Extending CORBA Interfaces with Protocols

C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga.
{canal,lff,ernesto,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 relative order in which they expect their methods to be called. 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 at the automated checking of protocol interoperability between CORBA objects in open component-based environments. In addition, 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

Component-Based Software Engineering is an emergent discipline that is generating tremendous interest due to the development of plug-and-play reusable software, which has led to the concept of '*commercial off-the-shelf*' (COTS) components. Under this new setting, constructing an application now involves the use of prefabricated pieces, perhaps developed at different times, by different people unaware of each other, and possibly with different uses in mind. The ultimate goal, once again, is to be able to reduce development costs and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated.

This approach moves organizations from application *development* to application *assembly*. The development effort now becomes one of gradual discovery about the components, their capabilities, their internal assumptions, and the incompatibilities that arise when they are used in concert. Therefore, software components search and retrieval have become crucial in these environments, since we need to be able to check whether a given component can successfully replace another in a particular application, or whether the behavior of two components is compatible for them to interoperate.

Interoperability deals with these issues, 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 [Wegner, 1996]. Basically, three main levels of interoperability between objects can be distinguished: the *signature level* (names and signatures of operations), the *protocol level* (relative ordering between exchanged messages and blocking conditions), and the *semantic level* (the "meaning" of the operations) [Vallecillo et al., 2000].

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), and bridges among them. However, all parties are starting to recognize that this sort of interoperability is not sufficient for ensuring the correct development of large applications in open systems.

Object interoperability should be studied in general at the semantic level, which deals with operational semantics and behavioral specifications, as well as with agreements on names, context-sensitive information, agreements on concepts (ontologies), etc. [Heiler, 1995]. The problem is that this semantic level covers a very broad set of issues, and there is not even a consensus on the full scope of what those issues are. In the literature, most of the authors that say "semantics" really mean just "operational semantics" or "behavioral specifications", and propose different formalisms to deal with object interoperability at this level, such as pre/post conditions, temporal logic, or refinement calculus (see [Leavens and Sitaraman, 2000] for a comprehensive survey on these proposals). But even dealing with behavioral specifications needs very heavy machinery, which makes most approaches impractical for real applications. Dealing with the full semantics is a too broad and general problem to be tackled in full.

As a first step, in this paper we concentrate on the interoperability of reusable components at the protocol level, —which deals with the relative order among their incoming and outgoing messages, and their blocking conditions—, 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 leading commercial object platforms.

Our work proposes an extension to the CORBA IDL that allows the description of certain 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, deployers, and implementors of objects.

Protocols are described using a 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. In addition, having this protocol information available at run time also allows to define dynamic compatibility checks in open and extensible applications. Therefore, we will be able to reason about the compatibility and substitutability of their components, and to infer some safety properties directly from the description of the behavior of their constituent components.

In general, software *components* are binary units of possibly independent production, acquisition and deployment that interact to form a functioning system [Szyperski, 1998]. In this CORBA context, the term component will simplistically refer to CORBA distributed objects.

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 discusses what sort of information should protocols include, while section 5 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., some kind of behavioral subtyping [America, 1991, Nierstrasz, 1995]), and on proving some safety properties of applications (e.g. absence of deadlocks). Additionally, 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. Later, section 6 discusses some of the limitations and problems that this sort of proposals introduce, and section 7 relates our work to other similar approaches. Finally, section 8 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. *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, there are two main 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 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 being 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 [Ólafsson and Bryan, 1996]. At most, in some CORBA applications some IDL modules contain not only the servers' interfaces, but also the definition of "call-back" interfaces that need to be satisfied by clients (e.g., in the CORBA Event service [OMG, 2000]). However, this only happens at the `module` level, while there is no syntactic support for this distinction between offered and required interfaces at the individual component level.
2. Typical IDLs provide just the *syntactic* descriptions of the objects' public methods, i.e., their signatures. However, nothing is said about the correct way (e.g. the order) in which the objects expect their methods to be called, or their blocking conditions [Yellin and Strom, 1997].
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 object adaptation, which are among the most commonly required facilities for building component-based applications in open and independently extensible systems [Szyperski, 1998].

2.1 CORBA and its IDL

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

The Common Object Request Broker Architecture (CORBA) specification describes the interfaces and services that must be provided by compliant ORBs [OMG, 1999].

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

Even though an object reference identifies a particular object, it does not necessarily describe anything about the object's interface. Before an application can make use of an object, it must know what services the object provides. CORBA defines an IDL to describe object interfaces, a textual language with a syntax resembling that of C++. The CORBA IDL provides basic data types (such as `short`, `long`, `float`, ...), constructed types (`struct`, `union`) and template types (`sequence`, arrays). These are used to describe the interface of objects, defined by set of types, attributes and the signature (parameters, return types and exceptions raised) of the object methods, grouped into `interface` definitions. Finally, the construct `module` is used to hold type definitions, interfaces, and other modules for name scoping purposes.

2.2 A case study

In order to illustrate our proposal, let us describe a simple E-commerce application: a distributed auction system. Three main players are involved in this application: an auctioneer that sells items on behalf of their customers, bidders that bid for the items being auctioned, and banks that deal with money transactions.

