A distributed implementation of Mobile Maude ⋆

Francisco Durán, [a] Adrián Riesco, [b] and Alberto Verdejo [b]

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

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

Abstract

We present a new specification/implementation of the mobile agent language Mobile Maude. This new version uses the external *sockets* provided by Maude since its 2.2 version, thus obtaining a really distributed implementation of the mobile language, where messages and mobile objects now may travel from one machine to another one in a transparent way. We also show how, even though the complexity of the Mobile Maude specification and the use of reflection, we have managed to use the Maude's model checker to prove properties about mobile agents applications.

Keywords: Mobile agents, Mobile Maude, model checking.

1 Introduction

Mobile Maude is a mobile agent language extending Maude and supporting mobile computation. It was first presented in [3], and a significant application appeared in [4].

Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances of mobile agent behavior. The formal semantics of Mobile Maude is given by a rewrite theory in rewriting logic. Since this specification is executable, it can be used as a prototype of the language, in which mobile agent systems can be simulated. The two key notions are *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects have their own code, can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. Mobile Maude's key characteristics include: (1) reflection as a

⋆ Research partially supported by the MCyT Spanish projects *MIDAS* (TIC 2003-01000) and TIN2005-09405-C02-01.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

way of endowing mobile objects with "higher-order" capabilities; (2) object-orientation and asynchronous message passing; and (3) a simple semantics without loss in the expressive power of application code.

The code of a mobile object is given by (the metarepresentation of) an object-based 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. Maude configurations become located computational environments where mobile objects can reside. Mobile objects can interact with other ones (possibly in different locations), and can move from one location to another.

In [3], Durán, Eker, Lincoln, and Meseguer first introduced Mobile Maude. In that work, the authors presented a 'simulator' of Mobile Maude, an executable Maude specification on top of Maude 1.0.5, in which the system code was written entirely in Maude, and thus locations and processes were encoded as Maude terms. In the same paper, the authors also gave a development plan including two development efforts: a first step in which a single-host executable was implemented, and a second implementation effort focussing on true distributed execution.

The release of Maude 2.0 allowed taking the first step. This implementation effort was completed in a very short time, utilizing the builtin object system, for object/message fairness, just by simplifying and extending the previous specification. This new version was developed by Durán and Verdejo, and used in several examples, one of which was reported in [4].

The present work summarizes our results in the second development effort. The built-in string handling and internet socket module available in Maude 2.2 has allowed us to build a really distributed implementation. The Maude 2.2 socket modules support non-blocking client and server TCP sockets (at the OS level). In this implementation effort, a Mobile Maude server runs on top of a Maude interpreter and performs the following tasks: keeps track of the current locations of mobile objects created on a host, handles change of location messages, reroutes messages to mobile objects, and runs the code of mobile objects by invoking the metalevel. In fact, we have made a quite significant number of changes on Mobile Maude. Processes and locations are no longer part of the specification of Mobile Maude, now we talk about Maude processes—not terms, OS processes, which may be running on different machines—and IP addresses. We have also introduced the notion of *root objects* as managers of the configurations of mobile objects in the different processes.

We explain below the design of processes and mobile objects and their rewriting semantics, based on a formal specification of Mobile Maude written in Maude.

The fundamental notions of Mobile Maude, namely processes, mobile objects, and messages are introduced in Section 2. In Section 3, we give a flavor of the rewriting semantics of Mobile Maude. In Section 4 we discuss on the

connections via sockets between the different processes in a distributed configuration; in particular, we introduce Maude sockets, we explain buffered sockets and then introduce a very simple sample architecture. Section 5 presents a Mobile Maude application code example in which we specify the search of the best offer between several distributed alternatives. Section 6 explains how we have used the model checker to check properties on our Mobile Maude specifications. Section 7 wraps this piece of work with some final conclusions.

2 Processes, mobile objects, and messages

The key entities in Mobile Maude are *processes* and *mobile objects*. Mobile objects are modeled as distributed objects in the class `MobileObject`. A *distributed configuration* is made up of located configurations. Each located configuration is executed in a Maude process. Such processes can therefore be seen as *located* computational environments *inside which* mobile objects can reside, execute, and send and receive messages to and from other mobile objects located in different processes. We assume that each located configuration has one (and only one) *root object*, of class `RootObject`, which keeps information on the location of the process, on the mobile objects in such a configuration, and on the whereabouts of the mobile objects created in it, which may have moved to other processes. We assume uniqueness of root object names in a distributed configuration.

Mobile objects carry their own internal state and code (rewrite rules) with them, can move from one process to another, and can communicate with each other by asynchronous message passing. The *names* of root objects range over the sort `Loc`, and have the form `l(IP, N)` with `IP` the IP address of the machine in which the process is being executed and `N` a number. The names of mobile objects range over the sort `Mid` and have the form `o(L, N)` with `L` the name of the root object of the process in which it was created and `N` a number. Figure 1 shows several mobile objects in two processes, with (mobile) object `o(l(IP, 0), 1)` moving from the process with root object `l(IP, 0)` to the process of root object `l(IP', 0)`, and with object `o(l(IP, 0), 0)` sending a message to `o(l(IP', 0), 0)`.

Mobile objects are specified as objects of the following class `MobileObject`:[1]

```
class MobileObject |
  mod : Module,                *** rewrite rules of the mobile object
  s : Term,                    *** current state
  gas : Nat,                   *** bound on resources
  hops : Nat,                  *** number of hops
  mode : Mode .                *** objects in motion cannot be active
```

[1] We use here the Full Maude object-oriented notation for defining classes. However, the actual implementation of Mobile Maude is made in Core Maude, because Full Maude does not support external objects. The complete code for Mobile Maude including the corresponding declarations in Core Maude for the classes `MobileObject` and `RootObject` can be found in http://maude.sip.ucm.es/mobilemaude.

