

Mobile Maude: An Prototype Interactive Implementation

Carolyn Talcott

Computer Science Laboratory, SRI International, Menlo Park, CA 94025 USA

July 26, 2004

Abstract

An implementation of the Mobile Maude language founded on the Russian Dolls framework for reflective distributed objects and the IMAude environment for interactive Maude specifications.

1 Introduction

Mobile Maude is a Mobile Agent language extending the rewriting logic language Maude and supporting mobile computation. Rewriting logic [6, 8], is a simple logic well-suited for distributed system specification, that is executable and reflective (capable of faithfully representing important aspects of its own syntax and deductive/computation mechanisms, see [3]). The Maude system [2, 1] (`maude.cs.uiuc.edu`), is an implementation of rewriting logic and its reflective capabilities. Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances of mobile agent behavior. 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.

The Mobile Maude implementation described here is a collection of Maude modules specifying infrastructure to support modeling, prototyping and analysis of mobile agents. The initial design of Mobile Maude was presented in [4]. This implementation is founded on the Russian Dolls reflective distributed object framework [7] and the IOP+IMaude [5] environment for interacting and interoperating with Maude specifications. This report is organized as follows. In S 2 we summarize the initial design. In S 3 we describe the basic ideas of our Reflective Russian Dolls (RRD) framework. In S 4 we describe our implementation of Mobile Maude. In S 5 we explain how to specify mobile agent systems and give a small example, a DataMobile system. In S 6 we explain how to prototype and interact with Mobile Maude agent systems at different levels of abstraction and illustrate the techniques using the example DataMobile system.

The Maude modules, examples, and documentation are available at <http://www-formal.stanford.edu/clt/FTN/MobileMaude>. The Maude system (sources, binaries, and documentation) is available at <http://maude.cs.uiuc.edu>. The IOP platform, needed for distributed execution of Mobile Maude can be found at mcs.une.edu.au/~iop. In the following we assume the reader is familiar with the basics of rewriting logic and Maude. The primer available on the Maude web site is an excellent starting point.

2 Principles of Mobile Maude

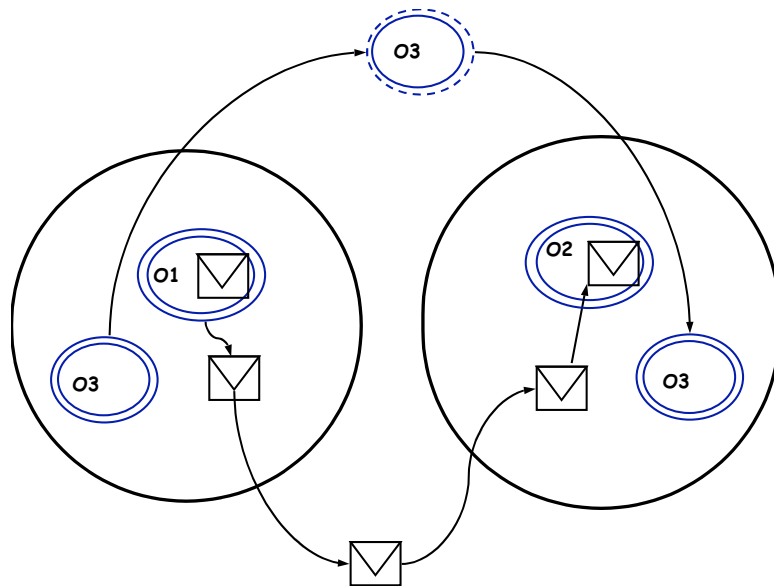


Figure 1: Mobile Maude: object and message mobility

Figure 1 illustrates a prototypical Mobile Maude system with two located processes and three mobile agents, where agent 03 is moving from one location to another. In addition agent 01 is sending a message to agent 02. Each Mobile Maude object is created by some process, and that process is responsible for keeping track of the location of the mobile object in order that messages may be delivered.

Formally, a located process (briefly a process) is an object whose attributes include an configuration of mobile objects and messages, and a mapping from identifiers of objects it has created to the objects last known location, and the number of hops the object took to reach that location. Each time a mobile object arrives at a new location (as the contents of a message) the receiving process notifies the object’s creator of the new location and hop count. The hop count is used to identify stale location information. A process also keeps information to allow it to generate a new identifier each time it creates a mobile object. A located process extracts messages from its contained configuration, and processes them. A request by a mobile object to go to a new location is turned into a message addressed to that process and transmitted.

Messages sent from one mobile object to another are routed using the location maps. If the addressee is local, the message is delivered. If the addressee belongs to the sending process the current location and hop count is obtained from the forwarding table, and the message and hop count are sent in a routing request to that location. Otherwise the message is sent in a routing request to the creator. A new object request message is processed by creating a new object, adding it to the contained configuration, and sending a reply containing the identifier of the new object to the requester.

A mobile object carries its code and state with it when it moves to a new location. The code for a Maude object is the module containing the declarations for the object attributes and the rewrite rules specifying its behavior. Reflection is used to transform code and state into data that can be understood by any located process. In particular a mobile object has two levels a meta level (called the mobile object wrapper) and a base level (call the base object or base agent). A mobile object wrapper has the same attributes for all mobile objects. These consist of metarepresentations of the base object and of the module specifying the base object behavior, an integer recording the number of hops that the object has made (initially 0), a bound on the number of rewrites the object can use in any execution step, and auxiliary data used to keep track of the wrappers processing state. The bound on rewrites is a form of resource management and is used to ensure that all mobile objects in a given location will have a fair share of the available resources.

The rules for a mobile object wrapper specify how it manages its base level. Messages sent by the base level to peers are forwarded to the peer wrapper for delivery, and messages for delivery received from peer wrappers are delivered to the base object, using the reflected evaluation and rewriting functions. A request for new object creation is transformed into a request for creation of a new mobile object (adding the wrapper). The heart of mobility is the treatment of requests from the base level to move to a new location. The mobile wrapper packages its metarepresented module and base object state, along with the object identifier and additional information describing the wrappers current state, and turns itself in to a request to go to the location specified by the base object, containing the packaged information. This information is used by the process at the new location to reconstitute the mobile object and let it resume execution.