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

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

```
interface Account {
  exception NotEnoughMoney {float balance;};
  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 called `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 extract more money than currently available in the account.

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

```
interface Bidder {
  bool wannaBid(in string itemdesc, in float price);
  void youGotIt(in string itemdesc, in float price, in Account acc, out string address);
  void itemSold(in string itemdesc);
};
```

This interface defines three methods, to be invoked by the auctioneer. They allow the bidder to decide whether he is interested in a given item or not, to be notified that he got the item he was bidding for, and to notify all bidders (including the one that got it) that the item has been finally sold. Method `youGotIt()` has four arguments: the item description and its price, the auctioneer's account so the bidder can deposit the money there, and the address where the bidder wants the item to be delivered at.

Finally, an auctioneer is an object that maintains a list of bidders, and that is able to sell items on behalf of a customer.

```
interface Auctioneer {
  void register(in Bidder b);
  void unregister (in Bidder b);
  bool sell(in string itemdesc, in float minprice, in float maxprice, in Account acc,
            out float finalprice);
};
```

In this interface, methods `register()` and `unregister()` are for adding and deleting references from the list of bidders that the auctioneer keeps[1]. Method `sell()` implements the main service offered by the object. The customer specifies an item, the range of prices she is happy to sell it for, and the account where the auctioneer will deposit the money once the item is sold. This method returns a boolean value indicating whether the operation has succeeded or not, and the final price for which the item was sold.

As we can see, it is very difficult to learn the behavior of those objects just from their interfaces. Actually, from them and from their description in natural language we could at most try to *guess* the bidding protocol they follow, and how they use each other's services. Although this sort of specification is the one currently used by the OMG for specifying CORBA services, many experiences show how insufficient and ambiguous it can be [Bastide et al., 1999, Bastide and Sy, 2000]. For instance, does the auctioneer follow a normal or a dutch bidding protocol? (in a dutch auction the price is reduced by the auctioneer until a buyer is found). May bidders join an ongoing sell as soon as they register, or do they have to wait until an auction for a new item starts? What happens if a bidder receives a `youGotIt()` message for an item that he is not bidding for? How do money transactions happen between the seller, the bidder, and the auctioneer? Those (and many others) are questions that are left unanswered from the IDL descriptions of the participant objects.

3 Extending CORBA interfaces with protocols

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 description of the object's interactions, and the rules governing those interactions.

Protocols will be described using a sugared subset of the polyadic π-calculus, a process algebra particularly 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, it is more appropriate than other process algebras such as CCS or CSP.

[1] This is a very simplictic way of registering objects, but we will use it here for the sake of clarity; other methods for object registration/unregistration can be found in [OMG, 2000].

In our approach we can also define component interaction protocols that make use of operation parameters and return values when considering alternatives, which is an important issue concerning the expressiveness of protocol specifications, and that represent some advantages over other approaches, such as those based on state-machines.

Another extra benefit of this formal notation is that it allows us 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. In this sense, π-calculus also offers this advantage over other formal notations for describing mere protocol information, such as Message Sequence Charts (MSCs), for instance.

Finally, we think that the subset of the polyadic π-calculus we have chosen, together with the syntactic sugar added, provides a quite natural notation for describing protocols, and that it should not present major difficulties for CORBA component designers and developers —not generally familiar with process calculi— when describing their interaction protocols. It may not be as intuitive as MSCs (e.g., it is not graphical), but it is more expressive (as previously stated) and counts with the formal underpinnings lacking in MSCs. And definitely is not more difficult to learn than other formal notations for describing protocols, such as Petri nets, or temporal logic. In this sense, we think that the π-calculus is indeed a valid alternative for describing protocols in a formal way.

3.1 The polyadic π-calculus

The π-calculus was originally proposed by Milner, Parrow and Walker in 1992 [Milner et al., 1992]. 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 can be considered as an extension to CCS, where not only values can be passed around, but also channel names.

The polyadic π-calculus [Milner, 1993] 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 labeled transitions.

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 `Q{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.

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`. This scope can be extended to include other processes by simply sending the new name in an output action, as in `(^z)ch!(z).S`.

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 communication in the π-calculus:

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

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, 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, and the usual `head` and `tail` operations). 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. In order to make the manipulation of data structures easier, we will also use the "`=`" symbol for representing term unification, i.e., an expression like `t1=t2` may produce a binding on variables appearing either in `t1` or `t2`. We will also use the construct `[else]`, that provides some syntactic sugar for expressing that a process given by

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

behaves 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 if we do not include the `[else]` part and all guards are false, the semantics of the matching operator in the π-calculus makes the process deadlock.

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 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 channel named `excp` has to be sent along within the return channels tuple. The server object may 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 can be 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:

- Method invocation `ref!(m,(args),(reply))` is abbreviated to the simpler form `ref!m(args,reply)`, or even to `ref!m(args)` if the reference and reply channels are the same.
- Similarly, on the server side we write `ref?m(args,reply)` for accepting the invocation of method `m(args)`, instead of writing `ref?(meth,(args),(reply)).[meth='m']...`

Protocols are defined using the 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.

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

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 then 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.

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 (the `withdraw()` method may also raise an exception), and behaves again as an account. Please note how extra channel `notEnough` is used by method `withdraw` to raise the exception, as discussed in section 3.2.

This protocol description only shows the interactions of `Account` objects with their clients, but without any of the object's behavioral semantics, such as how the operations handle the balance, or when the object decides to raise the exception (e.g. maybe the account allows some credit). Protocols have been designed for describing objects' interactions mainly, although they also have the ability to specify some semantic information too by means of process variables that store the object's internal states. Thus, in addition to the specific protocol information, the designer could specify 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. In general, this sort of semantic information to be included in a protocol description will depend on the level of detail that the component's designer wants to specify. More on this issue will be discussed later in section 4.

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

