Adding Roles to CORBA Objects

Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José M. Troya, and Antonio Vallecillo, *Member*, *IEEE*

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. Some of the existing proposals try to add protocol information to object interfaces, but most of them fail to do so in a modular way. 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, based on the concept of *roles*. Roles allow the modular specification of the observable behavior of CORBA objects, reducing the complexity of the compatibility tests. Our main aim is the automated checking of protocol interoperability between CORBA objects in open component-based environments, using similar techniques to those used in software architecture description and analysis. In addition, our proposal permits the study of substitutability between CORBA objects, as well as the realization of dynamic compatibility tests during their runtime execution.

Index Terms—Interface definition languages, software components, component-based software development, protocols, compatibility and substitutability of components.

1 Introduction

Component-Based Software Development (CBSD) is gaining recognition as a key enabling technology for the construction of high-quality, evolvable, large software systems, developed in timely and affordable manners. CBSD advocates the development and usage of plug-and-play reusable software, which has led to the concept of "*commercial off-the-shelf*" (COTS) components, and to the emergence of a growing market of software components.

The idea of COTS components is highly attractive, offering the possibility of producing cheaper and higher quality software products, and helping develop complex and distributed applications deployed on a wide range of component platforms (CPs) that support the coordination of components, like J2EE or CORBA. However, neither CPs, nor any other technology for deploying standalone components, currently define any methodology for the development of component-based software in open distributed environments. As a consequence, established software development processes have to be reviewed according to the new requirements imposed by the existence of a component market advertising COTS components ready to be inserted inside the architecture of applications [40]. Here, the outstanding question is how to describe the software architecture of distributed applications around COTS components. Basically, there are two major alternatives for addressing COTS-based software development: 1) define the architecture first and then proceed to reuse components, or 2) search for COTS components first and then adapt and glue them together.

In the first approach, application designers may draw the system architecture according to different kinds of requirements to produce a preliminary design, and then search in a component repository for those that fit the abstract components. Software Architecture (SA) is a discipline that focuses on the design of the system architecture, represented as a high-level abstraction of the structure of an application in terms of a set of abstract components and their relationships. Architectural Description Languages (ADLs)—such as Rapide [23], Wright [2], Darwin [24], Unicon [35], C2 [25], or LEDA [9]—have been defined for providing accurate and unambiguous description of software architectures, defining the abstract component interfaces and the way they are interconnected. In addition, ADLs allow checking that the resulting composition is error-free and even proving some properties about the application right from its description, such as consistency and user requirements checking [12], [14]. Beyond mere academic exercises and prototypes, ADLs have also been extensively used for the specification and validation of existing commercial systems [3], [14], [36].

The main benefit of this approach is that the resulting system will have a custom-made architecture, organized around the user requirements and without redundant components. However, reusing COTS components or any legacy code inside an established architectural frame can be a difficult task. Even if the suitable functionality is provided by a COTS component, the cost of adapting it in order to match the architectural constraints and the user requirements may be too high, if possible at all. Thus, we may be forced to write the components from scratch, losing the benefits of component reusability that we were pursuing.

An alternative approach starts by designing the architecture of the system from third-party components available in software repositories, using a bottom-up approach. Instead of specifying abstract component interfaces a priori, they are obtained after analyzing the software components offered in the component marketplace. 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. This results in a more realistic approach, more adapted to achieve reuse in a component mass market,

• *The authors are with the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, ETSI Informática, 29071 Málaga, Spain. E-mail: {canal, lff, ernesto, troya, av}@lcc.uma.es.*

Manuscript received 22 Dec. 2000; revised 10 Oct. 2001; accepted 26 Aug. 2002.
Recommended for acceptance by E. Clarke.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 113354.

which moves organizations from application *development* to application *assembly*. The ultimate goal, once again, is to be able to reduce developing 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.

Following this second approach, the aim of this work is to enrich COTS components interfaces with useful information about their behavior, in order to facilitate the search and communication processes between unknown components inside an open environment. In this sense, our present work proposes an extension to the CORBA IDL that allows the description of certain dynamic behavior of CORBA objects or components[1] in addition to the structural 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 in which 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. Although our proposal could be applied to any theoretical or academic model, we have chosen CORBA, one of the leading commercial object platforms.

The notation used to enrich CORBA IDLs is based on a formalism since we need to be able to check whether a given CORBA off-the-shelf component can successfully replace another in a particular application (component substitutability), or whether the behavior of two components is compatible and, thus, they are able to interoperate (component compatibility). For that, we will use similar ideas and techniques to the ones used by software architects in their ADLs, showing how they can be successfully applied in CORBA environments.

Our proposal is focused on improving IDLs with component behavior and context dependencies. IDLs of most current well-known platforms (such as CORBA or EJB) provide poor documentation, restricted to the signature of the services offered. All parties are now starting to recognize that this kind of (*signature*) interoperability is not sufficient for enabling component search and selection, and for ensuring the correct development of component-based applications in open systems [42]. Current approaches to overcome this limitation try to add *semantic* information to interfaces, using different notations (pre/post conditions, temporal logic, Petri nets, refinement calculus, etc.), and are also concerned about compatibility and substitutability of components (see [22] for a comprehensive survey on these proposals).

Apart from these signature and semantic levels, another possibility is to concentrate just on the interactions among the components, defining their service access protocols, and the way each component uses the services provided by others. This approach provides more than just signature information, and allows the definition of compatibility and substitutability checks among components—and at a lower computational cost than other semantic tests. Some authors have dealt with component interoperability at this "protocol" level [6], [10], [21], [41], and have shown its benefits. Notations such as Petri nets, process algebras or state machines have been used for describing object service access protocols and for reasoning about them. However, the existing approaches for documenting components at this level present some limitations:

a. The description of the observable behavior of components is not modular: Each component is assigned a single protocol description, which defines all of its interactions with the rest of the system. This mixes up all interactions, and usually forces the introduction of irrelevant details into the protocol specification, e.g., the interleaving among unrelated interactions.
b. The computational complexity of most tests is still very high, due in part to the need of checking "full" protocols (i.e., those describing all the component's interactions with the rest of the components in the system). However, typical pairwise component interactions are very simple and this should reflect in simpler compatibility tests.

In this paper, we use the notion of *role* for describing the dynamic behavior of components, in the same sense as it is used in ADLs. Roles are partial protocol specifications, in which we only pay attention to the behavioral interface that a component presents to another one. This allows modular specification of component behavior and permits more efficient pairwise compatibility checks: the full protocols of all of the application's components need not be considered at the same time, just the shared roles of the two components involved.

This paper is a continuation of our previous work [10], improving it in two main ways. First, it allows the declaration of "provided" and "required" component interfaces, in the style of the new CORBA Component Model (CCM [30])—allowing also an easier integration with CORBA and COM IDLs. And second, protocols are described in a modular way using the aforementioned roles. We have been very careful when defining those extensions, so backwards compatibility with our previous work is kept and therefore its benefits are maintained.

Roles are described using a sugared subset of the polyadic π-calculus and are defined separately from 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 runtime also allows the definition of dynamic compatibility checks in open and extensible applications. Thus, we will be able to reason about component compatibility and substitutability and to infer some safety properties of applications directly from the description of the behavior of their parts (e.g., that no messages are lost, or the absence of deadlocks).

The structure of this paper is as follows: After this introduction, Section 2 briefly describes the CORBA IDL and discusses its current limitations. Section 3 describes our proposal, showing how to enrich interfaces with protocol information. An example is introduced in Section 4, which will be used throughout the paper to illustrate our approach. Section 5 presents the sort of checks that can be carried out with our proposal, both at design and during

1. In this CORBA context, we will not distinguish between objects and components, so we refer to objects with the term component—as binary units of possibly independent production, acquisition, and deployment.

runtime. Sections 6 and 7 are devoted to discuss the sort of information to be included in role descriptions and some other open issues that need further investigation. Section 8 compares our work with other similar approaches. Finally, Section 9 draws some conclusions and describes some future research activities.