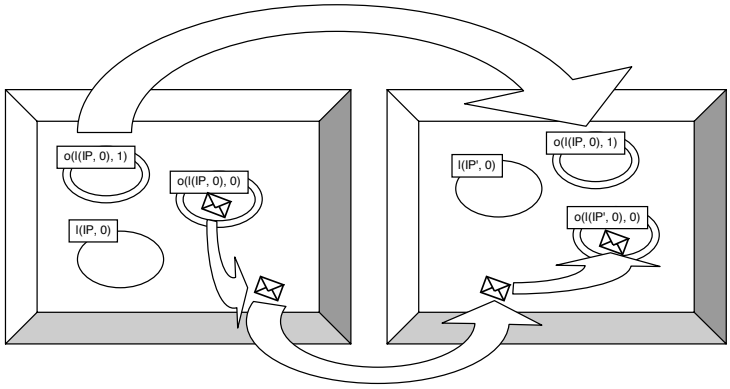

Fig. 1. Object and message mobility.

The sorts `Module` and `Term`, associated to the attributes `mod` and `s`, respectively, are sorts in the module `META-LEVEL`. The mobile object's *module* must be object-based, and the mobile object's *state* must be the metarepresentation of a pair of configurations meaningful for that module and having the form `C & C'`, where `C` is a configuration of objects and messages—unprocessed incoming messages and inter-inner-objects messages, see below—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 sometimes refer to this object as the *main* one, which in most cases will be the only one. Therefore, we can think of a mobile object as a *wrapper* that encapsulates the state and code of its inner object and mediates its communication with other objects and its mobility. For this reason, Figure 1 depicts mobile objects by two concentric circles, with the inner object and its incoming and outgoing messages contained in the inner circle.

To maintain the forwarding information up to date (see below), the definition of the class `MobileObject` includes the attribute `hops`, which stores the number of "hops" from one process to another. To guarantee that all mobile objects eventually have some activity, and as a bound on the resources they can consume, they have a `gas` attribute. Finally, an object's `mode` is only `active` inside the belly of a process: on-transit objects are `idle`.

The class `RootObject` of root objects is declared as follows:

```
class RootObject |
  cnt : Nat,                    *** counter to generate mobile obj. names
  guests : Set{Oid},            *** objects in the location
  forward : Map{Nat, Tuple{Loc, Nat}},    *** forwarding information
  state : RootObjectState,      *** idle, waiting-connection, or active
  neighbors : Map{Loc, Oid},    *** associates a socket to each location
  defNeighbor : Default{Oid} . *** default socket
```

We assume that each located configuration contains one and only one root object, plus the messages and mobile objects *currently residing* in such a

process. Located configurations, running on different Maude processes, make up a distributed configuration. Mobile objects can *move* from one process (located configuration) to another.

The root object of each process keeps information about the mobile objects currently in it in the `guests` attribute. Mobile objects are named with identifiers of the form `o(L, N)`. The attribute `cnt` stores a counter to generate such unique new mobile object names. Since mobile objects may move from one process to another, reaching them by messages is nontrivial. The solution adopted in Mobile Maude [3] is that, when a message's addressee is not in the current process, the message is forwarded to the addressee's parent process (the process it was created at). Each root object stores forwarding information about the whereabouts of its children in its `forward` attribute, a partial function in `Map{Nat, Tuple{Loc, Nat}}` that maps child number n to a pair consisting of the name of the located process in which the object currently resides, and the number of "hops" to different processes that the mobile object has taken so far. The number of hops is important in disambiguating situations when old messages (containing old location information) arrive after newer messages containing current location. The most current location is that associated with the largest number of hops. Whenever a mobile object moves to a new process, the object's parent process is always notified. Note that this mechanism does not guarantee message delivery in the case that objects move more rapidly than messages.

In the previous version of Mobile Maude [3,4], all the processes were in the same Maude object-oriented configuration, and reaching a particular process was represented by one single rule. However, in this new version, when a mobile object moves to a different location, or a message is sent to a mobile object in a different location, since we use TCP sockets to connect processes, we need to know which of the sockets must be used to send the information. The root object in the process is in charge of sending it through the appropriate socket. [2] Assuming that all processes are directly connected to each other is not realistic, would be very limited in the number of processes we could connect, and would make the task of connecting a new process a really expensive one. Fortunately, connectivity between two nodes does not necessarily imply a direct connection between them. An indirect connectivity may be achieved among a set of cooperating nodes. Nevertheless, just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the nodes of the network between them—switchers and routers—use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward

[2] As we will see in the coming sections, root objects send messages through buffered sockets. We discuss the used of sockets and buffered sockets in Section 4.

messages toward the destination node based on its address—which is usually called *routing*—is nontrivial.[3] Here, we assume a very simple, although quite general, approach consisting in having a routing table in each root object. Such a table gives the socket through which a message must be sent if one wants to reach a particular location. If there is a socket between the source and the target of the message then it reaches its destination in a single step; otherwise the forwarding have to be repeated several times. The `neighbors` attribute maintains such a routing table as a map associating socket object identifiers to location identifiers. That is, the attribute `neighbors` stores in a partial function `Map{Loc, Oid}` information on the sockets through which sending the data to reach a particular location.