```
protocol Bidder {
 Bidder(ref,auctioneer) = auctioneer!register(ref) . auctioneer?() . Bidding(ref,auctioneer) ;

 Bidding(ref,auc) = WaitingRound(ref,auc) + auc!unregister(ref) . auc?() . Bidder(ref,auc) ;

 WaitingRound(ref,auc) =
   ref?wannaBid(item,price,reply) . Deciding(ref,reply,auc)
 + ref?itemSold(item) . Bidding(ref,auc) ;

 Deciding(ref,reply,auc) =
   tau . reply!(TRUE) . Waiting(ref,auc)
 + tau . reply!(FALSE) . Bidding(ref,auc) ;

 Waiting(ref,auc) =
   ref?youGotIt(item,price,acc,reply) .
     acc!deposit(price) . acc?(receipt) . (^addr) reply!(addr) . Bidding(ref,auc)
 + WaitingRound(ref,auc)
};
```

This process has two arguments: the object reference channel, and the reference channel of the auctioneer. It specifies that the bidder starts by registering itself, then it waits for a confirmation from the auctioneer and behaves as process `Bidding`. This process waits for the auctioneer to start a new round for selling an item (process `WaitingRound`), but it also gives the bidder the possibility of leaving the current auction (`unregister`). In process `WaitingRound` the bidder is either informed that the actioneer is ready for receiving bids for an item (`ref?wannaBid`) or that an item has been sold (`ref?itemSold`). In the first

case (`Deciding`), and provided the bidder is interested in an item, it replies positively to the auctioneer and behaves as process `Waiting`, that keeps waiting until (a) he gets the item (`ref?youGotIt`), then pays for it, and replies to the auctioneer with the address he wants the item to be delivered at; or (b) a new round is started by the auctioneer (`WaitingRound`). Local decisions are modeled by internal actions `tau`, which leaves open to the object implementor the mechanism used to make the decisions.

Finally, a possible behavior of the `Auctioneer` object can be described as follows:

```
protocol Auctioneer {
 Auctioneer(ref,bidders) =
   ref?register(b,rep) . rep!() . Auctioneer(ref,bidders++<b>)
 + ref?unregister(rep) . rep!() . Auctioneer(ref,bidders--<b>)
 + [bidders!=<>] ref?sell(item,initial,giveup,acc,rep) Auctioning(ref,bidders,rep,acc) ;

 Auctioning(ref,bidders,rep,acc) =
   ( (^done) NotifyBid(bidders,NULL,done)
   | done?(winner) . EndOfRound(ref,bidders,winner,rep,acc) ) ;

 EndOfRound(ref,bidders,winner,rep,acc) =
   [winner != NULL] % there is at least one bid for the item
     tau . % and the auctioneer decides to sell the item
     (^item,price) winner!youGotIt(item,price,acc) . winner?(addr) .
     rep!(TRUE,price) . NotifySale(ref,bidders)
 + tau . % or to start a new round
     Auctioning(ref,bidders,rep,acc)
 + tau . % or to give up selling the item
     rep!(FALSE,0) . NotifySale(ref,bidders) ;

 NotifyBid(bidders,winner,done) =
   [bidders !=<> and b=head(bidders) and others=tail(bidders)]
     (^item,price) b!wannaBid(item,price) . b?(yesorno) .
       ( [yesorno = TRUE] NotifyBid(bidders,b,done)
       + [else] NotifyBid(others,winner,done) )
 + [else] done!(winner).zero ;

 NotifySale(ref,bidders)
   [bidders !=<> and b=head(bidders) and others=tail(bidders)]
     (^item)b!itemSold(item) . NotifySale(ref,others)
 + [else] Auctioneer(ref,bidders)
 };
```

In this example we can see how the auctioneer accepts operations `register()` and `unregister()` from the bidders, in order to update the list of bidders it knows about. When a `sell()` method is received, the auctioneer spawns a new process (`Auctioning`) that notifies the item price to the bidders currently registered (by means of process `NotifyBid`, which also waits for their replies). Once this notification process is over, it behaves as `EndOfRound`, that allows the auctioneer to: (a) decide to sell the item to one of the bidders, (b) start a new round, or (c) give up selling the item. In the first and last cases the bid is over, which is notified to all the bidders by process `NotifySale`.

It is important to notice that this is a *possible* behavior. Of course, alternative behaviors could have been specified. For instance, instead of using a list and notify the bidders according to the order in which they appear there, the order of notification could have been left unspecified (and hence left open to the implementor). On the other hand, the above protocol also leaves some other details open, e.g. both normal and dutch auctions could conform to it, and it does not state how the price is determined in each round.

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 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 as follows:

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

In particular, since every component interface should have a protocol associated, the first version of this protocol will usually have the same name as the interface it describes. In this case, the `describes` clause may be omitted.

At specification level a component is characterized by an interface and a protocol description. By changing the protocol description, and maintaining the interface, we can obtain different versions of a component. Likewise, for each protocol we may produce, or find inside a repository, different component implementations.

Protocols are defined to describe the behavior of a given interface, and they do not make sense in isolation (without the interface it refers to). However, 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.

The protocol description file `.ptl` will be used by application builders during the architectural design and verification, and at runtime by the `ProtocolTracer`, a π-calculus interpreter used by the CORBA object interceptors for verifying the correctness of input and output messages with respect to the object protocol (see Figure 1). This issue will be explained in more detail in section 5.2.

4 What information should be included in protocol descriptions?

In general, there are no precise guidelines about what should and should not be included in a protocol specification. It will depend, of course, on the level of abstraction or detail required. As an example, think of a queue and a stack. They are "functionally" different, although they may present the same access protocols. On the contrary, a bank could offer several access protocols for its transactions, e.g. two-phase commit or three-phase commit protocols. The "operational semantics" of each bank operation is the same, the only thing that changes is the access protocol. Although there is always a compromise, our suggestion is to include in protocol descriptions just the information about the components' interactions, keeping away as much as possible the "functional" aspects of components. However, the separation of concerns between computation and coordination aspects is often subtle, as the following example shows.

Suppose that we have the `Account` interface described in section 2.2. A possible behavior of this interface was already specified in section 3.3, although that `Account` protocol described a pretty abstract component in which we only paid attention to its coordination aspects. It did not even disclose how the operations modified the balance of the account, or when the exception was risen.

We could argue that protocol `Account` is therefore too abstract, and it does not represent precisely the behavior of the account we are trying to specify. Thus, we could write a slightly lower level (less abstract) protocol `Account2` as follows:

```
protocol Account2 describes Account {
  Account(ref,balance) =
    ref?getBalance(rep) . rep!(balance) . Account(ref)
  + ref?deposit(amount,rep) . (^receipt) rep!(receipt) . Account(ref,balance+amount)
  + ref?withdraw(amount,rep,notEnough) .
    ( [amount<=balance] (^receipt) rep!(receipt) . Account(ref,balance-amount)
    + [else] (^balance) notEnough!(balance) . Account(ref,balance)
    )
};
```

Please note the use of the second argument of the process to store 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. Furthermore, protocol `Account2` also deny money overdrafts by raising the exception when a user wants to withdraw more money than currently available. However, most people could argue now that we are overspecifying the interface of our account, since protocol `Account2` above describes not only the coordination pattern followed by the buffer but also some of its computational aspects, i.e., its semantics. Protocol `Account2` makes visible its internal state, and also its policy for overdrafts. It also rules out other different behaviors of an account, e.g., those behaviors that allow some credit, those that charge commissions on the account operations, or those that pay interests.

Which one is the correct level of detail for representing the behavior of the account? Unfortunately, we cannot give a definite answer to this question; it will depend on the system being specified and on the level of abstraction desired by the specifier. In order to check its compatibility or substitutability with other components, probably the first protocol `Account` is enough, but if we want to be more precise on the behavioral semantics of the account, protocol `Account2` may be required. All this can be applied to the protocol describing the behavior of the `Auctioneer`, too. For instance, we could have described how it increases or decreases the price of the item in each round, how the winner is selected, etc. Once again, the semantic details to be included in the protocol will depend on the particular requirements of the protocol specifier.

Another interesting possibility to be considered is the separation of behavioral aspects. Mixing all syntactic, protocol, and semantic aspects within one component specification will produce too large, complex, and brittle specifications to be of practical interest. Furthermore, using (imagined) internal states in the specification of interaction protocols (as in state machine based approaches) renders the specification more abstract, makes the specification less understandable and may discourage practitioners from using protocol specifications altogether. Therefore, it seems like a good idea to be able to specify each aspect of

the specification in a *modular* way, so different protocols can be associated to the same (functional) services, or that separate specification aspects can be composed to meet complex and changing requirements. Thus, signature, protocol, and semantic specifications can be separately specified, which may lead to more modular specifications, achieving in this way a beneficial separation of concerns.

5 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.

5.1 Static checks

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 binds among them (i.e., the structure of the application). This information can be used at design time to perform static analysis of some safety properties of the composed applications. With this approach we will also be able to address one of the shortcomings of the development of distributed applications using commercial platforms, such as CORBA or EJB: they do not provide the mechanisms for explicitly describing the architecture of the applications, just the interfaces of their components.

In our approach the π-calculus specifications of the components' protocols will be used for the checks, with the additional benefit that π-calculus tools (animators) can execute the specifications, hence automating all checks. It is our view that tools are indispensable for checking formal specifications, in particular in the production of *real* applications (with hundreds of components offering thousands of operations). Without those tools, the industrial use of formal methods will definitely fail to materialize.

We will discuss here some possible protocol checks, namely object compatibility and substitutability.

5.1.1 *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 [Yellin and Strom, 1997].

This is exactly what we do, since our approach allows to check the absence of deadlocks during the application's lifetime, which guarantees the correct interworking among all its components (i.e., their compatibility). Please note that these are the sort of checks commonly carried out by software architects using their ADLs (e.g. ACME [Garlan et al., 1997], Wright [Allen and Garlan, 1997], or LEDA [Canal et al., 1999]), testing that the architecture of an application is complete (there are no missing components or services), and deadlock-free. We will show how those tests can be also achieved with our proposal, right from the objects' protocol specifications.

Based on the previous example, suppose that we have an application with one bank, two bidders, one auctioneer, and one seller, and that we want to check whether this system is deadlock-free or not. 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() =
  (^af)     % AccountFactory's address
  (^b1,b2)  % Addresses of the two bidders
  (^au)     % Auctioneer's address
  (^sl)     % Seller's address
  ( AccountFactory(af) | Auctioneer(au,<>) | Bidder(b1,au) | Bidder(b2,au) | Seller(sl,af,au) )
```

