Extending CORBA Interfaces with π-Calculus for Protocol Compatibility

C. Canal, L. Fuentes, J.M. Troya, and A. Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga.
ETSI Informática. Campus de Teatinos s/n. 29071 Málaga. Spain.
{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 paper we propose 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. Furthermore, some advantages and disadvantages of our proposal are discussed, as well as some of the practical limitations encountered when trying to implement and use this sort of IDL extensions in open systems.

1 Introduction

Interoperability is one of the key aspects related to the construction of large object-oriented systems, and can be defined as the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment or the model abstraction [24]. Basically, three main levels of interoperability between objects can be distinguished: the *signature level* (names and signatures of operations), the *protocol level* (partial ordering between exchanged messages and blocking conditions), and the *semantic level* (the "meaning" of the operations) [21].

Interoperability is currently well-defined and understood at the signature level, for which middleware architects and vendors are trying to establish different interoperational standards, e.g. CORBA, JavaBeans, or DCOM. However, all parties are starting to recognize that this sort of interoperability is not sufficient for ensuring the correct development of object-oriented applications in open systems. This issue is beginning to be addressed by most international standardization bodies, like ISO and OMG. In particular, RM-ODP is one of the standards that is now trying to incorporate semantic interoperability between objects. Nevertheless, the study and usage of the protocol and semantic levels currently present serious challenges, from both the theoretical and practical points of view. There have been some partial advances in those two levels, but we are still far from reaching any satisfactory solution yet.

Although object interoperability should be studied in general at the semantic level, this is quite an ambitious and broad problem, very difficult to tackle in full. As a first step, in this paper we will concentrate mainly on the interoperability of reusable components at the protocol level, where the basic problems can be identified and managed, and where practical solutions can be proposed to solve them. And instead of using any theoretical or academic model on which to base our proposal we have chosen CORBA, one of the major and most widespread commercial object models.

Our work proposes an extension to the CORBA IDL that allows the description of the dynamic behavior of CORBA objects, in addition to the *static* description of the object services (i.e. method signatures) provided by the standard CORBA IDL facilities. Our approach enriches IDLs with information about the way objects expect their methods to be called, how they use other objects' methods, and even some semantic aspects of interest to users and implementors of objects, and that may be present in their behavioral descriptions.

Protocols are described using a textual sugared subset of the polyadic π-calculus, and defined separately from the IDLs. In this way current repositories can be easily extended to account for this new information and to manage it. Besides, having this protocol information available at run time also allows to define dynamic compatibility checks in open and extensible applications, therefore being able to rea-

son about the compatibility and substitutability of their components, and to infer some of their safety and liveness properties directly from the description of the behavior of their constituent components.

The structure of this paper is as follows. After this introduction, section 2 briefly describes the CORBA IDL and introduces an example application, that will be used throughout the paper to illustrate our proposal. Section 3 describes our contribution in detail, showing how the π-calculus can be used for our purposes. Section 4 is dedicated to discuss the sort of checks that can be carried out once we have extended traditional IDLs with protocol information. In particular, we concentrate on object compatibility and substitutability checks (i.e. behavioral subtyping [17]), and on proving some safety and liveness properties of applications (e.g. absence of deadlocks). Besides, we discuss *when* those checks can be carried out, since we are not only concerned with static analysis of applications during design time, but also with the dynamic checks needed in open and reactive systems during their lifetime. Once our contribution has been described, section 5 discusses some of the limitations and problems that this sort of proposals introduce, while section 6 relates our work to other similar existing approaches. Finally, section 7 draws some conclusions and describes some future research activities.

2 Object interfaces and IDLs

Traditional object interfaces provide a description of an object functionality and capabilities, in terms of the attributes and the signature of the operations offered by the object. On top of them, *Interface Description Languages* (IDLs) have been defined for describing those object interfaces at the signature level. Apart from providing a textual description of the objects functionality, two are the major benefits of using IDLs. First, IDL descriptions can be stored in repositories, where service traders and other applications can locate and retrieve components from, and use them to dynamically learn about object interfaces and build service calls at run time. And second, IDL descriptions can be 'compiled' into platform-specific objects, providing a clear frontier between object specification and implementation, which facilitates the design and construction of open heterogeneous applications.

