edition of the proceedings. Such example was proposed by Cardelli in [2] as a challenge for any wide area language to demonstrate its usability, although it was previously used by different authors.

Mobile Maude is described in Section 2. Section 3 includes a description of the conference reviewing system, and a detailed explanation of our Mobile Maude implementation. Section 4 discusses some related work. In Section 5, we present some conclusions and future work.

2 Mobile Maude

Mobile Maude was first described in [7]. We present here a brief discussion on the language, showing its key notions and the primitives used in the following sections.

Mobile Maude is a mobile agent language that extends the rewriting logic language Maude for supporting mobile computation. In its design, a systematic use of reflection is made, obtaining a simple and general declarative mobile agent language. Mobile Maude has been formally specified by means of a rewrite theory. Since this specification is executable, it can be used as a prototype of the language, in which mobile agent systems can be simulated.

Mobile Maude is based on two key notions: *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects can move between different processes, can communicate asynchronously with each other by means of messages, and can evolve. In Mobile Maude, a mobile object can communicate with objects in the same process or in different processes—some mobile agent languages forbid this latter kind of communication, allowing only communication within a process.

Mobile objects travel with their own data and code. The code of a mobile object is given by (the metarepresentation of) an object-oriented module—a rewrite theory—and its data is given by a configuration of objects and messages that represent its state. Such configuration is a valid term in the code module, which is used to execute it.

Processes and mobile objects are modeled as distributed objects in classes `P` and `MO`, respectively. The names of processes range over the sort `Pid`, and the declaration of the class `P` of processes is as follows.

```
class P | cf : Configuration, cnt : Int, guests : Set(Mid),
          forward : PFun(Int, Tuple(Pid, Int)) .
```

The `cf` attribute represents an *inner* configuration, where mobile objects and messages can reside. Thus, a mobile agents system will consist of a configuration of processes (and messages in transition between processes), with a configuration of mobile objects inside each process. The `guests` attribute keeps the names of the mobile objects currently in the process's internal configuration. The `cnt` attribute is a counter to generate new mobile object names. The names of mobile objects range over the sort `Mid`, and they have the form `o(PI, N)`, where `PI` is the name of the object's home or parent process, that is, the process where it was created, and `N` is a number that distinguishes the children of `PI`. This number is provided by the `cnt` attribute, which gets increased after the creation of a new mobile object. The dispatching of messages is nontrivial, since mobile objects can move from one process to another. To solve this problem each process keeps forwarding information about the whereabouts of its children in its `forward` attribute, a partial function that maps a child number to a pair consisting of the name of the process where the object currently is, and the number of "hops" that the object has taken to reach it. The number of hops is used to know the age of the information kept in the parent. Each time a mobile object moves to a different process it sends a message to its home process announcing its new location.

The class of mobile objects is declared as follows:

```
class MO | mod : Module, s : Term, p : Pid, hops : Int, mode : Mode .
```

The `s` attribute keeps (the metarepresentation of) the mobile object's state, and has to be of the form `C & C'`, where `C` is a configuration of objects and messages—unprocessed incoming messages and inter-inner-objects messages—and `C'` is a multiset of messages—the *outgoing* messages tray. One of the objects in the configuration of objects and messages is supposed to have the same identifier as the mobile object it is in. We refer to this object as the *main* one. The `mod` attribute is the metarepresentation of the Maude module that describes the behavior of the object. The `p` attribute keeps the name of the process where the object currently resides. The number of hops from one process to another performed by the object is stored in the `hops` attribute. Finally, the object's `mode` can be `active` if the object is inside a process, and `idle` if it is moving.

The semantics of Mobile Maude is specified by an object-oriented rewrite theory containing the definitions of the above classes and rewrite rules that describe the behavior of the different primitives: object mobility, message passing, and object and process creation. This specification is the *system code* of Mobile Maude, which works as a prototype on which to execute Mobile Maude applications. Such applications need of course to satisfy certain requirements, as being object-oriented, using the `_&_` constructor for sending messages out of the mobile objects, and using the primitive messages for moving to other processes.

We will present now the Mobile Maude primitives used in the application.

