Principles of Mobile Maude

Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer

SRI International, Menlo Park, CA 94025, USA

Abstract. Mobile Maude is a Mobile Agent language extending the rewriting logic language Maude and supporting mobile computation. 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. Mobile Maude's key novel characteristics include: (1) reflection as a way of endowing mobile objects with "higher-order" capabilities; (2) object-orientation and asynchronous message passing; (3) a high-performance implementation of the underlying Maude basis; (4) a simple semantics without loss in the expressive power of application code; and (5) security mechanisms supporting authentication, secure message passing, and secure object mobility. Mobile Maude has been specified and prototyped in Maude. Here we present the Mobile Maude language for the first time, and illustrate its use in applications by means of Milner's cell-phone example. We also discuss security and implementation issues.

1 Introduction

Use of the Internet has exploded in recent years, and current technological trends may lead to new systems and business models based substantially on mobile code and mobile agents [9, 7]. It seems likely that within a few years most major Internet sites will be hosting some form of mobile code or mobile agents. As more and more applications come to depend on mobile computing, new risks for unintentional or malicious failures and for compromises of vital information must be avoided. Declarative mobile languages seem particularly promising to achieve high levels of confidence and security in mobile computing. This is because, by being directly based on formalisms with a precise semantics, there is a much shorter conceptual distance between the formal properties that must be ensured and the code. Furthermore, such formalisms can be *intrinsically concurrent*, further facilitating the programming and reasoning tasks.

One approach recently favored in declarative mobile language design is using *mobile calculi* that extend or modify the π-calculus [12] with new features, including mechanisms for encryption and security. Calculi of this kind include, among others, the Spi Calculus [1], and the Ambient Calculus [2]. In addition, there is a broader body of work favoring declarative approaches, including work in the related field of coordination languages (see Springer LNCS 1061, 1282, and 1594). There has also been a great expansion of the capabilities and security of agent-based languages such as OAA [10] and D'Agents [6].

Mobile Maude is an extension of Maude [3] supporting mobile computation that uses reflection in a systematic way to obtain a simple and general declarative mobile language design. 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 novel characteristics include:

- *Based on rewriting logic*, a simple first-order formalism that is intrinsically concurrent and has a clear mathematical semantics [11].
- *Extends Maude*, a high performance interpreter (1.66 million rewrites per second on a 500Mhz Alpha) and compiler (13 million rewrites per second on a 500MHz Pentium) implementation of rewriting logic.
- *Object-oriented and asynchronous*, with (mobile) objects as first-class entities in the language, and with direct support for asynchronous message-passing communication.
- *Reflective*: using rewriting logic reflection, the application code in a mobile object—that is, a rewrite theory $\mathcal{R}$—is metarepresented as *data*—namely, as a term $\overline{\mathcal{R}}$. This endows mobile objects with powerful "higher-order" capabilities within a simple first-order framework.
- *Simple rewriting semantics without loss in expressiveness*: the semantics of mobility is defined in an application-independent way by a small set of rewrite rules axiomatizing Mobile Maude's *system code*; however, *application code* inside mobile objects can be defined in Maude with great freedom and expressiveness.
- *Secure*, with underlying encryption primitives supporting authentication, secure message passing, and secure object mobility.

The above characteristics distinguish Mobile Maude from mobility calculi and from the other languages described above, and offer some novel advantages not available in such languages. In this paper, after briefly introducing rewriting logic, Maude, and reflection (Section 2) we give an overview of Mobile Maude and its rewriting semantics based on the current Mobile Maude simulator (Section 3) and discuss a simple mobile phone application (Section 4). Section 5 then discusses the design of Mobile Maude's security infrastructure; and Section 6 outlines our implementation plans. We end with some concluding remarks.