However, traditional IDLs were originally defined for closed client-server applications, and therefore they present some limitations when tried to be used in open component-based applications:

1. IDLs describe the services that objects offer, but not the services that they require from other objects in order to accomplish their tasks.
2. Typical IDLs provide just the *syntactic* descriptions of the objects' public methods, i.e. their signatures. However, nothing is said about the ordering in which the objects expect their methods to be called, or their blocking conditions.
3. In general, the use of IDL descriptions during run-time is quite limited. They are mainly used to discover services and to dynamically build service calls. However, there are no mechanisms currently in place to deal with automatic compatibility checks or dynamic component adaptation, which are among the most common facilities required for building component-based applications in open and independently extensible systems [19].

2.1 CORBA and its IDL

CORBA is one of the major distributed object platforms. Proposed by the OMG, 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 [3, 23].

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`, `string`). These are used to describe the interface of objects, defined by set of types, attributes and the signature (parameters, re-

turn types and exceptiones raised) of the object methods, grouped into `interface` definitions. Finally, construct `module` is used to hold type definitions, interfaces, and other modules for name scoping purposes.

2.2 An example application

In order to show an example of the CORBA IDL, 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 an user. This application will be used throughout the paper for illustrating our proposal.

Let us start with the description of the interface of each of those objects in the CORBA IDL. The first one defines the bank services:

```
interface AccountFactory {
  Account create();
};
interface Account {
  exception NotEnoughMoney
      {float balance; float requestedAmount;};
  float  balance();
  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 in the account.

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 an 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. Besides, method `remove()` has been defined as `oneway` to illustrate how this mechanism can be modeled in our proposal.

3 Extending CORBA interfaces with protocols

At the beginning of this paper we mentioned some of the problems presented by this sort of specifications, in which only the signature descriptions of the objects are described. In this section we will concentrate on how to add protocol information to the description of the CORBA object interfaces. By protocol we mean the partial order in which the objects expect their messages to be called, and the order in which they should invoke other objects' methods. Our main aim is to solve some of the problems previously identified, namely describing at the IDL level the services that objects require from other objects during their execution, providing protocol information beyond the description of the methods' signature, and making all this information available at run time for dynamic compatibility checks in open applications.

Protocols will be 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 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, the π-calculus 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 (i.e. the dynamic object's behavior), 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 the reasons we decided to use π-calculus instead of other formal notations for describing mere protocol information (like message state-charts, for instance).

3.1 The polyadic π-calculus

The π-calculus was originally proposed by Milner, Parrow and Walker in 1992 [15]. 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 [14] is a generalized version of the basic π-calculus, extended to allow tuples and compound types to be sent along channels. Semantics is done in terms of both a reduction system and a version of labeled transitions called *commitment*.

A very brief description of the calculus follows. If `ch` is a channel name, then `ch!(v).P` represents a process that sends value `v` along `ch` and then proceeds as process `P`. Conversely, `ch?(x).Q` is the process that waits for a value `v` to be received by channel `ch`, binds the variable `x` to the value received and then proceeds as `Q{v/x}`, where `{v/x}` indicates the substitution of the name `x` with `v` in the body of `Q`. Process communication is synchronous, and channel names can be sent and received as values.

On the other hand, the special process `zero` represents inaction, internal actions (also called *silent* actions) are noted by `tau`, and the creation of a new channel name `z` in a process `R` is represented as `(^z)R`, where the scope of `z` is restricted to `R`.

The parallel composition operator '`|`' is defined in the usual way: `P | Q` consists of processes `P` and `Q` acting in parallel. The summation operator '`+`' is used for specifying alternatives: `P + Q` may proceed to `P` or to `Q` (but not to both). The choice can be globally or locally taken. In a global choice, two processes agree in the commitment to complementary transitions, matching synchronously two complementary actions. This provides the main rule of computation in the π-calculus:

$$(\cdots + \texttt{x!(z).P} + \cdots) \mid (\cdots + \texttt{x?(y).Q} + \cdots) \xrightarrow{\tau} \texttt{P} \mid \texttt{Q\{z/y\}}$$

On the other hand, local choices are expressed by combining operator '`+`' with silent actions. Thus, a process like `(tau.P + tau.Q)` may proceed to `P` or `Q` with independence of its context by performing one internal action `tau`. We use local and global choices for stating the responsibilities for action and reaction, respectively.

There is also a *matching* operator, used for specifying conditional behavior. Thus, the process `[x=z]P` behaves as `P` if `x=z`, otherwise as `zero`.

Although the standard polyadic π-calculus does not provide for built-in data types and process parameters, we will assume that these can be simulated. Therefore we will use numbers and some basic data types, such as lists (with operators `<>` and `++` for list creation and concatenation) and sets (with {} and `++` for set creation and union). Besides, 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 allows a process given by

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

to behave as any process $\texttt{P}_i$ for which guard $[\texttt{G}_i]$ is true, or as $\texttt{P}_0$ if all guards $\texttt{G}_j$ $(1 \leq j \leq n)$ are false. Please note that this special construct is needed, because if we do not include it and all guards are false, the semantics of the matching operator in the π-calculus makes the process proceed as `zero`.

3.2 Modeling approach

The main modeling techniques that we propose and that we shall put into action in the next sections 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 we allow that request and reply channels are 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 input (and in-out) parameters, and `r` is a tuple containing the return channel and other reply channel names (for possible exceptions).
- Once the method has been served, the normal reply consists of a tuple sent by the 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.

Besides, we have added some syntactic sugar for the sake of clarity and brevity when writing the specifications of the objects' behavior:

- 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)` for accepting the invocation of method `m(args)`, instead of writing

