Cardelli's Challenge in Mobile Maude: A Conference Reviewing System

Francisco Durán[1] and Alberto Verdejo[2]
[1] ETSI Informática, Universidad de Málaga, Spain. duran@lcc.uma.es
[2] Facultad de Informática, Universidad Complutense, Madrid, Spain. alberto@sip.ucm.es

Technical Report 124-02
Dpto. Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain

May 12, 2002

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 suports 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.

Contents

1 Cardelli's description of the problem

The ambitious wide area application we discuss in this section was first described by Luca Cardelli in [1] as *a challenge for any wide area language to demonstrate its usability.*

The problem description, as presented in [1], is the following one.

> **Description of the problem**
>
> The problem consists in managing a virtual program committee meeting for a conference.
>
> **Announcement**
>
> A conference is announced, and an electronic submission form, signed by the conference chair, is publicized.
>
> **Submission**
>
> Each author fetches the submission form, checks the signature of the conference chair, and activates the form. Once activated, the form actively guides most of the reviewing process. Each author fills an instance of the form and attaches a paper. The form checks that none of the required fields are left blank, electronically signs the paper with a signature key provided by the author, encrypts the attached paper, and finds its way to the program chair. The program chair collects the submissions forms, and gives them a decryption key so that they can decrypt the attached papers and verify the signatures of the authors. (All following communications are signed and encrypted; we omit most of these details from now on.)
>
> **Assignment**
>
> The program chair then assigns the submissions to the committee members, by instructing each submission form to generate a review form for each assigned member. The review forms incorporate the paper (this time signed by the program chair) and find their way to the appropriate committee members.
>
> **Review**
>
> Each committee member is a reviewer, and may decide to review the paper directly, or to send it to another reviewer. The review form keeps tracks 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 the subreviewers. Eventually a review is filled. The form performs various consistency checks, such as verifying that the assigned scores are in range and that no required fields are left blank. Then it finds its way back to the program chair.
>
> **Report generation**
>
> Once the review forms reach the program chair, they become report forms. The various report forms for each paper merge with each other incrementally to form a single report form that accumulates the scores and the reviews. The program chair monitors the report form for each paper. If the reviews are in agreement, the program chair declares the form an accepted paper report form, or a rejected paper report form.
>
> **Conflict resolution**
>
> If the reports are in disagreement, the program chair declares the form an unresolved review form. An unresolved review form circulates between the reviewers and the program chair, accumulating further comments, until the program chair declares the paper accepted or rejected.
>
> **Notification**
>
> The report form for an accepted or rejected paper finds its way back to the author (minus the confidential comments), with appropriate congratulations or regrets.
>
> **Final versions**
>
> Once it reaches the author, an accepted paper report form spawns a final submission form. In due time, the author attaches to it the final version of the paper and signs the copyright release notice. The completed final submissions form finds its way back to the program chair.
>
> **Proceedings**
>
> The final submission forms, upon reaching the program chair, merge themselves into the proceedings. The program chair checks that all the final versions have arrived, sorts them into a conference schedule, attaches a preface, and lets the proceedings find their way to the conference chair.
>
> **Publication**
>
> The conference chair files the copyright release forms, signs the proceedings, and posts them to public sites.

Mobile Maude was first described in [2]. 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 : MachineInt,
          guests : Set[Mid],
          forward : PFun[MachineInt, Tuple[Pid, MachineInt]] .
```

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 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 the `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 another process it sends a message to its home process announcing its new location. The number of hops guarantees that if the messages are received out of order then only the useful ones are considered.

The class of mobile objects is declared as follows:

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

The `s` attribute keeps (the metarepresentation of) the mobile object's state, and have to be of the form `C & C'`, where `C` is a configuration of objects—one of them, the *main* one, has the same identifier as the mobile object it is in—and messages—unprocessed incoming messages and inter-inner-objects messages—, and `C'` is a multiset of messages—the *outgoing* messages tray. 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 process is inside a process, and `idle` if it is moving.

As said above, 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, etc.

We will present now the Mobile Maude primitives used in the example, together with some rewrite rules describing them.

When a mobile object wants to send a message to another one, it has to create a message of the form `to MO : MSG-CNTS`, 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 2.2). That is, the minimum information needed to dispatch a message is the receiver identity; if the sender wants to communicate its name, it has to include it in the message content. 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. We show some of the rewrite rules handling the dispatching of messages. The following rule, where terms appear metarepresented (for example, the term `T'` metarepresents the name of the addressee object), initiates this process by pulling out the messages from the mobile object.

```
rl [message-out-to] :
  < M : MO | mod : MOD, s : '_&_[T, 'to_:_[T', T'']], mode : active >
  => < M : MO | mod : MOD, s : '_&_[T, {'none}'MsgSet] >
     (to downMid(T') { T'' }) .
```

If the message is sent to a mobile object `o(PI', N)` in a different process it must be sent to the parent process `PI'` which will appropriately redirect it using the forwarding information.

```
crl [msg-send] :
  < PI : P | cf : C (to o(PI', N) { T }), guests : SMO, forward : F >
  => < PI : P | cf : C >
     to o(PI', N) hops null in PI' { T }
  if (not o(PI', N) in SMO) and PI =/= PI' .
```

When the message arrives to the process where the addressee object is, it is first put in its inner configuration, and finally it is put in the belly of the target mobile object.

```
crl [msg-arrive-to-proc] :
  to o(PI, N) hops H in PI' { T' }
  < PI' : P | cf : C, guests : SMO >
  => < PI' : P | cf : C (to o(PI, N) { T' }) >
  if o(PI, N) in SMO .
rl [msg-in] :
  to M { T }
  < M : MO | mod : MOD, s : '_&_[T', T''] >
  => < M : MO | s : '_&_['__['to_:_[up(M), T], T'], T''] > .
```

We want to point out that there are three kinds of communication between objects. Objects inside the same mobile object can communicate with each other by means of messages of any kind; a mobile object can communicate with another mobile object in the same process; and a mobile object can communicate with another mobile object in a different process. In these last two kinds of communication, messages of the form `to MO : MSG-CNTS` are used, and the actual kind of communication is transparent to the mobile objects.

When a mobile object wants to move to another process it puts in its outgoing messages tray a `go(PID)` message, where `PID` is the target process identifier. The following rule initiates the movement: When a mobile object has an outgoing `go` message, a new `go` message is sent, with the mobile object as one of its arguments, after removing the outgoing message and setting the state to `idle`.

```
rl [message-out-move] :
  < M : MO | s : '_&_[T, 'go[T']], mode : active >
  => go(downPid(T'), < M : MO | s : '_&_[T, 'none.MsgSet], mode : idle >) .
```

This message has to travel to the desired process, going out to the outer configuration of processes if the current process is different from the desired one. When the message reaches the destination process, the mobile object is put into it (in `active` mode), and the parent process is informed, or just updated, if it is the parent process itself.

```
rl [arrive-proc] :
  go(PI, < o(PI', N) : MO | hops : N' >)
  < PI : P | cf : C, guests : SMO, forward : F >
  => if PI == PI'
     then < PI : P | cf : C < o(PI', N) : MO |
                               p : PI, hops : N' + 1, mode : active >,
                     guests : o(PI', N) . SMO,
                     forward : F[N -> (PI, N' + 1)] >
     else < PI : P | cf : C < o(PI', N) : MO |
                               p : PI, hops : N' + 1, mode : active >,
                     guests : o(PI', N) . SMO >
          (to PI' @ (PI, N' + 1) { N })
     fi .
```

In the previous `go` message, the mobile object has to indicate the process where it wants to go. But sometimes, a mobile object wants to go where another object is, but it only knows the name of the object it wants to catch up. In this case, the `go-find` message can be used. When this message is used by object `M`, it takes as arguments the `Mid` of the object that `M` wants to reach, and the `Pid` of a tentative process where it may be. Rewrite rules similar to the above ones describe how the target object is reached, asking to its parent process where it is whenever necessary.

When an object wants to create a 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 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. This is done in this way because Mobile Maude assumes the convention of naming the 'main' object in the belly of a mobile object as such a mobile object, so that it just have to move messages down into its belly. Although the creation of new mobile objects could be simplified following a different approach, for example, making each mobile object to keep the identifier of its main object, and then changing the addressee of the messages before taking them in, this approach simplifies the allocation of objects and mobile objects.

A `kill` message is used by an object when it has finished its job and it wants the mobile-object it is in to be disappeared (see, for example, Section 2.2). Mobile Maude handles this message updating the whole state of the system. The mobile object encapsulating the inner, dead object disappears, and the `guest` attribute of the process where it is and the `forward` attribute of its parent are updated.

The execution of mobile objects uses reflection and is accomplished by the following rule.

