Maude meets CORBA

Antonio Albarrán[1], Francisco Durán[2], and Antonio Vallecillo[2]

[1] Junta de Andalucía. Sevilla, Spain.
{aalbarran}@ceh.junta-andalucia.es
[2] ETSI Informática. Universidad de Málaga. Spain.
{duran,av}@lcc.uma.es

Abstract. In this paper we show how to connect the worlds of a formal specification language such as Maude and of a commercial object platform for open distributed systems such as CORBA. In addition to the usual advantages provided by the use of formal specifications, by allowing objects in any of these worlds (specification and implementation) to seamlessly interoperate we can obtain several interesting advantages, such as softening the gap between system specifications and their implementations, building prototypes in which specifications and final implementations are combined, and directly using Maude specifications for testing component implementations.

1 Introduction

There seems to be a growing consensus on the use of ready-to-use components for building software systems, in order to reduce development time and efforts, and to improve software quality. Whether using components or not, software systems have to be developed following rigorous guidelines to have a chance of being dependable, finished on time, and easy to maintain. Typically, the software development process is seen as the construction of a sequence—up to backtracking and iterations—of more and more detailed descriptions of the software under development, leading to a final set of elements that contain a set of executable programs and their documentation.

Making use of formal specifications is a demanding process and should be suitably targeted. From a logical point of view, the correctness of a program is a nonsense without a formal specification against which correctness can be proved. However, there are many cases where formal specifications would be a mere luxury. In most projects where formal specifications are used, formal and informal specifications are mixed, so that some components or refinement steps are formalized and others not. The decision to use formal specifications mainly depends on the criticality of the component, in terms of consequences of a fault (human lives, costs, etc.) and on the complexity of its requirements or its development. Besides, formal and informal specifications cannot be smoothly integrated, nor can they easily interoperate. Therefore, the verification of formal specifications must be accomplished in isolation, that is, they can be proved to be correct but it cannot be proved that they will do what they are supposed

to do when combined with other components informally specified—unless these and their interactions are also specified formally.

The formal or informal global specification of a system is valuable by itself as a way of establishing a description of the system on which the developer and the client agree. In component software, we generally begin with a high-level graphical description of the system, using, for example, a UML-like notation for identifying the different components and their relations. Component interfaces are then identified and specified. Such specifications can be viewed as contracts between clients of an interface and the provider of an implementation of it, and they are used as descriptions of the components for specifying the interactions among the different components via some coordination or architecture description language. Typically, interface specifications include a description of the functional aspects of the operations, including its syntax and its semantics, given by invariants and pre- and post-conditions. It can be expected that non-functional specifications will also be included into such contracts (see, e.g., [13]).

Over the last years, there has been a big effort by the research community for increasing the usability of formal methods, developing simpler to use and more powerful tools which can be more easily integrated into the software development process. Theorem proving, prototyping, model checking, static analysis, code generation, and testing are the kinds of activities where formal specifications have shown to play a useful role, with notations and tools such as B [1], PVS [10], SPIN [6], etc. For the present paper, the possibility of prototyping is particularly relevant: if the specifications can be executed (or at least animated) they may be used for getting rapid prototypes of the systems, which may then be used for completing and clarifying such specifications.

One of the benefits of component-based systems is the clear separation of functionality between their constituent parts, and the 'loose' connection among them. With the use of component platforms such as CORBA [9], different components can communicate without any assumption on their implementation language, the hardware they run on, or their physical location. This kind of component interaction (only through their interfaces) may help the reasoning process about the systems being built.

In this paper we explore the possibility of connecting the worlds of a formal specification language such as Maude [3] and of a commercial object platform for open distributed systems such as CORBA. Maude is an executable rewriting logic language particularly well suited for the specification of object-oriented open and distributed systems [8]. CORBA is one of the leading object and component platforms in the market, specially well suited for the implementation of distributed applications in open systems. By allowing objects of both worlds to seamlessly interoperate we can obtain several interesting results.

In the first place, suppose that we have the specification of a system written in Maude, in terms of the specifications of its constituent components. We can prove some properties about such a system specification, do some model-checking, or consider it as a prototype of the system by simply executing its Maude specifications. However, there is a gap between the system specification

and a running implementation of it. In our proposal, Maude objects can be transparently replaced by CORBA objects (i.e., their implementations) one by one, so that objects in both worlds coexist, while still being able to reason about such a system. In other cases it may be just a question of time, we do not need to wait until all the components are available, we can keep using their specifications in the running system until we get their final implementations.