```
ref?(method,(args),(reply)).[method='m'] ...
```

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. 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.

3.3 Extending the example specification

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

```
protocol AccountFactory {
  AccountFactory(ref) =
    ref?create(balance,rep) .
      (^acc)
      ( Account(acc,balance)
      | ( rep!(acc) . AccountFactory(ref) ) )
  + [else]
      AccountFactory(ref)
};
protocol Account {
  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)
    )
  + ref?balance(rep) .
      rep!(balance) .
      Account(ref,balance)
  + [else]
      Account(ref,balance)
};
```

There are two protocol descriptions, one for each of the two interfaces previously defined. 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 refers to 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 that replies to the request by the given channel and goes back to the original state. In case the 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 how the object decides to raise the exception (e.g. maybe the account could allow some credit). As we previously mentioned, using this formal notation allows to specify, in addition to the specific protocol information, some of the details of the object's behavior and semantics that are relevant to its potential users, while hiding those which we want to leave open to possible implementations.

Going back to the example, protocol `Bookshop` defines a possible behavior of an object compliant with interface `Bookshop`:

```
protocol Bookshop {
  Bookshop(ref,bank) =
    (^rep) bank!create(0,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) .
      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. As we can see, this process waits for the `deliver()` method call from a customer after answering the corresponding `order()` from that customer, using an special state called `Pending`. 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 (`bookstoresDB`), 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`. Besides, the behavior of process `Buy` is recursively defined by using structural induction over the list of bookshops. Thus, process `Buy` iterates over the list of bookshops until it finds one that has a book in stock that matches its requirements, or the list is empty. In the former case the process defines the order in which it accesses both the bookshop and the bank, in order to get the book delivered to the user if available. Otherwise the bookshop is skipped and the process `Buy` proceeds with the rest of bookshops in the list, until this list is empty.