2 Rewriting Logic, Maude and Reflection

Rewriting logic [11] is a very simple logic in which the state space of a distributed system is formally specified as an algebraic data type by means of an equational specification consisting of a signature of types and operations Σ and a collection of conditional equations E. The *dynamics* of such a distributed system is then specified by rewrite rules of the form $t \to t'$, where t, t' are Σ-terms, that describe the *local, concurrent transitions* possible in the system, namely, when a part of the distributed state fits the pattern t, then it can change to a new local state fitting the pattern t'. Therefore, a *rewrite theory* is a triple (Σ, E, R), with (Σ, E) an equational specification axiomatizing a system's distributed state space, and R a collection of rewrite rules axiomatizing the system's local transitions. Rewriting logic has proved to be a general and flexible semantic framework for specifying a wide variety of models of concurrency [11], and in particular for specifying concurrent object systems.

Maude [3] is a multiparadigm executable specification language based on rewriting logic. Maude is a high-level reflective language and high-performance interpreter and compiler supporting rewriting logic specification and programming for a wide range of applications. Maude integrates an equational style of functional programming with an object-oriented programming style for highly concurrent object systems. Maude modules are rewrite theories whose basic axioms are rewrite rules.

2.1 Object-Oriented Modules

In Maude, object-oriented systems are specified by object-oriented modules in which *classes* and *subclasses* are declared. Each class is declared with the syntax

$$\texttt{class}\ C\ \texttt{|}\ a_1\texttt{:}\ S_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}\ S_n$$

where C is the name of the class, and for each $a_i : S_i$, a_i is an attribute identifier, and S_i is the sort over which the values of such an attribute identifier must range. Objects of a class are then record-like structures of the form

$$\texttt{<}\ O\texttt{:}\ C\ \texttt{|}\ a_1\texttt{:}\ v_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}\ v_n\ \texttt{>}$$

with O the name of the object, $v_1, \ldots, v_n$ the current values of its attributes, and with v_i of sort S_i for $1 \leq i \leq n$. Objects can interact with each other in a variety of ways, including the sending of messages. A message declaration

$$\texttt{msg}\ m : p_1 \ldots p_n \to \texttt{Msg}$$

defines the name of the message and the sorts of its parameters. Often, the first parameter refers to the object the message is sent to. The state of a concurrent object system is called a *configuration*. Typically, a configuration is a multiset of objects and messages. The multiset union operator for configurations is denoted with empty syntax (juxtaposition). It is associative and commutative so that order and parentheses do not matter, and so that rewriting is multiset rewriting supported directly in Maude. The *dynamic behavior* of a concurrent object system is then axiomatized by specifying each of its basic concurrent transition patterns by a corresponding labeled rewrite rule that rewrites a multiset of objects and messages into a new multiset of objects and messages, perhaps including new object identifiers and new messages.

2.2 Reflection and the META-LEVEL

Rewriting logic is reflective in the precise sense that there is a finitely presented rewrite theory $\mathcal{U}$ which is *universal*, that is, for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

where $\overline{\mathcal{R}}$, $\overline{t}$, and $\overline{t'}$ are terms representing, respectively, $\mathcal{R}$, t, and t' as data elements of $\mathcal{U}$.

Reflection is systematically exploited in the Maude design and implementation [3], providing key features of the universal theory $\mathcal{U}$ in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{\mathcal{R}}$ of a term t and a module $\mathcal{R}$ have sorts `Term` and `Module`, respectively.[1] Furthermore, `META-LEVEL` provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `meta-apply`, `meta-reduce` [3].

3 Mobile Maude

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 which can be used as a language simulator.

3.1 Processes and Mobile Objects

The key entities in Mobile Maude are processes and mobile objects; both are modeled as distributed objects in classes `P` and `MO`, respectively. Processes are *located* computational environments *inside which* mobile objects can exist, execute, and communicate. 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. Figure 1 shows several processes in two locations, with (mobile) object $o3$ moving from one process to another, and with object $o1$ sending a message to $o2$. The *names* of processes range over the sort `Pid`, whereas the names of mobile objects range over the sort `Mid` and have the from `o(PI,N)` with `PI` the name of the object's *parent* process, in which it was created, and `N` a number.

The class `P` of Mobile Maude processes is declared as follows,

```
class P | cnt : MachineInt,cf : Configuration,guests : Set[Mid],
          forward : PFun[MachineInt, Tuple[Pid, MachineInt]] .
```

[1] The key operator for the sort `Term` is of the form `_[_] : Qid TermList -> Term` where the sort `Qid` of quoted identifiers is used to metarepresent operator names. Then a term such as, for example, `X + Y` is metarepresented as `'_+_['X,'Y]` .

