From Maude Specifications to SOAP Distributed Implementations: A Smooth Transition

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 the formal specification of a system written in Maude can be easily connected to its Web-based distributed implementation using SOAP. 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 bridging 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

The easy access to the Internet and the exponential growth of the number of servers connected to it have made of the World Wide Web and private Intranets the natural candidate for developing and deploying distributed applications. The Internet currently provides an exceptional infrastructure for exchanging information and locating resources and data. However, current enterprise applications and services require distributed access and processing, which is not directly supported by the Internet technologies as yet.

One of the approaches to deal with this problem is based on the use of distributed component platforms—such as CORBA, J2EE, or .NET—that provide the coordination mechanisms and services for distributed heterogeneous components to interoperate. But what we are now seeing as a result of efforts at W3C (the World Wide Web Consortium) and different Web-based service providers, is a gradual move to other kind of interoperation mechanisms, more naturally integrated with the Internet specific characteristics and evolving requirements. SOAP [14] (Simple Object Access Protocol) is an example of those. SOAP provides a simple and lightweight protocol for exchanging information between peers in a decentralized, distributed environment using XML. It allows heterogeneous distributed components to interoperate using the Internet resources in a natural way, and is gaining acceptance among Internet-based system developers and service providers for accessing their services.

An important issue to consider when building distributed systems is the use of formal methods and methodologies. There seems to be a growing consensus that 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.

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 tools such as B [1], PVS [11], SPIN [7], 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.

In addition, component-based software development has provided many advantages for the development of complex distributed systems, such as the clear separation of functionality between the constituent parts of an application, and the 'loose' connection among them. The use of component interfaces allows to decouple service definitions and access from final implementations. Thus, different components can communicate and cooperate 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) also helps the reasoning process about the systems being built.

There is, however, a big gap between system specifications and their implementations. Even if a system has been formally specified, and its specification validated for correctness, it is usually very hard to prove that a given implementation of a system conforms to its specification. Probably, the main reason is because specifications and implementations live in separate *worlds*. And even when techniques and mechanisms such as refinement or generative programming are used, there is always the final gap between both worlds (*formal* and *real*).

In this paper we explore the possibility of connecting the specification of a system written in a formal language such as Maude [4] with an Internet-based distributed implementation of it using SOAP. Maude is an executable rewriting logic language particularly well suited for the specification of object-oriented open and distributed systems [9]. By connecting both worlds so objects in one of them can seamlessly interoperate with objects in the other we can obtain several interesting benefits.

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 specification. 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 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 Internet components providing a service 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 their actual implementations, thus testing their behavior.

Third, given a running system and the specification of a new component (or a new version of one of them), we can validate the new system without having to implement such a new component. 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 other components distributed over the Internet, is now available for execution without having to wait for all critical components to be implemented.

In this paper we show how Maude can be smoothly integrated with distributed Internet-based applications, obtaining all these benefits. Sections 2 and 3 briefly describe SOAP 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 SOAP

SOAP is a proposal to the W3C for the definition of a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to remote procedure call (RPC) ones.

SOAP consists of three main parts:

?? The SOAP envelope construct defines an overall framework for expressing what there is in a message; who should deal with it, and whether it is optional or mandatory.
?? The SOAP encoding rules define a serialization mechanism that can be used to exchange instances of application-defined datatypes.
?? The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

Although these parts are described together as part of SOAP, they are functionally orthogonal. In particular, the envelope and the encoding rules are defined in different namespaces in order to promote simplicity through modularity.

In addition to the SOAP envelope, the SOAP encoding rules and the SOAP RPC conventions, the SOAP specification defines two protocol bindings that describe how a SOAP message can be carried in HTTP messages either with or without the HTTP Extension Framework.