Second, black-box test suites for CORBA objects can be easily specified (or automatically generated from their specifications) in Maude based on their interfaces only. Those specifications can be later executed directly against the actual implementations of the CORBA objects, thus testing their behavior.

Third, given a running CORBA system and the specification of a new CORBA object (or a new version of one of its components), we can validate the new system without having to implement such a new object. We can just 'drop' the new specification into the running system, and check its behavior.

Finally, and most importantly, in safety-critical systems the vital parts can be formally specified and their behavior formally verified, while non-critical components can be just informally specified and implemented. The system, composed of Maude specifications and some CORBA objects, is now available for execution without having to wait for all critical components to be implemented.

In this paper we show how the worlds of Maude and CORBA can be smoothly integrated, obtaining all these benefits. Sections 2 and 3 briefly describe CORBA and Maude, respectively. Then, Section 4 presents a simple example application to illustrate our proposal, and Section 5 briefly describes how to connect both worlds. Finally, Section 6 draws some conclusions and describes some future research activities.

2 CORBA

CORBA is one of the major distributed object platforms. Proposed by the OMG (`www.omg.org`), the Object Management Architecture (OMA) attempts to define, at a high level of description, the various facilities required for distributed object-oriented computing. The core of the OMA is the Object Request Broker (ORB), a mechanism that provides transparency of object location, activation and communication. The Common Object Request Broker Architecture (CORBA) specification describes the interfaces and services that must be provided by compliant ORBs [9].

In the OMA model, objects provide services, and clients issue requests for those services to be performed on their behalf. The purpose of the ORB is to deliver requests to objects and return any output values back to clients, in a transparent way to the client and the server. Clients need to know the *object reference* of the server object. ORBs use object references to identify and locate objects to redirect requests to them. As long as the referenced object exists, the ORB allows the holder of an object reference to request services from it.

Even though an object reference identifies a particular object, it does not necessarily describe anything about the object's interface. Before an application can

make use of an object, it must know what services the object provides. CORBA defines an IDL to describe object interfaces, a textual language with a syntax resembling that of C++. The CORBA IDL provides basic data types (such as `short`, `long`, `float`, ...), constructed types (`struct`, `union`) and template types (`sequence`, arrays, ...). These are used to describe the interface of objects, defined by set of types, attributes and the signature (parameters, return types and exceptions raised) of the object methods, grouped into `interface` definitions. Finally, the construct `module` is used to hold type definitions, interfaces, and other modules for name scoping purposes.

3 Rewriting Logic and Maude

Rewriting logic [7] is a logic in which the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of types (sorts) and operations, and E is a set of (conditional) equational axioms. The dynamics of such a system is then specified by rewrite *rules* of the form $t \longrightarrow t'$, where t and t' are Σ-terms that describe the local, concurrent transitions possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

Maude [3] is a high-level language and a high-performance interpreter and compiler that supports equational and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. This logic can naturally deal with state and with highly nondeterministic concurrent computations; in particular, it supports very well concurrent object-oriented computation [8].

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 "`class` $C \mid a_1 : S_1, ..., a_n : S_n$", where C is the name of the class, a_i are attribute identifiers, and S_i are the corresponding sorts of the attributes. Objects of a class C are then record-like structures `<` $O : C \mid a_1 : v_1, ..., a_n : v_n$ `>`, where O is the name of the object, and the v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing.

In a concurrent object-oriented system the concurrent state, which is called the *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the *communication events* between some objects and messages. The general form of such rewrite rules is

$$
\begin{array}{l}
\texttt{crl}\ [r]\ :\ M_1 \ldots M_m \\
\qquad\quad < O_1 : C_1 \mid atts_1 > ... < O_n : C_n \mid atts_n > \\
\qquad\quad \texttt{=>}\ < O_{i_1} : C'_{i_1} \mid atts'_{i_1} > ... < O_{i_k} : C'_{i_k} \mid atts'_{i_k} > \\
\qquad\qquad\quad < Q_1 : D_1 \mid atts''_1 > ... < Q_p : D_p \mid atts''_p > \\
\qquad\qquad\quad M'_1 \ldots M'_q \\
\qquad\quad \texttt{if}\ \ Cond\ .
\end{array}
$$

where r is the rule label, M_i are messages, O_i and Q_j are object identifiers, C_i, C'_j and D_h are classes, $i_1, \ldots, i_k$ is a subset of $1 \ldots n$, and $Cond$ is a boolean condition (the rule 'guard'). The result of applying such a rule is that: (a) messages $M_1 \ldots M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1}, \ldots, O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1, \ldots, Q_p$ are created; and (e) new messages $M'_1 \ldots M'_q$ are created, i.e., they are sent.