```
rl [do-something] :
  < M : MO | mod : MOD, s : T, mode : active >
  => < M : MO | s : meta-rewrite(MOD, T, 1) > .
```

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 Maude specification may be easily made mobile.

2 Conference Reviewing System

In the following description we have wanted to include all the complete modules, as they are introduce in the Maude system, and in this same order. Although this is not the best way to understand the reviewing process, or how the different rewrite rules are applied, we want to show how the application code is structured in different modules. A different presentation, that follows the reviewing process sequentialization, can be found in [3].

2.1 Auxiliary modules

We start presenting some auxiliary modules.

These are auxiliary modules defining sorts `Paper` and `Name`, and several constants of each one.

```
(fmod PAPER is
  sort Paper .
  ops p1 p2 p3 p4 p5 : -> Paper .
  op final-version : Paper -> Paper .
  op rejected : Paper -> Paper .
 endfm)

(view Paper from TRIV to PAPER is
  sort Elt to Paper .
 endv)

(fmod NAME is
  sort Name .
  ops author1 author2 author3 author4 conf-chair program-chair
      pc-member1 pc-member2 reviewer1 reviewer2 reviewer3 : -> Name .
 endfm)

(view Name from TRIV to NAME is
  sort Elt to Name .
 endv)

(fmod SET-MID is
   pr SET[Mid] * (op __ : Set[Mid] Set[Mid] -> Set[Mid] to _._) .
 endfm)
```

The following module defines an operation `broadcast` used to send the same message content to a set of the mobile objects .

```
(mod BROADCAST is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr SET-MID .

  op broadcast : Set[Mid] Contents -> MsgSet .

  var O : Mid .
  var OS : Set[Mid] .
  var MC : Contents .

  eq broadcast((O . OS), MC) = (to O : MC) broadcast(OS, MC) .
  eq broadcast(mt, MC) = none .

 endm)
```

Module `CHOOSE-REVIEWERS` defines the `choose` operation, that nondeterministically chooses a given number of mobile object identifiers from a given set.

```
(mod CHOOSE-REVIEWERS is
   inc MOBILE-OBJECT-ADDITIONAL-DEFS .
   pr SET-MID .

  op choose : Set[Mid] MachineInt -> Set[Mid] .

  var O : Mid .
  var OS : Set[Mid] .
  var I : MachineInt .

  rl [choose-reviewers] :
     choose(OS, 0) => mt .
  crl [choose-reviewers] :
     choose(O . OS, I) => O . choose(OS, _-_(I, 1)) if I > 0 .
endm)
```

We assume that a reviewer can only accept or reject the paper. In the current version there are no more possibilities and no comments.

```
(fmod SCORE is
  sort Score .
  ops accept reject : -> Score .
  op opp : Score -> Score .
  eq opp(accept) = reject .
  eq opp(reject) = accept .
 endfm)

(view Score from TRIV to SCORE is
  sort Elt to Score .
 endv)
```

2.2 Review Forms

These are the messages interchanged between a review form and a reviewer:

```
(omod REVIEW-FORM-REVIEWER-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr PAPER .
  pr SCORE .
  pr (LIST * (op nil to no-id, op __ to _&_))[Mid] .
  pr SET-MID .

  op review_excluding_from‘:_ : Paper Set[Mid] Mid -> Contents .
  op delegate-to_ : Mid -> Contents .
  op review-result_ : Score -> Contents .
  op check-review__from‘:_ : Paper Score Mid -> Contents .
  op check-review-result_ : Score -> Contents .
  op cannot-review : -> Contents .
  op finish : -> Contents .
  op unresolved : -> Contents .
  op unresolved-review_from‘:_reviewers_ : Paper Mid List[Mid] -> Contents .
  op unresolved-delegate : -> Contents .
  op unresolved-review-result_ : Score -> Contents .
 endom)
```

and the message that a review form sends to the program chair:

```
(omod REVIEW-FORM-PROGR-CHAIR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr PAPER + SCORE .
  op review-result__from‘:_ : Paper Score Mid -> Contents .
 endom)


(omod REVIEW-FORM is
  inc REVIEW-FORM-REVIEWER-MSGS .
  inc REVIEW-FORM-PROGR-CHAIR-MSGS .
  pr NAME .
  pr DEFAULT[Score] .
```

In order to distinguish the different states in which a review form can be, the class `ReviewForm` has an attribute `state` of sort `ReviewFormState` with the following possible values.

```
  sort ReviewFormState .
  ops inactive towards-reviewer back-from-review
      unresolved-towards-reviewer unresolved-back-from-review : -> ReviewFormState .
```

A reviewer may decide to review a paper assigned to him/her 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 the 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 id of the program committee assigned to it, and all the succesive subreviewers are added to this list. When the paper is finally reviewed, the form finds its way back by extracting the ids from the lists in a *last-in-first-out* manner. Nevertheless, instead of discarding these ids, they are included in the list in the second attribute, `chain-back`, so that it can follow its reviewing path again if necessary.

```
  class ReviewForm | paper : Paper,
                     program-chair : Mid,
                     chain : List[Mid],
                     chain-back : List[Mid],
                     refused : Set[Mid],
                     score : Default[Score],
                     state : ReviewFormState .

  var P : Paper .
  var PI : Pid .
  var I : MachineInt .
  vars O O' : Mid .
  vars OL OL' : List[Mid] .
  var OS : Set[Mid] .
  var Sc : Score .
  var Conf : Configuration .
```

When the mobile object form is created, the name of the form object in its belly must be started up, initializing its name and putting it in the `towards-reviewer` state.

```
  rl [start-up] :
     < tmp-id : ReviewForm | paper : P, program-chair : O,
          chain : o(PI, I), chain-back : no-id,
          refused : mt, score : null, state : inactive >
     (to tmp-id : start-up(O')) Conf & none
     => < O' : ReviewForm | paper : P, chain : o(PI, I),
             chain-back : no-id, refused : mt, score : null,
             state : towards-reviewer, program-chair : O >
        Conf & go-find(o(PI, I), PI) .
```

We assume that, once the `go-find` command is given in the start-up rule, the review 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 assigned program committee member's process. This prevents from starting the exchange of messages before arriving to the author's object, which seems not to have much sense in this context. However, there is no reason to forbid this systematically, since there could be applications in which this is useful.

The review form keeps 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 the subreviewers.

When a review form gets to the process of a reviewer, which would be a program committee member in the first place, it sends a review message to the given reviewer (saying which reviewers he cannot delegate on) and waits for an answer.

```
op listToSet : List[Mid] -> Set[Mid] .
eq listToSet(no-id) = mt .
eq listToSet(O & OL) = (O . listToSet(OL)) .

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) .
```

There can be three different messages back: delegating the review to some subreviewer, refusing the review, or giving the review itself. If delegated, the form must find its way to the indicated subreviewer.