In case there is no socket associated to a particular location in the map `neighbors`, there can be a default socket stored in the attribute `defNeighbor`. Nevertheless, the value of the `defNeighbor` attribute may also be unspecified. The sort `Default{X}` declared in the module `DEFAULT-ELEMENT` below adds a default value to the sort used in the instantiation of the module. We define the parameterized functional module `DEFAULT-ELEMENT{X :: TRIV}` in which we declare a sort `Default{X}` as a supersort of the sort `Elt` of the parameter theory, and a constant `null` of sort `Default{X}`.

```
fmod DEFAULT-ELEMENT{X :: TRIV} is
  sort Default{X} .
  subsort X$Elt < Default{X} .
  op null : -> Default{X} [ctor] .
endfm
```

Then, since `defNeighbor` is declared of sort `Default{Oid}`, it can take as value either an object identifier or `null`.

If there is no socket associated to a particular location and a default one has not been specified then the data is not delivered. Note that this model allows us to represent many different network architectures, and, although we do not care here about it, the routing information may be updated and used in a very flexible way. We will explain how to build a very simple architecture in Section 4.2.

Finally, a root object may be in state `idle`, `waiting-connection`, or `active`. The attribute `state` will take one of these values. Root objects are only idle when they are created, being their first action either being activated as a client or server socket. They stay in `waiting-connection` until they get the confirmation from the server socket, passing then to `active` mode, state in which they will develop their normal activity.

Mobile Maude *system code* is specified by a relatively small number of rules (about 40) for root objects, mobile objects, mobility, and message passing.

[3] We only consider the case of a source node wanting to send a message to a single destination node (*unicast*). The cases of *multicasting*—the source node wants to send a message to some subset of the nodes on the network—and *broadcasting*—the source node wants to send a message to all the nodes on the network—could similarly be specified.

Such rules work in an *application-independent* way. Application code, on the other hand, can be written as Maude object-based modules with great freedom, except for being aware that, as explained in Section 2, the top level of the state of a mobile object has to be a pair of configurations, with the second component containing outgoing messages and the first containing the inner object(s) and incoming messages.

```
sort MobObjState .
op _&_ : Configuration Configuration -> MobObjState [ctor] .
```

The messages being pulled in or out of a mobile object must be of the form `to O : C`, `go(L)`, `go-find(O, L)`, `newo(Mod, Conf, O)`, or `kill`, for `L` a location (of sort `Loc`), `O` a mobile object identifier (of sort `Mid`), `C` a term of sort `Contents`, `Mod` a term of sort `Module`, and `Conf` a term of sort `Configuration`. Such messages may in fact be understood as commands that the object—or one of the objects—in the inner configuration of a mobile object gives to it. Thus, an object may send a message with contents `C` to the object `O` with the message `to O : C`; may request to move from its current location to a given location `L` with the `go(L)` message; may request going to the location in which the mobile object `O` resides, which is possibly `L`, with the message `go-find(O, L)`; may request creating a new mobile object with module `Mod`, initial state `Conf`, and temporal identifier of the main object in such a configuration `O`, with the message `newo(Mod, Conf, O)`; or may request the destruction of the mobile object it resides into with the message `kill`. The definition of all these ingredients are defined in the module `MOBILE-OBJECT-ADDITIONAL-DEFS`, which is assumed to be imported by the user in all his Mobile Maude applications.

Note that messages being sent to other mobile objects must be of the form `to_:_`, with the addressee of the message as first argument and a term of sort `Contents` as second argument. The definition of such a sort is left to each particular application (see Section 5), which in fact let the user the freedom to define any kind of message, with the restriction of having the identifier of the addressee as first argument.

3 Mobile Maude's rewriting semantics

The entire semantics of Mobile Maude can be defined by a relatively small number of rewrite rules written in Maude. We should think of such rules as an implementation/specification of the *system code* of Mobile Maude, that operates in an application-independent way providing all the object creation and destruction, message passing, and object mobility primitives.

We give the flavor of Mobile Maude's rewriting semantics by commenting on some of its rules. In particular, we focus on the rules in charge of delivering inter-object messages since, in addition to illustrating the general approach (a more detailed discussion may be found in [3] and [4]), it is directly related to the main novelty in the new implementation: sockets. The complete specification, including other rules in the same style can be found in

http://maude.sip.ucm.es/mobilemaude.

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 such 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 through messages of the form `to_:_`, as explained above. 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 then send the message to the addressee object. First the message is pulled out of the object's outgoing tray.

```
rl [message-out-to] :
  < O : V@MobileObject |
      mod : MOD, s : '_&_[T, 'to_:_[T', T'']], mode : active, AtS >
  => < O : V@MobileObject |
       mod : MOD, s : '_&_[T, 'none.Configuration], mode : active, AtS >
     (to downTerm(T', o(1("null", 0), 0)) { T'' }) .
```

Once the message is out of the mobile object, it can be appropriately delivered. The `msg-send` rule below redirects messages addressed to mobile objects in different locations.

```
crl [msg-send] :
  < L : V@RootObject  | state : active, guests : OS, forward : F, AtS >
  (to o(L, N) { T })
  => < L : V@RootObject  | state : active, guests : OS, forward : F, AtS >
     Send(p1(F[N]), L, to o(L, N) hops p2(F[N]) in p1(F[N]) { T })
  if (p1(F[N]) =/= L) /\ (not o(L, N) in OS) .
```

Notice the use of the message

```
op Send : Oid Oid Msg -> Msg [ctor msg] .
```

to send messages to the appropriate locations. The first and second arguments of the `Send` message are, respectively, the addressee and sender of the message, and the third argument is the message being sent. We will see in Section 4 how the `Send` messages will be used to send the corresponding data through the appropriate sockets.