When several objects or messages appear in the left-hand side of a rule, they need to synchronize in order for such a rule to be fired. Thus, such rules are called *synchronous*, while rules involving just one object and one message in their left-hand side are called *asynchronous* rules.

4 A Case Study

In order to illustrate our proposal, let us suppose that we have already designed a system (using UML or any other notation), and that we have decided to implement it in CORBA. Among other entities, the system contains a bank account object, which uses a calculator for doing some computations. This is a simple example, but will serve us to illustrate how to model the typical client-server interactions between two objects. The CORBA interfaces of these components are as follows.

```
interface Account {
 exception NotEnoughMoney{};
 float balance();
 void deposit(in float amount);
 void withdraw(in float amount)
          raises (NotEnoughMoney);
};
```

```
interface Calculator {
 float plus(in float a, in float b);
 float minus(in float a, in float b);
};
```

Imagine that they are critical components, whose behavior needs to be formally specified in order to be able to reason about them. Using Maude, their specifications can be written as given below.

First, let us consider the specification of a very basic calculator, with support for plus and minus operations only, as described in its CORBA interface. We declare a class `Calculator` without any attributes, which can receive messages `plusRequest` and `minusRequest`, with the integer numbers to operate on plus the identifiers of the addressee and the sender objects as arguments. Responses to such messages are given as messages `plusResponse` and `minusResponse`, each of which have, in addition to the arguments of the requested operation, its corresponding result.

```
(omod CALCULATOR is
  protecting CORBA-BASIC-TYPES .
  class Calculator | .
  msgs plusRequest minusRequest : float float Oid Oid -> Msg .
  msgs plusResponse minusResponse : float float float Oid Oid -> Msg .
```

```
  vars O O' : Oid .  vars X Y : float .

  rl [plus]: < O : Calculator | >
             plusRequest(X, Y, O, O')
             => < O : Calculator | >
                plusResponse(X, Y, (X + Y), O', O) .

  rl [minus]: < O : Calculator | >
              minusRequest(X, Y, O, O')
              => < O : Calculator | >
                 minusResponse(X, Y, (X - Y), O', O) .
endom)
```

With regard to the class `Account` in the module `BANK-ACCOUNT`, it is declared with attributes `bal`, `client`, and `calc` for storing, respectively, its balance, its current client (if any), and a reference to the calculator it will use if necessary. The module `DEFAULT[Oid]` provides the sort `Default[Oid]` for object identifiers with a default value `null`. Note that each of the methods gives rise to a pair of messages request-response as in the case of the calculator, and that the exception `notEnoughMoney` is modeled as an additional message to be sent in case a withdraw operation cannot take place.

```
(omod BANK-ACCOUNT is
  protecting CORBA-BASIC-TYPES + DEFAULT[Oid] .
  class Account | bal : float, client : Default[Oid], calc : Oid .
  msgs depositRequest withdrawRequest : float Oid Oid -> Msg .
  msgs balanceRequest : Oid Oid -> Msg .
  msgs balanceResponse : float Oid Oid -> Msg .
  msgs depositResponse withdrawResponse notEnoughMoney : Oid Oid -> Msg .
  vars C A O : Oid .  vars B X Z : float .

  rl [balance]: < A : Account | bal : B >  balanceRequest(A, O)
                => < A : Account | > balanceResponse(B, O, A) .

  rl [deposit]: < A : Account | bal : B, client : null, calc : C >
                depositRequest(X, A, O)
                => < A : Account | client : O >
                   plusRequest(B, X, C, A) .

  rl [depositResponse]: < A : Account | bal : B, client : O >
                        plusResponse(B, X, Z, A, C)
                        => < A : Account | bal : Z, client : null >
                           depositResponse(O, A) .

  crl [withdraw]: < A : Account | bal : B, client : null, calc : C >
                  withdrawRequest(X, A, O)
                  => < A : Account | client : O >
                     minusRequest(B, X, C, A)
                  if B >= X .
```

```
 rl [withdrawResponse]: < A : Account | bal : B, client : O >
                        minusResponse(B, X, Z, A, C)
                        => < A : Account | bal : Z, client : null >
                           withdrawResponse(O, A) .

 crl [withdraw-notEnoughMoney]: < A : Account | bal : B, client : null >
                                withdrawRequest(X, A, O)
                                => < A : Account | >
                                   notEnoughMoney(O, A)
                                if B < X .
endom)
```