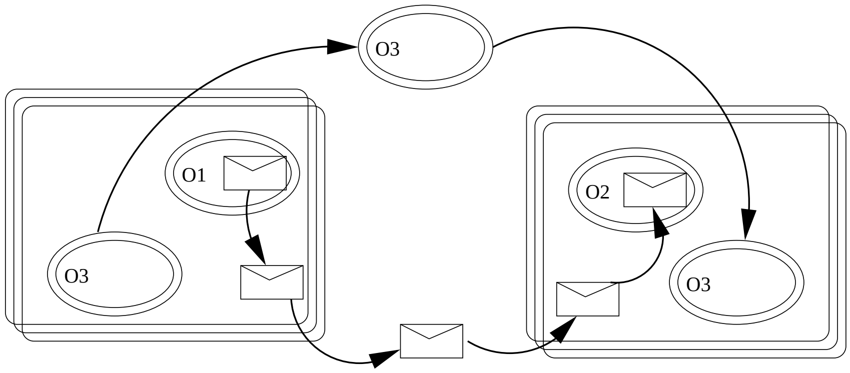

Fig. 1. Object and message mobility

Note the interesting fact that the attribute `cf` is itself a *configuration*. This means that processes exist as objects of an *outer* configuration of processes (plus messages and possibly mobile objects in transit) but also contain "in their belly" an *inner* configuration consisting of mobile objects and messages currently residing inside the process. The counter attribute `cnt` is used for creating new mobile objects. However, mobile objects can *move* from one process to another. For this reason, each process keeps information about the mobile objects currently in its belly in the `guests` attribute.

Since mobile objects may be moving from one process to another, reaching them by messages is nontrivial. The solution adopted in Mobile Maude is that, when a message's addressee is not in the current process, the message is forwarded to the mobile object's parent process. Each process stores forwarding information about the whereabouts of its children in its `forward` attribute, a table modeled as a partial function in
`PFun[MachineInt, Tuple[Pid, MachineInt]]` that maps child number n to a pair consisting of the name of the process in which the object currently resides, and the number of "hops" to different processes that the mobile object has taken so far. Whenever a mobile object moves to a new process, the object's parent process is always notified, so that forwarding information is updated.

Mobile objects are specified as objects of the following class `MO`,

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

Note that the sorts `Module` and `Term`, associated to the attributes `mod` and `s`, respectively, are sorts in the module `META-LEVEL`, that is imported by the specification of Mobile Maude. They metarepresent Maude modules and terms in such modules. The mobile object's *module* must be object-oriented, 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'`, with `C'` a multiset of *outgoing messages* that must be pulled out, and `C` containing unprocessed *incoming messages* and an *inner object*, with the *same identity* as that of the mobile object containing it. 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 mail contained in the inner circle. The remaining attributes are quite straightforward. The process where the object currently resides is stored in the `p` attribute. The number of "hops" from one process to another is stored in the `hops` attribute. It is also important to bound resource consumption; this is done with the `gas` attribute. Finally, an object's `mode` is only `active` inside the belly of a process: moving objects are `idle`.

3.2 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. Such a specification is executable and can be used as a Mobile Maude simulator. We should think of such rules as the *system code* of Mobile Maude, that operates in an application-independent way providing all the object and process creation, message passing, and object mobility primitives. By contrast, all *application code* is encapsulated in a metarepresented form within mobile objects and is executed at the metalevel inside such objects.

We give the flavor of Mobile Maude's rewriting semantics by commenting on four rules: three for object mobility, and one for mobile object execution. There are of course other rules in the same style dealing with message communication, mobile object and process creation, and so on. The entire specification can be found in `http://maude.csl.sri.com`.

The three rules below govern object mobility. Such mobility is initiated by the mobile object's inner object, which puts the metarepresentation `'go[T']` in the second component (i.e., as an outgoing message) of the state. The term `T'` metarepresents the name of the process where the object wants to go. The rule `message-out-move` indicates how such a name is decoded by the `downPid` function, and shows in its righthand side the mobile object ready to go—which is indicated by being enclosed inside a `go` operator.