In this case the `Seller` process models the *environment*, by simply representing an object that calls the auctioneer method `sell()` trying to sell an item, and waits for its reply. More precisely, its observable behavior can be described as follows:

```
protocol Seller {
  Seller(ref,bank,auctioneer) =
    bank!create() . bank?(acc) .
    (^item,minprice,maxprice) auctioneer!sell(item,minprice,maxprice,acc) .
    auctioneer?(yesorno,finalprice) . zero
};
```

In a first approach we could consider that a software system, specified by a π-calculus process, is deadlock-free when it terminates without requiring any interaction with its environment, i.e., when it always performs a finite number of silent actions `tau` leading to the inaction `zero`. However, most server components are not terminating, since they provide their services running on an infinite loop. Therefore, we must extend this definition in order to accommodate also infinite sequences of silent actions. Thus, we give a negative definition, saying that a π-calculus process *fails* when, considered as isolated from its environment, it may perform a finite sequence of silent transitions leading to a process which is not the inaction nor it can perform any transition by itself. Now, we can say that a process is *deadlock-free*, or that it *succeeds*, when it does not fail. Formal definitions of these notions of success and failure in the context of the π-calculus can be found in [Canal et al., 2001].

The distinction between global and local choices, to which we referred in section 3.1, plays a significant role in deadlock-freedom. Suppose that a certain component performs a local choice; then the rest of the system must be able to follow this decision, otherwise the system would deadlock. Thus, every local choice must be taken into account when analyzing absence of deadlocks. However, if a certain component presents a global choice, this choice will only take place if another component in the system presents the complementary action, that is, if there is a component willing to accept the choice. If not, the global decision will not occur. Hence, only global decisions which are common (in the form of complementary actions) to two or more system components must be taken into account for determining deadlock-freedom.

With all this, the absence of deadlocks in the application gets reduced to analysing the π-calculus process `Appl()` and proving that it is deadlock-free.

On the other hand, it is important noticing here that we are strongly relying on the application's internal structure; there is no problem with that for closed applications, but this is definitely a very strong assumption for open and evolvable applications.

5.1.2 *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 [Nierstrasz, 1995]. Substitutability and compatibility are the two flip sides of the *object interoperability* coin. This issue is not difficult to solve at the signature level, 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 (hereinafter called *outgoing operations*) when implementing the old object's methods 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 object with a new one.
- And second, the protocols (relative order among incoming and outgoing messages, and blocking conditions) 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 method names read (resp. written) by the object through all the channels it uses.

With regard to the 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 [Nierstrasz, 1995]. For this reason, we make use of a specific mechanism for behavioral subtyping of processes, which we have called protocol inheritance (less restrictive than bisimilarity) defined in [Canal et al., 2001], 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.

The detailed description of this relation of protocol inheritance is beyond the scope of this paper, we are here more interested in showing its applicability than its technical aspects in the π-calculus. However, we may roughly say that a certain protocol $\mathtt{P}_2$ *inherits* from another one $\mathtt{P}_1$ if (a) $\mathtt{P}_2$ preserves the semantics of behavior of $\mathtt{P}_1$ (i.e., if any global choice offered by $\mathtt{P}_1$ is also offered by $\mathtt{P}_2$); (b) $\mathtt{P}_2$ does not extend $\mathtt{P}_1$ (i.e., if any action present in $\mathtt{P}_2$ has been inherited from $\mathtt{P}_1$); and (c) $\mathtt{P}_2$ terminates when $\mathtt{P}_1$ does also terminate. If these conditions are fulfilled we can ensure that $\mathtt{P}_2$ may replace $\mathtt{P}_1$ in any context, that is, when combined with any other component. Since the just mentioned second condition seems to be too restrictive, we have also introduced a relation of protocol *extension*, derived from that of inheritance, that also allows the child protocol to extend its parent by adding new methods that do not interfere with the inherited behavior, therefore ensuring safe substitutability.

In order to illustrate the notion of protocol substitutability, let us consider again one of the objects in our example: the `Auctioneer`. In the first place, from its protocol description we can easily compute its incoming and outgoing operations: as incoming messages we find just the component's supported

operations, apart from the replies it gets when calling external methods; and as outgoing operations, the object calls methods `wannaBid()`, `youGotIt()`, and `itemSold()` in the corresponding `Bidder` objects.