As we can see in this specification, a call to the account's `deposit` or `withdraw` methods cause a call to the calculator's `plus` or `minus` methods to do the calculations, if no exception is raised. The bank account is then a client of the calculator, whose reference is kept as an account object attribute. Note also the use of the `client` attribute to make sure that no more than one request is attended at a time by allowing a new deposit or withdraw only when there is no calculation going on requested by some client. In such case the `client` attribute has value `null`. In Maude, as described in [3], those attributes of an object that are not relevant for an axiom do not need to be mentioned, and the attributes mentioned only on the left hand side of a rule are preserved unchanged.

It is important to note how CORBA object interactions have been modeled in Maude:

1. First, the `CORBA-BASIC-TYPES` Maude module contains the specifications of the CORBA basic types (such as "long", "float", etc.). CORBA constructed types are naturally modeled in Maude by using its facilities for specifying ADTs. Generic sorts, such as sequences, sets, lists, arrays and partial functions are also easily specifiable in Maude.
2. The natural communication mechanism between CORBA objects is based on RPCs, following a client-server style. We have modeled this kind of object interactions by using two Maude messages for each RPC, one that carries the request and one for the reply.
3. Maude does not require the source and target objects to be specified in the messages. However, in a CORBA method invocation the target object should be specified, as well as the object taking care of the reply (it is implicit in a normal RPC invocation, or explicitly mentioned in CORBA *asynchronous* calls [9]). In order to model this in a systematic way, two special arguments have been included at the end of all Maude messages representing CORBA calls or responses: the identifiers of the target and the source objects.
4. CORBA methods may also raise exceptions. We have modeled exceptions by using additional messages, that may be returned by the server instead of the normal response message. Thus, a new message is defined for each exception, whose name is the name of the exception, and whose arguments are the ones defined by the exception—together with the source and target

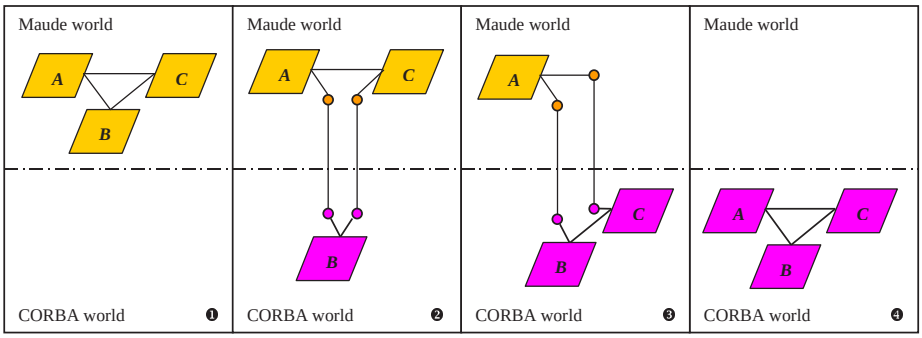

Fig. 1. Replacing Maude objects by their CORBA implementations.

object identifiers, as usual. When serving a method, the server object may decide to terminate normally—then sending the normal response message to the method originator—or to raise one of its declared exceptions—thus sending the corresponding message.

5 Connecting Both Worlds

Once we have specified a system and proved that it is safe, or that it satisfies any other property we are interested in, we are ready to build an implementation for it. Instead of going from the specification to the implementation in one single step (implementing the different components or picking up components already available), we propose doing it gradually. In this section we illustrate how we can replace one by one the components in a system specification and being able to execute and test the running system in each of the steps. Let us suppose, for example, that we have sourced a CORBA implementation of the calculator interface from a third party, and that we want to execute it, testing it against the specifications of its client, i.e., the bank account. Let us see how both worlds—CORBA and Maude—can be integrated, and how their objects interoperate.

Our integration is based on the use of proxies, that replace the object in one world, capture their incoming calls and pass them to another proxy in the second world. This second proxy simulates the client calls in the other world, awaits for the server response, and then passes it back to the first proxy, which builds the final reply from it. Communication between objects and proxies is by using the natural communication mechanisms available in each world, and communication between proxies is implemented via TCP, using the features available in Maude 2.0 [4].