The arrival of an inter-object message to a location where the addressee object is, is handled by the following rule. The message is just put in the location so the object can get it.

```
rl [msg-arrive-to-loc] :
  to o(L, N) hops H in L' { T' }
  < L' : V@RootObject | state : active, guests : (o(L, N), OS), AtS >
  => < L' : V@RootObject | state : active, guests : (o(L, N), OS), AtS >
     to o(L, N) { T' } .
```

Once the message reaches its addressee object, the message must be in-

serted in—*push into*—the state of such a mobile object. To make sure that the mobile object will remain in a valid state, we check that the metarepresentation of the corresponding message is a valid message in the module of the object.

```
rl [msg-in] :
  to O { T }
  < O : V@MobileObject | mod : MOD, s : '_&_[T', T''], AtS >
  => if sortLeq(MOD, leastSort(MOD, 'to_:_[upTerm(O), T]), 'Msg)
        or sortLeq(MOD, 'Msg, leastSort(MOD, 'to_:_[upTerm(O), T]))
     then < O : V@MobileObject |
           mod : MOD, s : '_&_['__['to_:_[upTerm(O), T], T'], T''], AtS >
     else < O : V@MobileObject | mod : MOD, s : '_&_[T', T''], AtS >
     fi .
```

4 Socket handling

Maude 2.2 supports rewriting with external objects and an implementation of sockets as the first such external object. Rewriting with external objects is started by the command `erewrite` (abbreviated `erew`) which is like `frewrite` except it allows messages to be exchanged with external objects that do not reside in the configuration.

Sockets are accessed using the messages declared in the module `SOCKET`, which can be found in the file `socket.maude` distributed with Maude. We briefly describe here Maude sockets. For a complete explanation of Maude sockets, their use, and examples, we refer the reader to the Maude manual [2]. Currently only IPv4 TCP sockets are supported; other protocol families and socket types may be added in the future.

The external object named by the constant `socketManager` is a factory for socket objects. To create a client socket, a message `createClientTcpSocket (socketManager, ME, ADDRESS, PORT)` has to be sent to the `socketManager`, where `ME` is the name of the object the reply should be sent to, `ADDRESS` is the name of the server you want to connect to, and `PORT` is the port you want to connect to (say 80 for HTTP connections). The reply will be the message `createdSocket(ME, socketManager, SOCKET-NAME)` where `SOCKET-NAME` is the name of the newly created socket. All errors on a client socket are handled by closing the socket.

You can then send data to the server with a message `send(SOCKET-NAME, ME, DATA)` which elicits the message `sent(ME, SOCKET-NAME)`. Similarly you can receive data from the server with a message `receive(SOCKET-NAME, ME)` which elicits the message `received(ME, SOCKET-NAME, DATA)`.

To have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port. To create a server socket, you send `socketManager` a message

```
createServerTcpSocket(socketManager, ME, PORT, BACKLOG)
```

where `PORT` is the port number and `BACKLOG` is the number of queue requests

for connection that you will allow. The response is the message

```
createdSocket(ME, socketManager, SERVER-SOCKET-NAME).
```

Here `SERVER-SOCKET-NAME` refers to a server socket. The only thing you can do with a server socket is to accept clients, by means of the message

```
acceptClient(SERVER-SOCKET-NAME, ME)
```

which elicits the message

```
acceptedClient(ME, SERVER-SOCKET-NAME, ADDRESS, NEW-SOCKET-NAME).
```

Here `ADDRESS` is the originating address of the client and `NEW-SOCKET-NAME` is the name of the socket you use to communicate with that client. This new socket behaves just like a client socket for sending and receiving.

As we have seen in Section 3, the specification of Mobile Maude does not know about sockets. The only place where we get close to sockets is when using the `Send` messages, which is in fact not a socket message, but a *buffered socket* one. We introduce in Section 4.1 buffered sockets, a kind of filter class that makes Mobile Maude independent of sockets at the same time it adds some additional functionality. As we will see in Section 6, this independence is precisely what allows us to model check Mobile Maude specifications in a rather clean way. Section 4.2 talks about the architecture of the systems, on how processes get connected, and show how to do it for a very simple architecture.

4.1 Buffered sockets

TCP sockets do not preserve message boundaries. Thus, sending e.g. messages "ONE" and "TWO" might result in the reception of messages "ON" and "ETWO". Although not relevant in other applications, in the current case we need to guarantee that messages are received as originally sent; for instance, if a mobile object is sent through a socket, we need to be able to recover a valid object, in the same valid state in which it was sent, upon the reception of the message. To guarantee message boundaries we use a filter class `BufferedSocket`, defined in the module `BUFFERED-SOCKET`. This module is completely independent of Mobile Maude, and can therefore be used in other applications. We interact with buffered sockets in the same way we interact with sockets, with the only difference that all messages in the module `SOCKET` have been capitalized to avoid the confusion, being the boundary control completely transparent to the user.

When a buffered socket is created, in addition to the socket object through which the information will be sent, a `BufferedSocket` object is also created on each side of the socket (one in each one of the configurations between which the communication is established). All messages sent through a buffered socket are manipulated before they are sent through the socket underneath. When a message is sent through a buffered socket, a mark is placed at the end of it; the `BufferedSocket` object at the other side of the socket stores all messages

received on a buffer, in such a way that when a message is requested the marks placed say which part of the information received must be given as the next message.

An object of class `BufferedSocket` has three attributes: `read`, of sort `String`, which stores the messages read, `bState`, which indicates whether the filter is `idle` or `active`, and `waiting`, which indicates if we are waiting for a `sent` message (when we are waiting, we do not allow sending new messages).

```
sort BState .
ops idle active : -> BState [ctor] .
class BufferedSocket | read : String, bState : BState, waiting : Bool .
```

We do not give here all the rules, but only those related to the sending of messages.