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

3.4 Where do protocols live?

Before we finish this section, let us describe how protocols are created, stored, and assigned to interfaces. In the first place, analogously to where object IDLs live, each protocol resides inside a text file (with extension `.ptl`). As previously mentioned, each `protocol` description corresponds to one CORBA

`interface` declaration, and serves as the specification of its behavior. Unless otherwise stated, the name given to the protocol description is used to identify the interface it relates to. However, 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 use the keyword '`describes`' in the protocol definition as follows:

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

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.

4 Checking protocols

Once we have enriched IDLs with protocol information, this section discusses 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 and behavioral subtyping; proving safety and liveness properties of the composed applications (e.g. no deadlocks); and even achieving process refinement.

On the other hand, there are situations in which protocol compatibility needs to be checked at runtime, as it happens in open systems where components may dynamically evolve. Thus, special runtime compatibility checks are needed in those environments, in which both the components and the application internal topology may change with time. In those cases we will see that it is also possible to carry out protocol compatibility checks, and how to achieve them.

4.1 Static checking

Static checks are those carried out during the design phase of applications, prior to the execution of the components, and are based on the IDL of the constituent components and the internal structure of the application. We will discuss here three possible protocol checks:

- proving safety and liveness properties of applications;
- checking object substitutability (i.e. behavioral subtyping); and
- checking object compatibility (i.e. protocol matching).

In order to define them, we will consider components within a component framework context, which contains architectural information about the system structure and its internal interconnections. This information can be used at design time to perform static analysis of protocol compatibility among the IDLs of the components that take part of the system. In our approach, the π-calculus specifications of the components' behavior will be used for the checks, that will be automated by using π-calculus tools (animators) that will execute the specifications.

It is our view that tools are indispensable for checking those formal specifications. Not only they help during the development phase of the specifications, but also they are necessary for the applicability and effectiveness of formal methods in the production of *real* applications (with hundreds of components offering thousands of operations). We think that without those tools, the industrial use of formal methods will definitely fail to materialize.

Let us discuss now in detail the static checks that are possible with our approach, and how to achieve them using the standard π-calculus mechanisms.

4.1.1 *Proving safety and liveness properties of applications*

The first issue that can be checked with our approach is the absence of deadlocks during the application's lifetime, which guarantees the correct interworking among all the components of a given application. Please note that these are the sort of checks that were commonly carried out by software architects using their ADLs (as in Wright [2] or LEDA [4]), testing that the architecture of an application is complete and deadlock-free. We will show how this can be achieved by using our proposed extension of the CORBA IDLs of the objects that take part in a given application.

Based on the previous example, suppose that we have an application with 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:

```
(^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 *enviroment*, by simply representing an object that calls the book-broker method `getABook()` looking for a given book and waits for its reply. More precisely, its behavior can be described as follows:

```
protocol User {
  User(ref,bookbroker) =
    (^auth,title,price,addr)
    bookbroker!getABook(auth,title,price,addr) .
    bookbroker?(yes,when) .
    zero
};
```

Therefore, the absence of deadlocks in the application can be easily checked. However, we are strongly relying on the application's internal structure. There is no problem for closed applications, but this is a very strong assumption for open and evolvable applications.

4.1.2 *Object substitutability*

Object substitutability refers to the ability of an object to replace another, in such way that the change is transparent to external clients, i.e. so that the new object offers the same services as the old one [17].

At the signature level this issue is not difficult to solve, it is just a matter of checking that the interface of the new object contains all methods of the object to be replaced. However, the situation is different at the protocol level:

- In the first place, we also need to check that the services required by the new object when implementing its methods (hereinafter called *outgoing operations*) are a subset of the outgoing operations of the old one. Otherwise, we may require to add some additional components to the application when replacing the old component with a new one (cf. [20]).
- And second, the ordering among incoming and outgoing messages should be consistent between the old and the new versions of the object.

The first issue can be easily managed in our proposal. Since all methods are called and responded using channels, the set of the incoming (resp. outgoing) operations of an object can be calculated as the union of all the values read (resp. written) by the object through all the channels it uses.

With regard to the partial order of operations, the π-calculus offers the standard axiomatization of bisimilarity, which supports the replacement of processes. However, bisimulation is too strict for our purposes since it forces the behavior of both objects to be undistinguishable. Effective replacement of a software object often implies that it must be adapted or specialized to accommodate it to new requirements [17]. For this reason, we make use of an specific mechanism for behavioral subtyping of processes (less restrictive than bisimilarity) defined in [5], that allows to decide whether a given object with protocol description $\mathtt{P}_1$ can be replaced by another one with protocol description $\mathtt{P}_2$, while keeping the object's clients unaware of the change.

In order to illustrate all this, let us use of one of the objects in our example, namely the `Bookshop`. In the first place, taking a look at its protocol description we can easily compute its incoming and outgoing operations:

- Apart from the channel it uses for receiving requests, this object makes use of two additional channels (called `bank` and `ch`) to communicate with the `AccountFactory` object, plus the reply channels that the client objects may specify. Through all these channels, the incoming operations are the methods declared in the object IDL, together with the return of the `create()` method call received along the `ch` channel.
- With regard to the outgoing operations, apart from this `create()` method call, we only have the object responses to the invocations of its methods.

Now, let us suppose that we have another `Bookshop` object, whose behavior is defined by:

```
protocol BetterBookshop describes Bookshop {
  Bookshop(ref,bank) =
    (^rep) bank!create(0,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)
};
```

We can see in this example that the incoming and outgoing operations of both objects are the same, but that their behavior is not. Actually, using the relation of behavioral subtyping defined in [5], it is not difficult to prove that the behavior described by protocol `Bookshop` is more restrictive than that of `BetterBookshop`. Thus, we can replace a component exhibiting the former behavior by one compliant with the latter protocol without affecting safety and liveness properties in the system; a client of `Bookshop` could perfectly be a client of `BetterBookshop`, but not vice versa. The main difference is that `BetterBookshop` presents a completely concurrent behavior, while the original protocol `Bookshop` waits for the `deliver()` method call from a customer after answering the corresponding `order()` from that customer. This latter behavior does not allow the processing of further services in-between, unlike the behavior described by protocol `BetterBookshop`.

4.1.3 *Object compatibility*

Object compatibility can be described as the ability of two objects to work properly together if connected, i.e. that all exchanged messages between both objects are understood by each other, and that their communication is deadlock free [25]. Substitutability and compatibility are the two flip sides of the *object interoperability* coin.

The π-calculus is a formal notation very appropriate for checking object compatibility, since it allows to know whether two protocol descriptions are *compatible* by just checking than their parallel composition is deadlock free (due to the semantics of the π-calculus, this is the only test needed). However, objects may simultaneously interact with more than one other object, so in order to define compatibility between just two objects we need to talk about *roles*. Roles are abstract and partial descriptions of the behavior (i.e. the rôle) an object plays in its interactions with another object. Hence, roles are partial protocol specifications, in which we only pay attention to the behavioral interface that a certain component presents to another one, thus allowing pairwise checks of compatibility, with reduced complexity. A role is obtained from the corresponding object protocol by restricting it to a certain set of communication channels (those significant to the component connected to the role), while hiding the rest of channels [5]. The restriction of channel names is expressed by means of clause '`restricts`' which defines the set of channels and/or methods that will be visible to the role, using the following notation:

- '`*.m1`' Method `m1` sent or received by any channel.
- '`ref.*`' All methods sent or received through channel `ref`, i.e. any method of the object referenced by `ref`.
- '`ref.m1`' Method `m1` of the object referenced by channel `ref`.

For instance, we can divide the specification of our `BookBroker` into several different roles: one for indicating its behavior with respect to the addition and removal of bookstores, a second one with respect to the operations related to bookstores, and a third one representing its behavior as seen from users. This last role can be defined in our approach as follows:

```
protocol RBB restricts BookBroker [ref.getaBook]
```

This produces a new protocol `RBB`, which is automatically obtained from `BookBroker` by applying a series of transformations: (a) all communication actions not significant to the users' point of view have been abstracted, being replaced by silent `tau` actions or even omitted when they do not imply a local decision; (b) input variables that came through hidden channels are now fresh variables; and (c) guards have been also replaced accordingly. In this case, the new protocol obtained is the following:

```
protocol RBB {
  RBB(ref) =
    ref?getaBook(auth,title,price,addr,rep) .
    ( Buy(ref,auth,title,price,addr,rep)
    | RBB(ref) )
  Buy(ref,auth,title,price,addr,rep) =
    tau . rep!(FALSE,NIL) . zero
  + tau . (^date) rep!(TRUE,date) . zero
};
```

In order to check compatibility of both objects, we can simply execute the following π-calculus process:

```
(^ref,bb) ( User(ref,bb) | RBB(bb) )
```

and check that it evaluates to `zero` (i.e. it is deadlock-free).

Not only pairwise compatibility can be checked in our proposal, but also that three or more protocols are compatible among themselves. In order to do that, we need just to restrict the protocols to their shared communication channels and evaluate their parallel composition. The most general situation of checking the compatibility among *all* the

components of an application gets simply reduced to evaluate the parallel composition of all the protocols without channel restrictions (all channels are used). Please note that this case clearly corresponds to the situation described in section 4.1.1, i.e., that the application is deadlock free.

4.2 Run-time checking

Apart from the static checks, there are many situations in which protocol compatibility has to be checked at run time. Typical cases are the applications developed in open and *independently extensible* systems [19], in which the evolution of the system and its components is unpredictable: new components may suddenly appear or disappear, while others are replaced without previous notification. Internet is probably the most well-known example of those systems. The Web has re-educated us all into expecting systems to be more dynamic, and to be able to load new components into an already running system and have the system reconfigure itself to accommodate the new services which these objects supply, while old components graciously retire when their services are no longer required. Unfortunately, in most situations the architecture of the applications is not made explicit anywhere, and therefore in those cases the static checks previously mentioned are no longer valid, since they cannot be used for dynamic attachments among unknown components.

In order to clarify the situation in this sort of open environments, it is important to realize some of the issues that have to be considered in them, namely:

(a) components are defined independently from their context and the applications they will be part of;
(b) the connections among components are not explicitly stated anywhere; and
(c) components may dynamically join or leave the applications.

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. This sort of information is very useful for system debugging, and it may help components to make run time decisions about their behavior within an application.

Components entering in a deadlock state can be notified by an error event about the situation, so they can react accordingly if they wish. For instance, if in the previous example it is detected that an invalid request has been issued to a `Bookshop` object that obeys a `Bookshop` protocol, the request can be rejected, hence avoiding the deadlock situation.

To implement this facilities in CORBA we have used a reflexive facility that some ORB vendors provide: Filters. This mechanism was originally defined and implemented in Orbix [3], and allows a programmer to specify some additional code to be executed before or after the normal code of an operation. This code may perform security checks, provide debugging traps or information, maintain an audit trail, etc. Although less powerful than other object reflexive facilities (such as Composition Filters [1] or Object Filters [12]), they provide the mechanism that we need, since they allow the interception and observation of the messages exchanged among components. Thus, a filter can be defined for each object that captures incoming and outgoing messages, reproduces its run-time trace, and checks that received messages are compatible with the behavior defined for that object.

4.3 How and when to check

Summarizing, we can identify two different stages where compatibility between the components that form part of an open application can be checked:

1. At design time, in which a static analysis of components can be made prior to their execution.
2. At run time, in which all the messages exchanged between the components are checked for consistency with their current states, detecting deadlock or starvation situations.

Both checks are possible with our proposal, but the question is whether they are useful and practically achievable. For instance, design time compatibility checks are very useful in closed applications, but rather limited for open applications, in which components evolve over time and 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 done on the fly by the object filters 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 also delayed until