3 Reflective Russian Dolls

Reflective Russian dolls [7] is a generic formal model of distributed object reflection, that combines logical reflection with a structuring of distributed objects as nested configurations of metaobjects (a la Russian Dolls) that can reason about and control their subobjects.

In simple situations the state of a distributed object-based system can be thought of as an unstructured “soup” of objects and messages interacting in various ways. However, there are often good reasons for having boundaries that circumscribe parts of a distributed object state. For example, the Internet is not really a flat network, but a network of networks, having different network domains, that may not be directly accessible except through specific gateways, firewalls, and so on. Thus we model distributed state not as a flat soup, but as a “soup of soups”, each enclosed within specific *boundaries* with well-defined interaction points. Reflective distributed architectures based on the Russian Dolls paradigm are already quite expressive,

in the sense that metaobjects can perform a wide range of services and can control interactions of their subobjects in many different ways. To provide even more powerful features such as control of execution, code morphing, runtime dynamic adaptation, and mobility, the simple nesting model is extended using *logical reflection*, allowing some subobjects and their behavior to be metarepresented as data, again allowing the metarepresented subobjects to be nested if appropriate.

To illustrate the ideas in a little more detail, we briefly sketch how reflective object systems are represented. We use Maude syntax, to also illustrate the formal notation. An object is represented as a data structure containing an object identifier, a class identifier, a set of attributes representing its state, and a set of interfaces making explicit its points of interaction with its environment. For simplicity in the following we restrict attention to input/output interfaces. We write

$$[o : C \mid a_1: v_1, \dots, a_n: v_n \mid in: inQ, out: outQ]$$

for an object with identifier o , class identifier C , attributes $a_1: v_1, \dots, a_n: v_n$, and interfaces $in: inQ, out: outQ$ where inQ and $outQ$ are lists of messages. The set of messages includes standard messages of the form $msg(o, o', mb)$ (a message to an object with identifier o , apparently from an object with identifier o' with contents mb), and $newo(cid,atts)$ (a request to create a new object with class named cid and initial attributes $atts$).

An object's behavior is specified by rewrite rules determining what the object might do in given situations. For example a rule processing the first message in the input queue has the general form

$$\begin{aligned} & [o : C \mid atts \mid in: M inQ, out: outQ] \\ & \Rightarrow \\ & [o : C' \mid atts' \mid in: inQ, out: outQ outQ'] \text{ if } cond \end{aligned}$$

where M is a message and $cond$ is a boolean condition that must evaluate to `true` for the rule to fire.

Nesting is represented using a metaobject with distinguished attribute, for example, $\{ _ \}$ that holds a configuration of objects and messages:

$$[mid : MC \mid \{ configuration \}, other\text{-attributes} \mid interfaces]$$

giving the containing metaobject mid control over interaction of the contained configuration with the outside environment. Objects within the configuration may also be nested. Any or all of the elements of the configuration may be metarepresented, giving the containing metaobject greater ability to reason about and modify the behavior of the contained configuration.

Rules for moving messages across object boundaries (to the input queue and from the output queue) can be specified in the container of an object configuration, or as part of the composition of a collection of objects into a configuration.

4 Mobile Maude

The Mobile Maude language specification has three component: the basic syntax, mobile object wrappers, and located processes. Each component is discussed in turn in the following subsections.

4.1 Basic Syntax

The basic syntax for objects and messages in the RRD framework is specified in the module `CONFIG`. This module differs from the module `CONFIGURATION` supplied in the Maude library, `prelude.maude`, in that it provide an alternative syntax for objects and specifies some standard attributes and message constructors. The declaration of sorts `Oid` (object identifiers), `Cid` (class identifiers), `Msg` (messages), `Object`, `Attribute`, `AttributeSet`, `Configuration`, and the constructors for `AttributeSet` and `Configuration` are the same as in the module `CONFIGURATION`. In particular the declarations for attribute sets are

```
sorts Attribute AttributeSet .
subsort Attribute < AttributeSet .
op none : -> AttributeSet .
op __, _ : AttributeSet AttributeSet -> AttributeSet
        [assoc comm id: none] .
```

The declaration of the concatenation operator `__,_` uses Maude's mix fix syntax in which each `_` is an argument place holder. It is declared to be ACI (associative and commutative with identity), thus an attribute set is a multiset of attributes where `none` is the empty attribute set. Similarly a configuration is a multiset of objects and messages with empty syntax for concatenation.

```
op __ : Configuration Configuration -> Configuration
        [assoc comm id: none] .
```

Syntax for objects with explicit interfaces is declared by

```
op [_:_|_|_] : Oid Cid AttributeSet AttributeSet -> Object
```

Interface attributes are specified as follows

```
sort MsgQ . subsort Msg < MsgQ .
op nil : -> MsgQ .
op __, _ : MsgQ MsgQ -> MsgQ [assoc id: nil] .
ops in:_ out:_ : MsgQ -> Attribute .
```

Peer-to-peer messages have a standard format consisting of the addressee, the sender, and a message body.

```
sort MsgBody .
op msg : Oid Oid MsgBody -> Msg .
```

An object creation request contains the identifier of the requester, the class identifier of the requested object, and the objects initial attribute set. The reply contains the requester (message destination) and the identifier of the newly created object.

```
op newo-req : Oid Cid AttributeSet -> Msg .
op newo-reply : Oid Oid -> Msg .
```

Several utility functions are also defined including `tgt` to access the target (addressee) of a message, `identity` to access the identity of an object, and `present` to test whether a object with a given identity is contained in a configuration.