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

rl [go-proc] : < PI : P | cf : C go(PI', < M : MO | >),guests : M . SMO >
  => if PI =/= PI'
      then < PI : P | cf : C, guests : SMO > go(PI',< M : MO | >)
      else < PI : P | cf : C < M : MO | p : PI,mode : active > > fi .

rl [arrive-proc] : go(PI,< o(PI', N) : MO | hops : N' >)
   < PI : P | cf : C,guests : SMO,forward : F >
  => if PI == PI' then
```

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

The rule `go-proc` then initiates the move of the object from its current process (`PI`) to another process `PI'` by extracting it from `PI` and putting it in the outer configuration. Of course, if the process where the object wants to go is the process in which it is currently located, the process remains where it is (with the right process name in its `p` attribute, and in active mode) and the `go` operator is removed. The `arrive-proc` rule then finishes the motion by inserting the mobile object inside the belly of the target process in active mode and with updated information about its current process and number of hops, and includes the object's name in the set of current guests. However, if the destination process happens to be the parent process, then the forwarding information has to be updated; otherwise, the parent process has to be informed of the mobile object's whereabouts by means of the message `to PI' @ (PI, N' + 1) { N }`.

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

```
crl [do-something] : < M : MO | mod : MOD, s : T, gas : N, mode : active >
   => < M : MO | s : meta-rewrite(MOD,refill(T, N), 1),gas : N - 1 >
      if (N > 0) and (refill(T, N) =/= meta-rewrite(MOD, refill(T, N), 1)) .
```

The idea is that we advance the execution of the object by meta-rewriting its state `T` at the metalevel using the rules represented in the module `MOD`. However, two precautions must be taken. Firstly, to avoid "runaway" mobile objects, the number of rewrites must be bound by the amount of gas available; for this, we refill the state with "gas" before performing the computation. Secondly, we should avoid wasting gas in trying to recompute idle states that cannot make any progress; this is accomplished by checking in the condition that we have indeed reached a new state.

4 A Mobile Cell Phone Example

In [12], Milner presents a simple mobile telephones example to illustrate the π-calculus. We use a variant of this example to illustrate how mobile *application code* can be written in Maude and can be wrapped in mobile objects. As already explained in Section 3, Mobile Maude *systems code* consist of a relatively small number of rules for processes, mobile objects, mobility, and message passing. Such rules work in an *application-independent* way. Application code, on the other hand, can be written as Maude object-oriented modules with great freedom, except for being aware that, as explained in Section 3.1, 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 and incoming messages.

The example consists in a set of *cars*, with mobile phones in them, which move around a large geographical area. The mobile phones are always in contact with one of the *bases*, which communicate among themselves through a control *center*. When the center detects that a car is approaching a new base, it sends messages to the base currently in contact with the car asking it to release the car, and to the new base, asking it to get in contact with it.

We represent cars, bases, and centers as objects of respective classes `Car`, `Base`, and `Center`. Such objects in the application code will then be embedded as *inner objects* of their corresponding mobile objects. The rules of Mobile Maude discussed in Section 3.2 will then support the mobility and message passing mechanisms of the mobile objects.

We assume that the mobile objects encapulating different bases are located in different process, and that the mobile object encapsulating a car is in the process of the base it is currently in contact with. We also assume that there is a single mobile object encapsulating the center, which is located in a different process. The center causes a car mobile object to move from the process of the base it is currently in contact with to the process of the new base it switches to. Figure 2 shows a possible configuration of two bases, the center, and two cars.

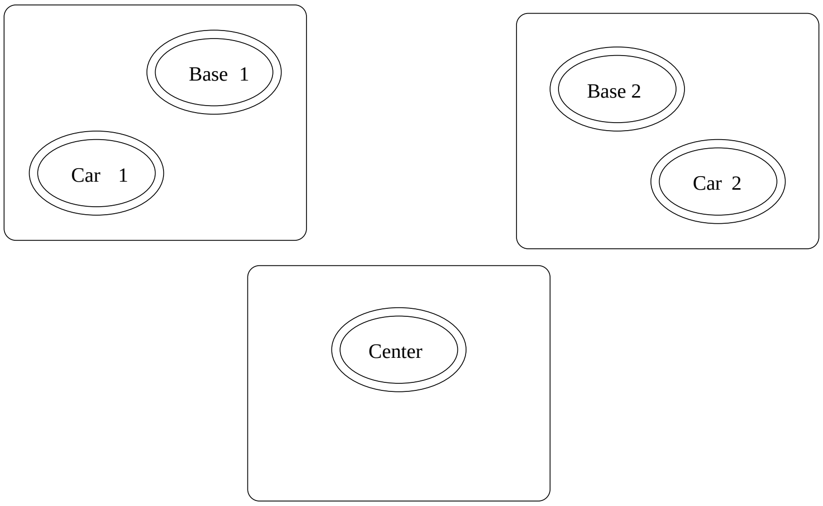

Fig. 2. A configuration with two cars, two bases, and one center.

An objects of the `Car` class has two attributes, `base`, storing the name of the base it is currently in contact with, and `phone-book`, storing the names of other cars it knows about and can call.