just before they happen, which may be unacceptable in some specific situations (nevertheless, we are able in this way to check and know about the occurrence of those problems, which is better than completely ignoring them).

5 Open issues

In this section we will discuss further issues related to object interoperability at the protocol level, but for which we do not know about any easy solution yet.

Adaptors. Once we have characterized the behavior of the components, we can check their compatibility. However, in case we decide that the behavior of two components is incompatible, a new question arises: is it possible to build extra components that *adapt* their interfaces, compensating their differences? Those extra components are called *adaptors* [25], and their automated construction right from the description of the interfaces of two components is a difficult problem: it requires not only proving the existence of the adaptor, but also deriving its interfaces and code (cf. [25]).

Checking compatibility at connection time. On top of the benefits for open system debugging and prototyping that we can obtain with the previous protocol checks (both static and during system execution), we would like to explore a third possibility, and see whether we could perform static analysis at *connection time*, just when 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 an intermediate state.

Conformance to specifications. The last issue that we will discuss here is about how to test that a given implementation of an object conforms to a given specification of its behavior. There is no problem at the signature level, where it is a matter of checking that all methods defined in the interface are actually implemented by the object. However, it is a completely different situation at the protocol level. In that case we need to check that the actual *implementation* conforms to the behavior specified in the protocol, but this is usually impossible: we are dealing with black-box components, whose code is unaccessible.

Nevertheless, there is one possible way to partially deal with this problem, using the previously defined filters. They were used for checking that incoming messages to an object were valid with regard to its current state. But they can also be used for checking that the object's behavior is valid with regard to the protocol it is supposed to implement. In this sense, filters can be used to *enforce* behaviors, in a similar way to Minsky's Law Governed Architectures [16].

6 Related Work

The contributions we have presented in this paper fall into two main categories: the extension of object IDLs for coping with the semantic aspect of their behavior, and the use of formal notations for describing those behaviors.

Several authors have provided a number of proposals that try to overcome the limitations that current IDLs present, defining IDL extensions that usually cope for the semantic aspects of object interfaces and behavior. For instance, Doug Lea's PSL [13] proposes 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 the object may need from other objects, neither it is supported by standard proving tools.

Protocol Specifications [25] 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.

Another approach by Cho, McGregor and Krause [6] 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. Jun Han [9] also 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. They are both similar approaches, although none of them is associated to any commercial component platform like CORBA or EJB, or supported by tools.

Reuse Contracts [18] is another well known proposal, although it is based on textual annotations to facilitate the reuse by humans, not by computer programs during object execution.

Finally, we will also mention Architectural Description Languages (ADLs), since they usually include the descriptions of the protocols that determine the access to the components they define. It is worth mentioning the use of standard notations for protocol description in some ADL proposals, that derive from process algebras (like CSP, CCS or π-calculus) for describing what they call the 'dynamic' aspects of the systems. One of the benefits of using standard calculi is that reasoning about system behavior and correctness can be done using appropriate tools. Wright [2] and LEDA [4] are examples of ADLs that make use of the π-calculus for describing the behavior of the components of a system.

With regard to the second topic, the use of formal notations in commercial environments, we share the thesis by P. Henderson [10] that formal methods (such as the π-calculus) are mature enough to be used in the design and validation of components of large distributed systems, and that the use of such methods will lead to the better design of components and of component-based applications in open systems. What we have seen here is how formal tools and commercial products do not have to live in separate worlds. It is possible to combine them to improve the component-based software development process.