The module `MMCONF` extends `CONFIG` with declarations mobile object specific sorts and messages. It declares `Oid` subsorts `Pid` (process identifiers), and `Mid` (mobile object identifiers). Process identifiers are constructed from two strings (using the network convention of naming a process by its host and port), and mobile object identifiers are constructed from a process identifier (its creator) and a string giving the local identifier.

```
op p : String String -> Pid .    *** Host Port -- Process Identifier
op o : Pid String -> Mid .      *** Mobile-Object Identifier
```

A base object go request has the following syntax

```
op go : Pid -> Msg .
```

and print requests have the syntax

```
op print : String -> Msg .
```

As a simple example the following is an object who's class is `Echo`. It has no attributes and has a single message in its input buffer and an empty output buffer.

```
[ o(p("localhost", "5555"), "e") : Echo | none
  | in: msg(o(p("localhost", "5555"), "e"), user, mb("hello")),
    out: nil]
```

The `Echo` class has one rule that prints the contents of each input message `mb(s)`, by placing `print(s)`, in the output buffer. (The full example can be found in the `MM` directory.)

4.2 Mobile Object Wrappers

The mobile object wrapper class is specified in the module `MOBILE-OBJECT`. The class identifier is `MO`. As explained in S 2 the attributes `mod:` and `ob:` contain the metarepresentations of the base level module and base object respectively, while `gas:` contains a bound on the number of rewrites the base agent can do each step, and `hops:` contains the number of hops the object has made.

```
op mod:_ : Module -> Attribute .
op ob:_ : Term -> Attribute .
op gas:_ : Nat -> Attribute .
op hops:_ : Nat -> Attribute .
```

The mobile wrapper layer has a new constructor for go requests, that takes the destination process identifier, the mobile object identifier, its class identifier, an attribute set, and two message queues (input and output). In addition, there is a message body constructor `dlv` to wrap the metarepresentation of the body of a base level message being sent to a peer.

```

op dlv : Term -> MsgBody .
op go : Pid Mid Cid AttributeSet MsgQ MsgQ -> Msg .

```

The processing state of a mobile wrapper is recorded in the attribute `mode`:

```

sort Mode .
ops deliver send tick yield : -> Mode .
op send : Term -> Mode .
op init : Module Term Term Nat -> Mode .
op mode:_ : Mode -> Attribute .

```

The operator `mkmo` constructs a mobilized wrapping agent given the mobile object identifier, the name of the module defining the agent behavior, its class identifier and initial attributes, and the value of the `gas`: attribute.

```

var ?mid : Mid . var ?cid : Cid . var ng : Nat .
var modname : Qid . var atts : AttributeSet .
op mkmo : Mid Qid Cid AttributeSet Nat -> Object .
eq mkmo(?mid,modname,?cid,atts,ng) =
  [ ?mid : MO | mode: init([modname],upTerm(?cid),upTerm(atts),ng)
    | in: nil, out: nil ] .

```

The function `upTerm` is provided by the Maude META-LEVEL. It returns the metarepresentation of its argument.

The modes can be thought of as states of a state machine and a mobile object wrapper has the following state transitions

$$\begin{array}{ccccccc}
 & & \text{inQ}==\text{nil} & & \text{noOutput} & & \\
 \text{init}(\text{?mod},\text{cidT},\text{attsT},\text{ng}) & \rightarrow & \text{deliver} & \text{----} \rightarrow & \text{tick} & \rightarrow & \text{send} & \text{----} \rightarrow & \text{yield} \\
 & & \begin{array}{c} \wedge \\ \vee \end{array} & & & & \begin{array}{c} \wedge \\ | \\ \vee \end{array} & & \\
 & & & & & & \text{send}(\text{mT}) & &
 \end{array}$$

In `yield` mode the wrapper waits for its containing process to move it to `deliver` mode to restart its processing cycle. This is simply a form of task scheduling.

The transition from `init` mode initialized the wrapper attributes using the module `?mod`, and constructing a metarepresentation of the base object from the metarepresentations `cidT` of its class identifier and `attsT` of its attribute set. The `gas`: is initialized using the parameter `ng`.

In `deliver` mode, messages in the wrappers input queue are transformed and meta-delivered into the base objects input queue. Messages of the form

```
msg(mid,mid',dlv(mbT))
```

are transformed to

```
'msg[midT,midT',mbT]
```

the metarepresentation of the message

```
msg(mid,mid', mB)
```

where `midT` is the metarepresentation of `mid`, `midT'` is the metarepresentation of `mid'`, and `mbT` is the metarepresentation of `mB`. Other messages are simply transformed to their metarepresentation using the metafunction `upTerm` discussed above.

In `tick` mode the wrapper metarewrites the base object using `bound` stored in the `gas:` attribute and then moves to `send` mode.

In `send` mode the wrapper extracts (metarepresented) messages from the output queue of the base object, transforms them and puts them in its output queue. As explained in S 2, `go` messages cause the wrapper to fold itself into a wrapper layer `go` request, `newo-req` messages are converted to requests for wrapped objects, and `msg(...)` messages are converted to mobile peer `dlv` requests. All other metarepresented messages are simply transformed to their base-level form using the metafunction `downTerm` provided by the Maude metalevel.

Note 1. The uniform passing of messages unrelated to mobility to and from the containing environment allows for new interactions with the local execution environment without modification of the mobile object wrapper.