```
class Car | base : Oid, phone-book : Set[Oid] .
```

An object of the `Base` class has two attributes, `cars`, storing the names of the cars the base is in contact with, and `center`, the name of the center.

```
class Base | cars : Set[Oid],center : Oid .
```

The center is an object of class `Center`. The connection information is a partial function, mapping each car name to the base it is currently in contact with, stored in the `control` attribute. In addition, the center stores the names of all bases and cars in the `bases` and `cars` attributes.

```
class Center | control : PFun[Oid, Oid], bases : Set[Oid], cars : Set[Oid] .
```

Cars talk to each other through their respective bases, with the center mediating the conversations between cars in different bases. For simplicity, we model any words said to a car object `O` by means of the term `talk(O)`. A car object can initiate a conversation by choosing another car name in its phone book and placing a message addressed to its current base with such a request in the outgoing messages part of the state, according to the rule

```
rl [talk] : < O : Car | base : O', phone-book : O'' + OS > C & none
                          => < O : Car | > C & (to O' : talk(O'')) .
```

Other rules then govern the handling of the request by the base, which forwards it to the other car if it is also in contact with it, or otherwise asks the center to forward it to the appropriate base. Such car talking rules, and the entire example, can be found in `http://maude.csl.sri.com`.

We focus instead on the application code rules that—when such code is encapsulated in corresponding mobile objects—cause car mobile objects to move from one process to another. We assume that, by GPS or some other mechanism not modeled here, the center can know the positions of the cars and the bases, and therefore can detect that a car is approaching a new base different from the one it is in contact with, and can then start the switching of the connections. The center then sends a `release` message to the base connected to such a car and an `alert` message to the new base. When a base receives a `release` message it sends a `switch` message to the car with the identifier of its new base. When a base receives the `alert` message it saves the identifier of the new car connected with it. The reception of a `switch` message by a car causes it to connect to a new base, by updating its `base` attribute, and by placing the `go` command in the outgoing messages part of the state, causing the mobile object encapsulating the car to move to the process in which the new base is located (see rule `message-out-move` in Section 3.2).

```
crl [switch] : < O : Center | control : PF,bases : O' + O''' + OS,
                              cars : O'' + OS' > C & none
  => < O : Center | control : PF[O'' -> O'] > C & (to O' : alert(O''))
     (to O''' : release O'' to O') if PF[O''] == O''' .
```

```
rl [release] : < O : Base | cars : O' + OS >
               (to O : release O' to O'') C & none
  => < O : Base | cars : OS > C & (to O' : switch(O'')) .

rl [alert] : < O : Base | cars : OS > (to O : alert(O'))
  => < O : Base | cars : O' + OS > .