```
rl [review-delegation] :
   < O : ReviewForm | state : inactive, chain : OL >
   (to O : delegate-to o(PI, I)) Conf & none
   => < O : ReviewForm | state : towards-reviewer,
        chain : OL & o(PI, I) >
      Conf & go-find(o(PI, I), PI) .
```

If a reviewer refuses a review, the review form has to go back to the last reviewer who delegated, adding the refuser to 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 he/she is the program committee member himself/herself—or back to the reviewer that delegated on this one.

```
rl [review-result] :
   < O : ReviewForm | state : inactive, chain : O',
        chain-back : OL, program-chair : o(PI, I) >
   (to O : review-result Sc) Conf & none
   => < O : ReviewForm | state : back-from-review, score : Sc,
        chain : no-id, chain-back : O' & OL >
      Conf & go-find(o(PI, I), PI) .

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-from-review, 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 request a checking from the reviewer, which may confirm or contradict the review. We assume that the score given by a reviewer prevails over the score given by his/her subreviewers. Another possibility would be to send it back to the subreviewer, asking for confirmation, or even for rectification.

```
rl [way-back] :
   < O : ReviewForm | paper : P, chain : OL & O',
        state : back-from-review, score : Sc >
   Conf & none
   => < O : ReviewForm | state : inactive > Conf &
      (to O' : check-review P Sc from : O) .

rl [get-checked-review] :
   < O : ReviewForm | state : inactive, program-chair : o(PI, I),
        chain : O', chain-back : OL >
   (to O : check-review-result Sc) Conf & none
   => < O : ReviewForm | state : back-from-review, score : Sc,
        chain : no-id, chain-back : O' & OL >
      Conf & go-find(o(PI, I), PI) .

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

At some point, following the chain of reviewers, the review form reaches the program chair, which receives the review result.

```
rl [back-to-the-program-chair] :
   < O : ReviewForm | paper : P, chain : no-id,
        state : back-from-review, score : Sc, program-chair : O' >
   Conf & none
   => < O : ReviewForm | state : inactive > Conf &
      (to O' : review-result P Sc from : O) .
```

The program chair can say to a review form that it has finished.

```
rl [dead] :
   < O : ReviewForm | chain : no-id, state : inactive >
   (to O : finish) & none
   => (none).Configuration & kill .
```

The program chair can declare the review form unresolved.

```
rl [unresolved] :
   < O : ReviewForm | chain : no-id,
        chain-back : o(PI,I) & OL, state : inactive >
   (to O : unresolved) Conf & none
```

```
    => < O : ReviewForm | chain : o(PI,I),
         chain-back : OL, state : unresolved-towards-reviewer >
        Conf & go-find(o(PI,I), PI) .

  rl [unresolved-ask-for-review] :
    < O : ReviewForm | paper : P, chain : OL & O',
         chain-back : OL', state : unresolved-towards-reviewer >
    Conf & none
    => < O : ReviewForm | state : inactive >
        Conf & (to O' : unresolved-review P from : O reviewers OL') .

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

When an unresolved review form receives a new result, it goes directly to the program chair.

```
  rl [unresolved-review-result] :
    < O : ReviewForm | chain : OL, program-chair : o(PI,I),
         chain-back : OL', state : inactive >
    (to O : unresolved-review-result Sc) Conf & none
    => < O : ReviewForm | score : Sc, chain : no-id,
         chain-back : OL & OL', state : unresolved-back-from-review >
        Conf & go-find(o(PI,I), PI) .

  rl [unresolved-back-to-the-program-chair] :
    < O : ReviewForm | paper : P, chain : no-id,
         state : unresolved-back-from-review, score : Sc, program-chair : O' >
    Conf & none
    => < O : ReviewForm | state : inactive > Conf &
       (to O' : review-result P Sc from : O) .

 endom)
```

2.3 Submission Form

These are the messages that a submission form and an author interchange:

```
(omod SUBMISSION-FORM-AUTHOR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr NAME + PAPER .
  op subm-form-gets-to-author`from`:_ : Mid -> Contents .
  op activate-subm-form : -> Contents .
  op request-author-name : -> Contents .
  op author-name : Name -> Contents .
  op request-paper : -> Contents .
  op paper : Paper -> Contents .
 endom)
```

and the messages that a submission form and the program chair interchange:

```
(omod SUBMISSION-FORM-PROGR-CHAIR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr NAME + PAPER .
  pr SET-MID .
  op submission___from`:_ : Name Mid Paper Mid -> Contents .
  op generate-review-forms : Set[Mid] -> Contents .
 endom)
```

Once activated, the submission form actively guides most of the submission process. Each author fills an instance of the form and attaches a paper. The form checks that none of the required fields are left blank and finds its way to the program chair.

```
(omod SUBMISSION-FORM is
  inc SUBMISSION-FORM-AUTHOR-MSGS .
  inc SUBMISSION-FORM-PROGR-CHAIR-MSGS .
  pr REVIEW-FORM .
  pr DEFAULT[Name] .
  pr DEFAULT[Paper] .
  pr CHOOSE-REVIEWERS .
```

In order to distinguish the different states in which a form can be, class `SubmForm` has an attribute `state` of sort `SubmFormState` with the following possible values.

```
  sort SubmFormState .
  ops inactive towards-author waiting active towards-pc
      finishing finished :
      -> SubmFormState .
```

In fact, it is possibly enough with two states, so that we are able to detect the moment at which a form object arrives to the process of an author or program chair and then starts the communication with it.

```
  class SubmForm | conf-chair : Mid,
                   author : Mid,
                   program-chair : Mid,
                   author-name : Default[Name],
                   paper : Default[Paper],
                   state : SubmFormState .

  vars O O' O'' : Mid .
  var OS : Set[Mid] .
  var PI : Pid .
  var I : MachineInt .
  var Conf : Configuration .
  var N : Name .
  var P : Paper .
  var DN : Default[Name] .
  var DP : Default[Paper] .
```

When the mobile object submission form is created, the name of the form object in its belly must be started up, initializing its name and putting it in the `towards-author` state. Note that changing the name of an object implies destroying the object in the lhs of the rule and creating a new one in the rhs with a different name, and therefore all attributes must be explicitly copied. This rule does not work for objects in subclasses in which new attributes are added.

```
  rl [start-up] :
     < tmp-id : SubmForm |
          author : o(PI, I), state : (inactive).SubmFormState,
          conf-chair : O',  program-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',  program-chair : O'',
               author-name : DN,  paper : DP >
        Conf & go-find(o(PI, I), PI) .
```

Once the form is in the `towards-author` state, it can initiate the validation process. However, we assume 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. This prevents from starting the exchange of messages before arriving to the author's object, which seems not to have much sense in this context.

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

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

When the author accepts the form, that is, when it receives the `subm-form-gets-to-author` message from the form, then the author sends an `activate-subm-form` message to the form. The information about the author and the paper are then requested. Note that in the current version this is the only information being requested, and that since the communication between the form and the author is already established these messages do not include the form's identifier. Note also that the form goes to the active state.

```
rl [activation] :
   < O : SubmForm | author : O', state : waiting >
   (to O : activate-subm-form)
   Conf & none
   => < O : SubmForm | state : active > Conf &
      (to O' : request-author-name)
      (to O' : request-paper) .
```

When the submission form receives the messages with the name and the paper the corresponding attributes are updated.

```
rl [author-name] :
   < O : SubmForm | state : active > (to O : author-name(N))
   => < O : SubmForm | author-name : N > .

rl [paper] :
   < O : SubmForm | state : active > (to O : paper(P))
   => < O : SubmForm | paper : P > .
```

When the submission form has the author's name and paper, 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,
           program-chair : o(PI, I),
           state : (active).SubmFormState >
   Conf & none
   => < O : SubmForm | state : towards-pc >
      Conf & go-find(o(PI, I), PI) .
```

When a submission form reaches the program chair the form gives the information about the submission to the program chair.