Figure 1 shows a gradual transition from a Maude specification into a running CORBA system. The three Maude components in the left are replaced by their corresponding implementations step by step. All objects are of course unaware of the replacement process, since we use proxies that substitute the

Maude components being replaced. This is just an example, since the number of components that can be replaced in each step may vary depending on the implementor's choice.

5.1 Implementing the Calculator

Let us see what those proxies look like in the particular cases of the bank account and the calculator. Suppose that we want to replace the calculator specification by an implementation of it, making the bank account specification interact with it. In the first place we need a proxy of the calculator in the Maude world. In the following module `CALCULATOR-PROXY`, `TCPHandler` is the Maude module providing TCP communication, with messages `sendTCP` and `receivedTCP` for sending and receiving messages, respectively. Note that in addition to the predefined `System` object reference and the sender and addressee objects references, the message to be sent by TCP itself is given as a string. We use XML for describing messages, although any other notation could be used. Note also that we give the rules for the messages related to the `plus` method only. There are similar rules for `minus`.

```
(omod CALCULATOR-PROXY is
  protecting TCPHandler + STRING + DEFAULT[Oid] + CORBA-BASIC-TYPES .
  class Calculator | CORBAProxyAddr : String, client : Default[Oid] .
  msgs plusRequest minusRequest : float float Oid Oid -> Msg .
  msgs plusResponse minusResponse : float float float Oid Oid -> Msg .
  vars O O' : Oid . vars X Y Z : float . var A : String .

  rl [plusRequest]:
    < O : Calculator | CORBAProxyAddr : A, client : null >
    plusRequest(X, Y, O, O')
    => < O : Calculator | client : O' >
       sendTCP(System, O, A, "<m:plusRequest>" +
                             "<a type=\"float\">" + X + "</a>" +
                             "<b type=\"float\">" + Y + "</b>" +
                             "</m:plusRequest>") .
  rl [plusResponse]:
    < O : Calculator | client : O' >
    receivedTCP(O, System, "<m:plusResponse>" +
          "<a type=\"float\">" + X + "</a>" +
          "<b type=\"float\">" + Y + "</b>" +
          "<result type=\"float\">" + Z "</result>" +
          "</m:plusResponse>")
    => < O : Calculator | client : null > plusResponse(X, Y, Z, O', O) .

  *** rules for "minus" omitted for brevity

endom)
```

An object of the class `Calculator` in this module is the one that provides now the calculator, which accepts the same methods as the original calculator.

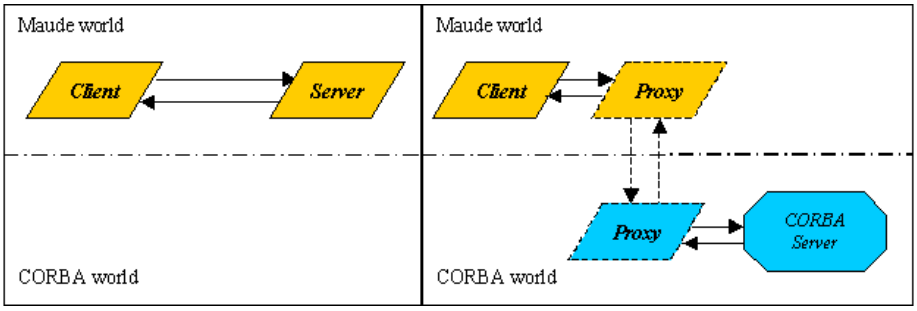

Fig. 2. Replacing a Maude server by its CORBA implementations.

However, it does not carry out the computations any longer: it just passes the information around, packing and unpacking messages into the appropriate formats. Upon reception of a method call, this proxy sends by TCP the request to the appropriate CORBA proxy (whose IP address it keeps as an attribute). And once the response from the CORBA proxy is received, it sends back the answer to the original client. Figure 2 depicts this situation.

It is important to notice that the generation of this proxy can also be automated directly from the CORBA IDL of the original object.