2 Object Interfaces and IDLs

Traditional object interfaces provide the description of an object functionality and capabilities, in terms of its attributes and the signature of the operations it offers. *Interface Description Languages* (IDLs) have been defined for specifying those object interfaces at the signature level. Apart from providing a textual description of the object 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 runtime. 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.

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 [31].

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 in order 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 a 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.

As an example, let us describe a simple CORBA object given by the following interface:

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

As we can see, a bank account offers three methods that allow the user to know the current balance, to deposit, or to withdraw some money from the account. Method `withdraw()` can also raise an exception in case we want to withdraw more money than currently available in the account.

2.2 Limitations of IDLs

Traditional IDLs were originally defined for closed client-server applications and, therefore, they present some problems when being used in open component-based applications.

In the first place, IDLs describe the services that objects offer, but not the services that they require from other objects in order to accomplish their tasks [29]. At most, in some CORBA applications, the IDL modules contain not only the servers' interfaces, but also the definition of "callback" interfaces that need to be satisfied by clients (e.g., see the CORBA Event service specification [32]). 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.

The second limitation is about IDLs providing just the *structural* 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 [41].

And finally, we think that the use of IDL descriptions during runtime 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 [37].

3 Extending CORBA Interfaces with Protocols

In this section, we will concentrate on how to add protocol information to the description of CORBA object interfaces. Let's start by defining the concepts of protocol and role that will be used here.

CORBA objects interact by exchanging messages in a client-server manner. A given component may act as a server (for its clients), as a client (when invoking other objects' methods), or usually as both. The set of "acquaintances" of a component refers to the set of components that may interact (i.e., exchange messages) with it during its lifetime. Please note that these components can interact

because either they implement some of the interfaces required by the component or because they call some of the component's provided interfaces.

With this, a *protocol* is "a specification of the set of legal sequences of messages that are exchanged between a component and its acquaintances during their interactions." Interfaces serve for describing the services implemented by server objects at the signature level. Protocols describe the valid flow of information (i.e., messages) between interacting components [38], [41].

Following the approach used in most ADLs for describing the behavior of components and connectors, protocols will be described in terms of a set of *roles*. This allows the modular specification of complex protocols. Roles describe the expected local behavior of each of the interacting parties. That is, they provide the specification that determines the obligations of each component participating in an interaction [2].

A protocol may consist of one or more role descriptions. Binary (or dyadic) protocols, involving just two participants, are very common and also the simplest to specify. One advantage of these protocols is that only one role needs to be specified in each component. Just one role may also be used for specifying more complex protocols involving several participants in the case that all interactions (and their corresponding interleaving traces) need to be considered. However, there are situations in which protocols are broken down into a set of roles, each one describing separate interactions between unrelated components. In this sense, roles can be seen as "projections" of a component protocol over subsets of the component acquaintances, which partition the set of all the protocol traces. Here, each role models the behavior of its component from the point of view of its counterpart components in a given interaction, hiding both the internal details of the component and its interactions with the rest of its acquaintances.

The use of roles provides both modularity and abstraction to the specification and analysis of complex behaviors. However, in some situations, we may also lose expressiveness since the information on how the different roles interleave may be lost. Section 6 contains a more thorough discussion on this important issue.

Roles are specified using a process algebra to model traces of communication events exchanged among components participating in an interaction. Specifically, roles are 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 proven 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, in which the possible dynamic changes have to be decided statically (i.e., a priori). For instance, it is difficult to express in CCS or CSP a common mechanism used in OO applications: object factories. They allow the creation of new objects servers that dynamically join the application and whose reference is created and then passed to the clients to deal with their requests. However, this is something that can be modeled in the π-calculus in a natural way since it is just a matter of creating a fresh channel name for the new object, spawning the π-calculus process that determines the behavior of the new object and passing the newly created name (i.e., the object reference) to the clients [8].

In our approach, we can also define interaction protocols between components that make use of operation parameters and return values when considering alternatives, which is an important issue concerning the expressiveness of protocol specifications. This represents 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.

3.1 The Polyadic π-Calculus

The π-calculus was originally proposed by Milner, Parrow, and Walker in 1992 [27]. 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 [26] 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` by `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:

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

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, 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

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

behaves as any process $\mathtt{P}_i$ for which guard $[\mathtt{G}_i]$ is true, or as $\mathtt{P}_0$ if all guards $\mathtt{G}_j$ $(1 \leq j \leq n)$ 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 modeling approach we propose and that we shall put into action in the next sections is the following:

- Each object interface will be handled using a π-calculus channel, that will be used for sending and receiving method calls. In particular, each reference to a CORBA object will be modeled by a π-calculus channel.
- Together with every method request, the calling object should include a channel name through which the called object will send the results. Although channels are bidirectional in the π-calculus, in this way 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 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 is modeled by recursive equations, where the various parts of the object state (i.e., the state variables we want to make visible to exhibit the object behavior) are parameters.

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

- Method invocation `ref!(m,(args),(reply))` can be abbreviated to `ref!m(args,reply)`, or even to `ref!m(args)` if the reference and reply channels are the same.
- Similarly, we can write `ref?m(args,reply)` on the server side for accepting the invocation of method `m(args)`, instead of writing

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

Component protocols are defined using the construct "`protocol`," that consists of a name, followed by three main sections enclosed between curly brackets. First, we find the set of *provided* interfaces that the component supports (implements). Each provided interface is indicated in a `#provides` clause. Second, we have the set of the external interfaces that the component *uses* when implementing its services, each one indicated in a `#uses` clause. Finally, there is the description in textual π-calculus of each role the component plays in its interactions with other components. Each role is indicated by a `#role` clause and described as one or more π-calculus processes. As we can see, the first two sections contain information at the signature level only, while the last one is in charge of describing the observable behavior of the component at the protocol level. Fig. 1 describes the precise grammar used to express protocols.

It is worth noting that every CORBA object provides just one interface [31]. Nevertheless, we have preferred to define our interfaces in a more general way, to be able to cope with other component models that allow components to provide more than one interface (such as COM, or the CORBA Component Model CCM [30]).

Using this syntax, a possible behavioral description of the `Account` interface of our previous example can be written as follows:

```
protocol AccountBehav {
 #provides Account
 #role Accounting(Account ref, float balance)
   = ref?getBalance(rep) . rep!(balance) .
       Accounting(ref,balance)
   + ref?deposit(amnt,rep) . rep!() .
       Accounting(ref,balance+amnt)
   + ref?withdraw(amnt,rep,notEnoughMoney) .
```