```
rl [get-to-pc] :
   < O : SubmForm | author-name : N, author : O'', paper : P,
           program-chair : O', state : towards-pc >
```

```
    Conf & none
    => < O : SubmForm | state : waiting > Conf &
       (to O' : submission N O'' P from : O) .
```

The program chair then assigns the submissions to the committee members, by instructing each submission form to generate a review form for each assigned member.

```
  op create-review-forms : Paper Mid Set[Mid] -> MsgSet .

  eq create-review-forms(P, O', mt) = (none).MsgSet .
  eq create-review-forms(P, O', O'' . OS) =
        newo(up(REVIEW-FORM),
             < tmp-id : ReviewForm |
                   state : (inactive).ReviewFormState, score : null,
                   program-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 | program-chair : O', paper : P >
     (to O : generate-review-forms(OS)) Conf & none
     => < O : SubmForm | state : finishing >
        Conf & create-review-forms(P, O', OS) .
```

A submission form kills itself after generating the review forms.

```
  rl [dead] :
     < O : SubmForm | state : finishing > Conf & none
     => < O : SubmForm | state : finished > Conf & kill .

 endom)
```

2.4 Author

These are the message contents that the conference chair and an author interchange. `announcement` is the message that the conference chair sends to each author announcing a conference (and providing the conference chair id), and `submission-request` is the author's response (providing his id).

```
(omod CONF-CHAIR-AUTHOR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  op announcement`from`:_ : Mid -> Contents .
  op submission-request`from`:_ : Mid -> Contents .
 endom)
```

And the messages that an author interchange with a report form or a final submission form:

```
(omod REPORT-FORM-AUTHOR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  op congratulations : -> Contents .
  op regrets : -> Contents .
 endom)

(omod FINAL-SUBMISSION-FORM-AUTHOR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr PAPER .
  op final-subm-form-gets-to-author`from`:_ : Mid -> Contents .
  op final-paper : Paper -> Contents .
 endom)
```

We suppose that each author has at most one paper.

```
(omod AUTHOR is
  inc SUBMISSION-FORM-AUTHOR-MSGS .
  inc CONF-CHAIR-AUTHOR-MSGS .
  inc REPORT-FORM-AUTHOR-MSGS .
  inc FINAL-SUBMISSION-FORM-AUTHOR-MSGS .
  pr DEFAULT[Mid] .
  pr DEFAULT[Paper] .
  pr NAME .

  class Author | name : Name,
                 paper : Default[Paper],
                 submission-form : Default[Mid] .

  vars O O' : Mid .
  var Conf : Configuration .
  var N : Name .
  var P : Paper .
```

When an author receives a conference's announcement, he/she decides whether requesting a submission form or not.

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

In order to see what is happening when executing the system we assume that if an author has a paper then he/she requests a submission form. Hence, we omit the following rule:

```
        rl [author-gets-announcement] :
              < O : Author | > (to O : announcement from : O')
           => < O : Author | > .
```

When the submission form mobile object reaches the process in which there is an author, the form sends a message `subm-form-gets-to-author` to let the author know about it, which responds activating the form to initiate the submission process. The author should check the chairman's signature before activating the submission form.

```
  rl [activate-form] :
     < O : Author | > (to O : subm-form-gets-to-author from : O')
     Conf & none
     => < O : Author | submission-form : O' > Conf &
        to O' : activate-subm-form .
```

The author sends its name and paper to the submission form when requested.

```
  rl [request-name] :
     < O : Author | name : N, submission-form : O' >
     (to O : request-author-name) Conf & none
     => < O : Author | > Conf & to O' : author-name(N) .

  rl [request-paper] :
     < O : Author | paper : P, submission-form : O' >
     (to O : request-paper) Conf & none
     => < O : Author | > Conf & to O' : paper(P) .
```

An author can receive congratulations or regrets from a report form.

```
  rl [get-congratulations] :
     < O : Author | > (to O : congratulations) Conf & none
     => < O : Author | > Conf & none .

  rl [get-regrets] :
     < O : Author | paper : P > (to O : regrets) Conf & none
     => < O : Author | paper : rejected(P) > Conf & none .
```

When the final submission form is created by an accepted paper report form, it sends a message `final-subm-form-gets-to-author` to the author, which responds sending it the final version of the paper.

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

endom)
```

2.5 Reviewers

```
(omod REVIEWER is
   inc REVIEW-FORM-REVIEWER-MSGS .
   pr AUTHOR .
   pr SET[Paper] * (op __ : Set[Paper] Set[Paper] -> Set[Paper] to _._) .

   class Reviewer | subreviewers : Set[Mid], done : Set[Paper] .
```

Any reviewer can be interested in submitting a paper to the conference.

```
   subclass Reviewer < Author .

   var P : Paper .
   var PS : Set[Paper] .
   vars O O' O'' : Mid .
   vars OS OS' : Set[Mid] .
   var OL : List[Mid] .
   var Sc : Score .
   var Conf : Configuration .
```

A reviewer can give three different answers to a review message: it can delegate in some other reviewer, it can refuse the review because it has already participated in the review of the given paper, or it can do the review and send back the result, which can be accept or reject.

A reviewer should only be able to delegate in someone who has not accepted or delegated the paper yet. The review form keeps this information, and send it in the "excluding" part of the review request.

```
   crl [delegate-review] :
      < O : Reviewer | subreviewers : O' . OS, done : PS >
      (to O : review P excluding OS' from : O'') Conf & none
      => < O : Reviewer | done : PS . P > Conf &
         (to O'' : delegate-to O')
      if not (O' in OS') .

   rl [review-accept] :
      < O : Reviewer | done : PS >
      (to O : review P excluding OS from : O') Conf & none
      => if (P in PS) then
           (< O : Reviewer | > Conf & (to O' : cannot-review))
```

```
        else
          (< O : Reviewer | done : PS . P > Conf &
          (to O' : review-result accept))
        fi .
   rl [review-reject] :
      < O : Reviewer | done : PS >
      (to O : review P excluding OS from : O') Conf & none
      => if P in PS then
           < O : Reviewer | > Conf & (to O' : cannot-review)
         else
           < O : Reviewer | done : PS . P > Conf &
           (to O' : review-result reject)
         fi .
```

In the way back of the review form, each reviewer checks the review.

```
   rl [agree-review-check] :
      < O : Reviewer | > (to O : check-review P Sc from : O') Conf & none
      => < O : Reviewer | > Conf & (to O' : check-review-result Sc) .

   rl [disagree-review-check] :
      < O : Reviewer | > (to O : check-review P Sc from : O') Conf & none
      => < O : Reviewer | > Conf & (to O' : check-review-result opp(Sc)) .
```

When an unresolved review form asks for a new review, the reviewer can delegate again, or give a result.

```
   rl [unresolved-delegate-review] :
      < O : Reviewer | >
      (to O : unresolved-review P from : O' reviewers O'' & OL) Conf & none
      => < O : Reviewer | > Conf & (to O' : unresolved-delegate) .

   rl [unresolved-review-accept] :
      < O : Reviewer | >
      (to O : unresolved-review P from : O' reviewers OL) Conf & none
      => < O : Reviewer | > Conf & (to O' : unresolved-review-result accept) .

   rl [unresolved-review-reject] :
      < O : Reviewer | >
      (to O : unresolved-review P from : O' reviewers OL) Conf & none
      => < O : Reviewer | > Conf & (to O' : unresolved-review-result reject) .

 endom)
```

2.6 Conference Chair

The following module defines the message sent by the program chair to the conference chair when the proceedings are complete.

```
(omod CONF-CHAIR-PROGR-CHAIR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  pr SET[Paper] * (op __ : Set[Paper] Set[Paper] -> Set[Paper] to _._) .
  op proceedings : Set[Paper] -> Contents .
 endom)

(omod CONF-CHAIR is
  pr SUBMISSION-FORM .
  pr CONF-CHAIR-AUTHOR-MSGS .
  inc CONF-CHAIR-PROGR-CHAIR-MSGS .
  inc REVIEWER .
```

```
pr BROADCAST .

class ConfChair | mailing-list : Set[Mid],
                  program-chair : Mid,
                  proceedings : Set[Paper] .
```

A conference chair can also be a reviewer and an author.

```
subclasses ConfChair < Author Reviewer .
```

The `start` message is used when simulating an example. It is sent to the conference chair in order to start all the process.

```
op start : -> Contents .

var MS : MsgSet .
vars O O' O'' : Mid .
var OS : Set[Mid] .
var Conf : Configuration .
var MC : Contents .
var PS : Set[Paper] .
```

The conference review process starts with the broadcasting of the announcement message to all the potential authors in its mailing list.