The following example shows a "`plus`" SOAP request sent to a service that implements a simple calculator. The request takes the two operators as parameters. The SOAP Envelope element is the top element of the XML document representing the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers. The example illustrates the HTTP bindings, and it is worth noting that the rules governing XML payload format in SOAP are entirely independent of the fact that the payload is carried in HTTP.

```
POST /StockQuote HTTP/1.1
Host: www.calculator.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <m:plus xmlns:m="http://www.calculator.com/interface">
    <a>17</a>
    <b>2</b>
  </m:plus>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Following is the response message containing the HTTP message with the SOAP message as the payload:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle=
             "http://schemas.xmlsoap.org/soap/encoding/"/>
 <SOAP-ENV:Body>
   <m:plusResponse xmlns:m=
             "http://www.calculator.com/interface">
     <result>19</result>
   </m:plusResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP messages are fundamentally one-way transmissions from a sender to a receiver, but as illustrated above, SOAP messages are often combined to implement patterns such as request/response. As a matter of fact, one of the design goals of SOAP is to encapsulate and exchange RPC calls using the extensibility and flexibility of XML. SOAP also defines a uniform representation of remote procedure calls and responses:

?? A method invocation is modeled as a struct, named and typed identically to the method name. Each [in] or [in/out] parameter is viewed as an accessor, with a name corresponding to the name of the parameter and type corresponding to the type of the parameter. These appear in the same order as in the method signature.
?? A method response is modeled as a struct. The method response is viewed as a single struct containing an accessor for the return value and each [out] or [in/out] parameter.
?? The first accessor is the return value followed by the parameters in the same order as in the method signature.
?? Each parameter accessor has a name corresponding to the name of the parameter and type corresponding to the type of the parameter. The name of the return value accessor is not significant. Likewise, the name of the struct is not significant. However, a convention is to name it after the method name with the string "`Response`" appended.
?? A method fault is encoded using the SOAP "`fault`" element, that allows to model RPC errors: communication problems, server internal errors, client erroneous invocations, and method exceptions.

SOAP implementations can be optimized to exploit the unique characteristics of particular network systems. For example, the HTTP binding provides for SOAP response messages to be delivered as HTTP responses, using the same connection as the inbound request.

3 Rewriting Logic and Maude

Rewriting logic [8] 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 (S,E), where

S 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 ? t', where t and t' are S-terms. Such rules 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 [4] 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 non-deterministic concurrent computations; in particular, it supports very well concurrent object-oriented computation [9].

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 `|` 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 `|` 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 (intuitively known as a *soup*) 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

$$
\begin{array}{ll}
\texttt{crl}\,[r]: & M_1 \ldots M_m \\
 & < O_1 : C_1 \mid atts_1 > \ldots < O_n : C_n \mid atts_n > \\
 & \texttt{=>} < O_{i1} : C'_{i1} \mid atts'_{i1} > \ldots < O_{ik} : C'_{ik} \mid atts'_{ik} > \\
 & \quad < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\
 & \quad M'_1 \ldots M'_q \\
 & \texttt{if}\ Cond\ \texttt{.}
\end{array}
$$

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, $\{i1,\ldots,ik\}$ is a subset of $\{1,...,n\}$, 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_{i1},..., O_{ik} may change;
(c) all the other objects O_j vanish;
(d) new objects Q_1,...,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 a decentralized, distributed environment using SOAP for exchanging information. Among other entities, the system contains a bank account and a calculator as depicted in Figure 1. The `Calculator` class provides operations `plus` and `minus`, and instances of it will be used by the account objects for doing some computations. The class `Account` has operations `getBalance`, `deposit`, and `withdraw`, which may raise an exception `notEnoughMoney`. This is a simple example, but will serve us to illustrate how to model the typical client-server interactions between two objects.

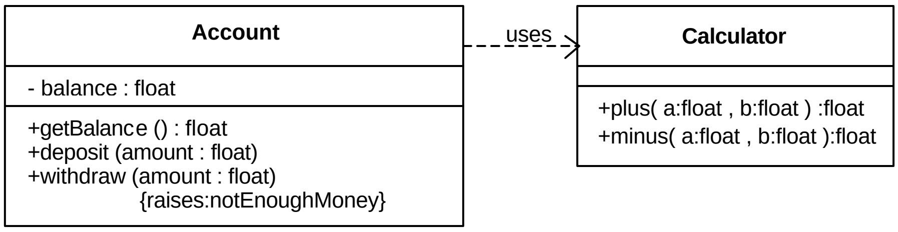
Account
- balance : float
+getBalance () : float
+deposit (amount : float)
+withdraw (amount : float)
{raises:notEnoughMoney}
uses
Calculator
+plus(a:float , b:float) :float
+minus(a:float , b:float):float

Fig 1. The UML class diagram of the account and the calculator classes.

4.1 The Maude Specifications

Imagine that both the calculator and the bank account are critical components, whose behavior and interactions need to be formally specified in order to be able to reason about them. The following shows how the specification of both objects can be written, using Maude as formal notation.

First, let us consider the specification of the calculator. We declare a class `Calculator` with an attribute (to store the balance), which can receive messages `plus` and `minus`, with the float 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`, which return the corresponding results. We assume a module `SOAP-BASIC-TYPES` containing the SOAP basic types (`float`, etc.) which we do not include here.