$$
\begin{array}{lll}
\mathit{Protocol} & ::= & \texttt{protocol}\ \mathit{id}\ \{\ \mathit{Provides}^{+}\quad \mathit{Uses}^{*}\quad \mathit{Roles}^{+}\} \\
\mathit{Provides} & ::= & \texttt{\#provides}\ \mathit{id} \\
\mathit{Uses} & ::= & \texttt{\#uses}\ \mathit{id} \\
\mathit{Roles} & ::= & \texttt{\#role}\ \mathit{id}\ \texttt{(}\mathit{argsDeclaration}\texttt{)} = \mathit{Process}\ \mathit{AuxProcess}^{*} \\
\mathit{AuxProcess} & ::= & \texttt{;}\ \mathit{id}\ \texttt{(}\mathit{argsDeclaration}\texttt{)} = \mathit{Process} \\
\mathit{Process} & ::= & \mathit{Process}\ \texttt{|}\ \mathit{Process} \\
 & | & \mathit{Process}\ \texttt{+}\ \mathit{Process} \\
 & | & \texttt{[}\mathit{boolExpr}\texttt{]}\,\mathit{Process}\quad \{\texttt{+[}\mathit{boolExpr}\texttt{]}\,\mathit{Process}\}^{*}\quad [\texttt{+[else]}\,\mathit{Process}] \\
 & | & \mathit{id}\ (\mathit{args}) \\
 & | & \texttt{(\^{}}\mathit{id}\texttt{)}\,\mathit{Process} \\
 & | & \alpha\ \texttt{.}\ \mathit{Process} \\
 & | & \texttt{zero} \\
\alpha & ::= & \texttt{tau} \\
 & | & \mathit{id}\texttt{?(}[\mathit{id},]([\mathit{args}])[,([\mathit{args}])]\texttt{)} \\
 & | & \mathit{id}\texttt{!(}[\mathit{id},]([\mathit{args}])[,([\mathit{args}])]\texttt{)} \\
 & | & \mathit{id}\texttt{?}[\mathit{id}]([\mathit{args}]) \\
 & | & \mathit{id}\texttt{!}[\mathit{id}]([\mathit{args}]) \\
\mathit{args} & ::= & \mathit{id}\{,\mathit{id}\}^{*} \\
\mathit{argsDeclaration} & ::= & \mathit{id}\ \mathit{id}\{,\mathit{id}\ \mathit{id}\}^{*}
\end{array}
$$

Fig. 1. Grammar for describing protocols.

```
      ( tau . % normal response
        rep!() . Accounting(ref,balance-amnt)
      + tau . % exception raising
        notEnoughMoney!(balance) .
        Accounting(ref,balance)   )
};
```

This protocol describes the observable behavior of a component that provides the services defined by interface `Account`, as stated in the clause `#provides`. It requires no external services, so no `#uses` declarations appear in the protocol description.

The observable behavior of an `Account` object is defined by just one role, named "`Accounting`," specified by one π-calculus process with two (typed) arguments. The first one (`ref`) is the object reference channel used for handling the methods specified in the CORBA interface `Account`, while the second argument (`balance`) is a number used to store the internal state of the object.

Although the π-calculus is a typeless calculus, in our approach process parameters are declared using CORBA types. This is mainly needed to identify the channel names associated to every interface that a role handles and to differentiate them from the state variables. Therefore, neither type compatibility nor type/subtype checking are considered here since parameter types are used for identification purposes only.

Apart from typed parameters, processes describing roles are standard π-calculus processes. In this case, role `Accounting` is recursively defined, and starts by reading from its reference channel, waiting for a tuple with the invocation of one of its implemented services. Once a service call is received, it is answered (or an exception is raised) and the process goes back to its original state.

As we have said, the process `Accounting` above uses its second argument to store the internal state of the object. Thus, we are specifying not only protocol information, but also some of the object's behavioral semantics, i.e., how the three methods modify the balance of the account. On the contrary, we decided not to specify when the object decides to raise the exception (e.g., maybe the account could allow some credit) and wrote it as an internal (local) choice. Although all of this semantic information is ignored during protocol analysis, the specifier has the option to state the details of the object's behavior and semantics that are relevant to its potential readers, while hiding those which (s)he wants to leave open to possible implementations.

Roles allow the separated specification of the different (unrelated) interactions in which a component is engaged in during its lifetime. By using roles, we can keep the specification free of any unnecessary details, apart from those specific to the interactions among the application components. Although roles usually consist of pairwise interactions between just two components, there are situations in which we need to specify multiparty interactions within the same role description. Our notation is flexible enough to cover from the simplest case of dyadic interactions, to full protocol descriptions, as will be discussed later on in Section 6.

3.3 Where do Protocols Live?

Analogous to where object IDLs live, each protocol resides inside a text file (with extension "`.ptl`"). As previously mentioned, each `protocol` description provides the specification of the observable behavior of one component, describing how it handles its supported and required interfaces (only one supported `interface` in the particular case of CORBA).

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, in a similar way as it is done by ADLs. Furthermore, this information is used at runtime by a tool we have developed for implementing the dynamic tests. This tool—named `ProtocolTracer`—is a π-calculus interpreter that acts as a filter that can be attached to any CORBA object, intercepting its input and output messages and verifying their correctness with respect to the protocol declared for that object. This tool, as well as the dynamic tests that it allows, will be explained in Section 5.3 with more detail.

4 An Example

In order to illustrate our proposal, we will use a real example, namely, the CORBA Transaction Service [33], one of the several CORBA Services currently defined and standardized. Our intention is to use roles for formally documenting component interactions and service access protocols, so ambiguities and under-specifications can be avoided. Moreover, once we have the description of the roles of the CORBA objects that comprise the service, we will also be able to perform compatibility and substitutability tests that can help checking whether a given object uses the Transaction Service in a valid way or not, or whether a given implementation of one of the objects in the Transaction Service conforms to its specified behavior or not.

4.1 The CORBA Transaction Service

The Transaction Service (TS) provides transaction synchronization across the elements of a distributed client-server application. A transaction can involve multiple objects performing multiple requests. In a typical scenario, a client first begins a transaction by issuing a request to a `Current` object defined by the TS which establishes a transaction context. The client then issues requests to other objects (called transactional objects), which may be associated to the client's transaction context. Eventually, the client decides to end the transaction by issuing a request to the `Current` object. If there were no failures, the changes produced as a consequence of the client's requests would then be committed and are made permanent; otherwise, the changes would be rolled back.

The TS supports several kinds of scenarios, depending on the client's requirements. For instance, we could allow the client to directly control the propagation of the transaction context; or require that all requests be performed within the scope of a transaction. We could also allow both flat and nested transactions, or even deal with *heuristic* exceptions, due to unilateral decisions made by one or more participants. For space reasons, in this paper, we will concentrate just in the core functionality of the TS and simplify the number of objects and its interfaces. For a complete description of the rest of the TS interfaces, see [33].

The subset of the TS we are going to deal with defines two CORBA interfaces, `Current` and `Resource`:

```
//Subset of the Transaction Service V1.1
// May 2000 A-1
#include <Corba.idl>
module CosTransactions {
  // DATATYPES
  enum Status
   {StatusActive,StatusPrepared,...};
  enum Vote
   {VoteCommit,VoteRollback,VoteReadOnly};
 // Transaction-specific exceptions
 exception SubtransactionsUnavailable {};
 exception NotPrepared {};
 exception NoTransaction {};
 // INTERFACES
 interface Current : CORBA::Current {
  void begin()
    raises(SubtransactionsUnavailable);
  void commit()       raises(NoTransaction);
  void rollback()     raises(NoTransaction);
  void rollback_only() raises(NoTransaction);
  Status get_status();
 };
 interface Resource {
  Vote prepare();
  void rollback();
  void commit()   raises(NotPrepared);
 };
}; // End of CosTransactions Module
```

The `Current` interface defines operations that allow any client of the TS to begin and end transactions, and to obtain information about the current transaction. It may be shared with other object services (e.g., security) and is obtained by using a "`resolve_initial_references ("TransactionCurrent")`" operation on the `CORBA::ORB` interface.

A typical *transaction originator* (e.g., a bank account client) uses the `Current` pseudo object to begin a transaction, which becomes associated with the transaction originator's thread. The transaction originator then issues requests. Some of these requests involve *transactional objects* (e.g., bank accounts). Using the `Current` pseudo object, the transactional object can inquire about the current state of the transaction and can also unilaterally rollback the transaction.

Some transactional objects are *recoverable* objects. A recoverable object has persistent data that must be managed as part of the transaction and is considered as a `Resource` object in the transaction. The resource represents the recoverable object's participation in the transaction; each resource is implicitly associated with a single transaction.

After the computations involved in the transaction have been completed, the transaction originator (i.e., the client) uses the `Current` object to request that the changes be committed. The TS commits the transaction using a *two-phase commit protocol* (2PC) wherein a series of requests are issued to the registered resources. The operations that support the 2PC protocol are the ones described in the `Resource` interface.