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

Since in order to simplify the specification most of the messages do not contain any data, it does not make much sense to encrypt them. However, it is assumed that all messages are encrypted. Such an encryptation could be easily added by using encrypt and decrypt functions as follows. If an object `A` sends a message `M` to another object `B` we generally have a message `(to B : M from : A)`. Instead, if we assume `KU` and `KP B`'s public and private keys, respectively, we can consider a message `(to B : encrypt(M, KU) from : A)`, which can then be decrypted by `B` using its private key with an equation as `decrypt(encrypt(M, KU), KP) = M`. By using the keys in the reverse order we have the usual signing, that is, we can use private keys to sign messages and then used the corresponding public one to read it `decrypt(encrypt(M, KP), KU) = M`.

Each of the authors receives the announcement and decides whether requesting a submission form or not. They request a form by sending a `submission-request` message to the conference chair, who then creates the corresponding forms and sends them to the authors.

When the conference chair receives a `submission-request` message he/she creates a submission form, which finds its way to the corresponding author. It could have been specified in other ways, e.g., the forms could be created by the respective authors after receiving the message from the conference chair.

Note that the conference chair does not keep track of the potential authors to which he/she is sending the submission forms. We may assume that such forms are destroyed if certain deadline is reached.

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

Here, we are assuming that `newo` takes a module (a term of sort `Module` metarepresenting a module), a configuration (which will be the initial configuration to put in the belly of the mobile object to be created), and the provisional identifier of the main object in the configuration given as second argument. We assume that the first action accomplished by the conference chair when it detects the `newo` message is to create a new mobile object with the configuration given as second argument of the `newo` message in it, and then sending 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. Mobile objects take their names when they are created, and they are of the form `o(P, N)`, where `P` is the name of the mobile object's home process and `N` is a number. This is done in this way because we follow the convention of naming the 'main' object in a mobile object as such a mobile object, so that it just have to move messages down into its belly. Another alternative would be, for example, to make each mobile object keep the identifier of this main object, and then change the addresse of the messages before taking it in.

When all the review process has finished, the conference chair receives the proceedings from the program chair. Note that we make sure that this message is received only if there are no proceedings already available.

```
  rl [receive-proceedings] :
     < O : ConfChair | proceedings : mt >
     (to O : proceedings(PS))
     => < O : ConfChair | proceedings : PS > .

 endom)
```

2.7 Final Submission Form

A final submission form communicates its final paper to the program chair by using the message `final-paper`.

```
(omod FINAL-SUBMISSION-FORM-PROGR-CHAIR-MSGS is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  inc PAPER .
  op final-paper : Mid Paper -> Contents .
 endom)

(omod FINAL-SUBMISSION-FORM is
  inc MOBILE-OBJECT-ADDITIONAL-DEFS .
  inc FINAL-SUBMISSION-FORM-AUTHOR-MSGS .
  inc FINAL-SUBMISSION-FORM-PROGR-CHAIR-MSGS .

  sort FinalSubFormState .
  ops inactive waiting towards-pc finished : -> FinalSubFormState .

  class FinalSubmissionForm | author : Mid,
                              state : FinalSubFormState,
                              paper : Paper,
                              program-chair : Mid .

  vars O O' O'' : Mid .
  vars P P' : Paper .
  var PI : Pid .
  var I : MachineInt .
  var Conf : Configuration .

  rl [start-up] :
     < tmp-id : FinalSubmissionForm | author : O',
          state : inactive, paper : P, program-chair : O'' >
     (to tmp-id : start-up(O)) Conf & none
     => < O : FinalSubmissionForm | author : O',
```

```
        state : waiting, paper : P, program-chair : O'' >
      Conf & (to O' : final-subm-form-gets-to-author from : O) .
```

When the final submission form gets the final version of the paper from the author, it goes towards the program chair.

```
  rl [get-final-version] :
     < O : FinalSubmissionForm | paper : P, state : waiting,
         program-chair : o(PI, I) >
     (to O : final-paper(P')) Conf & none
     => < O : FinalSubmissionForm | paper : P', state : towards-pc >
        Conf & go-find(o(PI,I), PI) .
```

Once the final submission form reaches the program chair, it sends the final paper to him and dies.

```
  rl  [send-final-version] :
     < O : FinalSubmissionForm | author : O', paper : P, state : towards-pc,
         program-chair : O'' > Conf & none
     => < O : FinalSubmissionForm | state : finished > Conf
         & (to O'' : final-paper(O', P)) .

  rl  [dead] :
     < O : FinalSubmissionForm | state : finished > Conf & none
     => none & kill .

 endom)
```

2.8 Report Form

```
(omod REPORT-FORM is
  inc REPORT-FORM-AUTHOR-MSGS .
  pr FINAL-SUBMISSION-FORM .
  pr SCORE .

  sort ReportFormState .
  ops inactive towards-author finished : -> ReportFormState .

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

  vars O O' O'' O''' : Mid .
  var PI : Pid .
  var I : MachineInt .
  var Conf : Configuration .
  var P : Paper .
  var Sc : Score .
```

When a report form is created, the name of the form object in its belly must be started up, initializing its name and putting it in the `towards-author` state.

```
  rl [start-up] :
     < tmp-id : ReportForm |
            paper : P, author : o(PI, I), score : Sc,
            state : (inactive).ReportFormState, program-chair : O' >
     (to tmp-id : start-up(O)) Conf & none
     => < O : ReportForm |
            paper : P, author : o(PI, I), score : Sc,
            state : towards-author, program-chair : O' >
        Conf & go-find(o(PI, I), PI) .
```

When an accepted paper report form reaches the author, it sends congratulations and spawns a final submission form, which will ask the author for the final version of the paper. After that, the final submission form dies.

```
  rl [send-congratulations] :
     < O : ReportForm | paper : P, author : O', score : accept,
        state : towards-author, program-chair : O'' >
        Conf & none
     => < O : ReportForm | state : (finished).ReportFormState > Conf
        & (to O' : congratulations)
               newo(up(FINAL-SUBMISSION-FORM),
                    < tmp-id : FinalSubmissionForm |
                       author : O', state : (inactive).FinalSubFormState,
                       paper : P, program-chair : O'' >,
                    tmp-id) .
```

When a rejected paper report form reaches the author, it sends regrets and dies.

```
  rl [send-regrets] :
     < O : ReportForm | author : O',
           score : reject, state : towards-author >
        Conf & none
     => < O : ReportForm | state : (finished).ReportFormState > Conf
         & (to O' : regrets) .

  rl [dead] :
     < O : ReportForm | state : (finished).ReportFormState > Conf & none
     => none & kill .

 endom)
```

2.9 Program Chair

The submission info that the program chair maintains consists of the name and address of the contact author, title of the paper, and the set of scores received so far for the given paper.

```
(view Set‘[Mid‘] from TRIV
     to SET[Mid] * (op __ : Set[Mid] Set[Mid] -> Set[Mid] to _._) is
  sort Elt to Set[Mid] .
 endv)
```