There are three kinds of communication between objects. Objects inside the same mobile object can communicate with each other by means of messages with any format, and the communication may be synchronous or asynchronous. Objects in different mobile objects may communicate when such mobile objects are in the same process and when they are in different processes; in these cases, the actual kind of communication is transparent to the mobile objects, but such communication must be asynchronous, and messages must be of the form `to_:_`, where the first argument is the identifier of the addressee object, and the second argument is the message contents, a value of sort `Contents` built with free user-defined syntax (see, for example, Section 3.2). That is, the minimum information needed to dispatch a message is the receiver's identity; if the sender wants to communicate its identifier, it has to include it in the message contents. If the addressee is an object in a different mobile object, then the message must be put by the sender object in the second component of its state (the outgoing messages tray). The system code will send the message to the addressee object.

When a mobile object wants to move to another process it puts in its outgoing messages tray a `go(PI)` message, where `PI` is the target process identifier. When a mobile object has an outgoing `go` message, a new *inter-mobile-objects* `go` message is sent, with the mobile object as one of its arguments, after removing the outgoing message and setting the state to `idle`. This ensures that the mobile object is inactive while on the move.

If the message's sender and addressee are in different processes, then this message has to travel to the desired process, going out to the outer configuration of processes. When the message reaches the destination process, the mobile object is put into it (in `active` mode), and the parent process is informed, so it can update its forwarding information, or just updated, if it is the parent process itself.

In the `go` message, the mobile object indicates the process where it wants to go. Sometimes, a mobile object wants to reach another object, but it only knows the identifier of the object it wants to catch up, not the process it is in. In this case, the `go-find` message can be used. When this message is used by an object `M`, it takes as arguments the identifier of the mobile object that `M` wants to reach, and the identifier of a tentative process where it may be.

When an object wants to create a new mobile object, it sends a `newo` message to the system (by putting it in the second component of its state). The `newo` message takes as arguments (the metarepresentation of) a module M, a configuration C (which will be the initial configuration to be put in the belly of the mobile object to be created, and which is a valid term in the module M), and the provisional identifier of the main object in the configuration C. The first action accomplished by the system when it detects the `newo` message is to create a new mobile object with the configuration C as its state and the module M as its code, and then sends a `start-up` message to the main object with its new name, so it coincides with the name of the mobile object it is in.

The code describing the behavior of mobile objects is called *application code*, and our main purpose in this work is to illustrate how such construction can be used when developing our applications, and how a *normal* object-oriented specification may be easily made mobile.

The system code for Mobile Maude, plus some related information, can be found in `http://maude.csl.sri.com/mobile-maude`.

3 Conference Reviewing System

The ambitious wide area application we discuss in this section has been used by different authors [3,13], although we follow the presentation by Luca Cardelli in [2], where he describes it as *a challenge for any wide area language to demonstrate its usability.*

The problem consists in managing a virtual program committee meeting for a conference. Figure 1 shows (nearly all) the different steps in the conference reviewing process; the numbers in the following description correspond to the numbers in this figure. A conference is first announced (1), and an electronic submission form is publicized. If interested in the submission, an author fetches a submission form (2), which, after its activation (3, 4, 5), guides most of the reviewing process. Each author fills its instance of the form and attaches a paper (6). Once the submission form is completed, it finds its way to the program chair (7), who collects the submissions (8) and assigns them to committee members, by instructing each submission form to generate a