Once a connection has been established, and a `BufferedSocket` object has been created on each side, messages can be sent and received. When a `Send` message is received, the buffered socket sends a `send` message with the same data plus a mark [4] to indicate the end of the message.

```
rl [send] :
  < b(SOCKET) : V@BufferedSocket | bState : active,
                                   waiting : false, Atts >
  Send(b(SOCKET), 0, DATA)
  => < b(SOCKET) : V@BufferedSocket | bState : active,
                                      waiting : true, Atts >
     send(SOCKET, 0, DATA + "#") .
```

The key is then in the reception of messages. A `BufferedSocket` object is always listening to the socket. It sends a `receive` message at start up and puts all the received messages in its buffer. Notice that a buffered socket goes from `idle` to `active` in the `buffer-start-up` rule. A `Receive` message is then handled if there is a complete message in the buffer, that is, there is a mark on it, and results in the reception of the first message in the buffer, which is removed from it.

```
rl [buffer-start-up] :
  < b(SOCKET) : V@BufferedSocket | bState : idle, Atts >
  => < b(SOCKET) : V@BufferedSocket | bState : active, Atts >
     receive(SOCKET, b(SOCKET)) .

rl [received] :
  < b(SOCKET) : V@BufferedSocket | bState : active, read : S, Atts >
  received(b(SOCKET), 0, DATA)
  => < b(SOCKET) : V@BufferedSocket | bState : active,
                                      read : (S + DATA), Atts >
     receive(SOCKET, b(SOCKET)) .

crl [Received] :
  < b(SOCKET) : V@BufferedSocket | bState : active, read : S, Atts >
  Receive(b(SOCKET), 0)
```

[4] In the rules we use the string `"#"` as mark, but any other could be used. Note that the user data sent through the sockets should not contain such a mark.

```
   => < b(SOCKET) : V@BufferedSocket | bState : active, read : S', Atts >
      Received(0, b(SOCKET), DATA)
   if N := find(S, "#", 0) /\ DATA := substr(S, 0, N)
      /\ S' := substr(S, N + 1, length(S)) .
```

4.2 A client/server architecture

Although the specification of Mobile Maude presented in the previous sections allows different configurations of processes, we present here a very simple client/server architecture. We distinguish clients and servers by declaring two subclasses `ServerRootObject` and `ClientRootObject` of `RootObject`, with no additional attributes, although with different behavior.

```
class ClientRootObject .
class ServerRootObject .
subclasses ClientRootObject ServerRootObject < RootObject .
```

The architecture we present here consists in a process with a server root object, and several processes with client root objects. The server is connected to all clients, and a client is connected only to the server. If a mobile object residing in a client process—a process with a client root object in it—wants to move to (or send a message to a mobile object in) another client process, then it will be sent to the server process, and from there to its final destination. That is, we have a very simple star network, with a server root object in the middle redirecting all messages.

When a `ServerRootObject` is created it send an `AcceptClient` message indicating that it is ready to accept clients through the server socket. When a `ClientRootObject` is created it first tries to establish a connection with the sever by sending a `CreateClientTcpSocket` message. In the rule `acceptedClient` below, in addition to sending messages `AcceptClient` and `Receive` indicating, respectively, that it is ready to accept new clients through the server socket, and messages through the new socket, the server root object that gets the `AcceptedClient` message sends a start-up message `new-socket` communicating its identifier. Notice that the client knows the address and port of the server root object, but not its identity. In this first message the server sends its name to its client, allowing to this one establishing the association between the socket and the identity of the object in it.

```
rl [acceptedClient] :
  < l(IP, N) : V@ServerRootObject | state : active, AtS >
  AcceptedClient(l(IP, N), SOCKET, IP', NEW-SOCKET)
  => < l(IP, N) : V@ServerRootObject | state : active, AtS >
     AcceptClient(SOCKET, l(IP, N))
     Receive(NEW-SOCKET, l(IP, N))
     Send(NEW-SOCKET, l(IP, N), msg2string(new-socket(l(IP, N)))) .
```

Since the third argument of a `Send` message is a `String`, the message being sent is transformed with the `msg2string` function; `string2msg` does the inverse transformation.

The response to a client root object's socket connection request is handled by the following rule `connected`, where a client also sends a `new-socket` message right after the socket is created.

```
rl [connected] :
  < l(IP, N) : V@ClientRootObject | state : waiting-connection, AtS >
  CreatedSocket(O, SOCKET-MANAGER, SOCKET)
  => < l(IP, N) : V@ClientRootObject | state : active, AtS >
     Receive(SOCKET, l(IP, N))
     Send(SOCKET, l(IP, N), msg2string(new-socket(l(IP, N)))) .
```

The attributes `neighbors` and `defNeighbor` are key for sending messages through the appropriate sockets. The reason why the first message sent through a socket after its creation is the message `new-socket` is to initialize these attributes. When it is received, depending on whether the receiver is a client or a server, and whether there is already a default neighbor or not, one or another action is taken.