The so far received scores are kept in a set of tuples (Mid, Score). The module `SET-RESULT` defines several operations to work with these sets.

```
(view Result from TRIV
      to TUPLE(2)[Mid, Score] * (sort Tuple[Mid, Score] to Result) is
  sort Elt to Result .
 endv)

(fmod SET-RESULT is
  pr SET[Result] * (op __ : Set[Result] Set[Result] -> Set[Result] to _._) .
  pr SET-MID .
  pr SCORE .

  op size : Set[Result] -> MachineInt .
  op agreement : Set[Result] -> Bool .
  op all-equal : Score Set[Result] -> Bool .
  op reviewers : Set[Result] -> Set[Mid] .
  op decision : Set[Result] -> Score .
  op majority : Set[Result] -> Score .
```

```
  var R : Result .
  var RS : Set[Result] .
  vars O O' : Mid .
  vars Sc Sc' : Score .

  eq size(mt) = 0 .
  eq size(R . RS) = 1 + size(RS) .

  eq agreement(mt) = true .
  eq agreement((O, Sc) . RS) = all-equal(Sc, RS) .

  eq all-equal(Sc, mt) = true .
  eq all-equal(Sc, (O, Sc') . RS) = (Sc == Sc') and all-equal(Sc, RS) .

  eq reviewers(mt) = mt .
  eq reviewers((O, Sc') . RS) = O . reviewers(RS) .

  eq decision(mt) = accept .
  ceq decision((O, Sc) . RS) = Sc if agreement((O, Sc) . RS) .
  ceq decision(RS) = majority(RS) if not agreement(RS) .

  ceq majority(RS) = decision(RS) if agreement(RS) .
  ceq majority((O, reject) . (O', accept) . RS) = majority(RS)
      if not agreement((O, reject) . (O', accept) . RS) .
 endfm)

(view Set‘[Result‘] from TRIV to SET-RESULT is
  sort Elt to Set[Result] .
 endv)

(view SubmissionInfo from TRIV
      to TUPLE(5)[Name, Mid, Paper, Set‘[Result‘], MachineInt]
            * (sort Tuple[Name, Mid, Paper, Set‘[Result‘], MachineInt]
                    to SubmissionInfo) is
  sort Elt to SubmissionInfo .
 endv)

(omod PROGR-CHAIR is
  inc SUBMISSION-FORM-PROGR-CHAIR-MSGS .
  inc REVIEW-FORM-PROGR-CHAIR-MSGS .
  inc CONF-CHAIR-PROGR-CHAIR-MSGS .
  inc REVIEWER .
  pr SET[SubmissionInfo] .
  pr BAG[Score] .
  pr REPORT-FORM .
  pr BROADCAST .
  pr CHOOSE-REVIEWERS .

  class ProgrChair | conference-chair : Mid,
                     committee-members : Set[Mid],
                     number-of-reviewers : MachineInt,
                     submissions : Set[SubmissionInfo],
                     accepted : Set[Mid],
                     proceedings : Set[Paper] .
```

A program chair can also be an author and a reviewer.

```
  subclasses ProgrChair < Author Reviewer .

  var N : Name .
```

```
var P : Paper .
vars O O' O'' CM CM' : Mid .
vars OS OS' : Set[Mid] .
var Conf : Configuration .
var Subms : Set[SubmissionInfo] .
vars Sc Sc' : Score .
var PS : Set[Paper] .
vars I I' : MachineInt .
var MC : Contents .
var TS : Set[Result] .
```

When the program chair gets a submission then he/she assigns it to two committee members (hopefully, following some criteria, although here they are chosen nondeterministically; it could be two, three, or any other number of referees, or even a nonfix number). The program chair asks the submission form to create a review form for each assigned member.

```
rl [pc-gets-submission] :
   < O : ProgrChair | committee-members : OS, submissions : Subms,
                      number-of-reviewers : I >
   (to O : submission N O'' P from : O') Conf & none
   => < O : ProgrChair | submissions : ((N, O'', P, mt, 0) Subms) >
      Conf & (to O' : generate-review-forms(choose(OS, I))) .
```

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

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

If the reviews are in disagreement, the program chair declares the review forms unresolved unless three review rounds have been done yet.

```
crl [disagreement] :
   < O : ProgrChair | submissions : ((N, O', P, TS, I) Subms),
                      number-of-reviewers : I' >
   Conf & none
   => < O : ProgrChair | submissions : ((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 | submissions : ((N, O', P, TS, I) Subms),
                      accepted : OS,
                      number-of-reviewers : I' > Conf & none
   => < O : ProgrChair | submissions : 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).ReportFormState, program-chair : O >,
            tmp-id)
        broadcast(reviewers(TS), finish)
    if (size(TS) == I') and (agreement(TS) or (I == 2)) .
```

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.

```
  rl [get-final-versions] :
     < O : ProgrChair | proceedings : PS, accepted : O' . OS >
     (to O : final-paper(O', P))
     => < O : ProgrChair | proceedings : PS . P, accepted : OS > .
```

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.

```
  crl [send-proceedings] :
     < O : ProgrChair | proceedings : PS, submissions : mt, accepted : mt,
                        conference-chair : O' > Conf & none
     => < O : ProgrChair | proceedings : mt >
        Conf & (to O' : proceedings(PS))
     if PS =/= mt .

 endom)
```

2.10 Program committee members

```
(omod PROGRAM-COMMITTEE-MEMBER is
  inc REVIEWER .
  inc AUTHOR .
  class PCMember .
  subclass PCMember < Reviewer Author .
 endom)
```

2.11 Example

It is supposed all communications are signed and encrypted!

```
(omod CONF-REVIEW is
  inc MOBILE-MAUDE .
  pr CONF-CHAIR .
  pr PROGRAM-COMMITTEE-MEMBER .
  pr PROGR-CHAIR .

  op m# : Qid -> Module .
  op m# : Qid MobObjState -> Term .

  op initial : -> Configuration .
```