In our simplified scenario, we will only have transaction originators (bank clients), `Current` pseudo-objects for managing transactions and the transaction context, and recoverable objects that act as resources and bank accounts,

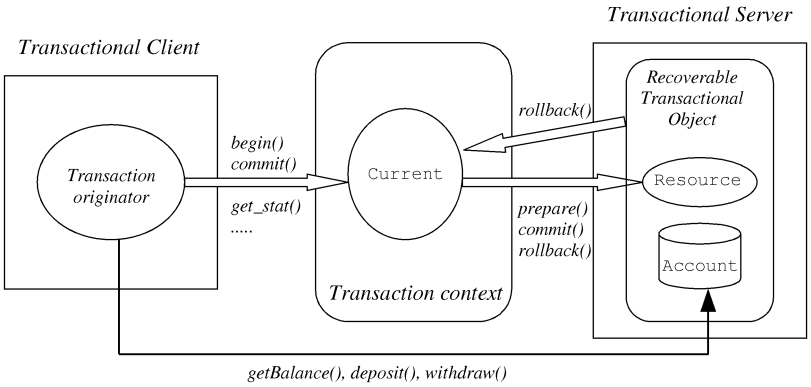

Fig. 2. Example of transactional application including its basic elements.

see Fig. 2. `Current` objects will also take the role of the Coordinator objects defined in the CORBA TS, which have been omitted here for simplicity. Thus, a transaction originator will make use of the `Current` interface to initiate a transaction, while a "transactional" bank account (i.e., a bank account able to act as a recoverable transactional object in a transaction) needs to implement both `Account` and `Resource` interfaces:

```
interface TransactionalAccount
 : Account, Resource {};
```

The `Resource` interface defines the operations invoked by the TS on each resource. Note that, in the case of failure, the completion sequence will continue after the failure is repaired. Thus, a resource should be prepared to receive duplicate requests for the commit or rollback operation and to respond consistently.

The `prepare()` operation is invoked to begin the two-phase commit protocol on the resource. The resource can respond in several ways, represented by the `Vote` result. If no persistent data associated with the resource has been modified by the transaction, the resource can return `VoteReadOnly`. After receiving this response, the TS is not required to perform any additional operations on this resource and the resource can forget all knowledge of the transaction. If the resource is able to write (or has already written) all of the data needed to commit the transaction to stable storage, it can return `VoteCommit`. After receiving this response, the TS is required to eventually perform either the commit or the rollback operation on this object. The resource can return `VoteRollback` under any circumstances, including not having any knowledge about the transaction (which might happen after a crash). If this response is returned, the transaction must be rolled back, the TS is not required to perform any additional operations on this resource, and the resource can forget all knowledge of the transaction. Resources, as transactional objects, can also use the `Current` interface to obtain information about the transaction, or to commit or roll back a transaction (as it happens, for instance, when an object detects a fault and decides to abort the transaction).

4.2 Specifying Transactions

Let us see how the protocol followed by the CORBA objects in the TS can be specified in terms of roles. We shall start with the `Current` interface, for which a possible behavior can be written as follows:

```
protocol CurrentBehav
 #provides Current
 #uses Resource
 #role WithAClient(Current ref) =
   ref?begin(rep, SubtransUnavailable) .
   rep!() . TransactionsStarted(ref)
 #role TransactionsStarted(Current ref) =
   ref?commit(rep, NoTransaction) .
   ( tau . rep!() . zero            % success
   + tau . NoTransaction!() . zero % exception
   )
 + ref?rollback(rep, NoTransaction) .
   ( tau . rep!() . zero
   + tau . NoTransaction!() . zero )
 + ref?rollback_only(rep, NoTransaction) .
   ( tau . rep!() . zero
   + tau . NoTransaction!() . zero )
 + ref?get_status(rep) .
   (^status) rep!(status).
   TransactionsStarted(ref)
 + ref?begin(rep,SubtransUnav) .
   SubtransUnav!() .
   TransactionsStarted(ref)

 #role CoordinatingAResource(Resourcersc) =
   (^rep) rsc!prepare(rep) . rep?(vote) .
   ( [vote=VoteCommit] (^rep)
     ( tau . rsc!rollback(rep) .
       rep?() . zero
    + tau .rsc!commit(rep, NotPrepared).
       ( rep?().zero + NotPrepared?().zero )
    + rsc?rollback(r) . r!() . zero
    )
    + % vote=VoteReadOnly or vote=VoteRollBack
     [else] zero )
};
```

A `Current` object may have interactions with three different kinds of objects (clients, transactional objects, and resources) and, therefore, defines three different roles. Clients should begin the transaction and then they may behave as other transactional objects, that may interrogate

about the state of the transaction, or terminate it. Note how a session identifier is reused for specifying the interaction protocol with transactional objects, a subset of the interaction protocol with a transaction originator.

The behavior associated to a `Resource` can be described as follows:

```
protocol ResourceBehav {
 #provides Resource
 #uses Current
 #role BeingCoordinated(Resource ref)=
    ref?prepare(rep) .
     ( tau . rep!(VoteReadOnly) . zero
     + tau . rep!(VoteRollBack) . zero
     + tau . rep!(VoteCommit) . Ready(ref) )
  + ref?rollback(rep) . rep!() . zero
  + ref?commit(rep,NotPrepared) .
     NotPrepared!() . zero
  ;
  Ready(Resource ref) =
    ref?rollback(rep) . rep!() . zero
  + ref?commit(rep,NotPrepared) .
    ( tau.rep!().zero
    + tau.NotPrepared!().zero
    )
 #role withCurrent(Current cur) =
  (^rep,SubtransUnavailable)
  cur!begin(rep,SubtransUnavailable) .
  ( rep?() . % start transactions
    TransactionsStarted(cur)
  + SubtransactionsUnavailable?() . zero
  ) ;
  TransactionsStarted(Current cur) = tau .
  (^r,NoTrans)
  ( tau . cur!commit(r,NoTrans) .
     ( r?().zero + NoTrans?().zero )
  + tau . cur!rollback(^r,NoTrans) .
    ( r?().zero + NoTrans?().zero )
  + tau . cur!rollback_only(^r,NoTrans) .
    ( r?().zero + NoTrans?(). zero )
  + tau . (^r) cur!get_status(r) . r?(status) .
   TransactionsStarted(cur) )
  };
```

Please note how resources are forced by the TS specification to always be able to accept requests for `commit` or `rollback` operations and to respond to them consistently. Therefore, those two operations have been included as global choices in all interactions in the role of the resource.

On the other hand, in our context a `Resource` may behave as a transactional object and, thus, it may interrogate the `Current` object about the state of the transaction, or terminate the transaction if a problem is detected. This is specified by role `withCurrent`, which shows how the resource may use the `Current` object services.

5 Checking Protocols

The approach presented here allows the clear description of the interactions among the objects participating in a system and, thus, it helps providing an enhanced documentation for CORBA objects and applications. However, the more complex these interactions are, the more difficult is to reason about them and to prove the correctness of the composed application. Once we have enriched CORBA object interfaces 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 those purposes.

As mentioned at the beginning of this paper, we are especially interested in object compatibility and substitutability checks, based on the information now available. 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).

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 over time. In those cases, we will see that it is also possible to define protocol compatibility checks, and how to carry them out.

Components are described by their interfaces and, therefore, all tests will be based on the information they contain. The first thing to do is to define what will constitute a component interface in our environment. In this setting, the *interface* of a component C will consist of a protocol description of its behavior and, hence, it will be determined by three sets (P, U, R), where the first one contains the set of *provided* interfaces of the component $P = \{P_1, \ldots, P_p\}$, the second one contains the set of *required* (or *used*) interfaces $U = \{U_1, \ldots, U_u\}$, and R is the set of *roles* defined in the component's protocol description, $R = \{R_1, \ldots, R_r\}$.