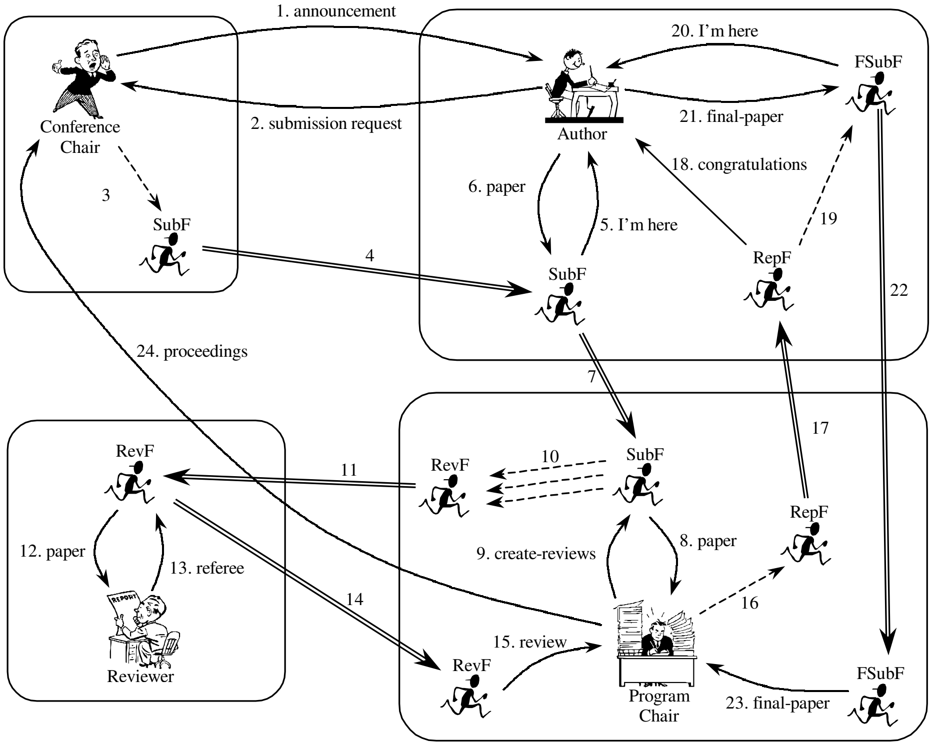

Fig. 1. Conference reviewing process.

review form for each assigned member (9, 10). Each review form then finds its way to the committee member it has been assigned to (11, 12), who may decide to review the paper directly, to refuse it, or to delegate it to another reviewer. The review forms keep track of the reviewers they have passed by, so that they can find their way back when completed or refused, allowing each reviewer to check the work of its subreviewers. Eventually, a review is filled (13). Then, the form performs various consistency checks, such as verifying that the assigned scores are in range and that no required fields are left blank, and it finds its way back to the program chair (14). Once the review forms reach the program chair, this one accumulates the scores and the reviews for each paper (15). Once the program chair has collected all the reviews for a paper, this one is resolved. If the reviews are in agreement, the program chair declares the form an accepted or rejected paper report form (16). If the reports are in disagreement, the program chair declares the form an unresolved review form, which circulates between the reviewers and the program chair, accumulating further comments, until the program chair declares the paper accepted or rejected. The program chair creates a new report form with the result of the revision for such a paper, which then finds its way back to the author (17), with appropriate congratulations (18) or regrets. Once it reaches the author, it informs the author about the result of the reviewing process. An accepted paper report form spawns then a final submission form (19, 20), where the author attaches the final version of the paper (21). The completed final submission form finds its way back to the program chair (22, 23), who merges them into the proceedings, and lets them find their way to the conference chair (24).

From the previous description, we can identify different actors, which may move freely from one process to another, and therefore they should be represented as mobile objects. In the Mobile Maude approach the specification of the system consists of objects embedded inside mobile objects, which communicate to each other via messages. In addition to the term representing its state, each mobile object carries the *code* managing the behavior of the configuration of objects and messages representing such a state. The main difference with respect to the specification of systems in Maude is that these objects must be aware of the fact that they are inside mobile objects, and that in order to communicate with (objects in) other mobile objects or to use some of the system messages available, they must follow the appropriate procedure.

In the description of the system above there are also real people, and it is assumed some kind of interaction between the mobile agents and these people. Agents ask people diverse information, like a name, a paper, or a review report. In our representation of the system these people are also represented by mobile agents that have all the required information.

In order to simplify, we have minimized the information contained in messages, and we are not encrypting them. However, it is assumed that all messages are encrypted. Such an encryption could be easily added by using encrypt and decrypt functions as in [6].

In the following we present a sketch of the application code for the different mobile objects that appear in the system. Although Mobile Maude requires that each mobile object carries the module with the specification of the classes and the behavior of the objects in its belly, because of presentation purposes, we prefer to give here fragments of each of the modules following the natural sequence in which the process happens. The complete specification, together with execution examples, can be found in [8].

3.1 The Classes of the System

From the description above, we identify the following actors: authors, reviewers, conference chairs, program chairs, program committee members, and different types of forms, namely, submission forms, review forms, report forms, and final submission forms.