In this initial configuration there are one conference chair, one program chair, and four authors. Each of these objects is in a different process. These processes are distributed in three locations as follows:

```
  ***    'l0
  ***      p('l0, 0)
  ***        o(p('l0, 0), 0) ConfChair
  ***        o(p('l0, 0), 1) Reviewer
  ***      p('l0, 1)
```

```
***        o(p('l0, 1), 0) Author
***        o(p('l0, 1), 1) PCMember
***    'l1
***      p('l1, 0)
***        o(p('l1, 0), 0) Author
***      p('l1, 1)
***        o(p('l1, 1), 0) Author
***        o(p('l1, 1), 1) PCMember
***    'l2
***      p('l2, 0)
***        o(p('l2, 0), 0) ProgrChair
***      p('l2, 1)
***        o(p('l2, 1), 0) Author
***        o(p('l2, 1), 1) Reviewer
***      p('l2, 2)
***        o(p('l2, 2), 0) Reviewer

rl [initial-state] :
  initial
  => < 'l0 : R | cnt : 2 >

     < 'l1 : R | cnt : 2 >

     < 'l2 : R | cnt : 3 >

     < p('l0, 0) : P |
        cnt : 2,
        cf : < o(p('l0, 0), 0) : MO |
                   mod : up(CONF-CHAIR),
                   s : up(CONF-CHAIR,
                          < o(p('l0, 0), 0) : ConfChair |
                              hidden-gas : 300,
                              submission-form : null,
                              name : conf-chair,
                              paper : null,
                              subreviewers : o(p('l0, 0), 1),
                              done : mt,
                              mailing-list :
                                 (o(p('l0, 1), 0) .
                                  o(p('l1, 0), 0) .
                                  o(p('l1, 1), 0) .
                                  o(p('l2, 1), 0)),
                              program-chair : o(p('l2, 0), 0),
                              proceedings : mt >
                           (to o(p('l0, 0), 0) : start)
                          & none),
                   p : p('l0, 0),
                   gas : 300,
                   hops : 0,
                   mode : active >
              < o(p('l0, 0), 1) : MO |
                   mod : up(REVIEWER),
                   s : up(REVIEWER,
                          < o(p('l0, 0), 1) : Reviewer |
                              hidden-gas : 300,
                              submission-form : null,
                              name : reviewer1,
                              paper : null,
                              subreviewers : mt,
                              done : mt >
                          & none),
```

```
            p : p('l0, 0),
            gas : 300,
            hops : 0,
            mode : active >,
    guests : o(p('l0, 0), 0) . o(p('l0, 0), 1),
    forward : (0, (p('l0, 0), 0)) (1, (p('l0, 0), 0)) >

< p('l0, 1) : P |
    cnt : 2,
    cf : < o(p('l0, 1), 0) : MO |
            mod : up(AUTHOR),
            s : up(AUTHOR,
                   < o(p('l0, 1), 0) : Author |
                       hidden-gas : 300,
                       submission-form : null,
                       name : author1,
                       paper : p1 >
                   & none),
            p : p('l0, 1),
            gas : 300,
            hops : 0,
            mode : active >
         < o(p('l0, 1), 1) : MO |
            mod : up(PROGRAM-COMMITTEE-MEMBER),
            s : up(PROGRAM-COMMITTEE-MEMBER,
                   < o(p('l0, 1), 1) : PCMember |
                       hidden-gas : 300,
                       submission-form : null,
                       name : pc-member1,
                       paper : null,
                       subreviewers : o(p('l0, 0), 1),
                       done : mt >
                   & none),
            p : p('l0, 1),
            gas : 300,
            hops : 0,
            mode : active >,
    guests : o(p('l0, 1), 0) . o(p('l0, 1), 1),
    forward : (0, (p('l0, 1), 0)) (1, (p('l0, 1), 0)) >

< p('l1, 0) : P |
    cnt : 1,
    cf : < o(p('l1, 0), 0) : MO |
            mod : up(AUTHOR),
            s : up(AUTHOR,
                   < o(p('l1, 0), 0) : Author |
                       hidden-gas : 300,
                       submission-form : null,
                       name : author2,
                       paper : p2 >
                   & none),
            p : p('l1, 0),
            gas : 300,
            hops : 0,
            mode : active >,
    guests : o(p('l1, 0), 0),
    forward : (0, (p('l1, 0) ,0)) >

< p('l1, 1) : P |
    cnt : 2,
    cf : < o(p('l1, 1), 0) : MO |
```

```
                mod : up(AUTHOR),
                s : up(AUTHOR,
                       < o(p('l1, 1), 0) : Author |
                           hidden-gas : 300,
                           submission-form : null,
                           name : author3,
                           paper : p3 >
                       & none),
                p : p('l1, 1),
                gas : 300,
                hops : 0,
                mode : active >
           < o(p('l1, 1), 1) : MO |
                mod : up(PROGRAM-COMMITTEE-MEMBER),
                s : up(PROGRAM-COMMITTEE-MEMBER,
                       < o(p('l1, 1), 1) : PCMember |
                           hidden-gas : 300,
                           submission-form : null,
                           name : pc-member2,
                           paper : null,
                           subreviewers : o(p('l2, 1), 1),
                           done : mt >
                       & none),
                p : p('l1, 1),
                gas : 300,
                hops : 0,
                mode : active >,
    guests : o(p('l1, 1), 0) . o(p('l1, 1), 1),
    forward : (0, (p('l1, 1), 0)) (1, (p('l1, 1), 0)) >

< p('l2, 0) : P |
    cnt : 1,
    cf : < o(p('l2, 0), 0) : MO |
                mod : up(PROGR-CHAIR),
                s : up(PROGR-CHAIR,
                       < o(p('l2, 0), 0) : ProgrChair |
                           hidden-gas : 300,
                           conference-chair : o(p('l0, 0), 0),
                           committee-members :
                              (o(p('l0, 1), 1)
                               . o(p('l1, 1), 1)
                               . o(p('l2, 0), 0)),
                           number-of-reviewers : 3,
                           submissions : mt,
                           accepted : mt,
                           proceedings : mt,
                           submission-form : null,
                           name : program-chair,
                           paper : null,
                           subreviewers :
                              (o(p('l2, 2), 0) . o(p('l2, 1), 1)),
                           done : mt >
                       & none),
                p : p('l2, 0),
                gas : 300,
                hops : 0,
                mode : active >,
    guests : o(p('l2, 0), 0),
    forward : (0, (p('l2, 0) ,0)) >

< p('l2, 1) : P |
```

```
          cnt : 2,
          cf : < o(p('l2, 1), 0) : MO |
                    mod : up(AUTHOR),
                    s : up(AUTHOR,
                           < o(p('l2, 1), 0) : Author |
                               hidden-gas : 300,
                               submission-form : null,
                               name : author4,
                               paper : p4 >
                           & none),
                    p : p('l2, 1),
                    gas : 300,
                    hops : 0,
                    mode : active >
               < o(p('l2, 1), 1) : MO |
                    mod : up(REVIEWER),
                    s : up(REVIEWER,
                           < o(p('l2, 1), 1) : Reviewer |
                               hidden-gas : 300,