```
(omod CALCULATOR is
   protecting SOAP-BASIC-TYPES .
   class Calculator | .

   msgs plus minus : Oid float float Oid -> Msg .
   msgs plusResponse minusResponse : Oid float Oid -> Msg .

   vars O O' : Oid .
```

```
    vars X Y : float .

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

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

With regard to the `Account` class, which is specified in the module `BANK-ACCOUNT` below, 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 method invocation gives rise to a pair of messages (request-response) as in the case of the calculator, and that the exception `notEnoughMoney` is modeled as a possible response message to be sent in case a withdraw operation cannot take place.

```
 (omod BANK-ACCOUNT is
    protecting SOAP-BASIC-TYPES + DEFAULT[Oid] .
    class Account | bal : float, client : Default[Oid],
            calc : Oid .
    msgs deposit withdraw : Oid float Oid -> Msg .
    msg getBalanceResponse : Oid float Oid -> Msg .
    msgs getBalance depositResponse : Oid Oid -> Msg .
    msgs withdrawResponse notEnoughMoney : Oid Oid -> Msg .

    vars C A O : Oid . vars B X Z : float .

    rl [getBalance]:
       < A : Account | bal : B >
       getBalance(A, O)
       => < A : Account | >
          getBalanceResponse(O, B, A) .

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

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

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

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

As we can see in this specification, a call to the account's `deposit` or `withdraw` methods causes 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 —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`. On the other hand, when writing the Maude rules we have used the facility that those object attributes 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 [4].

4.2 Modeling Object Interactions in Maude

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

1. First, a Maude module contains the specifications of the basic types (such as “long”, “float”, etc.) available in the platform we are modeling. In the previous example, the `SOAP-BASIC-TYPES` module contains the specifications of SOAP basic types, including the SOAP compound types, which are naturally modeled in Maude by using its facilities for specifying abstract data types.
2. The remote procedure call (RPC) communication mechanism is modeled by using two Maude messages for each RPC, one that carries the request, and one for the reply. This is done in a way similar to the one proposed for SOAP.
3. Maude does not require the source and target objects to be specified in the messages. However, in order to model message passing in a systematic way, we have included as the first and the last arguments of all Maude messages the identifiers of the target and the source objects of the message, respectively.

4. Exceptions are modeled by 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 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 (*Adding some SOAP to the Maude Soup*)

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 SOAP implementation of the calculator from a third party, or that we have implemented it following the specification in Section 4, 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—SOAP and Maude—can be integrated, and how their objects interoperate.