Authors have attributes to keep their personal data, their papers, and the identifiers of the submission forms they interact with.

```
class Author | name : Name, paper : Default(Paper),
               subm-form : Default(Mid) .
```

The attributes `paper` and `subm-form` are declared of sorts `Default(Paper)` and `Default(Mid)`, respectively, in such a way that they may have a default value `null`. Thus, an author may or may not have a paper—in fact, an author could have a set of papers but here we assume that an author is submitting at most one paper—and an author has the identifier of a submission form only while it is interacting with one.

A conference chair object has a mailing list, that is, a set of author identifiers, the identifier of the program chair of the conference, and the final version of the proceedings.

```
class ConfChair | mailing-list : Set(Mid), progr-chair : Mid,
                  proceedings : Set(Paper) .
```

In addition to the identifiers of the conference chair and the program committee members, a program chair object needs to keep information about the reviewing process, and a way to collect the final papers to compose the proceedings. The reviewing information consists of a set with the identifiers of the authors whose papers have been accepted but whose final papers have not been received yet and information regarding each of the submissions. This information is given by a term of sort `SubmInfo`, which is an *alias* for the sort `Tuple(Name,Mid,Paper,Set(Tuple(Mid,Score)),Int)`. That is, for each submission it is stored the name, identifier and paper of the (contact) author, the results from the referee reports and the number of rounds. The information on the refereeing of each paper is stored in the review forms, the program chair only keeps the identifier of such a review form and the score provided by it for the given paper. In case of disagreement, the program chair declares the paper unresolved and sends the review forms back to the reviewers. After three rounds without agreement, the program chair will take a decision. The final versions of the accepted papers are kept in the `proceedings` attribute.

```
class ProgrChair | conf-chair : Mid,      pc-members : Set(Mid),
                   reviewers : Int,       subms : Set(SubmInfo),
                   accepted : Set(Mid),   proceedings : Set(Paper) .
```

A reviewer can give three different answers to a review message: it can delegate in some other reviewer, it can refuse the review, or it can do it and send back the result. Thus, a reviewer has attributes for keeping the set of its subreviewers and the set of the papers it has already seen before, whatever it decided to review, delegate, or refuse them.

```
class Reviewer | subreviewers : Set(Mid), done : Set(Paper) .
```

Program committee members are particular cases of reviewers, they do not need any additional attributes. We may model this relationship by declaring the class `PCMember` a subclass of `Reviewer` as follows.

```
class PCMember .
subclass PCMember < Reviewer .
```

The submission process is guided by submission form objects. In addition to the identifiers of the conference chair, the program chair, and the author it is assigned to, a submission form will carry the author's information and submitted paper. The submission form needs to be aware of its state, given by a constant of sort `SubmFormState`, which may take values `inactive`, `active`, `towards-author`, `towards-pc`, and `finishing`.

```
class SubmForm | conf-chair : Mid,       author : Mid,
                 progr-chair : Mid,      author-name : Default(Name),
                 paper : Default(Paper), state : SubmFormState .
```

A reviewer object may decide to review a paper assigned to it directly, or to send it to another reviewer. The review form must keep track of the chain of reviewers so that it can find its way back when either completed or refused, and so that each reviewer can check the work of its subreviewers. In case there is no agreement on the scores for a paper, a review form must go back to its reviewer. Objects of class `ReviewForm` have two attributes of sort `List(Mid)`. The attribute `chain` contains initially the identifier of the program committee member assigned to it, and all the successive subreviewers are added to this list. When the paper is finally reviewed, the form finds its way back by extracting the identifiers from the list in a *last-in-first-out* manner. Nevertheless, instead of discarding these identifiers, they are included in the list in the attribute `chain-back`, so that it can follow its reviewing path again if necessary. As submission forms, review forms need to distinguish the different states in which they can be. They do so with an attribute `state` of sort `ReviewFormState` which may take values `inactive`, `towards-reviewer`, `back`, `unresolved-towards-reviewer`, and `unresolved-back`.

```
class ReviewForm | paper : Paper,           progr-chair : Mid,
                   refused : Set(Mid),      chain : List(Mid),
                   chain-back : List(Mid),  score : Default(Score),
                   state : ReviewFormState .
```

When a submission has been resolved, a report form is sent to the corresponding author object to inform it about the result of the refereeing process on its paper. The sort `ReportFormState` may take values `inactive` and `towards-author`.

```
class ReportForm | paper : Paper, author : Mid, score : Score,
                   progr-chair : Mid, state : ReportFormState .
```

