Adding Protocol Information to CORBA IDLs

C. Canal, L. Fuentes, J.M. Troya, and A. Vallecillo

ETSI Informática. Universidad de Málaga.
{canal,lff,troya,av}@lcc.uma.es

Abstract. Traditional IDLs were defined for describing the services that objects offer, but not those services they require from other objects, nor the partial ordering in which they expect their methods to be used. In this position paper we present an extension of the CORBA IDL that uses a sugared subset of the polyadic π-calculus for describing object service protocols, aimed towards the automated checking of protocol interoperability between CORBA objects in open component-based environments. This position paper for WOI'00 is a reduced version of [3].

1 Introduction

Current commercial IDL contain just syntactic information about the services provided by objects, but nothing is said about the behavior of the objects they describe. In a previous paper presented at WOI'99 [9] we proposed the use of π-calculus for extending component IDLs with protocol information, with the goal of describing the dynamic behavior of the components, in addition to the static description of their services, i.e., their method signatures. Our approach was based on a particular IDL that we had defined within our research group, enriching it with two types of information: the way objects expect their methods to be called, and how they use other objects' methods. One of the experiments suggested in our paper was the use of real commercial IDLs instead of toy ones. This paper present our experiences extending the CORBA IDL with protocols. A more detailed version of this work has been presented at TOOLS-33 in June this year [3].

2 An example application

In order to illustrate our proposal, let us describe a simple E-commerce application with three main players: a bank, a bookshop, and a book-broker that locates and buys books on behalf of a user.

The description of the interface of each of those objects in the CORBA IDL follows. The first one defines the bank services:

```
interface AccountFactory {
  Account create();
};
```

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

As we can see, bank accounts are created using an `AccountFactory` object, which has a method —`create()`— that returns the reference of a newly created account. Each account offers three methods, that allow the user to know the current balance, to deposit, or to withdraw some money from the account. Methods `deposit()` and `withdraw()` return a string that may serve as a receipt of the operation carried out, and method `withdraw()` can also raise an exception in case we want to withdraw more money than currently available.

The second player is a bookshop, defined by the following interface:

```
interface Bookshop {
  struct BookRef {string ISBN; float price;};
  BookRef inStock(in string title, in string author);
  void order(in BookRef b, out Account a, out string purchaseId);
  date deliver(in string purchaseId, in string receipt, in string addr);
};
```

This interface defines a data type and three methods. Type `BookRef` is a record with two fields, the ISBN and the price of the book being referenced. The operations allow the user to know whether a book is in stock, to order it (obtaining an identification of the purchase and an account number for the payment), or to ask the bookshop to deliver it to a given address once the user has paid for it.

Finally, a book-broker is an object that maintains a list of bookshops, and that is able to make use of their services in order to locate and buy a book on behalf of a user.

```
interface BookBroker {
  void add(in Bookshop b);
  oneway void remove(in Bookshop b);
  boolean getABook(in string author, in string title, in float maxprice,
                   in string addr, out date when);
};
```

In this interface, methods `add()` and `remove()` are for adding and deleting references from the list of bookshops that the book-broker keeps, while method `getABook()` implements the main service offered by the object. This method returns a boolean value indicating whether the operation has succeeded or not, and the date in which the book will be delivered to the user in case the order has been accepted. Method `remove()` has been defined as `oneway` to illustrate how this mechanism can be modeled in our proposal.

3 Extending CORBA interfaces with protocols

First of all, by protocol we mean the partial order in which the objects expect their methods to be called, and the order in which they should invoke other objects' methods. In our proposal protocols are described using a sugared subset of the polyadic π-calculus, a process algebra specially well suited for the specification of dynamic and evolving systems.

The π-calculus was originally proposed by Milner, Parrow and Walker in 1992 [8]. Although also called 'a calculus of mobile processes', no processes are actually moved around, just the identities (*names*) of the channels that processes use to communicate among themselves. It is quite similar to CCS, apart from the fact that not only values can be passed around, but also channel names. The polyadic π-calculus [7] is a generalized version of the basic π-calculus, extended to allow tuples and compound types to be sent along channels. Semantics is expressed in terms of both a reduction system and a version of labeled transitions called *commitment*.