For simplicity, we will write $C.P$, $C.U$, and $C.R$ to refer to these three sets. In our case, the elements of both $C.P$ and $C.U$ are CORBA IDL descriptions, containing just signature information about the services provided and required by the component being specified. The elements of set $C.R$ are the role descriptions we have introduced in this paper, defined as π-calculus processes.

Basically, component compatibility and substitutability tests will be defined at these two levels—signatures and protocols—given some definitions of compatibility and substitutability at those levels. Although the final checks are defined as a conjunction, we can think of signature tests as a "filter" that eliminates the obvious mismatches before trying the more expensive protocol tests, as suggested for instance in [16], [42].

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 specifications of the constituent components and the internal structure of the application. Here, we will discuss two possible protocol checks: compatibility and substitutability among objects. 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 the checks.

5.1.1 Component Compatibility

Component compatibility can be described as the ability of two components to work properly together if connected, i.e., that all exchanged messages between them are understood by each other and that their communication is deadlock-free [41].

At the signature level, compatibility between components A and B is a matter of checking that all the methods requested by A to B are implemented by B, and vice-versa. The sets of provided and used methods described in each component interface can be used for those purposes.

At the protocol level, the π-calculus is a formal notation very appropriate for checking component compatibility, since it allows to know whether two protocols are *compatible* by just checking than their parallel composition is *successful* (i.e., deadlock-free). This is the test we will use for checking compatibility between software components.

Since component behavior is defined in terms of set of roles, we need to establish the roles that two components share in their interactions in order to be able to check their compatibility. Therefore, we need to introduce "bindings" between components. Given components A and B, we define $BINDINGS(A.R, B.R)$ as the set of tuples of roles (α, β), with $\alpha \in A.R$ and $\beta \in B.R$, that specify the interactions between these components.

Each binding is represented by the π-calculus process obtained by composing in parallel the roles involved in the binding. At a first glance, we could consider that a binding is "successful" when this process 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 [10].

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 of this in mind, the absence of deadlocks between two roles, R_1 and R_2, gets reduced to analyzing the π-calculus process obtained by composing them in parallel $(R_1|R_2)$, and proving that the composed process is successful according to our previous definition. In this case, we shall say that roles R_1 and R_2 are *compatible* ($R_1 \bowtie R_2$). A set of sufficient conditions for helping prove the compatibility between roles can be found in [10]. Finally, we shall say that components A and B are *compatible* over a set of bindings $\mathcal{B} = BINDINGS(A.R, B.R)$ if and only if for every pair of roles $(\alpha, \beta) \in \mathcal{B}$ we have that $\alpha \bowtie \beta$.

5.2 Performing the Tests

The π-calculus offers good tool support, with several products for animating π-calculus specifications such as MWB [39] or epi [18]. 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. In addition, the π-calculus is more expressive than other process algebras such as CCS or CSP since it allows the description of dynamic relations between components and, therefore, more appropriate for modeling open systems applications, as we have mentioned before.

In order to illustrate how static checks are performed, we can test for instance that objects `Current` and `Resource` are compatible. To study their compatibility, we should first identify the bindings they use for interacting. As explained in Section 4.2, these two objects use each other's services (see the two arrows that join them in Fig. 2) via the following set of bindings $\mathcal{B}$:

```
{ ( ResourceBehav::BeingCoordinated ,
    CurrentBehav::CoordinatingAResource ),
  ( ResourceBehav::withCurrent ,
    CurrentBehav::withAClient ) }
```

Therefore, in order to prove their compatibility, we need just to check with the aid of the aforementioned tools that the following two π-calculus processes are successful:

```
(^res)( ResourceBehav::BeingCoordinated(res)
 | CurrentBehav::CoordinatingAResource(res) )
(^cur)( ResourceBehav::withCurrent(cur)
 | CurrentBehav::withAClient(cur) )
```

Since they are successful, we can conclude that

$$\texttt{ResourceBehav} \bowtie \texttt{CurrentBehav}$$

over the previous set of bindings $\mathcal{B}$, that is, that their interactions will be deadlock-free.

Another example comes from a Java program that implements a transactional client for a transactional account. It also serves to illustrate the usage of the CORBA TS:

```
public class ABankClient {
 public static void main( String [] args ){
  org.omg.CORBA.ORB orb =
    org.omg.CORBA.ORB.init(args,null);
  org.omg.CORBA.Object obj = null;
  try {
   obj = orb.string_to_object(args[1]));
   Account acc = AccountHelper.narrow(obj);
   org.omg.CosTransactions.Current current =
      null;
   obj = _orb.resolve_initial_references
      ("TransactionCurrent");
   current = org.omg.CosTransactions.
      CurrentHelper. narrow(obj);
   //transaction starts here
   current.begin();
   acc.deposit(100);
   acc.withdraw(acc.getBalance());
   current.commit(false);
   //transaction ends here
   ...
   } catch ( Exception e )
     {e.printStackTrace(); exit(1);}
 }
}
```

As we can see, the bank client above first locates the `Account` and the TS `Current` references: the first one is given as a parameter, while the second one is located by using the ORB services. It begins a transaction with the `Current` object and then commits to it. The transaction is defined by a set of three operations with the bank account, namely, a `deposit`, a `getBalance`, and a `withdraw`. A possible specification of its behavior (at the protocol level) can be written as follows:

```
protocol ABankClientBehav {
 #uses Current
 #uses Account
 #role 2PC(Current cur) =
   (^rep, SubtransUnavailable)
   cur!begin(rep, SubtransUnavailable) .
   ( SubtransUnavailable?() . zero
   + rep?() . (^NoTrans)commit!(rep,NoTrans) .
    ( rep?().zero + NoTrans?().zero )
   )
 #role Banking(Account bank) =
   (^rep,notEnough)
   bank!deposit(100, rep) . rep?() .
   bank!getBalance(rep) . rep?(bal) .
   bank!withdraw(bal,rep,notEnough).
   ( rep?().zero + notEnough?(balance).zero )
};
```

The compatibility of this client with the bank and the CORBA TS can be proven using the previous results, which means that we can prove that it is a valid client for the bank and for the CORBA TS, and that all objects will be able to successfully interoperate if they are put together.

5.2.1 Component Substitutability

Component substitutability refers to the ability of a component to replace another, in such a way that the change is transparent to external clients, i.e., so that the new component offers the same services as the old one [28]. This issue is not difficult to solve at the signature level, it is roughly a matter of checking that the interface of the new component contains all methods of the object to be replaced. We will use the subtyping rules defined for the CORBA IDL for checking CORBA objects substitutability at the signature level. However, the situation is different at the protocol level, where additional issues need to be addressed:

1. In the first place, we also need to check that the services required by the new component (hereinafter called *outgoing operations*) when implementing the original component's 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 the new one.
2. 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 component.

To check the first condition, we will use the sets of *used* interfaces of both components that describe their outgoing operations. For the second one that deals with the relative order of operations, in our approach protocols are split into separate roles, each of one described by means of the π-calculus processes. For reasoning about replacement of processes, the π-calculus offers the standard axiomatization of bisimilarity. However, bisimulation is too strict for our particular purposes because it forces the behavior of both objects to be undistinguishable. Effective replacement of a software component often implies that it must be adapted or specialized to accommodate it to new requirements [28]. 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). Protocol inheritance allows to decide whether a given component with role description R_1 can be replaced by another one R_2, while keeping the component's clients that use this role unaware of the change.