By way of illustration, we discuss the rules for the `tick` mode, for extracting messages from the base object in `send` mode, and the rules for processing `go` and `newo-req` messages. The following declares variables needed for these rules.

```
var ?mod : Module .
var ng : Nat .
var mo : Mid .
vars atts ifaces : AttributeSet .
var ?mode : Mode .
vars inQ outQ : MsgQ .
vars midT pidT attsT mT oT oT' cidT : Term .
var res? : ResultPair? .
var ?msg : Msg .
```

The rule for `tick` mode (labeled `do-something`) applies the `metaFrewrite` function to the module `?mod` (stored in the `mod:` attribute), and the object metarepresentation `oT` (stored in the `ob:` attribute). This function uses Maude bottom-up position-fair rewrite strategy that is more suitable for object systems. The argument `ng` specifies a bound on the number of position visits and the final argument (1) specifies a bound on the number of rewrites at each position. The value (`res?`) returned by `metaFrewrite` is either a pair containing the rewritten term and its type (and thus the membership `res? :: ResultPair` holds), or an error term indicating that the function could not be applied (usually because the term can not be understood in the context of the module). If rewriting succeeds, the rewritten object term `oT'` is used to update the `ob:` attribute. Otherwise the `ob:` attribute is left unchanged. In either case the wrapper object moves to `send` mode.

```

crl[do-something]:
  [mo : MO | mod: ?mod, ob: oT, gas: ng, mode: tick, atts | ifaces]
  =>
  [mo : MO | mod: ?mod, ob: oT', gas: ng, mode: send, atts | ifaces]
    if res? := metaFrewrite(?mod,oT,ng,1)
      /\ oT' := ( if res? :: ResultPair
                  then getTerm(res?)
                  else oT
                  fi ) .

```

To simplify manipulation of object interfaces, the module `CONFIG` defines functions `hasOutput` that returns `true` if the output queue is not empty, `getOutput` that returns the first message of the output queue (or `nil` if the queue is empty), `removeOutput` that removes a message from the output queue. The rule `msg-out` only applies if the base object output queue is not empty, determine by the test

```

getTerm(metaReduce(?mod,'hasOutput[oT])) == 'true.Bool

```

In this case, the extracted message term `mT` is stored in the `mode:` attribute and the `ob:` is updated by removing this message.

```

crl[msg-out]:
  [ mo : MO | mod: ?mod, ob: oT, mode: send, atts
    | out: outQ, ifaces ]
  =>
  [ mo : MO | mod: ?mod, ob: oT', mode: send(mT), atts
    | out: outQ, ifaces ]
  if getTerm(metaReduce(?mod,'hasOutput[oT])) == 'true.Bool
  /\ mT := getTerm(metaReduce(?mod,'getOutput[oT]))
  /\ sortLeq(?mod,leastSort(?mod, mT), 'Msg)
  /\ oT' := getTerm(metaReduce(?mod,'removeOutput[oT,mT])) .

```

In `send` mode, if the parameter is a metarepresented `go` message, then the mobile object wrapper turns itself into a `go` message containing the desired location `downTerm(pidT,unkOid)`, its identifier `mo`, its class identifier `MO`, its attributes with `mode:` updated to be `yield`, and the contents of its interface queues.

```

rl[send.go]:
  [ mo : MO | mod: ?mod, ob: oT, mode: send('go[pidT]), atts
    | out: outQ, in: inQ ]
  =>
  go(downTerm(pidT,unkOid),mo,MO,
     (mod: ?mod, ob: oT, mode: yield, atts), inQ,outQ) .

```

If the `send` parameter is a metarepresented `newo-req` message, then the request is modified to request creation of a mobilized object of the requested class and initial state by making the latter arguments to the `init` mode attribute in the modified request.

```

rl[send.newo]:
  [ mo : MO | mod: ?mod, ob: oT,
    mode: send('newo-req[midT,cidT,attsT]),
    gas: ng, atts
    | out: outQ, in: inQ ]
=>
  [ mo : MO | mod: ?mod, ob: oT, mode: send, gas: ng, atts
    | out: (outQ,
            newo-req(downTerm(midT,mo),MO,
                    (mode: init(?mod,cidT,attsT,ng)))),
    in: inQ] .

```

4.3 Located Processes

The class of located processes is specified in the module `MM-PROCESS`. The class identifier for a located process is `P` and its attributes are

```

op cnt:_ : Nat -> Attribute .
op cf:_ : Configuration -> Attribute .
op fwd:_ : FTable -> Attribute .
op sending:_ : MsgQ -> Attribute .

```

where `cnt`: stores the counter for generating new object identifiers, `cf`: in configuration managed by the process, `fwd`: is its forwarding table, and `sending`: stores a queue of messages extracted from the configuration to be processed.

A forwarding table (sort `FTable`) is a multiset of entries (sort `FEntry`), each entry having the form `fe(ix,pid,nhops)` where `ix` is a local identifier string, `pid` is the identifier of the objects location, and `nhops` is the number of hops the object made to reach that location. The function `locate` returns the location of an object locally identified by the string argument, if known. The third argument is a default to return if there is not entry for the object. The function `nhops` returns the number of hopes if known and `-1` otherwise. The function `update` updates an existing entry with new location and hop information or creates a new entry if there is no entry for the named object.

```

sorts FTable FEntry .
op fe : String Pid Nat -> FEntry .
subsorts FEntry < FTable .
op mt : -> FTable .
op ___ : FTable FTable -> FTable [assoc comm id: mt] .
op locate : String FTable Pid -> Pid .
op nhops : String FTable -> Int .
op update : FTable String Pid Nat -> FTable .

```

The operator `mkp` constructs a process, given its identifier and contained configuration.

```

var ?pid . var C : Configuration .
op mkp : Pid Configuration -> Object .

```

```

eq mkp(?pid, C) =
  [ ?pid : P | cnt: 0, cf: C, sending: nil, fwd: mt
    | in: nil, out: nil ] .

```

Messages exchanged between peer located processes have one of the following message bodies: a routing request (`route`), an installation request (`install`), or a location report (`at`), where the new location is the sender of the message.

```

op route : Msg Int -> MsgBody .
op install : Mid Cid AttributeSet MsgQ MsgQ -> MsgBody .
op at : Mid Int -> MsgBody .

```

The integer argument of a routing request is `-1` if this is sent to the addressee's creator for initial routing. Otherwise it indicates the number of hops to the last known location, and is used to avoid routing loops and to detect stale information.

As examples we discuss the two process rules for mobility and the rule for object creation. The variables used in these rules are:

```

var n : Nat .
var ix : String .
vars mo ?mid : Mid .
vars atts matts : AttributeSet .
vars inQ outQ inQ1 outQ1 sQ : MsgQ .
vars ?pid ?pid0 ?pid1 : Pid .
vars C C0 C1 : Configuration .
var ?ftable : FTable .
var ?msg : Msg .
var ?cid : Cid .