Finally, when the author receives an acceptance report form, a final submission form is created, which fetches a final paper and finds its way to the program chair. The sort `FinalSubmFormState` may take values `inactive`, `towards-pc`, and `finished`.

```
class FinalSubmForm | author : Mid,  state : FinalSubmFormState,
                      paper : Paper, progr-chair : Mid .
```

3.2 The Submission Process

All the process is initiated by the conference chair by broadcasting the announcement message to all the potential authors in its mailing list. The `review-process-starts` rule below is fired when a `ConfChair` object receives a `start` message. The broadcasting is accomplished by generating messages with contents `announcement from : O`, where `O` is the conference chair identifier, and with addressees each of the objects in the `mailing-list` set. The operation `broadcast` is in charge of generating such a set of messages.

```
rl [review-process-starts] :
   < O : ConfChair | mailing-list : OS > (to O : start) Conf & none
   => < O : ConfChair | > Conf & broadcast(OS, announcement from : O) .
```

Note the use of the `_&_` constructor. Since the announcement messages are sent to objects outside the mobile object in which the `ConfChair` object is located, they are placed in its righthand side. The rule `review-process-starts` is applied only if this outgoing messages tray is empty, making sure in this way that any previous message has been handled. The `_&_` operator is the top operator of the term representing the state of the mobile object, and therefore, since there might be other objects and messages in its lefthand side, we include a variable `Conf` of sort `Configuration` to match the rest. Note also how an object may communicate to objects in other mobile objects, which may be in different processes, in a completely transparent way. Most of the communications in the rest of the example will take place inside processes, moving the mobile objects to the processes in which the objects they want to communicate to are located.

When an author object receives the announcement, it decides whether requesting a submission form or not. Such a decision is modeled by the non-deterministic election between a pair of rules which may be applied. They request a form by sending a `subm-request` message to the conference chair, who then creates the corresponding forms and sends them to the authors.

```
rl [author-gets-announcement] :
   < O : Author | paper : P > (to O : announcement from : O') Conf & none
   => < O : Author | > Conf & (to O' : subm-request from : O) .
rl [author-gets-announcement] :
   < O : Author | > (to O : announcement from : O') => < O : Author | > .
```

Notice that since this last rule does not imply the sending of any message out of the mobile object, we do not need to use the `_&_` operator to wrap the whole state.

When the conference chair object receives a `subm-request` message it creates a submission form, which must then find its way to the corresponding author. The `newo` message allows an object in the belly of a mobile object to create another mobile object with a given initial configuration and a module. The module `SUBMISSION-FORM` includes the declaration of the `SubmForm` class given in Section 3.1 and the rules describing its behavior.

```
rl [conf-chair-receives-submission-request] :
   < O : ConfChair | progr-chair : O' >
   (to O : subm-request from : O'') Conf & none
   => < O : ConfChair | > Conf &
      newo(up(SUBMISSION-FORM),
           < tmp-id : SubmForm | author : O'',  conf-chair : O,
               progr-chair : O', author-name : null,
               paper : null,     state : inactive >, tmp-id) .
```

As described in Section 2, Mobile Maude follows the convention of naming the 'main' object in a mobile object as such a mobile object, so that it just has to move messages down into its belly. The rule `start-up` below allows a `SubmForm` object to change its provisional name when it receives a `start-up` message. Changing the name of an object implies destroying the object in the lefthand side of the rule and creating a new one in its righthand side with a different name, and therefore all attributes must be given explicitly.

```
rl [start-up] :
   < tmp-id : SubmForm | author : o(PI, I), state : inactive,
                         conf-chair : O',   progr-chair : O'',
                         author-name : DN,  paper : DP >
   (to tmp-id : start-up(O)) Conf  &  none
   => < O : SubmForm | author : o(PI, I), state : towards-author,
                       conf-chair : O',   progr-chair : O'',
                       author-name : DN,  paper : DP >
      Conf  &  go-find(o(PI, I), PI) .
```

When the submission form is created, it is put in the state `towards-author`, and it sends the message `go-find(o(PI, I), PI)` to communicate to its mobile object that it must find its way to the object with identifier `o(PI, I)`, the author it has been assigned to.