The detailed description of this relation of protocol inheritance is beyond the scope of this paper and can be found in [10]; here, we are more interested in showing its applicability than discussing its technical aspects in the π-calculus. However, we may roughly say that a certain role R_2 *inherits* from another one R_1 if the following three conditions hold: 1) R_2 preserves the semantics of the behavior of R_1 (any global choice offered by R_1 is also offered by R_2), 2) R_2 does not extend R_1 (any action present in R_2 is also present in R_1), and 3) R_2 terminates when R_1 also terminates (any successful trace of R_2 has a weak-similar trace in R_1).

If these conditions are fulfilled, we can ensure that R_2 may replace R_1 in *any* context, that is, when combined with

any other component that was compatible with R_1. However, the second condition seems to be too restrictive, constraining the new component to offer exactly the same methods as the replaced one. Hence, we have also introduced a relation between protocol descriptions called *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, thus ensuring safe substitutability. Therefore, we shall finally say that role R_2 can *substitute* R_1 ($R_1 \sqsubseteq R_2$) if R_2 extends R_1 according to the previous definition.

For instance, an alternative way of specifying role `CurrentBehav::WithAClient` is as follows:

```
#role WithAClient2(Current ref) =
  ref?begin(rep, SubtransUnavailable) .
   rep!() . TransactionsStarted(ref)
 + ref?rollback(rep, NoTrans) .
   NoTrans!() . zero
 + ref?rollback_only(rep, NoTrans) .
   NoTrans!() . zero
 + ref?get_status(rep) .
   (^status) rep!(status) . WithAClient(ref)
 ;
 TransactionsStarted(Current ref) =
   ref?commit(rep, NoTrans) . rep!() . zero
 + ref?rollback(rep, NoTrans) . rep!() . zero
 + ref?rollback_only(rep, NoTrans) .
   rep!() . zero
 + ref?get_status(rep) . (^status)
   rep!(status) . TransactionsStarted(ref)
 + ref?begin(rep,SubtransUnav) .
   SubtransUnav!() . TransactionsStarted(ref)
```

We can see in this example that the incoming and outgoing operations of both roles are the same, but that their behavior is not. The main difference is that role `WithAClient2` admits all operations in both states, while role `WithAClient` only allowed the `begin` operation at the beginning. In addition, role `WithAClient2` defines more clearly when the exceptions are raised. According to our definition, the relation between roles `WithAClient` $\sqsubseteq$ `WithAClient2` can be proven.

On the other hand, component protocols are defined in terms of sets of roles and, therefore, we need to extend the role substitutability operator "$\sqsubseteq$" to deal with sets of roles. Supposing that $\mathcal{R} = \{R_1, \ldots, R_n\}$ and $\mathcal{S} = \{S_1, \ldots, S_m\}$ are two sets of role descriptions, we shall say that $\mathcal{R} \sqsubseteq \mathcal{S}$ if there exists a subset $\{S_{i_1}, \ldots, S_{i_k}\}$ of elements of $\mathcal{S}$ such that $(R_1|\ldots|R_n) \sqsubseteq (S_{i_1}|\ldots|S_{i_k})$ as π-calculus processes.

We need this parallel composition of roles in order to cover the cases in which a role describes the component's interactions with two or more components simultaneously (i.e., multiparty interactions—see Section 6). However, in practice, different roles will usually describe separate and independent pairwise interactions and, therefore, protocol substitutability can be analyzed by checking the substitutability of just pairs of roles. This greatly simplifies the complexity of the analysis because it avoids the parallel composition of all the π-calculus processes, hence reducing the number of interleaved traces. Even in cases in which roles describe multiparty interactions, our experience shows that there is only a many-to-one or one-to-many relationship between the roles to be checked (as it happens for instance in the example in Section 6) and, therefore, the analysis of protocol substitutability can be done in a very tractable manner.

We are now in a position to define a substitutability relation between components at the protocol level and say that component B can *substitute* component A ($A \sqsubseteq B$) if: 1) all methods provided by A are also provided by B, i.e., $A.P \subseteq B.P$; 2) component B uses no more methods than component A, i.e., $B.U \subseteq A.U$; and 3) roles declared for component A are also replaceable by roles in component B, i.e., $A.R \sqsubseteq B.R$.

Here, the more efficient syntactic tests 1 and 2 can be conducted prior to checking the more expensive role substitutability test 3, narrowing the search space when looking for specific components that fit a given component specification.

The goal behind this definition of substitutability is to allow the replacements of components without affecting the safety and liveness properties of the system, independently from the system in which the replacement takes place. Of course, this context-independent definition may be too strict in some situations. But, note that, in those cases, context-dependent substitutability can be checked by ensuring that the new component is "compatible" with the rest of the existing system—and this is a matter of checking that the role-to-role bindings between the new component and the rest of the system are successful π-calculus processes (see previous section).

5.3 Run-Time Checking

Apart from the static checks, there are many situations in which protocol compatibility has to be checked at runtime. Typical cases are the applications developed in open and *independently extensible* systems [37], 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 runtime 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 can also be used in order to 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, which do not offer any support for dynamically handling incompatibility issues.

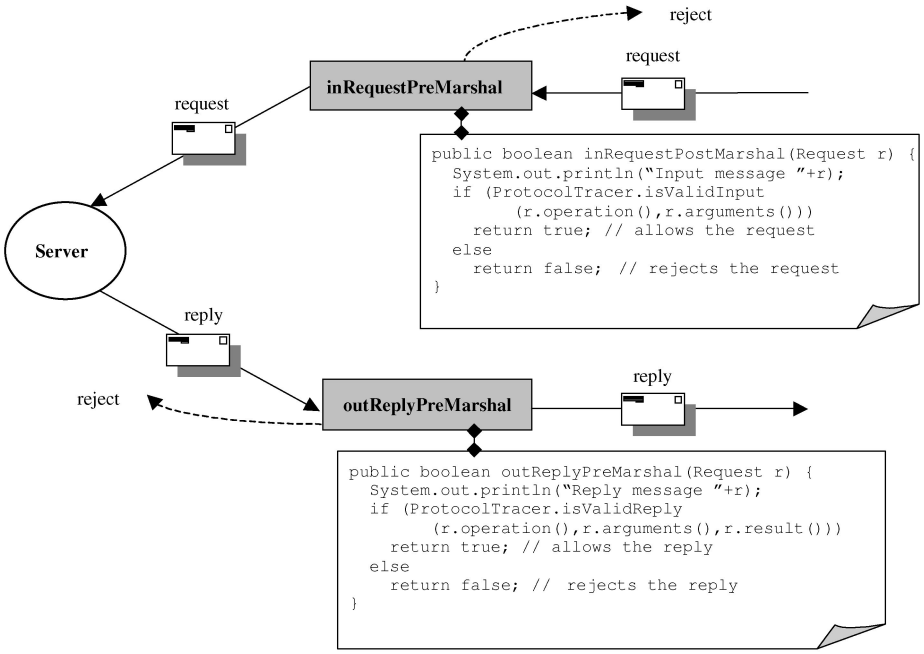

Fig. 3. Component interceptors.

5.3.1 Implementing the Runtime Tests

To implement these facilities in CORBA, we have used a reflective facility that some ORB vendors provide: interceptors [31] (also called *filters*). This mechanism was originally defined and implemented in Orbix [4] and allows a programmer to specify some additional code to be executed before or after the normal code of an operation. This additional 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 [1] or the Object Filters [20]), 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 runtime trace, and checks that received messages are compatible with the behavior defined for that object.

In Fig. 3, 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. `ProtocolTracer` is the name of the tool we have developed for reproducing the runtime trace of a given object's protocol (which is determined when the interceptor is created and associated to the object) and decides whether a 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. This is just a basic lexicographical analysis of the object protocol, for which the `ProtocolTracer` uses BYACC/J and JFLEX, standard tools available for Java programs.

Since the `ProtocolTracer` just *traces* the protocol, it is not burdensome; basically, it introduces the performance penalties due to message interception (see the discussion below). Each protocol execution also produces a runtime trace of the object, which is stored by the `ProtocolTracer` for possible postmortem analysis of the causes that led the system to the error situation.