Our integration is based on the use of proxies in the Maude side, which replace the Maude objects and capture their incoming calls, making the actual calls to the SOAP objects. The responses from the SOAP objects are also captured by the Maude proxies, which build the Maude responses from them. The communication between the Maude proxies and the SOAP objects they represent is done by using SOAP over HTTP (proxies can be seen as clients of their SOAP implementations), making use of the TCP features available in Maude 2.0 [5]. Thus, clients can continue interoperating with their servers, no matter whether they are Maude objects or their SOAP implementations. As an example of this, Figure 2 shows a transition by which a Maude specification of an object is replaced by its SOAP implementation—in a transparent way to its client.

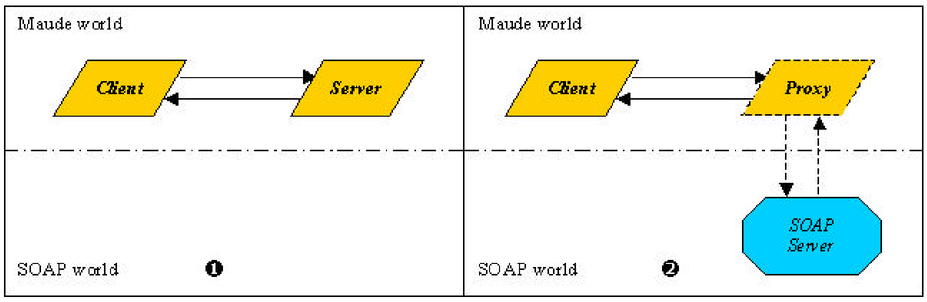

Fig 2. Replacing Maude objects by their SOAP implementations.

5.1 Implementing the Calculator

Let us see what these proxies may 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. Therefore, we need a proxy of the calculator in the Maude world, which is given in the module `CALCULATOR-PROXY` below. In it, `TCPHandler` is the Maude module providing TCP communication, with messages `sendTCP` and `receivedTCP` for sending and receiving messages, respectively (see [5]). Note that in addition to the predefined `System` object's reference and the sender and addressee objects' references, the HTTP message to be sent by TCP itself is given as a string. 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
   pr TCPHandler + STRING + DEFAULT[Oid] + SOAP-BASIC-TYPES .
   class Calculator | SOAPObjectAddr : String,
             client : Default[Oid] .
   msgs plus minus : Oid float float Oid -> Msg .
   msgs plusResponse minusResponse : Oid float Oid -> Msg .

   vars O O' : Oid .  vars X Y Z : float . vars A H : String .

   rl [plus]:
    < O : Calculator | SOAPObjectAddr : A, client : null >
    plus(O, X, Y, O')
    => < O : Calculator | client : O' >
       sendTCP(System, O, A,
         "POST /StockQuote HTTP/1.1 " +
         "Host: www.calculator.com " +
         "Content-Type: text/xml; charset=`"utf-8`" " +
         "Content-Length: nnnn " +
         "<SOAP-ENV:Envelope " +
          "xmlns:SOAP-ENV=
          `"http://schemas.xmlsoap.org/soap/envelope/`" " +
          "SOAP-ENV:encodingStyle=
           `"http://schemas.xmlsoap.org/soap/encoding/`">" +
            "<SOAP-ENV:Body>" +
              "<m:plus xmlns:m=
                `"http://www.calculator.com/interface`">" +
                "<a>" + X + "</a>" +
                "<b>" + Y + "</b>" +
              "</m:plus>" +
            "</SOAP-ENV:Body>" +
         "</SOAP-ENV:Envelope>") .

   rl [plusResponse]:
    < O : Calculator | client : O' >
    receivedTCP(O, System,
       H +  **** HTTP header ignored
       "<SOAP-ENV:Body>" +
        "<m:plusResponse xmlns:m=
          `"http://www.calculator.com/interface`">" +
         "<result>" + Z + "</result>" +
        "</m:plusResponse>" +
       "</SOAP-ENV:Body>" +
      "</SOAP-ENV:Envelope>")
    => < O : Calculator | client : null >
       plusResponse(O', Z, O) .

   *** similar rules for "minus" omitted
 endom)
```