```

A `go` message in the configuration is turned into an `install` request, unless the new location (`pid0`) is the current location (`pid0`), in which case the mobile object is simple reconstituted. The first argument to the `go` request is used as the message address and the remaining arguments are placed in the `install` message body.

```

rl[go] :
  [ ?pid : P | cf: (C go(?pid0,mo,?cid,matts,inQ1,outQ1)),
    atts | in: inQ, out: outQ ]
=>
  (if (?pid == ?pid0)
   then *** go to where you are is a no-op
     [ ?pid : P | cf: C [mo : ?cid | matts | in: inQ1, out: outQ1],
       atts
         | in: inQ, out: outQ ]
   else
     [ ?pid : P | cf: C, atts
       | in: inQ,
         out: (outQ,
              msg(?pid0,?pid,
                  install(mo,?cid,matts,inQ1,outQ1)))]
     fi) .

```

When an `install` request arrives, a mobile object is constructed from the `install` and added to the process's configuration. If this installing process is the mobile object's creator (`(?pid1 == ?pid)` holds), then the `fwd:` attribute is updated with the new location and hop count, otherwise a new location report

```
msg(?pid1,?pid, at(o(?pid1,ix), s n))
```

is sent to the object's creator.

```
crl[arrive]:
  [ ?pid : P | cf: C, fwd: ?ftable, atts
    | in: (msg(?pid, ?pid0,
            install(o(?pid1,ix), ?cid, (matts, hops: n),
                  inQ1, outQ1)),
          inQ),
    out: outQ ]

=>
(if (?pid1 == ?pid)
  then [ ?pid : P | cf: C1,
          fwd: update(?ftable, ix, ?pid, s n),
          atts
          | in: inQ, out: outQ ]
  else [ ?pid : P | cf: C1,
          fwd: ?ftable, atts
          | in: inQ,
            out: (outQ,
                  msg(?pid1,?pid, at(o(?pid1,ix), s n)))]
  fi)
if C1 := (C [o(?pid1,ix) : ?cid | (matts, hops: s n)
            | in: inQ1, out: outQ1]) .
```

To process a `newo-req` from a mobile object (assumed to be in the local configuration) a local identifier is generated from the `cnt:` attribute (`ix := string(n,10)`), and the value of `cnt:` is incremented. An object with identifier `o(?pid,ix)`, class identifier and attributes specified in the request, and empty input and output queues is added to the configuration, and a `newo-reply` containing the new object's identifier is input to the requester.

```
crl[newo] :
  [ ?pid : P | cf: C, cnt: n, sending: (?msg, sQ), atts
    | in: inQ1, out: outQ1 ]

=>
[ ?pid : P | cf: C1, cnt: s n, sending: sQ, atts
  | in: inQ1, out: outQ1 ]
if newo-req(mo,?cid,matts) := ?msg
  /\ ix := string(n,10)
  /\ ?mid := o(?pid,ix)
  /\ C1 := ( input(C, newo-reply(mo,?mid))
            [ ?mid : ?cid | matts | in: nil, out: nil] ) .
```

There is no need to update the forwarding table until the new object moves elsewhere.

Note 2. It would be nice to allow mobile objects to interact seamlessly with their stationary peers. One way to do this would be for a mobile wrapper to apply `downTerm` to peer-peer messages sent by its base object, as it does for service requests not related to mobility. However, for this to work, the `MOBILE-OBJECT` would have to be extended with the module specifying all the messages to be exchanged by base objects. In turn the module `MM-PROCESS` would need to import the extended wrapper module. In particular for an object to migrate the target process must understand messages it might emit. This violates the goals of modularity and separation of concerns, and so we support ‘non-mobile’ objects by insisting that the process object create all objects in its interior, thus all contained objects are mobilized whether or not they want to move.

5 Specifying Mobile Agents

5.1 Mobile Agent Specification: General

The specification of a mobile agent system is similar to specification of other interactive agent systems, with the added use of `go` requests. Thus to specify a mobile agent system one defines a module extending `MMCONF` that declare agent class identifiers, attributes, messages (by defining message body constructors) and behavior rules. The module may of course import submodules, to allow specifications to be modularized. A behavior rule should have one of the following two forms

```
rl[rname]:
  [ mid : cid | atts | in: inQ, out: outQ]
  =>
  [ mid : cid' | atts' | in: inQ', out: outQ' ] .
```

or

```
crl[crl.name]:
  [ mid : cid | atts | in: inQ, out: outQ]
  =>
  [ mid : cid' | atts' | in: inQ', out: outQ' ]
  if cond .
```

where `mid` is a term of sort `Mid`; `cid` and `cid'` are terms of sort `Cid`; `atts` and `atts'` are terms of sort `AttributeSet`; `inQ`, `inQ'`, `outQ` and `outQ'` are terms of work `MsgQ`; and `cond` is a term of sort `Bool`. Note that we allow the agents class to change, but not its identifier. Typically `inQ'` is obtained from `inQ` by removing zero, one or more messages while `outQ'` is obtained from `outQ` by adding zero, one or more messages. Although other transformations are allowed, rules taking messages from the output buffer are not likely to have a predictable consequence because the environment of an object can remove messages from the output buffer at any time.

A particular agent system is specified by a module together with an initial configuration.

5.2 Specification of the Data Mobile Agent System

A DataMobile is a mobile agent that travels from location to location, according to its specified itinerary. At each location visited the DataMobile expects to find a DataAgent object that queues user requests to store or retrieve data. Data is stored in a dictionary mapping keys to data elements. When a DataMobile object arrives at a new location, it sends a beep message to the local DataAgent. The DataAgent sends queued requests to the DataMobile, one at a time, until its queue is empty. Replies are printed. It then sends the DataMobile a done message, and the DataMobile goes to the next location on its itinerary. A DataMobile also sends print messages (to the user) so its travels can be monitored.