Once the form is created, it could initiate the validation process. However, the submission form must communicate to the author locally, that is, the interchange of messages between them must start once the form has reached the author's process. Since Mobile Maude guarantees that mobile objects moving from one process to another are idle, we know that, once the `go-find` command is given in the `start-up` rule, the form object will not be able to do anything until the mobile object in which it is embedded is set to active, that is, until it has reached the author's process. Therefore, since there is no rule taking a `SubmForm` object in `towards-author` state and a nonempty outgoing messages tray, this object will not do anything until it reaches its destination.

Once the form reaches its corresponding author, the validation process is started by sending a `subm-form-gets-to-author` message. Then, the form goes to the `inactive` state, and it is in such a state until it gets an `activate-subm-form` message from the author.

```
rl [request-activation] :
   < O : SubmForm | author : O', state : towards-author > Conf  &  none
   => < O : SubmForm | state : inactive > Conf &
      (to O' : subm-form-gets-to-author from : O) .
```

When an author receives such a message, it sends back a message to activate the form and in this way initiate the submission process. The submission form then requests the author's information and paper.

When a submission form receives the messages with the name and the paper, its corresponding attributes are updated, and then it moves to the program chair's process. Note that `N` and `P` are, respectively, variables of sorts `Name` and `Paper`, and therefore, if there is a match with this rule it is because the name and the paper are not `null` any more.

```
rl [move] :
  < O : SubmForm | author-name : N, paper : P, progr-chair : o(PI, I),
                   state : active > Conf & none
  => < O : SubmForm | state : towards-pc > Conf & go-find(o(PI, I), PI) .
```

A submission form object informs the program chair of its arrival by sending a message `to O : subm N O'' P from : O'`, with `N` an author's name, `O''` the identifier of the submission form object, and `P` a paper. Upon the reception of such a message, the `ProgrChair` object adds an entry to its set of registered submissions, and assigns it to the number of committee members given by its attribute `reviewers`. The program chair asks the submission form to create a review form for each committee member assigned by sending it a `gen-review-forms` with the chosen members of the program committee.

3.3 The Reviewing Process

The review forms are created with `newo` messages. The `create-review-forms` operation is in charge of generating these `newo` messages, one for each review form to be generated. Note that the identifier of the program committee member that the review form will try to reach after its creation is put as the first of the identifiers in the chain of reviewers.

```
op create-review-forms : Paper Mid Set(Mid) -> MsgSet .
eq create-review-forms(P, O', mt) = none .
eq create-review-forms(P, O', O'' . OS)
    = newo(up(REVIEW-FORM),
          < tmp-id : ReviewForm | state : inactive, score : null,
                progr-chair : O', chain : O'',
                chain-back : no-id, refused : mt, paper : P >, tmp-id)
      create-review-forms(P, O', OS) .
rl [gen-review-forms] :
  < O : SubmForm | progr-chair : O', paper : P >
  (to O : gen-review-forms(OS)) Conf & none
  => < O : SubmForm | state : finishing > Conf &
     create-review-forms(P, O', OS) .
```

When a review form gets to the process of the reviewer it was looking for, which would be a program committee member in the first place, it sends a `review` message to the given reviewer and waits for an answer. The `review` message includes as an argument the set of the reviewers it cannot delegate on, including all the reviewers it has passed by before, and which should not be used again in order to avoid loops, and those who have already rejected the review. The function `listToSet` returns a set with the elements of the list given as argument.

```
rl [ask-for-review] :
   < O : ReviewForm | paper : P, chain : OL & O',
        state : towards-reviewer, refused : OS, score : null > Conf & none
   => < O : ReviewForm | state : inactive > Conf &
      (to O' : review P excluding (listToSet(OL) . OS) from : O) .
```

If a reviewer refuses a review, the review form has to go back to the last reviewer who delegated, adding the refuser to the set in the `refused` attribute.

```
rl [review-refused] :
   < O : ReviewForm | state : inactive, chain : OL & o(PI, I) & O',
                      refused : OS >
   (to O : cannot-review) Conf & none
   => < O : ReviewForm | state : towards-reviewer, chain : OL & o(PI, I),
                         refused : OS . O' > Conf & go-find(o(PI,I), PI) .
```