At the CORBA side, the structure of the CORBA proxy is very similar. The following piece of code shows an implementation for it in Java.

```
Calculator calc = ....; // obtains the CORBA object ref. for calculator
String addr = "150.214.108.1:2020";          // IP:socket of MaudeProxy
for(;;) {
  String s = SocketReadLn(addr);
  if (getMethod(s).equals("<m:plusRequest>") {
    float a = (new Integer(getArg(s, 1))).intValue();
    float b = (new Integer(getArg(s, 2))).intValue();
    float res = calc.plus(a, b);           // call to the CORBA object!
    SocketWriteLn(addr, "<m:plusResponse>" +
                        "<a type=\"float\">" + a + "</a>" +
                        "<b type=\"float\">" + b + "</b>" +
                        "<result type=\"float\">" + res + "</result>" +
                        "</m:plusResponse>");
  } else if (getMethod(s).equals("<m:minusRequest>") {
     /* ... similar to "plus" ... */
  }
}
```

5.2 Calling Maude objects from CORBA

We have seen in the previous sections how a Maude object can be replaced by its CORBA implementation. Let us see now how CORBA objects can transparently

call Maude objects' services. Suppose a CORBA customer of the bank account, which wants to deposit and withdraw some money. This customer simulates the client of the final CORBA system, which is completely unaware whether it is 'talking' to the live CORBA system or to its specification in the Maude language:

```
Account account = ... ; // obtains the CORBA object ref. for the account
account.deposit(1000);
try { account.withdraw(200) } catch (notEnoughMoney e) {...}
```

In order to connect this call to the Maude specification of the bank account, we use a similar mechanism to the one above, based on the use of two proxies. The first one is a CORBA proxy that represents the Maude object in the CORBA world, and the second one is a Maude proxy that collects the calls from the CORBA proxy and passes them to the Maude object.

The following piece of code is part of the implementation of the CORBA proxy of the bank account object. This code usually resides in the `_AccountImpl` object that contains the implementation of the `Account` interface.

```
String addr = "150.214.108.1:2021"; // IP:socket of MaudeProxy
....
float  balance() {
    // sends the request to the MaudeProxy
    SocketWriteLn(addr, "<m:balanceRequest></m:balanceRequest>");
    // and then waits for the answer from the MaudeProxy
    String s = SocketReadLn(addr);
    while(!getMethod(s).equals("<m:balanceResponse>")
        s = SocketReadLn(addr);
    return (new Integer(getResult(s))).intValue();
}
void deposit(float amnt) {
    SocketWriteLn(addr,"<m:depositRequest>" +
                       "<amount type=\"float\">" + amnt + "</amount>" +
                       "</m:depositRequest>");
    String s = SocketReadLn(addr);
    while(!getMethod(s).equals("<m:depositResponse>")
        s = SocketReadLn(addr);
}
void withdraw(float amnt) throws notEnoughMoney {
    SocketWriteLn(addr,"<m:withdrawRequest>" +
                       "<amount type=\"float\">" + amnt + "</amount>" +
                       "</m:withdrawRequest>");
    String s = SocketReadLn(addr);
    while((!getMethod(s).equals("<m:depositResponse>") &&
           (!getMethod(s).equals("<m:notEnoughMoney>"))
        s = SocketReadLn(addr);
    if (getMethod(s).equals("<m:notEnoughMoney>"))
        throw new notEnoughMoney();
}
```

The Maude proxy of the Account follows a similar scheme: it waits for calls from the CORBA proxy, unpacks them, passes them to the Maude object, and once the Maude object sends the response, it is sent back to the CORBA proxy.

```
(omod ACCOUNT_PROXY is
  protecting TCPHandler + STRING .
  class AccountProxy | CORBAProxyAddr : String, client : Oid .
  vars A P C : Oid .
  vars M B : float .

  rl [balanceRequest]:
    < P : AccountProxy | client : A >
    receivedTCP(P, System, "<m:balanceRequest></m:balanceRequest>")
    => < P : AccountProxy | >
       balanceRequest(A, P) .

  rl [balanceResponse]:
    < P : AccountProxy | CORBAProxyAddr : C, client : A >
    balanceResponse(B, P, A)
    => < P : AccountProxy | >
       sendTCP(System,P,C,"<m:balanceResponse>" +
                          "<result type=\"float\">" + B + "</result>" +
                          "</m:balanceResponse>") .
  rl [depositRequest]:
    < P : AccountProxy | client : A >
    receivedTCP(P, System, "<m:depositRequest>" +
                           "<amount type=\"float\">" + M + "</amount>" +
                           "</m:depositRequest>")
    => < P : AccountProxy | >
       depositRequest(M, A, P) .

  rl [depositResponse]:
    < P : AccountProxy | CORBAProxyAddr : C, client : A >
    depositResponse(P, A)
    => < P : AccountProxy | >
       sendTCP(System, P, C,"<m:depositResponse></m:depositResponse>") .

  rl [withdrawRequest]:
    < P : AccountProxy | client : A >
    receivedTCP(P, System, "<m:withdrawRequest>" +
                           "<amount type=\"float\">" + M + "</amount>" +
                           "</m:withdrawRequest>")
    => < P : AccountProxy | >
       withdrawRequest(M, A, P) .

  rl [withdrawResponse]:
    < P : AccountProxy | CORBAProxyAddr : C, client : A >
    withdrawResponse(P, A)
    => < P : AccountProxy | >
       sendTCP(System,P,C,"<m:withdrawResponse></m:withdrawResponse>") .
```

```
 rl [notEnoughMoney]:
   < P : AccountProxy | client : A >
   notEnoughMoney(P, A)
   => < P : AccountProxy | >
      sendTCP(System, P, C, "<m:notEnoughMoney></m:notEnoughMoney>") .
endom)
```