rl [move] : < O : Car | > (to O : switch(o(PI, N))) C & none
  => < O : Car | base : o(PI, N) > & go(PI) .
```

5 Mobile Maude Security

In Mobile Maude, like in all mobile-agent systems, four key security issues should be addressed: protecting an individual machine (physical machine or in general any execution environment) from malicious agents, protecting a group of machines from various forms of attack, protecting agents from malicious hosts, and protecting groups of agents from one-another. For example, to protect an individual machine from malicious agents, we employ cryptographic authentication of the agent's authority, resource constraints and fairness management based on the agent's identity, and secure execution environments that provide strong partitioning (if not some level of noninterference) and enforce the decisions of the resource manager. No mobile agent systems today meet the strongest form of all these security needs. Mobile Maude provides a platform with some of these kinds of secure services built-in, and the Maude reflective framework provides an excellent vehicle for experimentation with techniques to provide a complete secure solution for mobile agents. Specifically, Mobile Maude supports the privacy of data through encryption, the authentication of communications through a public-key infrastructure service, the security of mobility (built on both of the above), and the reliability and integrity of computation by means of redundant checks.

5.1 Mobile Maude Security Infrastructure

The security features of Mobile Maude are provided as optional features for those applications which require one or more properties of security, privacy, authenticity, and integrity. The Mobile Maude object system is used to define subclasses of Mobile Maude objects, processes, and locations which can provide the required features as needed. For example, the birthplace of an object acts as the authoritative signature key repository, and birthplaces are authenticated in a hierarchy back to a predefined set of primordial sources of trust. Thus to authenticate a message, a receiver can contact the birthplace of the sending object, and securely obtain the public key of that object, and then authenticate that the message was properly signed using PGP or similar secure signature scheme. This is similar to the Telescript [13], D'Agents [6], and IBM Aglets [8] In Aglets, for example, migrating Java code is cryptographically signed and then standard Java security resource mechanisms are enforced. In Mobile Maude the communication to the birthplace must also be carried out using authentication, and thus authentication keys for all birthplaces must be obtained from the roots of trust entrusted with public keys. When a Maude mobile object changes locations, it can sign its own state, and then encrypt itself using the key of its intended destination process. In this way, as the object moves over untrusted communication paths, the object cannot be altered or interrogated, and can maintain valuable secrets (such as it's own signature key) even through multiple moves around an untrusted network.

The design of the security features minimizes the performance impact on Mobile Maude objects which do not utilize the features. By providing the main functionality only in subclasses of the Process and Mobile Object classes, Mobile Maude avoids most overhead costs. Certain minimal infrastructure is provided in all Mobile Maude processes and locations to support queries and to provide authoritative answers to secure object lookup requests.

5.2 Secure Message Passing and Mobility

There are several approaches to providing security for messages and mobility in a mobile context. A simple and efficient but relatively insecure approach is security by obscurity, where processes or objects are inaccessible if one doesn't know their name. By hiding the name of a location, process, or object, only those objects which already know the name can communicate with it. We adopt in addition the encryption of messages and objects in transit. Objects, processes, and locations which send or receive messages encrypt the communication and forward it so that malicious network manipulation cannot read or modify the communicated content.

5.3 Resource Bounds and Fairness

Another key aspect of Mobile Maude is the provision of fairness and other guaranteed bounds on resource allocation. The fairness provided by Mobile Maude prevents some kinds of denial-of-service attacks by ensuring that all processes are allowed to make progress, even in the presence of large numbers of spurious messages or objects. However, more aggressive bounds on object-generation and message-generation can be provided and can be combined with authentication techniques to ensure even stronger guarantees on performance.

6 Implementation Approaches

The implementation of Mobile Maude presents several technical challenges. We have thus far built a single-host prototype single-host specification environment on top of Maude 1.0 which is available at our web site
`http://maude.csl.sri.com/mobile-maude.html`. We plan to construct Mobile Maude in three stages. The first stage enables language and application development, by providing a single-host Mobile Maude specification and execution

environment. The second stage uses the existing Maude implementation, running on multiple hosts, extended with objects and socket-support to send mobile objects and messages over the network. The third stage is a full distributed mobile implementation of Mobile Maude, utilizing new protocols for object motion and management and separate processes for each Mobile Maude process, with new security features providing optional protection for mobile objects.