Now, let us suppose that we have another `Auctioneer` object, whose behavior is specified by the protocol `FairAuctioneer`, defined as follows:

```
protocol FairAuctioneer extends Auctioneer {
  redefines EndOfRound(ref,bidders,item,price,winners,rep,acc) =
    [winners!=<> and w=head(winners) and tail(winners)=<>] %only one bid, the item is knocked down
      w!youGotIt(item,price,acc) . w?(addr) . rep!(TRUE,price) .
      NotifySale(ref,bidders,item)
  + [winners != <> and tail(winners)!=<>] % several bids received, new round starts
      Auctioning(ref,bidders,item,price++,rep,acc)
  + [winners = <>] % no bids, the auctioneer gives up
      rep!(FALSE,0) . NotifySale(ref,bidders,item)
  + ref?register(b,rep) . rep!() . EndOfRound(ref,bidders++<b>,item,price,winners,rep,acc)
  + ref?unregister(rep) . rep!() . EndOfRound(ref,bidders--<b>,item,price,winners,rep,acc) ;

  redefines NotifyBid(bidders,item,price,winners,done) =
    [bidders!=<> and b=head(bidders) and others=tail(bidders)]
      b!wannaBid(item,price) . b?(yesorno) .
      ( [yesorno = TRUE] NotifyBid(bidders,item,price,winners++<b>,done)
      + [else] NotifyBid(others,item,price,winners,done) )
  + [else] done!(winners).zero
};
```

We can see in this example that the incoming and outgoing operations of both protocols are the same, but that their behavior is not. The `FairAuctioneer` protocol extends the `Auctioneer` protocol, detailing some of its features and also improving it. The protocol describes now a normal auction (dutch auctions do not conform to it), in which (a) the price for the item raises in each round from the amount originally indicated by the seller; and (b) the auction is fair, i.e., the auctioneer only knocks down the item when exactly one bidder remains bidding for it (in the previous version she could decide to assign it to a certain bidder, even when several others were bidding against him.) Notice also that now bidders may also join or leave the system while an auction which is in progress (at the end of each round), not only in the interlude between auctions. In order to specify this new behavior, two processes have been redefined. First, we have substituted the `tau` actions in the `EndOfRound` process, representing local choices, by guards describing how these decisions are made. In addition, we have modified the process `NotifyBid` so it returns a list with all the bidders having a bid (winners list), instead of answering with only one of them (as it was the case in protocol `Auctioneer`).

Now, if we make use of our relation of protocol inheritance, it is not difficult to prove that the behavior described by protocol `FairAuctioneer` inherits from that of `Auctioneer`. Thus, we can replace an object exhibiting the former behavior by one compliant with the latter protocol without affecting the safety and liveness properties in the system.

5.2 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

Figure 1: Component interceptors

[Szyperski, 1998], 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. The Internet is probably the most well-known example of those systems. Unfortunately, in those situations the architecture of the applications is not made explicit anywhere, and therefore the static checks previously mentioned are no longer valid, since they cannot be used for dynamic attachments among unknown components.

Protocol compatibility can be checked at run time by intercepting messages and verifying their correctness with regard to the current state of the 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. Likewise, the system could prevent illegal or incompatible messages to reach destination components, avoiding incompatibility issues and further deadlocks.

In addition, those tests also provide an exception handling mechanism when integrating systems from components and applications that were not designed for *open* environments, and hence do not offer any support for dynamically handling incompatibility issues.

To implement this facilities in CORBA we have used a reflective facility that some ORB vendors provide: interceptors [OMG, 1999] (also called *filters*). This mechanism was originally defined and implemented in Orbix [Baker, 1997], 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 reflective facilities (such as Composition Filters [Aksit et al., 1993] or the Object Filters [Joshi et al., 1997]), 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.

In Figure 1 we have shown a schema of an interceptor for an object called "`Server`", following the implementation structure defined in Orbix for object interceptors. The interceptor in the figure defines two methods, `InRequestPostMarshal` and `OutReplyPreMarshal`, that capture the incoming and outgoing messages, respectively. Each message is stored in a structure of type `Request`, from which the method name and parameters are accessible. As we can see in the figure, upon reception of a message those filtering methods invoke the `ProtocolTracer`, asking whether the message is valid or not. The `ProtocolTracer` is the tool we have developed, that simply reproduces the run-time trace of a given object's protocol (which is determined when the interceptor is created and associated to the object), and decides whether the message is valid with regard to its current state. If so, the message is allowed to proceed and the `ProtocolTracer` updates its state, moving to the next action indicated in the protocol; if not, the message is rejected and the state is unchanged. Since `ProtocolTracer` just *traces* the protocol it is not burdensome; it introduces basically the performance penalties due to message interception.

Another issue that can be addressed with our approach is about conformance to specifications, i.e., how to check that a given implementation of an object conforms to a given specification of its behavior. In general, 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. There is one possible way to deal with this problem using the aforementioned interceptors. They were used for checking that incoming messages to an object were valid with regard to its current state. But they can also capture outgoing messages (using `outRequestPreMarshal` and `inReplyPostMarshal` pre-defined methods of Orbix interceptors), and therefore be used for checking that the object's behavior is valid with regard to the protocol it is supposed to implement. In this sense, the interceptors can be used to *enforce* behaviors, in a similar way to Minsky's Law Governed Architectures [Minsky and Leichter, 1995].

5.3 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; and (2) at run time, in which all the messages exchanged between the objects 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 may evolve over time and there is no explicit framework context that defines the relations and binds among them.

Another problem is about the computational complexity of the static checks. Static protocol checks do not need such a heavy machinery as theorem provers, but still they are (theoretically) untractable because they need to explore all branches of the expansion tree of the protocols. However, most of the component protocol descriptions in real applications are usually very simple, and therefore even the protocol tests with an a priori exponential complexity can be practically used in these situations.