To avoid unintended loops in the delivering of messages, we assume that server root objects do not have default neighbors. For clients, the first connection is made the default one.

```
crl [Received] :
  < O : V@RootObject | state : active, neighbors : empty,
                       defNeighbor : null, AtS >
  Received(O, SOCKET, DATA)
  => < O : V@RootObject | state : active,
         neighbors : insert(L, SOCKET, empty),
         defNeighbor : if V@RootObject == ServerRootObject
                       then null
                       else SOCKET
                       fi,
         AtS >
     Receive(SOCKET, O)
  if new-socket(L) := string2msg(DATA) .

crl [Received] :
  < O : V@RootObject | state : active, neighbors : LSPF, AtS >
  Received(O, SOCKET, DATA)
  => < O : V@RootObject | state : active,
                          neighbors : insert(L, SOCKET, LSPF), AtS >
     Receive(SOCKET, O)
  if LSPF =/= empty /\ new-socket(L) := string2msg(DATA) .
```

If not a `new-socket` message, then the message is just left in the configuration.

```
crl [Received] :
  < O : V@RootObject | state : active, AtS >
  Received(O, SOCKET, DATA)
  => < O : V@RootObject | state : active, AtS >
     string2msg(DATA) Receive(SOCKET, O)
  if not new-socket(DATA) .
```

5 A buying printers example

In this section we present a simple application to illustrate how mobile *application code* can be written in Maude and can be wrapped in mobile objects. In this example we have printers, buyers, and sellers; a buyer agent visits several printer sellers which provide him information on their printers. The buyer looks for the cheapest printer, and once he has visited all the sellers, he goes back to the location of the seller offering the best price.

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 our sample application we have two different classes of mobile objects: sellers and buyers. A buyer visits several sellers. The buyer asks each seller he visits for the description of the seller's printer (represented here only by its price). The seller sends back this information, which the buyer keeps if it corresponds to a better (cheaper) printer. Otherwise he discards it. Once the buyer has visited all the sellers he knows, he goes back to the location of the best offer.

We represent sellers and buyers as objects of respective classes `Seller` and `Buyer`. Such objects in the application code will then be embedded as *inner objects* of their corresponding mobile objects.

The class `Seller` has a single attribute `description` with the printer price (a natural number).

```
class Seller | description : Nat .
```

Sellers receive messages of the form `get-printer-price(B)`, with `B` the identifier of the buyer mobile object sending the message. A seller can send messages of the form `printer-price(N)`, with `N` a natural number representing the printer's price. Both are defined of sort `Contents`, declared in the module `MOBILE-OBJECT-ADDITIONAL-DEFS`.

```
op get-printer-price : Mid -> Contents .
op printer-price : Nat -> Contents .
```

A seller's behavior is represented by the following single rewrite rule: when a seller receives a description (price) request, it sends the description back to the buyer.

```
rl [get-des] :
   (to S : get-printer-price(B))
   < S : V@Seller | description : N, AtS > Conf & none
   => < S : V@Seller | description : N, AtS > Conf
      & (to B : printer-price(N)) .
```

Note the use of the `_&_` constructor. Since the printer description is sent to an object outside the mobile object in which the `Seller` object is located, the message is placed in its righthand side. The rule `get-des` is applied only if the outgoing messages tray is empty, making sure in this way that any previous outgoing message has been handled. The `_&_` operator is the top operator of the term representing the state of the mobile object, and therefore, since there may be other objects and messages in the configuration 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 locations, in a completely transparent way.

A buyer has an attribute `sellers` with a list of the identifiers of the known sellers. It also has an attribute `status` with its current state: `onArrival`, `asking`, `done`, or `buying`. Finally, the buyer keeps information about the printer with the best price in the attributes `price` and `bestSeller` of sorts, respectively, `Default{Nat}` and `Default{Oid}`. Initially, these two last attributes are `null`.

```
class Buyer | sellers : List{Mid}, status : Status,
              price : Default{Nat}, bestSeller : Default{Oid} .
```

The first rewrite rule, `move`, handles the travels of the buyer to request information on printers: if it is not in the middle of a request (its status is `done`) and there is at least one seller name in the `sellers` attribute, it asks the system to take it to the host where the next seller is.

```
rl [move] :
   < B : V@Buyer | status : done, sellers : o(L, N) . OS, AtS >
   Conf & none
   => < B : V@Buyer | status : onArrival, sellers : o(L, N) . OS, AtS >
      Conf & go-find(o(L, N), L) .
```

Since Mobile Maude guarantees that mobile objects moving from one location to another are idle, we know that, once the `go-find` command is given in the `move` rule, the buyer 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 seller's process. Therefore, since there is no rule taking a `Buyer` object in `onArrival` state and a nonempty outgoing messages tray, this object will not do anything until it reaches its destination.

On arrival, the buyer asks the seller for the printer description.

```
rl [onArrival] :
   < B : V@Buyer | status : onArrival, sellers : S . OS, AtS >
   Conf & none
   => < B : V@Buyer | status : asking, sellers : S . OS, AtS >
      Conf & (to S : get-printer-price(B)) .
```

When the printer price arrives, if it corresponds to the first seller (the attribute `price` is `null`) the buyer keeps it as the best known price; or compares it with the best known printer and updates its information if needed. Notice that the first identifier in the list of known sellers gives us the identifier of the seller it is currently interacting with.

```
rl [new-des] :
   (to B : printer-price(N))
   < B : V@Buyer | status : asking, price : null, bestSeller : null,
                   sellers : S . OS, AtS >
   => < B : V@Buyer | status : done, price : N, bestSeller : S,
                      sellers : OS, AtS > .

rl [new-des] :
   (to B : printer-price(N))
   < B : V@Buyer | status : asking, price : N', bestSeller : S',
                   sellers : S . OS, AtS >
   => if (N < N')
      then < B : V@Buyer | status : done, price : N, bestSeller : S,
                           sellers : OS, AtS >
      else < B : V@Buyer | status : done, price : N', bestSeller : S',
                           sellers : OS, AtS >
      fi .
```

Notice that since these last rules do not imply the sending of any message out of the mobile object, we do not need to use the `_&_` operator and variable `Conf` to wrap the whole state.