The description of the polyadic π-calculus is beyond the scope of this paper. Here we will only describe the syntactic sugar we have added to it. In the first place, the standard polyadic π-calculus does not provide for built-in data types and process parameters, they can be easily simulated. Therefore we will use numbers and some basic data types, such as lists (with operators `<>`, `++` and `--` for list creation, concatenation and difference). Additionally, we have enriched the π-calculus matching operator so that within the square brackets we can use any logical condition, that acts as a guard for the process specified after the brackets. We will also use the construct `[else]`, that provides some syntactic sugar for expressing that a process given by

$$\texttt{([G}_1\texttt{]P}_1\texttt{ + [G}_2\texttt{]P}_2\texttt{ + } \cdots \texttt{ + [G}_n\texttt{]P}_n\texttt{ + [else]P}_0\texttt{)}$$

behaves as any process $\texttt{P}_i$ for which guard $\texttt{[G}_i\texttt{]}$ is true, or as $\texttt{P}_0$ if all guards $\texttt{G}_j$ $(1 \leq j \leq n)$ are false.

The π-calculus has proved to be a very expressive notation for describing the dynamic behavior of objects in applications with changing topologies (as those that live in open systems). In this sense, it is more appropriate than other process algebras such as CCS or CSP. An extra benefit of this formal notation is that it allows to specify, in addition to the specific protocol information, some of the details of the object's internal state and semantics that are relevant to its potential users, while hiding those which we want to leave open to possible implementations. This is one of the reasons why we decided to use π-calculus instead of other formal notations for describing mere protocol information (like message state charts, for instance).

3.1 Modeling approach

The main modeling techniques that we propose are the following:

- Each object is supposed to own a channel, through which it receives method calls. This channel will logically correspond to the object reference.
- Together with every request, the calling object should include a channel name through which the called object will send the results. Although channels are bi-directional in the π-calculus, in this way request and reply channels can be

kept separate to permit an object to accept several simultaneous calls, while using specific channels for replying.

- From the client's point of view, invocation of method `m` of an object whose reference is `ref` is modeled by one output action `ref!(m,(args),(r))`, where `m` is the name of the method, `args` is a tuple with its in and in-out parameters, and `r` is a tuple containing the return channel and, optionally, other reply channel names (for possible exceptions).
- Once the method has been served, the normal reply consists of a tuple sent by the called object through the return channel. That tuple consists of the return value of the method, followed by the out and in-out parameters. Arguments are transmitted in the same order they were declared.
- Exceptions are modeled by channels. For instance, if method `m` can raise exception `excp`, a new channel named `excp` has to be created before the method call, and sent along within the return channels tuple. The server object can either reply using the first return channel if the method is served without problems, or send the exception parameters through channel `excp` if the exception is raised.
- The state-based behavior of the objects is modeled by recursive equations, where the various parts of the object state (i.e. the state variables we want to make visible to exhibit the object behavior) are parameters.

We have also added some syntactic sugar for the sake of clarity and brevity when writing the specifications of the objects' protocols:

- We can write `ref!m(args,reply)` to model invocation of method `m(args)` instead of writing `ref!(m,(args),(reply))`.
- In case that the reference and reply channels are the same, we can simply write the invocation as `ref!m(args)`.
- Similarly, on the server side we can write `ref?m(args,reply)...` instead of `ref?(meth,(args),(reply)).[meth='m']...` for accepting the invocation of method `m(args)`.

Protocols are defined using special construct '`protocol`', that consists of a name, followed by the protocol description in textual π-calculus enclosed between curly brackets. Each `protocol` description corresponds to one CORBA `interface` declaration, and serves as the specification of its behavior —described as one or more π-calculus processes. Unless otherwise stated, the name given to the protocol description is used to identify the interface it relates to. We will later see how to associate different protocols to a given interface.