The Data Mobile Agent System is specified in the module `DATAMOBILE` which imports the mobile object configuration module `MMCONF` and the module `DICTIONARY` that defines the dictionary data type `Dict`, with lookup and updating operations. For simplicity we take both keys and data elements to be strings and use the empty string `" "` as the default value for key not in the table.

```
fmod DICT is
  inc STRING .
  sorts Row Dict .
  subsort Row < Dict .
  op none : -> Dict .
  op __ : Dict Dict -> Dict [assoc comm id: none] .
  op r : String String -> Row .

  op lookup : Dict String -> String .
  op update : Dict String String -> Dict .
  eq lookup(none,key:String) = "" .
  eq lookup((r(q:String, qs:String) d:Dict), key:String) =
    (if (q:String == key:String)
      then qs:String
      else lookup(d:Dict, key:String)
    fi) .

  eq update(none, key:String, new:String) = r(key:String, new:String) .
  eq update((r(q:String, qs:String) d:Dict), key:String, new:String) =
    (if (q:String == key:String)
      then (r(key:String, new:String) d:Dict)
      else (r(q:String, qs:String)
            update(d:Dict, key:String, new:String))
    fi) .
endfm
```

Notice that dictionaries generated from the empty dictionary (`none`) by updating will contain at most one entry for a given key.

The module `DATAMOBILE` defines two object classes: `DataMobile` and `DataAgent`. An object of class `DataMobile` has three attributes declared by

```
op data:_ : Dict -> Attribute .
```

```

op agent:_ : Mid -> Attribute .
op itin:_  : MidList -> Attribute .

```

The `data`: attribute holds the its dictionary of data, the `agent`: attribute holds the identifier of the data agent at the current location, and the `itin`: attribute holds the list of identifiers data agents at the remaining itinerary locations. Because data agents are stationary, a data agents location can be extracted from its identifier and need not be stored separately. The initial state of a `DataMobile` object with an itinerary of three locations `P0`, `P1`, `P2`, might look like

```

[ o(P0,"dm") : DataMobile
  | data: none, agent: o(P0,"da"), itin: (o(P1,"da"), o(P2,"da"))
  | in: nil, out: nil ]

```

The operator `mkdmobile` constructs a data mobile object in its initial state.

```

var ?mid ?dm : Mid .   var das : MidList .
op mkdmobile : Mid Mid MidList -> Object .
eq mkdmobile(?dm, ?mid, das) =
  [ ?dm : DataMobile | data: none, agent: ?mid, itin: das
    | in: nil, out: msg(?mid,?dm,beep) ] .

```

The messages used to interact with a `DataMobile` are declared by

```

op tellReq  : String String -> MsgBody .
op tellReply: String String -> MsgBody .

op askReq   : String -> MsgBody .
op askReply : String String -> MsgBody .

ops beep done : -> MsgBody .

```

When a `DataMobile` `dm` receives a request message of the form

```
msg(dm,da,tellReq(key,value))
```

it updates its data dictionary and sends a reply message

```
msg(da,dm,tellReply(key,value))
```

indicating the request has been processed. Similarly, when it receives a request message of the form

```
msg(dm,da,askReq(key))
```

it sends a reply message

```
msg(da,dm,askReply(key,value))
```

where value is the result of looking up key in its current dictionary. This request-reply pattern is a standard way of specifying method calls as patterns of asynchronous message passing. When a DataMobile receives a done message it travels to the next location in its itinerary. This is specified by the following rule

The variable declarations needed in the remainder of this section are

```

vars da dm : Mid .   var ?pid : Pid . var das : MidList .
var id : String .
var ?dict : Dict .
vars inQ outQ : MsgQ .
rl[done.dm]:
  [ dm : DataMobile | data: ?dict, agent: da,
                    itin: (o(?pid,id), das)
                    | in: (msg(dm,da,done), inQ), out: outQ]
=>
  [ dm : DataMobile
  | data: ?dict, agent: o(?pid,id), itin: (das, da)
  | in: inQ ,
  out: (outQ,
        print("dm going to " + pid2string(?pid)),
        go(?pid), msg(o(?pid,id),dm,beep),
        print("dm arrived at " + pid2string(?pid))) ] .

```

Notice the use of the go message. Before going, the DataMobile prints a message to the user. It also stores a beep message and another message to the user in its output queue to be sent upon arrival at the new location.

A data agent has class identifier DataAgent. It has attributes que: and pend: whose values are message queues, and st: whose value is the processing status (sort Status). A data agent's status is either waiting (for the data mobile to appear), sending(dm) (the data mobile dm has arrived), or wait4(msg) (waiting for a reply to msg from the data mobile).

The operator mkdagent constructs a data agent in its initial state.

```

var da : Mid .
op mkdagent : Mid -> Object .
eq mkdagent(da) =
  [ da : DataAgent |
    que: nil, pend: nil, st: waiting | in: nil, out: nil ] .

```

Rules for data agent behavior are summarized below. When a waiting data agent receives a beep it changes to sending status.

```

rl[beep]:
  [ da : DataAgent | que: rQ, st: waiting, atts
                    | in: (msg(da,dm,beep), inQ), out: outQ ]
=>
  [ da : DataAgent | que: rQ, st: sending(dm), atts
                    | in: inQ, out: outQ ] .