Finally, when the list of remaining sellers is empty, the buyer travels to find the best seller and reaches the `buying` status.

```
rl [buy-it] :
   < B : V@Buyer | status : done, sellers : no-id, bestSeller : o(L,N), AtS >
   Conf & none
   => < B : V@Buyer | status : buying, sellers : no-id,
                      bestSeller : o(L,N), AtS >
      Conf & go-find(o(L,N), L) .
```

Let us see an example of a distributed configuration, and how we can rewrite it by using the `erewrite` command. Our sample buyers/sellers configuration, shown in Figure 2, is constituted by three located configurations, each one to be executed in a Maude process—in this case the three processes run on the same machine, with IP address `IP`. The first located configuration (shown in the middle of the figure) contains a `ServerRootObject`, with identifier `l(IP, 0)`, and a mobile object with identifier `o(l(IP, 0), 0)` with a `Seller` in its belly. The Maude command to introduce the initial state of this configuration is as follows:

```
erew <> < l(IP, 0) : ServerRootObject |
            cnt : 1,
            guests : o(l(IP, 0), 0),
            forward : 0 |-> (l(IP, 0), 0),
            neighbors : empty,
```

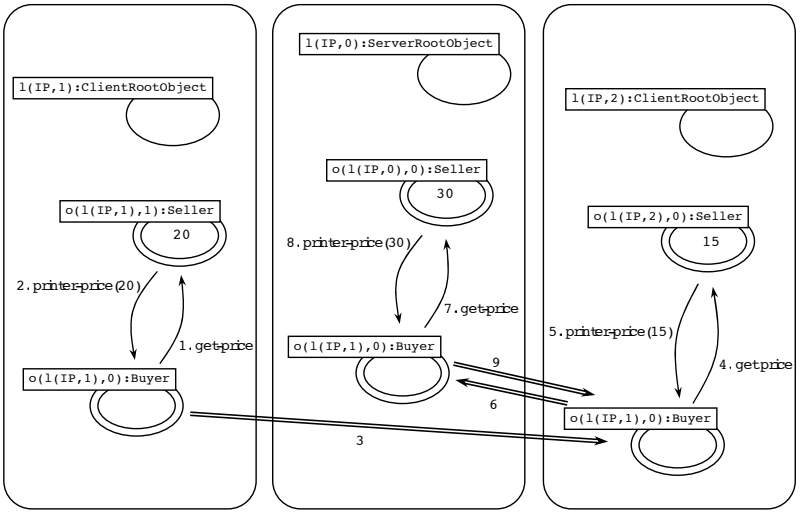

Fig. 2. Buyers and sellers configuration.

```
    state : idle,
    defNeighbor : null >
< o(l(IP, 0), 0) : MobileObject |
    mod : upModule('SELLER, false),
    s : upTerm(< o(l(IP, 0), 0) : Seller | description : 30 >
               & none),
    gas : 200,
    hops : 0,
    mode : active > .
```

Note how the function `upModule` is used to obtain the metarepresentation of the module `SELLER`, and how the function `upTerm` is used to metarepresent the initial state of the inner object.

This configuration must be executed before the other two ones because it contains the object `ServerRootObject`, which is in the central process of the star network.

The second located configuration (on the left in the figure) contains a `ClientRootObject`, a `Buyer` and a `Seller` with cheaper printers. Finally, the third located configuration (on the right) contains another `ClientRootObject` and a `Seller` with the cheapest printers. The Maude commands, introduced in other two different Maude process, are very similar to the previous one. [5]

[5] The execution of these three commands in three different Maude processes does not finish. And that is because of the blocking behavior of the socket messages like `receive`. An execution of a Mobile Maude application is not intended to finish since the located configurations are always waiting for messages or mobile objects to come in from other configurations. Due to this fact, it is recommended to execute these applications with the trace on. In this way we can see what is happening in each Maude process. When the execution of a concrete example seems to be finished because we do not see evolution in any of the involved processes, we can finish them by pressing `^C`. We are working on a graphic interface that supports execution of Mobile Maude applications.

Figure 2 shows how the order in which the different actions occur. First the buyer asks to the seller in his same location (price 20). Then the buyer travels to the location on the right (through the location with the `ServerRootObject`) and asks to the seller who sells printers costing 15. After that, the buyer travels to the middle location and asks to the seller there (price 30). Finally, the buyer travels to the right location to find the seller with the best offer.

6 Model checking Mobile Maude applications

Maude's model checker [5] allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a Maude system module is finite. This is supported in Maude by its predefined `MODEL-CHECKER` module and other related modules, which can be found in the `model-checker.maude` file distributed with Maude.

The properties to be checked are described by using a specific property specification logic, namely Linear Temporal Logic (LTL) [8,1], which allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property.

Using the model checker on Mobile Maude is not easy however. Mobile Maude configurations are distributed among several hosts, and therefore the model checker cannot be used directly to prove properties about these global configurations. On the other hand, we would like to check properties on the application code, which is metarepresented in the belly of the mobile objects. We show in the following sections how we have addressed both issues. The former problem has been solved by considering an algebraic specification of the sockets provided by Maude. The later one has been solved by considering two-level properties, stating different properties on each of the reflection levels.

6.1 Redefinition of the `SOCKET` module

To solve the distribution problem, we have provided an algebraic specification of sockets. We have redefined the `SOCKET` module, simulating the behavior of sockets on local configurations. This specification expresses processes as terms of a class `Process` with a single attribute `conf`. Processes work as hosts in the distributed version, keeping the configuration separated from the others in its attribute. Message passing is then defined between processes instead of between hosts.