On the other hand, run time compatibility can be done on the fly by the interceptors with even less 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. However, the performance penalties introduced by the filters that check protocols is not much, since they they only need to check that a given message is valid in certain state, but do not need to check the full protocol at all. Therefore, performance penalties are kept to a minimum.

6 Further issues

Apart from the issues discussed in this paper, there are still some other topics related to object interoperability which are worth mentioning here.

6.1 Mediation

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? Those extra components are usually called *adaptors* or *mediators*, and their automated construction right from the description of the interfaces of the original components is in general a difficult problem [Yellin and Strom, 1997].

6.2 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 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.

6.3 Role compatibility

We have previously discussed how to achieve compatibility checks among the constituent components of an application. However, we could also think of 'simpler' pairwise compatibility checks, just between two objects. Those tests are needed when we want to study the compatibility of two components in isolation, independently from the applications they will be part of (for instance, the compatibility between the `Bidder` and the `Account`). The problem is that objects may simultaneously interact with more than one other object, and protocols describe all interactions, and how they interleave. Therefore, 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 just 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. This allows pairwise compatibility checks, which are more efficient because the full protocols of all the application's components do not need to be considered, just the shared roles of the two components involved. A role can be obtained from the corresponding object protocol by restricting (projecting) it to a certain set of communication channels (those shared with the component connected to the role), while hiding the rest of channels. Each protocol description could be divided into different separate roles, each one describing pairwise object interactions. The extension of our current proposal to deal with roles, the sort of tests that can be carried out with them, and how they compare to full protocol descriptions is the subject of further research. Another interesting possibility which may be worth considering is the description of protocols in terms of roles only, instead of the full protocols. This would provide simpler descriptions and more efficient compatibility tests, which would greatly benefit our proposal. However, we may also lose some information, such as how the different (pairwise) roles interleave, which may be needed in some situations. Again, this is the subject of further research.

7 Related Work

The contributions we have presented in this paper fall into two main categories: the extension of object IDLs for dealing with some semantic aspects 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 extensions that usually cope for the semantic aspects of object interfaces and behavior. We will not cite here the proposals that try to deal with the behavioral semantics of components (the

interested reader can consult [Leavens and Sitaraman, 2000]), just the ones that cover the specification of the objects' service protocols.

In the first place, Doug Lea's PSL [Lea, 1995] 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 an object may need from other objects, neither it is supported by standard proving tools.

Protocol Specifications [Yellin and Strom, 1997] is a more general approach for describing object service protocols using finite state machines, that describe both the services offered and required by objects. It is based on a very simple notation and semantics that allow components to be easily checked for protocol compatibility. However, this approach does not support multi-party interactions (only contemplates two-party communications), and the simplicity that allows the easy checking also makes it too rigid and unexpressive for general usage in open and distributed environments.

The approach by Cho, McGregor and Krause [Cho et al., 1998] uses UML's OCL to specify pre and post conditions on the objects' methods, together with a simple finite state machine to describe message protocols. Similarly, Jun Han [Han, 2000] proposes an extension to IDLs that includes semantic information in terms of constraints and roles of interaction between components (but using no standard notation), that aims at improving the selection and usage processes in component-based software development. They are somehow similar approaches, although none of them is associated to any commercial component platform like CORBA or EJB, nor supported by standard proving tools.

Reuse Contracts [Steyaert et al., 1996] 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.

Bastide et al. [Bastide et al., 1999] use Petri nets to describe the behavior of CORBA objects, providing their full operational semantics, and supported by proving tools. This is a very powerful and expressive approach, very similar to ours, and that has been successfully used to detect inconsistencies in some CORBA services commercial implementations [Bastide and Sy, 2000]. The main difference between their work and ours in the notation used, and the implications that this carries along. For instance, the semantics of the behavior of operations and the interaction protocols are defined altogether in the Petri nets approach, without a clear separation of both semantic dimensions. This is something that can be avoided in the protocols defined using the π-calculus, that can only contain 'interaction' information. On the other hand, Bastide's proposal allow much richer information to be include in the objects' behavioral descriptions, which may be required in some cases.

Another interesting proposal is due to Büchi and Weck [Büchi and Weck, 1999], where a grey-box specification approach is used to specify the semantics of components in terms of pre-post conditions and invariants. This proposal successfully addresses many of the limitations that semantic specifications of components currently have. However, it mixes again protocol and semantic information altogether, it is devised only for static checks, and also implies very heavy compatibility proofs. One of the reasons for addressing component interoperability only at the protocol level was to try to avoid the complexity of the

tests that are needed when dealing with the whole semantic descriptions of components.

On the other hand, current model checkers (e.g. those based on Promela or SDL) also allow the specification of the observable behavior of objects. They are very good for cleaning up system designs quickly because of their efficiency (they work in polynomial time) and the quality and expressiveness of their feedback. However, a disadvantage of model checkers is their inability to deal with arbitrary number of instances, or with properties that involve universal quantifiers.

Message Sequence Charts (MSC) is also a notation that also permits the description of the interactions among objects. Now part of UML [Rumbaugh et al., 1999], they are very expressive for describing protocol interactions, but they do not allow the formal proof of (safety or liveness) properties about the system, or the inference of certain results.