```

When a waiting data agent receives a request (`isReq(msg)`) it puts the request in its que.

```
crl[enqueue]:
  [ da : DataAgent | que: rQ, st: waiting, atts
                    | in: (?msg, inQ), out: outQ ]
=>
  [ da : DataAgent | que: (rQ, ?msg), st: waiting, atts
                    | in: inQ, out: outQ ]
  if isReq(?msg) .
```

A data agent with status `sending(dm)` sends the next request in its queue to `dm` and changes to `wait4` status.

```
rl[req]:
  [ da : DataAgent | que: (msg(da,cust,req), rQ),
                    | st: sending(dm), atts
                    | in: inQ, out: outQ ]
=>
  [ da : DataAgent | que: rQ, st: wait4(msg(dm,cust,req)), atts
                    | in: inQ, out: (outQ, msg(dm,da,req)) ] .
```

If the queue is empty, the data agent sends a done message to `dm` and moves to `waiting` status. It also transfers the `pend` queue to the regular queue.

```
rl[done]:
  [ da : DataAgent | que: nil, st: sending(dm), pend: rQ, atts
                    | in: inQ, out: outQ ]
=>
  [ da : DataAgent | que: rQ, st: waiting, pend: nil, atts
                    | in: inQ, out: (outQ, msg(dm,da,done)) ] .
```

A data agent with `wait4` status accepts a reply, prints the result and moves to `sending(dm)` status to continue processing requests.

```
rl[reply]:
  [ da : DataAgent | que: rQ, st: wait4(msg(dm,cust,req)), atts
                    | in: (msg(da,dm,reply), inQ), out: outQ ]
=>
  [ da : DataAgent | que: rQ, st: sending(dm), atts
                    | in: inQ, out: (outQ, printReply(msg(cust,da,reply))) ] .
```

A data agent with `wait4` status puts arriving requests in the pending queue. This is to prevent requests from blocking a reply and also to ensure that only finitely many requests are processed each round.

```
crl[pend.enqueue]:
  [ da : DataAgent | que: rQ, st: wait4(msg(dm,cust,req)),
                    | pend: pQ, atts
```

```

| in: (?msg, inQ), out: outQ ]
=>
[ da : DataAgent | que: rQ, st: sending(dm),
  pend: (pQ, ?msg), atts
  | in: inQ, out: outQ ]
if isReq(?msg) .

```

6 Prototyping Mobile Agents

Mobile agent systems can be prototyped at different levels of abstraction (points of view) by using different configuration test modules that provide corresponding communication rules.

6.1 Soup-of-Agents Prototyping

The simplest (most abstract) is just to consider a multiset of agents and messages in a soup, abstracting away the location infrastructure. In this case we define a test module `MM-TEST` that adds rules to move messages from output queues to the soup and soup to the input queue of the target object.

```

var ?oid : Oid .
var ?cid : Cid .
vars inQ outQ : MsgQ .
var ?msg : Msg .
vars atts ifaces : AttributeSet .

rl[out]:
[ ?oid : ?cid | atts | out: (?msg, outQ), ifaces ]
=>
[ ?oid : ?cid | atts | out: outQ, ifaces ] ?msg .

crl[in]:
[ ?oid : ?cid | atts | in: inQ, ifaces ] ?msg
=>
[ ?oid : ?cid | atts | in: (inQ, ?msg), ifaces ]
  if tgt(?msg) == ?oid .

```

Messages such as `go(pid)` and `print(...)` are simply left in the soup.

6.2 Soup-of-Agents Prototyping: Data Mobile example

The module `DA-TEST` defines an initial `DATAMOBILE` system configurations with three locations, a data agent at each location, and a data mobile at the first location. For convenience we define some constants to name mobile and process objects.

```

ops P0 P1 P2 : -> Pid .
eq P0 = p("localhost", "5550") .

```

```

eq P1 = p("localhost","5551") .
eq P2 = p("localhost","5552") .

ops M0 M1 M2 MB user : -> Mid .
eq M0 = o(P0,"da") .
eq M1 = o(P1,"da") .
eq M2 = o(P2,"da") .
eq MB = o(P0,"dm") .
eq user = o(P0,"me") .

```

The base initial configuration `ic` has 3 data agents and one data mobile. The configuration `ic0` has a tell request to be submitted at each location. The configuration `ic1` adds an ask request to be submitted at each location.

```

ops ic ic0 ic1 : -> Configuration .
eq ic = ( mkdagent(M0) mkdagent(M1) mkdagent(M2)
          mkdmobile(MB, M0, (M1, M2)) ) .
eq ic0 = ( ic msg(M0, user, tellReq("a", "av"))
           msg(M1, user, tellReq("b", "bv"))
           msg(M2, user, tellReq("c", "cv")) ) .
eq ic1 = ( ic0 msg(M0, user, askReq("a"))
           msg(M1, user, askReq("b"))
           msg(M2, user, askReq("c")) ) .

```

These configurations can be executed using the position fair rewriting command. Executing

```
frew [75] ic0 .
```

results in the data mobile visiting each data agent, and delivery of each of the tell requests.

```

...
print("to localhost:5550:me from localhost:5550:da tellReply(a = av)")
print("to localhost:5550:me from localhost:5551:da tellReply(b = bv)")
print("to localhost:5550:me from localhost:5552:da tellReply(c = cv)")
...

```

Executing

```
frew [100] ic1 .
```

results in delivery all six requests. Because of the way messages are ordered, the asks are delivered before the tells, hence the empty answers.

```

...
print("to localhost:5550:me from localhost:5550:da askReply(a = )")
print("to localhost:5550:me from localhost:5550:da tellReply(a = av)")
print("to localhost:5550:me from localhost:5551:da askReply(b = )")
print("to localhost:5550:me from localhost:5551:da tellReply(b = bv)")
print("to localhost:5550:me from localhost:5552:da askReply(c = )")
print("to localhost:5550:me from localhost:5552:da tellReply(c = cv)")
...

```

The search command can be used to look for different possible executions. For example to see if there is a computation in which there is a non empty reply to an ask we can execute the following search.

```
search [1] ic1 =>+ C:Configuration
          msg(mid:Mid, mid1:Mid,askReply("a",?av:String))
          such that ?av:String != "" .
```

Search returns bindings for the variables in the search pattern on the right. Omitting the configuration binding, the result of the above search is

```
mid:Mid --> o(p("localhost", "5550"), "da")
mid1:Mid --> o(p("localhost", "5550"), "dm")
?av:String --> "av"
```

Full results of the above commands and others can be found in `MobileMaude/DataMobile/runs-da.txt`.