Please note how this way of functioning allows a transparent communication between the objects, independently from where they are actually implemented.

6 Concluding remarks

In this paper we have shown how a connection between a formal notation and a commercial component platform can be built. This connection can be very helpful for bridging the gap between system specification and implementation. Our proposal provides a transparent migration between objects in both worlds, so system specifications can be smoothly replaced by their implementations, as well as permitting new objects' specifications to be executed within running CORBA systems, allowing a very flexible way of systems/components prototyping.

There are two main fields of research that can be related to our proposal. First we have *refinement*, i.e., the process of transforming one specification into a more detailed one. The new specification can be referred to as a refinement of the original one, which needs to be verified. Most specification languages admit some sort of refinement, although it is particularly well defined for formal notations such as Z [11, 12, 14], B [1], or algebraic techniques [2]. Furthermore, specifications and their refinements do not necessarily need to coexist in the same system description. Precisely, what is meant by a more detailed specification will depend on the chosen specification language and, ultimately, by using the right notations, the refinement process may finally lead to a specification very close to the final implementation (if the last specification language is very close to a programming language). The refinement activity is somehow complementary to our proposal, since the Maude specification can be refined as usual. What our contribution provides is a bridge to go from a concrete Maude specification to a CORBA system.

Another field also related to ours is *generative programming*, which aims at the automated generation of program implementations directly from their formal (or informal) descriptions [5]. Again, our approach is not generative, because we are dealing with black-box binary components, usually sourced from third parties, that we want to *plug* into a specification. Or similarly, we deal with formal specifications of CORBA objects that we want to *drop* into a running CORBA system. Neither of these situations can be considered within a generative approach. However, the Maude specification could be used for generating code for particular components, although this falls out of the scope of our current proposal.

Once we have validated the feasibility of our approach, we plan to improve it by using SOAP, SCOAP, or some other Web-based RPC protocol. This will extend its capabilities beyond CORBA objects, allowing us to connect Maude specifications to other object and component platforms.

Acknowledgements

This work has been partially funded by the Spanish CICYT project TIC99-1083-C02-01. We would like to thank Ambrosio Toval, Narciso Martí-Oliet, and the anonymous referees for their insightful comments and suggestions, that greatly helped us improving the quality and presentation of the paper.

References

1. J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1986.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückenr. *Algebraic Foundations of Systems Specification*. Springer-Verlag, 1998.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
5. K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, June 2000.
6. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
7. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Comput. Sci.*, 96:73–155, 1992.
8. J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*. Kluwer Academic Publishers, 2000.
9. OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.4 edition, Nov. 2000. `http://www.omg.org/technology/documents/formal/corbaiiop.htm`.
10. S. Owre, N. Shankar, and J. Rushby. The PVS specification language. Technical report, Computer Science Laboratory, SRI International, February 1993.
11. J. Spivey. *The Z Notation. A Reference Manual. 2nd Ed.* Prentice Hall, 1992.
12. S. Stepney, D. Cooper, and J. Woodcock. More powerful Z data refinement: Pushing the state of the art in industrial refinement. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *Proc. of ZUM'98: The Z Formal Specification Notation*, number 1493 in LNCS, pages 284–307. Springer-Verlag, 1998.
13. C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
14. J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.