Finally, some Architectural Description Languages (ADLs) also include the descriptions of the protocols that determine the access to the components they define using standard notations that derive from process algebras (like CSP, CCS or π-calculus). One of the benefits of using standard calculi is that reasoning about system behavior and correctness can be done using appropriate tools. ACME [Garlan et al., 1997], Wright [Allen and Garlan, 1997], or LEDA [Canal et al., 1999] are examples of ADLs that make use of process algebras such as the π-calculus for describing the behavior of the components of a system. Our focus is somehow different, since we are more concerned with the specification of COTS components, independently from the applications they will be part of.

With regard to the second topic, the use of formal notations in commercial environments, we share the thesis 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. Having methods to describe the behavior of such systems formally gives us the ability both to explore alternative designs and to validate chosen designs to ensure that they have the behavior which we expect.

The π-calculus has also been successfully used for describing some aspects of the internal architecture of component models like COM [Feijs, 1999] or CORBA [Gaspari and Zabattaro, 1999]. In this paper we have shown how it can also be used to describe some of the semantics of the dynamic behavior of the components, not only of the model's communications mechanisms.

The π-calculus also offers good tool support, with several products for animating π-calculus specifications such as MWB [Victor, 1994] or epi [Henderson, 1997]. Both are executable versions of the polyadic π-calculus, with some enhancements added, and MWB is the one we have used for conducting our tests. However, we did not want to commit 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 into different executable versions of polyadic π-calculus. The only requirements are the support for basic data types, and the simulation of the syntactic sugar extensions we have defined.

8 Concluding remarks

In this paper we have outlined the importance of incorporating protocol information into object interface descriptions. Our proposal extends traditional IDLs with 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 issues previously described, there are some natural extensions of our work. 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 and extended service traders.

References

[Aksit et al., 1993] Aksit, M. et al. (1993). Abstracting object interactions using composition filters. In *Proc. of ECOOP'93*, number 791 in LNCS, pages 152–184. Springer-Verlag.

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249.

[America, 1991] America, P. (1991). Designing an object-oriented programming language with behavioral subtyping. In der Bakker, J., de roever, W., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages, 1990 REX School/Workshop, Noordwijkerhout, The Netherlands*, number 489 in LNCS, pages 60–90. Springer-Verlag.

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

[Bastide and Sy, 2000] Bastide, R. and Sy, O. (2000). Towards components that plug AND play. In Vallecillo, A., Hernández, J., and Troya, J. M., editors, *Proc. of the ECOOP'2000 Workshop on Object Interoperability (WOI'00)*, pages 3–12.

[Bastide et al., 1999] Bastide, R., Sy, O., and Palanque, P. (1999). Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 474–494. Springer-Verlag.

[Büchi and Weck, 1999] Büchi, M. and Weck, W. (1999). The greybox approach: When black-box specifications hide too much. Technical Report 297, Turku Center for Computer Science. http://www.abo.fi/∼mbuechi/publications/TR297.html.

[Canal et al., 1999] Canal, C., Pimentel, E., and Troya, J. M. (1999). Specification and refinement of dynamic software architectures. In Donohoe, P., editor, *Software Architecture*, pages 107–125. Kluwer Academic Publishers.

[Canal et al., 2001] Canal, C., Pimentel, E., and Troya, J. M. (2001). Compatibility and inheritance in software architectures. *Science of Computer Programming*. (Accepted for publication).

[Cho et al., 1998] Cho, I., McGregor, J., and Krause, L. (1998). A protocol-based approach to specifying interoperability between objects. In *Proceedings of TOOLS'26*, pages 84–96. IEEE Press.

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

[Garlan et al., 1997] Garlan, D., Monroe, R., and Wile, D. (1997). ACME: An architectural interchange language. In *Proc. of the 19th IEEE International Conference on Software Engineering (ICSE'97)*, Boston.

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

[Han, 2000] Han, J. (2000). Temporal logic based specifications of component interaction protocols. In Vallecillo, A., Hernández, J., and Troya, J. M., editors, *Proc. of the ECOOP'2000 Workshop on Object Interoperability (WOI'00)*, pages 43–52.

[Heiler, 1995] Heiler, S. (1995). Semantic interoperability. *ACM Computing Surveys*, 27(2):265–267.

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

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

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

[Leavens and Sitaraman, 2000] Leavens, G. T. and Sitaraman, M., editors (2000). *Foundations of Component-Based Systems*. Cambridge University Press.

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

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

[Minsky and Leichter, 1995] Minsky, N. and Leichter, J. (1995). Law-governed Linda as a coordination model. In Ciancarini, P., Nierstrasz, O., and Yonezawa, A., 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.

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

[Ólafsson and Bryan, 1996] Ólafsson, A. and Bryan, D. (1996). On the need for "required interfaces" of components. In *Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96*, pages 159–165. Dpunkt Verlag.

[OMG, 1999] OMG (1999). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.3 edition. http://www.omg.org.

[OMG, 2000] OMG (2000). *The CORBA Event Service Specification*. Object Management Group. http://www.omg.org.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling language Reference Manual*. Addison-Wesley.

[Steyaert et al., 1996] Steyaert, P., Lucas, C., Mens, K., and d'Hondt, K. (1996). Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.

[Vallecillo et al., 2000] Vallecillo, A., Hernández, J., and Troya, J. M. (2000). New issues in object interoperability. In *Object-Oriented Technology: ECOOP'2000 Workshop Reader*, number 1964 in LNCS. Springer-Verlag.

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

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

[Yellin and Strom, 1997] Yellin, D. M. and Strom, R. E. (1997). Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333.