On the other hand, the π-calculus has been successfully used for describing some aspects of the architecture of component models like COM [7] or CORBA [8]. In this paper we have shown how it can also be used to describe the semantics of the dynamic behavior of the components, not only of the model's communications mechanisms. The modeling of 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. Besides, as we have previously mentioned, this notation is more expressive than other process algebras such as CSP or CSP, and it also allows the description of more than just protocol information, which makes it more appropriate for our purposes than other protocol description notations, such as MSC.

Moreover, the π-calculus also offers good tool support, with several products for animating π-calculus specifications. Interesting examples are MWB [22] or epi [11]. Both are executable versions of the polyadic π-calculus, with some enhancements added. However, we did not want to commit ourselves to any particular tool. Following the CORBA IDL philosophy of producing platform-independent interface descriptions, we decided to produce tool-independent protocol specifications that could be easily translated (or compiled) into different executable versions of polyadic π-calculus. The only requirements are the support for basic data types, and the simulation of the extensions we have defined for the matching operator.

7 Discussion

In this paper we have outlined the importance of incorporating protocol information into object interface descriptions. Our proposal extends traditional IDLs with two different sorts of information: incoming and outgoing methods, and protocol descriptions (partial ordering of messages and blocking conditions) described using π-calculus. As major benefits, the information needed for object reuse is now available as part of their interfaces, and more precise interoperability checks can be achieved when building up applications.

Apart from the open issues previously described, there are still a significant number of major challenges ahead. We have seen how to add protocol information to CORBA IDLs, and the sort of benefits that can be obtained from it. The addition of this type of information to other object and component models —like DCOM, EJB or the new CORBA Component Model (CCM)— is an ongoing subject of research, as well as the development of new services and tools that make use of it —such as protocol repositories or extended service traders—, and the extension of this sort of information to cope for more semantic aspects of the functionality of components.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proc. of ECOOP'93*, number 791 in LNCS, pages 152–184. Springer-Verlag, 1993.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, July 1997.

[3] S. Baker. *CORBA Distributed Objects*. Addison-Wesley Longman, 1997.

[4] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Proc. of WICSA'99*, pages 107–125, February 1999.

[5] C. Canal, E. Pimetel, and J. Troya. Conformance and refinement of behavior in π-calculus. In *Proc. of the 2nd Int. Workshop on Component-Based Development in Computational Logic*, Paris, September 1999.

[6] 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.

[7] L. Feijs. Modelling Microsoft COM using π-calculus. In *Proceedings of FME'99*, number 1709 in LNCS, pages 1343–1363. Springer-Verlag, September 1999.

[8] M. Gaspari and G. Zabattaro. A process algebraic specification of the new asynchornous CORBA messaging service. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 495–518. Springer-Verlag, 1999.

[9] J. Han. Semantic and usage packaging for software components. In Vallecillo et al. [21], pages 25–34.

[10] P. Henderson. Formal models of process components. In *Proc. of the FSE'97 FoCBS Workshop*, pages 131–140, Zurich, September 1997.

[11] P. Henderson. From formal models to validated components in an evolving system. Technical report, University of Southampton, 1998.

[12] R. Joshi, N. Vivekananda, and D. J. Ram. Message filters for object-oriented systems. *Software–Practice and Experience*, 17(6):677–699, 1997.

[13] D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.

[14] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[16] N. Minsky and J. Leichter. Law-governed Linda as a coordination model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems*, number 924 in LNCS, pages 125–146. Springer-Verlag, 1995.

[17] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[18] P. Steyaert, C. Lucas, K. Mens, and T. d'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.

[19] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[20] J. M. Troya and A. Vallecillo. Specifying reusable controllers for software components. In *Proc. of FMOODS'99*, pages 131–148, Florence, February 1999. Kluwer.

[21] A. Vallecillo, J. Hernández, and J. M. Troya, editors. *Proc. of the ECOOP'99 Workshop on Object Interoperability*, June 1999.

[22] B. Victor. A verification tool for the polyadic π-calculus. Master's thesis, Department of Computer Systems, Uppsala University (Sweden), May 1994.

[23] S. Vinoski. New features for CORBA 3.0. *Commun. ACM*, 41(10):44–52, October 1998.

[24] P. Wegner. Interoperability. *ACM Comp. Surveys*, 28(1):285–287, March 1996.

[25] 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.