5.3.2 Performance Analysis

In order to observe the various overheads involved in the dynamic checks using `ProtocolTracer`, several tests were conducted. For the tests setup, a client and a server were implemented in Java for the `Account` interface, using VisiBroker's implementation of the CORBA ORB running on a Pentium III machine. To measure the overhead, we calculated the average time of a method call (from the moment the client calls the server until the answer is received) using five different configurations:

C1. Without interceptors in any of the two CORBA objects (the client and the server).
C2. With empty interceptors in both the client and the server (to measure the overhead introduced by the system when using the interceptors).
C3. With an interceptor in the client that uses the `ProtocolTracer` to check that its incoming and outgoing messages conform to its declared protocol, but with no interceptor in the server.
C4. With an interceptor in the server, checking its behavior against its declared protocol, but with no interceptor in the client side.
C5. With interceptors in both the client and the server, using the `ProtocolTracer` to check their behavior.

TABLE 1
Absolute Timings for the Five Configurations

Configuration number	C1	C2	C3	C4	C5
Absolute times (in miliseconds)	1.36	1.43	2.91	2.89	4.37
Overhead introduced (ratio)	1	1.05	2.13	2.13	3.21

Table 1 shows the absolute average timings for these five configurations. Times in the table were calculated by making the client call method `getBalance()` in the server a number of different times, ranging from 1 to 1,150 invocations. The body of the called method introduced no penalty at all since it just returned the value of a local variable. The overhead introduced is also showed. The overhead figures have been calculated by dividing the absolute times of each configuration by the time taken by configuration C1.

The following conclusions can be extracted from these figures:

- A call from a Java client to a CORBA object took an average time of 1.36 ms.
- The use of interceptors introduced a very small overhead: 0.07 ms. It is just 5 percent of the total time of a method call. This figure is in accordance with the estimated overhead obtained in previous performance studies for object filters [20].
- The use of `ProtocolTracer` for checking the behavior of an object (no matter whether it is the server or the client) introduces an average overhead of 1.47 ms—basically, the same as a null method call to a CORBA object that has an interceptor enabled.
- It is very important to note that each method call produces two activations of the object filters in every object, i.e., two invocations to the `ProtocolTracer`. In the case of the client, the first one happens when the outgoing message with the method call departs from the object and the second one when the answer comes back from the server. In the server side, if the `ProtocolTracer` is enabled, the first filter is activated when the call is received and the second one when the answer is returned to the client, respectively. So, the average time that the `ProtocolTracer` takes to check the conformance of a message to a given protocol is about half of a null method call to a CORBA object that has an interceptor enabled.
- Of course, the fact of running both the client and the server in the same (mono-processor) machine doubles the times obtained when enabling the `ProtocolTracer` in both objects: The same processor has to execute them both (see timings for configuration C5).
- The complexity of the work done by the `ProtocolTracer` when analyzing a given protocol depends on the number of branches in each state of the protocol, not on the number of its states: The `ProtocolTracer` just reproduces the protocol traces. This number corresponds to the number of methods that the component is able to receive in a given state, plus the number of methods it is able to send. In any case, this number—in the worst case—is the maximum between the number of methods implemented by the component and the number of external methods it may call. These numbers are usually small if the design of the component is properly done (i.e., the number of methods a component implements in a given role is reasonably small), so the figures shown here for the `Account` protocol will not differ much from the figures obtained for other (more complex) protocols.

Summarizing, we can conclude that the average overhead introduced by the use of the `ProtocolTracer` in an object by means of interceptors is basically of the same magnitude as a null call to a CORBA object in the same machine. This means that the performance penalties introduced by the filters when implementing the dynamic compatibility tests are quite moderate.

5.4 Conformance to Specifications

Another issue that can be addressed with the dynamic tests described here is the 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: 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.

Interceptors were used in Section 5.3 for checking that incoming messages to an object were valid with regard to its current state. But, they can also capture outgoing messages (e.g., using `outRequestPreMarshal` and `inReplyPostMarshal` predefined methods of Orbix interceptors [4]), and, therefore, be used for checking that the object's behavior is valid with regard to the protocol it is supposed to implement.

5.5 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 runtime, in which all of the messages exchanged between the objects are checked for consistency with regard to 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, by means of a specialized transition system such as the one proposed by Sangiorgi for bisimulation [34], it is possible to develop efficient tools for the automated checking of the relations proposed in this paper. In addition, 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 performed in these situations. For instance, recursion in π-calculus agents may lead to the definition of processes with infinite number of states, which may make the analysis untractable from a computational point of view; nevertheless, most of the CORBA servers and clients used in real applications (even if recursively specified) have a very short number of states, which alleviates the complexity of the analysis and makes it practical in most cases. In addition, roles help simplifying the tests since they eliminate the burden of having to deal with the full protocols of the components, but just with the much simpler bindings between roles.

On the other hand, runtime compatibility tests can be performed on the fly by the interceptors with even less heavy burden, checking the conformance to a given protocol message by message. This approach has the advantage of making the analysis tractable from a practical point of view, while allowing the management of dynamic attachments in open environments in a natural way. The main disadvantages are the additional accounting process that filters should carry out, and that detection of deadlock and other undesirable conditions—usually caused by a wrong protocol design—is delayed until just before they happen, when the undesirable condition cannot be avoided, instead of detecting and solving it during the design process itself. However, exception raising and even recovery actions (either automated or prompted by the user) can be taken even if the undesirable condition is detected in the last minute, hence minimizing its consequences.

5.6 An Application

In order to show the applicability of our proposal, we analyzed a commercial implementation of the CORBA TS. The following code shows a typical implementation of a CORBA resource, as shown in most texts and manuals on the CORBA TS (e.g., in the JavaORB documentation, one of the packages we analyzed).

```
public class TransactionalAccountImpl
      extends TransactionalAccountPOA {
 private float _currentBalance, _realBalance;
 private org.omg.CORBA.ORB _orb;
 public TransactionalAccountImpl
     (org.omg.CORBA.ORB orb,String name){
   _currentBalance = 0; _realBalance = 0;
   _orb = orb;
 }
 public float getBalance()
   { return _currentBalance; }
 public void deposit( float value ) {
   if ( value < 0 ) notifyRollbackOnly();
   else _currentBalance += value;
 }
 public void withdraw( float value ) {
   if ( _currentBalance - value < 0 ) {
     notifyRollbackOnly();
     throw
     new NotEnoughMoney(_currentBalance);}
   else _currentBalance -= value;
 }
 void notifyRollbackOnly() {
   try {
    org.omg.CORBA.Object obj =
      _orb.resolve_initial_references
       ("TransactionCurrent");
    org.omg.CosTransactions.Current current
      = org.omg.CosTransactions.
       CurrentHelper.narrow(obj);
    current.rollback_only();
   } catch ( Exception e )
     { e.printStackTrace();
 }
 public
   org.omg.CosTransactions.Vote prepare() {
   if ( _currentBalance == _realBalance )
     // No change
     return org.omg.CosTransactions.Vote.
          VoteReadOnly;
   if ( _currentBalance < 0 )
     // Something went wrong
     return org.omg.CosTransactions.Vote.
          VoteRollback;
   return org.omg.CosTransactions.Vote.
          VoteCommit;
 }
 public void commit() throws
     org.omg.CosTransactions.NotPrepared {
   // makes the changes permanent
   _realBalance = _currentBalance;
 }
 public void rollback() {
   // restores the balance
   _currentBalance = _realBalance;
 }
}
```

Although apparently correct, this implementation does not exactly correspond to the specification of the CORBA TS, neither does it conform to the `ResourceBehav` described in Section 4.2. For space reasons, we have not included its protocol specification, but it is basically a simplified version of the `ResourceBehav` protocol: It only has one state and raises less exceptions. The problem is that this implementation does not consider the case in which a `commit()` operation arrives just after the object has issued a `rollback_only()` operation to the `Current` object. This may happen not only for explicit violations of the TS protocol, but also due to accidental reasons, such as a

communication delay in the delivery of the `rollback_only()` message to the `Current` object. This situation was detected using the compatibility tests described in Section 5.1.1, which proves the benefits that can be obtained by our approach. And, even if no static analysis was conducted prior to the execution of the system, the dynamic tests perfectly detect this problem at runtime. In this case, after observing that a `rollback_only()` operation has been issued to the `Current`, the invalid `commit()` message is rejected by the interceptor and does not reach the `Current` object.