3.2 Extending the example specification

In this section we will show how the observable behavior of the objects in the previous example can be described. We will start with the bank services, for which a possible protocol description follows:

```
protocol AccountFactory {
  AccountFactory(ref) =
      ref?create(rep) .
        (^acc) ( Account(acc,0) | ( rep!(acc) . AccountFactory(ref) ) )
    + [else]
      AccountFactory(ref)
};
protocol Account {
  Account(ref,balance) =
      ref?getBalance(rep) .
        rep!(balance) . Account(ref,balance)
    + ref?deposit(amount,rep) .
        (^receipt) rep!(receipt) . Account(ref,balance+amount)
    + ref?withdraw(amount,rep,notEnough) .
        ( tau . (^receipt) rep!(receipt) . Account(ref,balance-amount)
        + tau . notEnough!(balance,amount) . Account(ref,balance) )
    + [else] Account(ref,balance)
};
```

There are two protocol descriptions, one for each of the two related interfaces. The first one describes the behavior of object `AccountFactory`, and uses a π-calculus process with only one argument: the name of the channel that the object will use (i.e. the object reference). The process starts by reading from that channel, and once a tuple has been read, it decides whether it matches the service that it implements, or not. If so, it creates a new channel name for referring to the new account (`^acc`), and spawns two processes: one that behaves as an `Account`, and another one that replies to the request through the given channel and goes back to the original state. In case the received request is not valid, it is ignored (as determined by the `[else]` part).

The `Account` protocol describes a behavior in which a process waits for a request to arrive through the object reference. In case it is a valid operation, the object replies to the request (or raises an exception) and behaves again as an account, otherwise the process ignores it. Please note the use of the second argument of the process to maintain the internal state of the object. In this case we are specifying not only protocol information, but also some of the object's behavioral semantics: how the three methods modify the balance of the account. However, we did not want to specify when the object decides to raise the exception (e.g. maybe the account could allow some credit).

The following protocol defines a possible behavior of an object compliant with interface `Bookshop`:

```
protocol Bookshop {
  Bookshop(ref,bank) =
      (^rep) bank!create(rep) . rep?(account) . SellingBooks(ref,account)
  SellingBooks(ref,account) =
      ref?inStock(title,author,rep) .
        (^bookref) rep!(bookref) . SellingBooks(ref,account)
    + ref?order(bookref,rep) .
        (^purchaseId) rep!(account,purchaseId) . SellingBooks(ref,account)
    + ref?deliver(pid,receipt,deliv,rep)
        (^date) rep!(date) . SellingBooks(ref,account)
    + [else] SellingBooks(ref,account)
};
```

This process has two arguments: the object reference channel, and the reference channel of the `AccountFactory` object to be used. The process starts by requesting an `AccountFactory` to create an account, and then behaves as process `SellingBooks`, that waits for valid requests and services them. The channel of the `Account` object used is obtained from the reply of method `create()`.

Finally, a possible behavior of the book-broker object can be described as follows:

```
protocol BookBroker {
  BookBroker(ref,bookstores) =
        ref?add(bs,rep) .
            rep!() . BookBroker(ref,bookstores++<bs>)
        + ref?remove(bs,rep) .
            BookBroker(ref,bookstores--<bs>)
        + ref?getABook(auth,title,price,addr,rep) .
            ( Buy(ref,auth,title,price,addr,rep,bookstores)
            | BookBroker(ref,bookstores) )
        + [else] BookBroker(ref,bookstores)
      Buy(ref,auth,title,price,addr,rep,stores) =
          [ stores = NIL ]
            rep!(FALSE,NIL) . zero
        + [ stores = <bs>++dB ]
            bs!inStock(title,auth) . bs?(book) .
            ( [(book!=NIL)&&(book.price<=price)]
                bs!order(book) . bs?(account,pid) .
                account!deposit(book.price) . account?(receipt) .
                bs!deliver(pid,receipt,addr) . bs?(date) .
                rep!(TRUE,date) . zero
            + [else] Buy(ref,auth,title,price,addr,rep,dB) )
    };
```

In this example we can see how a book-broker uses a list with the bookshops it knows about (`bookstores`), which is updated by operations `add()` and `remove()`. We can see the different behavior of both methods: the latter one does not send a reply, since it was defined as `oneway`. The behavior of process `Buy` is recursively defined by using structural induction over the list of bookshops.

It is important to notice that these are *possible* behaviors. Other protocols can be associated to those interfaces, hence defining different behaviors.