Once a review is obtained, depending on whether there is only one reviewer in the chain or more than one, the form goes back to the program chair—if there is only one then it is the program committee member itself—or back to the reviewer that delegated on this one.

```
rl [review-result] :
   < O : ReviewForm | state : inactive,
         chain : OL & O' & o(PI, I), chain-back : OL' >
   (to O : review-result Sc) Conf & none
   => < O : ReviewForm | state : back, score : Sc,
          chain : OL & O', chain-back : o(PI, I) & OL' >
      Conf & go-find(o(PI, I), PI) .
```

In its way back, each reviewer checks the review. The review form passes by the processes of each of the reviewers, in such a way that each time it reaches the process of a new reviewer it requests a checking from the reviewer, which may confirm or contradict the review.

When the review forms come back to the program chair's process, they send the results of the review to the program chair object, who saves this information together with the corresponding submission.

```
rl [pc-gets-review-result] :
   < O : ProgrChair | subms : ((N, O'', P, TS, I) Subms) >
   (to O : review-result P Sc from : O')
   => < O : ProgrChair | subms : ((N, O'', P, TS . (O', Sc), I) Subms) > .
```

When the program chair has all the referee reports for a submission, it must be resolved. If the reviews are in agreement, then the paper is accepted or rejected. Otherwise, that is, if the reviews are in disagreement, the program chair declares the review forms unresolved, and sends them back to the members of the program committee to which the submission was assigned. After three review rounds, the program chair resolves the submission deciding what the majority of the reviewers decided.

```
crl [disagreement] :
  < O : ProgrChair | subms : ((N, O', P, TS, I) Subms), reviewers : I' >
  Conf & none
  => < O : ProgrChair | subms : ((N, O', P, mt, I + 1) Subms) >
     Conf & broadcast(reviewers(TS), unresolved)
  if size(TS) == I' and I < 2 and not agreement(TS) .
```

If the reviews are in agreement or the number of rounds reaches the limit, the program chair creates a report form that will find its way back to the author and sends termination messages to all the review forms related to this paper. If there is an agreement then the `decision` function returns the agreed score, otherwise the majority decides. In case of disagreement with an even result then the paper is accepted.

```
crl [agreement] :
   < O : ProgrChair | subms : ((N, O', P, TS, I) Subms), accepted : OS,
                      reviewers : I' > Conf & none
   => < O : ProgrChair | subms : Subms,
                         accepted : if decision(TS) == accept
                                    then O' . OS else OS fi > Conf &
     newo(up(REPORT-FORM),
      < tmp-id : ReportForm | paper : P, author : O',
        score : decision(TS), state : inactive, progr-chair : O >, tmp-id)
   if size(TS) == I' and (agreement(TS) or I == 2) .
```

3.4 The Report Process

When an accepted paper report form reaches an author, it sends a congratulations message and it transforms itself into a final submission form, which will ask the author for the final version of the paper. Notice how we model the transformation of an object of class `ReportForm` into an object of class `FinalSubmForm`. The fact that such inner object keeps its identifer makes that what at some point is a report form mobile object becomes a final submission form mobile object.

```
rl [send-congratulations-and-transform-into-a-final-form] :
   < O : ReportForm | paper : P, author : O', score : accept,
       state : towards-author, program-chair : O'' > Conf & none
  => < O : FinalSubmForm | author : O', state : inactive,
       paper : P, program-chair : O'' > Conf & (to O' : congratulations) .
```

Finally, the final submission form with the final version of the paper sends it to the program chair, which will merge the papers into the proceedings. When the final versions of all accepted papers have arrived to the program chair, and there is no unresolved submission, the program chair sends the proceedings to the conference chair.

4 Related Work

Although Cardelli proposed this case study as a challenge for mobile languages, it is a well known example [3] often used in the areas of workflow management [10] and coordination languages.

The paper [10] describes this case study (such as we have implemented it) as an *administrative workflow*, that is, it involves repetitive, predictable processes with simple task coordination rules, where ordering and coordination of tasks can be automated.

In [15] it is proposed a simulation solution to this case study using Manifold, a strongly-typed, block-structured, event-driven coordination language, where communication is asynchronous and the separation of computation and communication concerns is strongly enforced. This approach structures the application into a hierarchy of processes distinguished in coordinator and coordinated processes, and focus on what can be automated and coordinated. Being a requirements specification, the paper does not deal with mobility, which the authors understand as an implementation issue.