For the single host executable specification, we will build upon the forthcoming Maude 2.0 interpreter/compiler, utilizing the builtin object system (MOS), for object/message fairness. The Mobile Maude system code will be written entirely in Maude, and thus locations and processes will be encoded as Maude terms. This implementation effort will be completed rapidly once Maude 2.0 is available, by simplifying and extending the existing specification.

For the second implementation effort, we will focus on true distributed execution. We will define and build a very simple Mobile Object Transfer Protocol (MOTP) 0.1 on top of TCP/IP. We will implement Mobile Maude servers written in Maude, using the built-in string handling and internet socket modules planned for Maude 2.0. The Maude 2.0 socket modules will support non-blocking client and server TCP sockets (at the OS level) and will make use of the concurrency inherent in the Maude object-message model rather than relying on threads as found in legacy programming languages. In this implementation effort, a Mobile Maude server will run on top of a Maude 2.0 interpreter, keeping track of the current locations of mobile objects created on a host, handle change of location messages, reroute messages to mobile objects and run the code of mobile objects by invoking the metalevel. In this implementation, processes could be simulated (i.e. encoded as Maude terms) or actual if forking new interpreters or JIT Maude compilation is made available from within Maude 2.0. The total implementation effort here is moderate — over and above the effort need to implement Maude 2.0, since much of the infrastructure including implementation of MOTP 0.1 is done using the Maude 2.0 language.

As we move toward an industrial stength implementation, we will define a backward-compatible MOTP 1.0 with error dectection and recovery, strong cryto/authenication, on-the-fly compression, persistent sockets, internationalization/unicode support time sensitive requests/responses, proxy support, proof-carrying code, and fault tolarence/redundancy/redirection. We will reimplement parts of the Mobile Maude infrastructure in a compiled imperative langauge such as C++. This will be simplified by using foreign modules to provide primitives for data intensive operations, and by the increasing availability of functionality in further releases of Maude, including the anticipated advances in Maude compilation techniques. The implementation effort here will be large — a general-purpose MOTP might have support for and/or interoperability with other mobile languages and require a lengthy design/review process.

Taken together, these implementations are the first steps toward a semantically-sound highly-dependable intelligent internet infrastructure for mobile code.

7 Concluding Remarks

We have presented the basic concepts of Mobile Maude, a new declarative mobile language extending Maude and based on rewriting logic reflection. We have explained the language's semantics based on its current specification, which serves also as a simulator for the language. We have also illustrated the use of Maude with a simple mobile cell phone application. Security and implementation issues have also been discussed.

Much work remains ahead. We are currently using the simulator to stress the language with more challenging applications, and we are using Maude to specify and prototype the security infrastructure. All this, together with the Maude 2.0 version currently under development, will give us a good basis on which to build the different implementations described in Section 6.

Mobile Maude's simple declarative semantics offers the promise of being able to reach high levels of assurance about the language itself and about specific applications through the use of a variety of formal methods. This promise has to be fulfilled by developing adequate formal methodologies, and by demostrating how it can be attained in practice for substantial applications. The encouraging experience using a flexible range of formal methods in Maude (see the survey [5]) and the formal tools already available [4] will certainly help in this regard. The planned implementations and experiments, together with their formal methods support, are aimed toward the construction of a semantically-sound highly-dependable intelligent internet infrastructure for mobile agents, an infrastructure capable of supporting with high assurance new classes of critical applications within the mobile agent computing paradigm.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148:1–70, 1999. An extended version of this paper appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998.
2. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of FoSSaCS'98: Foundations of Software Science and Computational Structures*, number 1378 in Lecture Notes in Computer Science, pages 140–155. Springer-Verlag, 1998. To appear in TCS July 2000.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.csl.sri.com`.
4. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.csl.sri.com`.
5. G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: the Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, pages 251–265. IEEE, 2000.

6. R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, LNCS 1419, pages 154–187. Springer-Verlag, 1998.
7. D. Kotz and R. S. Gray. Mobile agents and the future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, August 1999.
8. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
9. D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the Association for Computing Machinery*, 42:88–89, March 1999.
10. D. Martin, A. Cheyer, and D. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13:91–128, 1999. Available via `http://www.ai.sri.com/~cheyer/papers/aai/oaa.html`.
11. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
13. J. White. Telescript technology: the foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.