3.3 Where do protocols live?

Analogously to where object IDLs live, each protocol resides inside a separate text file (with extension `.ptl`). Keeping protocol descriptions separated from object IDLs permits the addition of protocol information to CORBA object interfaces in an incremental and independent manner. In this way, new CORBA tools, repositories and traders can be defined as extensions to the new ones, while keeping backwards compatibility with the current tools and applications that do not make use of this new protocol information.

Each `protocol` description corresponds to one CORBA `interface` declaration, and serves as the specification of its observable behavior. Different protocols can be associated to a given IDL. In case we want protocol `Prot` to describe the behavior of an interface named `Intfc` we can use the keyword '`describes`' in the protocol definition:

```
protocol Prot describes Intfc {/*...protocol description goes here...*/};
```

4 Checking protocols

Once we have enriched IDLs with protocol information, we need to think about the sort of checks that can be carried out, the moments in which those tests can be done, and the mechanisms required for that purposes.

We will distinguish between static and dynamic checks. The first ones are carried out during the design time of the applications, based just on the description of their constituent components and the binds among them (i.e. the architecture of the application). With them we are able to obtain several interesting advantages, such as high level reasoning about software composability, or proving safety properties of the composed applications (e.g. no deadlocks). With our approach this can be easily done using the protocol descriptions of the components. As an example, suppose that we have an application with some of the components previously defined, namely one bank, two bookshops, one book-broker, and one user, and that we want to check whether this system is deadlock-free. In order to do that, the behavior of the application can be modeled as a set of π-calculus processes running in parallel, whose channels are related according to the application's topology:

```
Appl() = (^ac)      // AccountFactory's address
         (^b1,b2)   // addresses of the two bookshops
         (^bb)      // Book-broker's address
         (^u)       // User's address
         ( AccountFactory(ac) | Bookshop(b1,ac) | Bookshop(b2,ac)
         | BookBroker(bb,<b1,b2>) | User(u,bb) )
```

In this case the `User` process models the *environment*, by simply representing an object that calls the book-broker method `getABook()` looking for a given book and waits for its reply. With this, the absence of deadlocks in the application can be easily checked by analysing the π-calculus process `Appl()` and proving that it is deadlock-free. Other static checks can also be carried out with our proposal, such as component compatibility and component substitutability [3].

On the other hand, there are situations in which protocol compatibility needs to be checked at run-time, as it happens in open systems where components may dynamically evolve. However, special run-time compatibility checks are needed in those environments, in which both the components and the application internal topology may change over time.

Protocol compatibility can be checked at run time by intercepting messages and verifying their correctness with regard to the current state of the destination component. In this way system inconsistencies and deadlock situations can be detected before they happen, and the appropriate actions can be taken beforehand. Thus, an interceptor can be defined for each object, which captures incoming and outgoing messages, reproduces its run-time trace, and checks that received messages are compatible with the behavior defined for that object.

Both static and dynamic checks are possible with our proposal, but the question is whether they are useful and practically achievable. For instance, static compatibility checks are very useful in closed applications, but rather limited for open applications, in which there is no explicit framework context that defines the relations and binds among the components. Another problem is that proving this sort of static checks is an NP hard problem, impractical for most real applications. On the other hand, run time compatibility can be checked on the fly by the interceptors with no such a heavy burden, checking the conformance to a given protocol message by message. This method delays analysis until run time, and has the advantages of making it tractable from a practical point of view, and to allow the management of dynamic attachments in open environments in a natural way. The main disadvantages are that it needs a lot of accountancy by the filters, and that detection of deadlock and other undesirable conditions is delayed until *just* before they happen, which may be unacceptable in some situations.

The detailed discussion on the sort of tests that can be carried out and how to achieve them is beyond the scope of this short paper, whose main intention is the presentation on how to describe the external behavior of CORBA distributed objects using π-calculus. Interested readers can consult the full paper [3].

5 Related Work

The contributions presented here fall into two main categories: the extension of object IDLs for dealing with protocol information, and the use of formal notations for describing those behaviors.

Regarding the first group, Doug Lea's PSL [6] proposes in 1995 an extension of the CORBA IDL to describe the protocols associated to an object's methods. This approach is based on logical and temporal rules relating situations, each of which describes potential states with respect to the roles of components, attributes, and events. Although it is a very good and expressive approach, it does not account for the services an object may need from other objects, neither it is supported by standard proving tools.