6.3 Soup-of-Mobilized Agents Prototyping

The second level of prototyping considers configurations that are a multiset of mobilized (wrapped) agents and messages in a soup, still abstracting away the location infrastructure. In this case we define a test module `MO-TEST` that adds rules for objects of class `MO` to move messages from output queues to the soup and soup to the input queue of the target object. In addition there is a rule to process `go` requests by reconstituting the traveling mobile object. Finally there is a rule to move the mobile object from `yield` to `deliver` mode. These rules abstract the behavior of a process object container.

```
var ?cid : Cid . var ?pid : Pid . var mo : Mid .
var atts : AttributeSet . var n : Nat .
vars inQ outQ : MsgQ .

rl[go]:
  go(?pid,mo,?cid,(atts, hops: n),inQ,outQ)
  =>
  [ mo : MO | atts, hops: s n | in: inQ, out: outQ ] .

rl[tick.mo]:
  [ mo : MO | mode: yield, atts | in: inQ, out: outQ ]
  =>
  [ mo : MO | mode: deliver, atts | in: inQ, out: outQ ] .
```

6.4 Soup-of-Mobilized Agents Prototyping: Data Mobile example

The module `MDA-TEST` defines initial `DATAMOBILE` system configurations for the Soup-of-Mobilized-Agents level of abstraction. The base configuration `mic` has three mobilized data agents, corresponding to three locations, and a mobilized data mobile at the first location. The

configuration `mic0` adds an initial `beep` (the data mobile arriving) and a tell request for each data agent to the base configuration. The configuration `mic1` adds an initial `beep` and a tell and ask for key "a" to the base configuration.

```
ops mic mic0 mic1 : -> Configuration .
eq mic =
  ( mkmo(M0, 'DATAMOBILE, DataAgent,
        (que: nil, pend: nil, st: waiting),
        10)
    mkmo(MB, 'DATAMOBILE, DataMobile,
        (data: none, agent: M0, itin: (M1, M2)),
        10)
    mkmo(M1, 'DATAMOBILE, DataAgent,
        (que: nil, pend: nil, st: waiting),
        10)
    mkmo(M2, 'DATAMOBILE, DataAgent,
        (que: nil, pend: nil, st: waiting),
        10)
  ) .
eq mic0 = ( mic msg(M0, MB, dlv(upTerm(beep)))
           msg(M0, user, dlv(upTerm(tellReq("a", "av"))))
           msg(M1, user, dlv(upTerm(tellReq("b", "bv"))))
           msg(M2, user, dlv(upTerm(tellReq("c", "cv")))) ) .
eq mic1 = ( mic msg(M0, MB, dlv(upTerm(beep)))
           msg(M0, user, dlv(upTerm(tellReq("a", "av"))))
           msg(M0, user, dlv(upTerm(askReq("a")))) ) .
```

As for the agent-in-soup level, these configurations can be rewritten using the position fair rewriting command. More rewriting is required for the data mobile to make a complete round due to the extra object layer. The state space here is too large for use of the search command. Results of sample executions can be found in `MobileMaude/DataMobile/runs-mds.txt`.

6.5 Soup-of-Processes Prototyping

The third level of prototyping considers configurations that are a multiset of located processes, each containing its configuration of mobilized (wrapped) agents. In this case we define a test module `MP-TEST` that adds rules for objects of class `P` to move messages from output queues to the soup and soup to the input queue of the target object and a rule that lifts the `tick` rule to processes.

6.6 Soup-of-Processes Prototyping: Data Mobile example

The module `PDA-TEST` defines initial `DATAMOBILE` system configurations for the Soup-of-Processes level of abstraction. Again, in the base configuration, `pic`, there are three locations, a process object representing each location and containing a mobilized data agent. The first

process also contains a mobilized data mobile. The configuration `pic0` adds a starter beep message and on tell request to the base configuration. The configuration `pic2` adds an ask request to `pic0`.

```
ops pic pic0 pic1 pic2 : -> Configuration .
eq pic = proc0 proc1 proc2 .
eq pic0 =
  ( pic
    msg(P0,P0,
      route(msg(M0, user, dlV(upTerm(tellReq("a", "av")))),0))
    msg(P0,P0, route(msg(M0, MB, dlV(upTerm(beep))),0))
  ) .
eq pic2 =
  ( pic0
    msg(P1,P0, route(msg(M1, user, dlV(upTerm(askReq("a")))),0))
  ) .
```

These configurations can be rewritten using the position fair rewriting command. Even more rewriting is required for the data mobile to make a complete round due to two extra object layers. Again the state space is too large for use of the search command. Results of sample executions can be found in `MobileMaude/DataMobile/runs-pda.txt`. Two process versions are also defined for simpler testing. The difference being the itinerary given the data mobile.

7 Interacting with Mobile Agent Systems

The prototyping methods discussed above allow one to define an initial configuration and either rewrite, using some builtin rewrite strategy, or apply search or possibly model-checking tools to explore the space of possible executions. However, it is not easy to interact with the specified system by sending messages at intermediate stages, possibly based on previously received replies.

We also provide two ways to interact with an agent system using the IOP+IMAude environment. In the first case an initial configuration is placed in the IMAude execution environment which provides commands for the user to send messages to the configuration, receive messages, control rewriting, and examine intermediate configurations. The IOP+IMAude environment is described in [5] and in `MobileMaude/Env/doc-ienv.txt`. Instructions for running the data mobile example can be found in `MobileMaude/DataMobile/README.txt`.

In the second case each mobile process object is placed in a node execution environment, each running in a separate IMAude process (on the same or different machines). The node execution environment provides the user interaction capabilities that the simple IMAude execution environment does. In addition it supports interprocess communication using client and listener sockets. Instructions for running the data mobile example in the node execution environment can be found in `MobileMaude/DataMobile/README.txt`.

References

- [1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.
- [2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. <http://maude.cs.uiuc.edu>.
- [3] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, 2002.
- [4] Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.
- [5] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMAude: An Interactive Extension of Maude. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.
- [6] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [7] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36, 2002. invited paper.
- [8] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.