The paper [14] presents a solution closer to ours, using the concept of active (mobile) documents. It uses the PageSpace architecture, a design paradigm for Web-based applications that are composed of autonomous agents performing their duties regardless of their physical positions. This architecture is instantiated with a coordination language with mobility facilities. It follows the MUD (Multi User Dungeon) metaphor, a cooperative interactive environment shared by several people to socialize and interact. It is based on the concepts of rooms, items and players (or users) which relate with our concepts of processes and mobile objects.

Real systems implementing this application have been also described. The paper [13] describes the electronic management systems developed for the Fourth and Fifth International World Wide Web Conferences.

5 Conclusions

We have showed how Mobile Maude, an extension of Maude supporting mobile computation, can be used to represent and specify ambitious wide area applications by providing an implementation of a conference reviewing system.

The semantics of Mobile Maude is defined in an application-independent way by a small set of rewrite rules. In addition to allowing us to execute mobile systems, to prove properties on them, etc., this executable Maude specification may allow us to identify possible deficiencies and experiment with different alternatives. For example, the `go-find` message was not in the original set of system messages. Moreover, we believe that new features should be introduced in Mobile Maude, specially allowing new kinds of agent communication and interaction. What happens, for example, if an object moves to another process while it is interacting with another object? Perhaps, there should be some kind of communication between the inner objects inside a mobile object and the process where it is, such as asking if another mobile object is also in the same process, or asking the process for notification when another object leaves or arrives. Also, we may consider supporting group-oriented events (where messages are sent to all the members in a group of mobile agents) and *collaboration* (where collaborator agents and agent groups are defined and interact with each other to obtain a shared objective) as in the *Concordia* system [16].

As future work, we want to consider the verification of properties of mobile systems, in particular those related to their reliability and security. Interesting contributions on this direction are the works on modal logics for rewriting logic

[9] and the work on spatial logics [1].

Acknowledgements
The authors are grateful to Steven Eker, Patrick Lincoln and José Meseguer, with which we are developing Mobile Maude, for the many discussions on the specification of the conference reviewing system. We are also very thankful to Narciso Martí-Oliet and Roberto Bruni for their constructive comments on several drafts of the paper and on the specification of the application. We would also like to thank the anonymous referees for leading us to related work.

References

[1] L. Caires and L. Cardelli. A spatial logic for concurrency (Part I). In *Theoretical Aspects of Computer Software, TACS 2001*, LNCS 2215, pages 1–37. Springer-Verlag, 2001.

[2] L. Cardelli. Abstractions for mobile computations. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, pages 51–94. Springer-Verlag, 1999.

[3] P. Ciancarini, O. Niestrasz, and R. Tolksdorf. A case study in coordination: Conference management on the Internet, 1998. `ftp://cs.unibo.it/pub/cianca/coordina.ps.gz`.

[4] P. Ciancarini and A. Wolf, editors. *Proceedings 3rd International Conference on Coordination Models and Languages*, LNCS 1594. Springer-Verlag, 1999.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 304, 2002. To appear.

[6] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.

[7] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000*, LNCS 1882. Springer-Verlag, Sept. 2000.

[8] F. Durán and A. Verdejo. Cardelli's challenge in Mobile Maude: A conference reviewing system. Technical Report 124.02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, May 2002.

[9] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99*, LNCS 1827, pages 438–458. Springer-Verlag, 2000.

[10] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[11] D. Kotz and R. Gray. Mobile agents and the future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, August 1999.

[12] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the Association of Computer Machinery*, 42:88–89, March 1999.

[13] G. Mathews and B. Jacobs. Electronic management of the peer review process. *Computer Networks and ISDN Systems*, 28(7–11):1523–1538, 1996.

[14] D. Rossi and F. Vitali. Internet-based coordination environments and document-based applications: A case study. In Ciancarini and Wolf [4], pages 259–274.

[15] A. Scutellà. Simulation of conference management using an event-driven coordination language. In Ciancarini and Wolf [4], pages 243–258.

[16] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Mobile Agents. First International Workshop, MA'97*, LNCS 1219, pages 86–97. Springer-Verlag, 1997.