Protocol Specifications [10] is a more general approach for describing object service protocols using state-based machines, that describes both the services offered and required by objects. However, some of its limitations (as recognized by their own authors) make it too rigid for its general use in open and distributed environments.

The approach by Cho, McGregor and Krause [4] uses UML's OCL to specify pre and post conditions on the objects' methods, together with a simple state machine construct to describe message protocols. Similarly, Jun Han [5] proposes an IDL extension to include semantic information in terms of constrains and roles of interaction between components (but using no standard notation), that aims to improve the selection and usage processes in component-based software development. The behavior od components is described in terms of constraints, that are expressed in a subset of temporal logic. They are somehow similar approaches, although none of them is associated to any commercial component platform like CORBA or EJB, nor supported by standard tools.

Bastide et al. [1] use Petri nets to describe the behavior of CORBA objects, providing operation semantics and interaction protocols altogether, without a clear separation of both semantic dimensions. The same happens to the proposal by Büchi and Weck [2], where a grey-box specification approach is used to specify the semantics of the components. However, this is not currently feasible due to the black-box approach intrinsic to CORBA environments.

On the other hand, we have shown how the π-calculus can be successfully used for describing some of the semantics of the dynamic behavior of the components. Furthermore, the modeling of CORBA object interaction mechanisms turned out to be easy and natural in the polyadic π-calculus, since object reference manipulation and client-server invocations have a very good semantic matching with the π-calculus. However, the use of different formal notations for expressing protocol information is open to debate, and we would like to heavily discuss this issue during the workshop.

6 Open issues

The following is a list of issues related to object interoperability at the protocol level that may require further investigation, and whose discussing we would also like to raise during the workshop:

Adaptors. Once we have characterized the protocols that components obey, we can check their compatibility. However, if the behavior of two components is incompatible, a new question arises: is it possible to build some extra components that *adapt* their interfaces, compensating their differences? In general, the automated construction of those *adaptors* right from the description of the interfaces of the original components is a difficult problem (cf. [10]).

Many-to-one substitutability. Everybody deals with one-to-one substitutability, studying when a component can be replaced by *just* another component. Searching for *several* components that can replace a given one is a difficult task, since it may introduce many different problems (service overlaps, gaps, extra services, etc.).

Checking compatibility at connection time. Apart from the static and dynamic checks that can be carried out with our proposal, we would like to explore a third possibility, and see whether we could perform static analysis at *connection time*, just before a new component joins an application. This is important in open systems, where a given user may decide to include a new type of component into a running application. We know about the application's architecture and the behavior of its constituent components, that are correctly operating. What we want to know is whether the new component will be compatible with the rest, but without using filters for dynamic checks in all those components, that may add an unnecessary burden to an application that has been previously tested for compatibility. Some sort of static checks would be ideal in this situation. However, the problem is that we may have to face compatibility checks between components which are at different states, i.e. we may have to check protocol compatibility between a source component (the new one we are just dropping in the system) which is in its initial state, and the rest of the application components, which are already running and that may be in intermediate states.

Conformance to specifications. The last issue that we would like to discuss is about how to test that a given implementation of an object conforms to a given specification of its behavior. In the case of CORBA we can count on interceptors to check that the object's behavior is valid with regard to the protocol it is supposed to implement. But in general it is a difficult problem for back-box components.

References

1. R. Bastide, O. Sy, and P. Palanque. Formal specifycation and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag, 1999.
2. M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.
3. C. Canal, L. Fuentes, J. Troya, and A. Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proc. of TOOLS'33*. IEEE Press, June 2000.
4. I. Cho, J. McGregor, and L. Krause. A protocol-based approach to specifying interoperability between objects. In *Proc. of TOOLS'26*, pages 84–96, 1998.
5. J. Han. Semantic and usage packaging for software components. In A. Vallecillo, J. Hernández, and J. M. Troya, editors, *Proc. of the ECOOP'99 Workshop on Object Interoperability (WOI'99)*, pages 25–34, June 1999.
6. D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.
7. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
9. A. Vallecillo, J. Hernández, and J. M. Troya. Object interoperability. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.
10. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.