An object of the `Calculator` class in this module is now the one providing the calculator services, hence accepting the same methods as the original calculator. However, it does not carry out the computations any longer: it just calls the SOAP object, packing and unpacking messages into the appropriate formats. It is important to notice that the generation of this proxy can be automated directly from the UML description of the original object.

5.2 Calling Maude Objects from the SOAP World

We have seen in the previous sections how a Maude object can be replaced by its SOAP implementation. Let us see now how a SOAP object can transparently call Maude objects' services. Suppose a SOAP customer of the bank account, which wants to deposit and withdraw some money. This customer simulates the client of the final system, which is completely unaware whether it is "talking" to the live SOAP system or to its specification in the Maude language. Actually, what it does is just send SOAP requests via HTTP, and wait for the SOAP responses.

In order to connect the SOAP calls to the Maude specification of the bank account, we use a similar mechanism to the ne above, based again on the use of a Maude proxy. This proxy waits for calls from the SOAP object, unpacks them, passes them to the Maude object, and once it receives the response from the Maude object, sends it back to the SOAP client.

```
(omod ACCOUNT_PROXY is
   protecting TCPHandler + STRING + SOAP-BASIC-TYPES .
   class AccountProxy | client : Oid, account : Oid .
   msg getBalance : Oid Oid -> Msg .

   vars A P C : Oid . vars M B : float .  var H : String .

   rl [getBalance]:
    < P : AccountProxy | client : C, account : A >
    receivedTCP(P, System,
      H + **** HTTP header ignored
      "<SOAP-ENV:Envelope " +
       "xmlns:SOAP-ENV=
        `"http://schemas.xmlsoap.org/soap/envelope/`" " +
       "SOAP-ENV:encodingStyle=
        `"http://schemas.xmlsoap.org/soap/encoding/`">" +
        "<SOAP-ENV:Body>" +
          "<m:getBalance xmlns:m=
              `"http://www.bank.com/interface`">" +
          "</m:getBalance>" +
        "</SOAP-ENV:Body>" +
      "</SOAP-ENV:Envelope>")
      => < P : AccountProxy | >
         getBalance(A, P) .

   rl [getBalanceResponse]:
    < P : AccountProxy | client : C >
    getBalanceResponse(P, B, A)
    => < P : AccountProxy | >
       sendTCP(System, P, C,
         "HTTP/1.1 200 OK " +
       "Content-Type: text/xml; charset=`"utf-8`" " +
       "Content-Length: nnnn " +
       "<SOAP-ENV:Envelope " +
        "xmlns:SOAP-ENV=
        `"http://schemas.xmlsoap.org/soap/envelope/`" " +
        "SOAP-ENV:encodingStyle=
        `"http://schemas.xmlsoap.org/soap/encoding/`">" +
          "<SOAP-ENV:Body>" +
           "<m:getBalanceResponse xmlns:m=
              `"http://www.bank.com/interface`">" +
             "<result>" + B + "</result>" +
           "</m:getBalanceResponse>" +
          "</SOAP-ENV:Body>" +
       "</SOAP-ENV:Envelope>") .

   *** Similar rules for "deposit" and "withdraw" omitted
   *** Exception handling modeled using the "fault" SOAP
   *** element in the SOAP response
 endom)
```

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 Internet-based distributed computing solution 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 SOAP systems, allowing a very flexible way of systems/components prototyping.

There are three 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 [12,13,15], or algebraic techniques [3]. Furthermore, specifications and their refinements do not necessarily need to coexist in the same system description. More 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 SOAP 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 [6]. 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 SOAP objects that we want to *drop* into a running SOAP 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.

Finally, we would like to mention the approaches that at the beginning of the nineties used the *abstraction function* to combine algebraic specifications with imperative implementations. The main goal of this abstraction function is to convert the values of the variables in a given programming language to their corresponding algebraic terms, hence allowing mixed executions. Basically our proposal differs from those ones in the "component-oriented" approach we have used, whereby components are the basic unit of specification and replacement. Thus, we do not try to combine any specification with any program, but to connect (by using their interfaces only) components in two different worlds.

This work is a sequel of a previous one [2], that showed how to connect the worlds of Maude and CORBA. Using SOAP instead of CORBA widens the scope of the work, allowing us to connect Maude specifications to other object and component platforms, in particular to the ones most commonly used for Internet distributed computing (e.g., CORBA objects can be easily accessed by SOAP clients [10]). The use of SOAP also simplifies the structure of our solution, since Maude objects can directly access SOAP objects using Maude's TCP facilities. And finally, we are allowing the increasing number of Internet services available via SOAP to be available also to Maude objects within Maude specifications, which greatly facilitates the development of prototypes of Internet-based systems and applications.

Acknowledgements

This work has been partially funded by 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] A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In *Proc. of ASSE'01*, Buenos Aires, Argentina, September 2001.

[3] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückenr. *Algebraic Foundations of Systems Specification*. Springer-Verlag, 1998.

[4] 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

[5] 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.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[8] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Comput. Sci*., 96:73—155, 1992.

[9] 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.

[10] OMG. *Simple CORBA Object Access Protocol (SCOAP)*. Object Management Group, Sept. 2000. OMG TC Document orbos/00-09-03.

[11] S. Owre, N. Shankar, and J. Rushby. The PVS specification language. Technical report, Computer Science Laboratory, SRI International, February 1993.

[12] J. Spivey. *The Z Notation. A Reference Manual*. 2nd Ed. Prentice Hall, 1992.

[13] 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.

[14] W3C. *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium, May 2000. http://www.w3.org/TR/SOAP.

[15] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.