                               submission-form : null,
                               name : reviewer2,
                               paper : null,
                               subreviewers : o(p('l2, 2), 0),
                               done : mt >
                           & none),
                    p : p('l2, 1),
                    gas : 300,
                    hops : 0,
                    mode : active >,
          guests : o(p('l2, 1), 0) . o(p('l2, 1), 1),
          forward : (0, (p('l2, 1) ,0)) (1, (p('l2, 1) ,0)) >

       < p('l2, 2) : P |
          cnt : 1,
          cf : < o(p('l2, 2), 0) : MO |
                    mod : up(REVIEWER),
                    s : up(REVIEWER,
                           < o(p('l2, 2), 0) : Reviewer |
                               hidden-gas : 300,
                               submission-form : null,
                               name : reviewer3,
                               paper : null,
                               subreviewers : mt,
                               done : mt >
                           & none),
                    p : p('l2, 2),
                    gas : 300,
                    hops : 0,
                    mode : active >,
          guests : o(p('l2, 2), 0),
          forward : (0, (p('l2, 1) ,0)) > .

 endom)

***( Execution of the above example

Maude> (MO grew [28] [10] initial .)
rewrites: 42756748 in 1010390ms cpu (1019800ms real) (42317 rewrites/second)

Result Configuration : < 'l0 : R | hidden-gas : 10 , cnt : 2 >
< 'l1 : R | hidden-gas : 10 , cnt : 2 >
```

```
< 'l2 : R | hidden-gas : 10 , cnt : 3 >
< p ( 'l0 , 0 ) : P | hidden-gas : 10 , cf : (
 < o ( p ( 'l0 , 0 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( CONF-CHAIR ) , s
    : up ( CONF-CHAIR ,
 < o ( p ( 'l0 , 0 ) , 0 ) : ConfChair | hidden-gas : 44 , paper : null ,
    program-chair : o ( p ( 'l2 , 0 ) , 0 ) , submission-form : null , name :
    conf-chair , done : mt , subreviewers : o ( p ( 'l0 , 0 ) , 1 ) ,
    proceedings : ( final-version ( p1 ) . final-version ( p2 ) . final-version
    ( p4 ) ) , mailing-list : ( o ( p ( 'l0 , 1 ) , 0 ) . o ( p ( 'l1 , 0 ) , 0
    ) . o ( p ( 'l1 , 1 ) , 0 ) . o ( p ( 'l2 , 1 ) , 0 ) ) > & none ) , mode :
    active , hops : 0 , gas : 43 , p : p ( 'l0 , 0 ) >
 < o ( p ( 'l0 , 0 ) , 1 ) : MO | hidden-gas : 0 , mod : up ( REVIEWER ) , s :
    up ( REVIEWER ,
 < o ( p ( 'l0 , 0 ) , 1 ) : Reviewer | hidden-gas : 33 , paper : null ,
    submission-form : null , name : reviewer1 , done : p2 , subreviewers : mt >
    & none ) , mode : active , hops : 0 , gas : 32 , p : p ( 'l0 , 0 ) > ) ,
    forward : ( ( 0 , ( p ( 'l0 , 0 ) , 0 ) ) ( 1 , ( p ( 'l0 , 0 ) , 0 ) ) ) ,
    guests : ( o ( p ( 'l0 , 0 ) , 0 ) . o ( p ( 'l0 , 0 ) , 1 ) ) , cnt : 6 >
< p ( 'l0 , 1 ) : P | hidden-gas : 10 , cf : (
 < o ( p ( 'l0 , 1 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( AUTHOR ) , s : up
    ( AUTHOR ,
 < o ( p ( 'l0 , 1 ) , 0 ) : Author | hidden-gas : 42 , paper : final-version (
    p1 ) , submission-form : o ( p ( 'l0 , 0 ) , 2 ) , name : author1 > & none
    ) , mode : active , hops : 0 , gas : 41 , p : p ( 'l0 , 1 ) >
 < o ( p ( 'l0 , 1 ) , 1 ) : MO | hidden-gas : 0 , mod : up (
    PROGRAM-COMMITTEE-MEMBER ) , s : up ( PROGRAM-COMMITTEE-MEMBER ,
 < o ( p ( 'l0 , 1 ) , 1 ) : PCMember | hidden-gas : 45 , paper : null ,
    submission-form : null , name : pc-member1 , done : ( p1 . p2 . p3 . p4 ) ,
    subreviewers : o ( p ( 'l0 , 0 ) , 1 ) > & none ) , mode : active , hops :
    0 , gas : 44 , p : p ( 'l0 , 1 ) > ) , forward : ( ( 0 , ( p ( 'l0 , 1 ) ,
    0 ) ) ( 1 , ( p ( 'l0 , 1 ) , 0 ) ) ) , guests : ( o ( p ( 'l0 , 1 ) , 0 )
    . o ( p ( 'l0 , 1 ) , 1 ) ) , cnt : 3 >
< p ( 'l1 , 0 ) : P | hidden-gas : 10 , cf :
 < o ( p ( 'l1 , 0 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( AUTHOR ) , s : up
    ( AUTHOR ,
 < o ( p ( 'l1 , 0 ) , 0 ) : Author | hidden-gas : 42 , paper : final-version (
    p2 ) , submission-form : o ( p ( 'l0 , 0 ) , 3 ) , name : author2 > & none
    ) , mode : active , hops : 0 , gas : 41 , p : p ( 'l1 , 0 ) > , forward : (
    0 , ( p ( 'l1 , 0 ) , 0 ) ) , guests : o ( p ( 'l1 , 0 ) , 0 ) , cnt : 2 >
< p ( 'l1 , 1 ) : P | hidden-gas : 10 , cf : (
 < o ( p ( 'l1 , 1 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( AUTHOR ) , s : up
    ( AUTHOR ,
 < o ( p ( 'l1 , 1 ) , 0 ) : Author | hidden-gas : 40 , paper : rejected ( p3 )
    , submission-form : o ( p ( 'l0 , 0 ) , 4 ) , name : author3 > & none ) ,
    mode : active , hops : 0 , gas : 39 , p : p ( 'l1 , 1 ) >
 < o ( p ( 'l1 , 1 ) , 1 ) : MO | hidden-gas : 0 , mod : up (
    PROGRAM-COMMITTEE-MEMBER ) , s : up ( PROGRAM-COMMITTEE-MEMBER ,
 < o ( p ( 'l1 , 1 ) , 1 ) : PCMember | hidden-gas : 45 , paper : null ,
    submission-form : null , name : pc-member2 , done : ( p1 . p2 . p3 . p4 ) ,
    subreviewers : o ( p ( 'l2 , 1 ) , 1 ) > & none ) , mode : active , hops :
    0 , gas : 44 , p : p ( 'l1 , 1 ) > ) , forward : ( ( 0 , ( p ( 'l1 , 1 ) ,
    0 ) ) ( 1 , ( p ( 'l1 , 1 ) , 0 ) ) ) , guests : ( o ( p ( 'l1 , 1 ) , 0 )
    . o ( p ( 'l1 , 1 ) , 1 ) ) , cnt : 2 >
< p ( 'l2 , 0 ) : P | hidden-gas : 10 , cf :
 < o ( p ( 'l2 , 0 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( PROGR-CHAIR ) , s
    : up ( PROGR-CHAIR ,
 < o ( p ( 'l2 , 0 ) , 0 ) : ProgrChair | hidden-gas : 97 , paper : null ,
    submission-form : null , name : program-chair , done : ( p1 . p2 . p3 . p4
    ) , subreviewers : ( o ( p ( 'l2 , 1 ) , 1 ) . o ( p ( 'l2 , 2 ) , 0 ) ) ,
    proceedings : mt , accepted : mt , submissions : mt , committee-members : (
    o ( p ( 'l0 , 1 ) , 1 ) . o ( p ( 'l1 , 1 ) , 1 ) . o ( p ( 'l2 , 0 ) , 0 )
```

```
    ) , conference-chair : o ( p ( '10 , 0 ) , 0 ) , number-of-reviewers : 3 >
    & none ) , mode : active , hops : 0 , gas : 96 , p : p ( '12 , 0 ) > ,
    forward : ( 0 , ( p ( '12 , 0 ) , 0 ) ) , guests : o ( p ( '12 , 0 ) , 0 )
    , cnt : 17 >
< p ( '12 , 1 ) : P | hidden-gas : 10 , cf : (
 < o ( p ( '12 , 1 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( AUTHOR ) , s : up
    ( AUTHOR ,
 < o ( p ( '12 , 1 ) , 0 ) : Author | hidden-gas : 42 , paper : final-version (
    p4 ) , submission-form : o ( p ( '10 , 0 ) , 5 ) , name : author4 > & none
    ) , mode : active , hops : 0 , gas : 41 , p : p ( '12 , 1 ) >
 < o ( p ( '12 , 1 ) , 1 ) : MO | hidden-gas : 0 , mod : up ( REVIEWER ) , s :
    up ( REVIEWER ,
 < o ( p ( '12 , 1 ) , 1 ) : Reviewer | hidden-gas : 41 , paper : null ,
    submission-form : null , name : reviewer2 , done : p2 , subreviewers : o (
    p ( '12 , 2 ) , 0 ) > & none ) , mode : active , hops : 0 , gas : 40 , p :
    p ( '12 , 1 ) > ) , forward : ( ( 0 , ( p ( '12 , 1 ) , 0 ) ) ( 1 , ( p (
    '12 , 1 ) , 0 ) ) ) , guests : ( o ( p ( '12 , 1 ) , 0 ) . o ( p ( '12 , 1
    ) , 1 ) ) , cnt : 3 >
< p ( '12 , 2 ) : P | hidden-gas : 10 , cf :
 < o ( p ( '12 , 2 ) , 0 ) : MO | hidden-gas : 0 , mod : up ( REVIEWER ) , s :
    up ( REVIEWER ,
 < o ( p ( '12 , 2 ) , 0 ) : Reviewer | hidden-gas : 33 , paper : null ,
    submission-form : null , name : reviewer3 , done : p2 , subreviewers : mt >
    & none ) , mode : active , hops : 0 , gas : 32 , p : p ( '12 , 2 ) > ,
    forward : ( 0 , ( p ( '12 , 1 ) , 0 ) ) , guests : o ( p ( '12 , 2 ) , 0 )
    , cnt : 1 >

)
```

References

[1] L. Cardelli. Abstractions for mobile computations. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[2] 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, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2000.

[3] F. Durán and A. Verdejo. Conference reviewing system in Mobile Maude. Submitted for publication.