Thus, we have specified sockets, socket managers and server sockets to deal with processes:

- The socket manager is now an instance of a class `Manager`, with a `counter` attribute to name the new sockets.
- The sockets are instances of a class `Socket` with attributes `source` (the

source `Process`), `target` (the target `Process`), and `socketState` (the socket state). Notice that although we talk about source and target, sockets are bidirectional.

- The server sockets are instances of the class `ServerSocket` with the attributes `address` (the server address), `port` (the server port), and `backlog` (the number of queue requests for connection that the server will allow). When one object want to create a server, we create one server socket at process level and the object receives a `createdSocket` message with the server socket identifier.

Note that there is no need for a client sockets class, they are only processes, so to create a client socket we create a socket with target the server and source the process.

The class `Process` allows to represent in a single term a whole distributed configuration. The rest of the above mentioned classes and the rewrite rules defined in the new module `SOCKET` allow to use the specification of Mobile Maude with no more changes. So in order to prove a property about a distributed configuration we have to prove it on the corresponding "local" configuration by using `Process`es.

6.2 Two-level atomic propositions for the buying printers example

To use the model checker we just need to make explicit two things: the intended sort of states, `Configuration`, and the relevant *state predicates*, that is, the relevant LTL atomic propositions.

To be able to model check Mobile Maude application code, we propose defining these predicates at two different levels: the processes level and the inner objects level. In the processes level we look for inner objects which have some properties; in the inner objects level we check such properties.

Let us see an example about the buying printers case study. Suppose we want to prove that the buyer always finds the best price, and that, when he has visited all sellers, he finishes in the process of the seller who has such a best price. If *bestPrice&Seller* represents the state predicate asserting that the buyer is in the process of the seller with the best offer, then the LTL formula we want to check is $\Diamond\Box$*bestPrice&Seller*, that is, it is always possible to reach an state where the property *bestPrice&Seller* is fulfilled and from that state the property keeps invariant.

First, we define when a top configuration of processes fulfills such a property. For it, we use an auxiliary predicate *bestPrice&Seller* with an argument, (the metarepresentation of) the best price, obtained by means of the auxiliary function `minPrice`.

```
op bestPrice&Seller : -> Prop .
op bestPrice&Seller : Term -> Prop .
eq C |= bestPrice&Seller = C |= bestPrice&Seller(minPrice(C)) .
```

The definition of `bestPrice&Seller(N)` recursively traverses all the pro-

cesses going inside each configuration looking for a seller with the given price and a buyer who has it as the best price.

```
op existsSeller : Term -> Prop .
op existsBuyer : Term -> Prop .

eq (C < PID : Process | conf : C' >) |= bestPrice&Seller(N)
  = (C |= bestPrice&Seller(N)) or
    ((C' |= existsSeller(N)) and (C' |= existsBuyer(N))) .
eq C |= bestPrice&Seller(N) = false [owise] .

eq (< O : MobileObject | s : ('_&_[TERM, TERM']), AtS > C)
      |= existsSeller(N)
  = (getTerm(metaReduce(upModule('EXAMPLE-PREDS, false),
               '_|=_[TERM, 'exSeller[N]])) == 'true.Bool)
    or (C |= existsSeller(N)) .
eq C |= existsSeller(N) = false [owise] .
```

The definition of `existsSeller(N)` uses the predicate `exSeller` defined at the inner objects level. The predicate `existsBuyer(N)` is defined in the same way. The module `EXAMPLE-PREDS` includes the definition of the predicates `exSeller` and `exBuyer`.

```
op exSeller : Nat -> Prop .
op exBuyer : Nat -> Prop .

eq < S : Seller | description : N, AtS > C |= exSeller(N) = true .
eq C |= exSeller(N) = false [owise] .

eq < B : Buyer | price : N, status : buying, AtS > C |= exBuyer(N)
  = true .
eq C |= exBuyer(N) = false [owise] .
```

Notice that these atomic propositions are defined at the level of the application code.

After having defined these predicates, the Maude command to use the model checker for examining whether an initial configuration `initial` fulfills the formula $\Diamond\Box$*bestPrice&Seller* is as follows:

```
Maude> red modelCheck(initial, <> [] bestPrice&Seller) .
result Bool: true
```

7 Conclusions

We have presented a distributed implementation of Mobile Maude where mobile objects, carrying its own code and internal state, can travel from one machine to another one. Sockets now provided by Maude are used to achieve this goal in a really distributed setting.

Although the main concepts and design decisions have been maintained as they were presented in the first implementation of the language [3], the parts regarding how the distributed state is represented and controlling how messages and mobile objects are transferred between different machines are

completely new. We have designed these new parts in a way as independent of the concrete underlying architecture as possible.

We have used the language to implement several case studies. Here we have shown an application where a printer buyer has to choose the seller offering the cheapest printer. The conference reviewing system presented in [4] has also been migrated to this new version of the language.

By explicitly representing the different processes in which a distributed application is allocated, we can represent the whole distributed state in a single term, and by redefining the predefined module `SOCKET` we can use the Mobile Maude implementation shown in this paper to execute/ simulate the behavior of such application. This compact representation can be used to prove properties it fulfills by means of Maude's model checker. However, model checking non-trivial examples takes too many time, so we are working on state space reduction techniques [6].

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. http://maude.cs.uiuc.edu/maude2-manual.

[3] 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*, pages 73–85. Springer, 2000.

[4] F. Durán and A. Verdejo. A conference reviewing system in Mobile Maude. In Gadducci and Montanari [7], pages 79–95.

[5] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In Gadducci and Montanari [7], pages 115–141.

[6] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. To appear in AMAST 2006.

[7] F. Gadducci and U. Montanari, editors. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specifications*. Springer-Verlag, 1992.