6 Roles vs. Full Protocols

In general, there is always an issue about what information should be included in the protocol specification of a given component. In most approaches that specify service access protocols—e.g., [6], [8], [21], [41]—just one protocol specifying the full behavior of the component is described. However, this usually forces the introduction of many irrelevant details to the specification and produces specifications with exponential complexity to check. Besides, by merging together all of the component's interactions with the other components usually makes the compatibility tests with individual components, as well as the overall readability of the specification difficult. Why should any individual service provider or user of a given component be aware of its interactions with the rest of the component's users or providers? Roles allow us to concentrate just on separate interactions between components. We like to consider them as "*projections*" of the full protocols over the channels involved in those interactions. The sort of information provided by roles include the relative order between the exchanged messages, their "signature types" (i.e., names, parameter types, and return values of operations), and the enabled messages in each state of the communication.

Roles are also the common way in which protocols are specified in Architecture Description Languages (ADLs) for describing the software architecture of software systems (e.g., in Wright [2], Unicon [35], or LEDA [9]). Furthermore, the need of roles has also been pointed out by some other authors (e.g., by Honda et al. in [19]), that claim the need of separate specifications for unrelated component interactions; they call the dyadic interaction between components *sessions* and propose a $\pi - calculus\text{-}based$ protocol specification language for them. Of course, sessions provide the benefits mentioned above over the full protocols. However, there are situations in which the dyadic sessions are not enough: The information on how separate sessions interleave is needed too.

This fact has been shown, for instance, in work-flow or transactional applications, for which the order between sessions is very important. Think, for example, in the CORBA TS, where sessions do not provide any information on how the 2PC protocol operations and the account operations interleave. More precisely, from the session information available from a client object (e.g., the one in Section 5.1.1) nobody could tell apart the operations on the account that have been subjected to transaction management from those that have not. Actually, an alternative behavior for that client could be written as follows:

```
protocol ABankClientBehav2 {
 #uses Current
 #uses Account
 #role FullBehav(Current cur, Account bank) =
   (^rep, SubtransUnavailable)
   cur!begin(rep, SubtransUnavailable) .
   ( SubtransUnavailable?() . zero
   + rep?() .
    bank!deposit(100, rep) . rep?() .
    bank!getBalance(rep) . rep?(balance) .
    (^notEnough)
    bank!withdraw(balance,rep,notEnough) .
    (rep?().(^NoTrans)commit!(rep,NoTrans).
     ( rep?() . zero + NoTrans?() . zero )
    + notEnoughMoney?(balance) . zero )
   )
};
```

It can be proven that this client can replace the previous one and that it is still compatible with the behavior of both the `Account` and the `Current` objects. It also contains information about the operations under transactional management, missing in the previous client.

Another issue related to the sort of information conveyed by sessions is their impossibility of proving some global properties of applications, such as absence of deadlocks. Sessions only deal with pairwise interactions, so they can prove that local dyadic interactions are error-free, but not deal with other global properties.

The tradeoff here is the amount of information to be included in the protocol descriptions of the objects versus their complexity (and the complexity of the corresponding tests). One of the benefits of our proposal is that it allows one to specify not only separate sessions between just pairs of components, but also multiparty interactions. It is up to the component specifier to determine the description of the information contained in each role. This permits one to obtain the benefits of both approaches—sessions and full protocols—mixing them according to the system's particular requirements.

7 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 and that we are currently investigating.

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, thus 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 [41]. Apart from studying mediation at design time, some works have started focusing on the use of interceptors for serving as dynamic adaptors, whereby interceptors could adapt or enforce behaviors in such a way that otherwise incompatible components can cooperate (e.g., [7]).

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*, i.e., 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 runtime checks in all of those components. 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.

8 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. Here, we will not cite the proposals that try to deal with the full operational semantics of components (the interested reader can consult [22]), just the ones that cover the specification of the objects' service access protocols.

In the first place, Lea proposed PSL in 1995 [21], 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 is it supported by standard proving tools.

Protocol Specifications [41] 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 multiparty 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 more complex, open, and distributed environments.

The approach by Cho et al. [11] uses UML's OCL to specify pre- and postconditions on the objects' methods, together with a simple finite state machine to describe message protocols. Similarly, Han [17] proposes an extension to IDLs that includes semantic information in terms of constrains 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.

Bastide et al. [6] 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 also been successfully used to detect inconsistencies in some CORBA services commercial implementations [5]. The main difference between their work and ours is 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 allows much richer information to be included in the objects' behavioral descriptions, which may be required in some cases.

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 permits the description of the interactions among objects, and now is part of UML. MSCs are very expressive for describing protocol interactions, but they do not allow the proof of (safety or liveness) properties of the system, nor the inference of certain results.

Finally, Architectural Description Languages (ADLs) usually include the descriptions of the protocols that determine the access to the components they define. Many ADLs use standard notations for describing component interactions (such as CSP, CCS, or π-calculus), which allow the simulation of the application's behavior, or the formal derivation of some of its safety or liveness properties. For instance, the Rapide ADL is well known as a rich pattern language for the simulation of architectural behavior. Wright [2] uses CSP for specification, and as a consequence is supported by model-checking tools such as FDR. Darwin [24] and LEDA [9] are examples of ADLs that make use of the π-calculus for describing the behavior of the components of a system. One of the benefits of using standard calculi is that reasoning about the system behavior and correctness can be done using appropriate tools. Our focus is slightly different since we are more concerned with the specification of COTS components independently from the applications they will be part of. What we have shown here is that we can achieve the same sort of tests that software architects carry out with their ADLs (such as those described in [12], [14]), right from the objects' protocol specifications. In addition, our proposal not only contemplates compatibility tests, but it also studies substitutability between components. Another novelty of our work is the dynamic tests that can be carried out with the interceptors,

checking at runtime that the behavior of a CORBA object conforms to its declared protocol. They help avoid incompatibility issues, deadlock situations, and possible violations to the protocols.

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 for describing 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 [13] or CORBA [15]. 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 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.

9 Concluding Remarks

With the rapid development of e-commerce and the Internet, the increasing amount of information available is becoming too large to be handled by human intermediaries. Soon, most of the search and retrieval processes needed for component mining in software repositories will have to rely on computers, possibly leaving to system designers just a few final decisions.

Signature information is not enough for these purposes. In this paper, we have highlighted 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 based on roles (relative ordering of messages and the blocking conditions that rule the component's interactions with other components) 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 conducted when building up applications from reusable components.

Apart from looking into the open 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 this. The addition of this type of information to other object and component models (such as DCOM, .NET, EJB, or CCM) is an ongoing subject of research. In addition, the development of new services and tools that make use of it, such as protocol repositories and extended service traders is also an interesting challenge, especially if they try to use the reflective facilities that component models are starting to incorporate. Finally, protocols cover only one partial aspect of the full specifications of components. Exploring how to "weave" protocols with other different specification aspects (e.g., the behavioral semantics of components, or other nonfunctional properties such as performance or real time) is an ambitious goal, but specifications need to be somehow combined if effective component trading is to be achieved.

Acknowledgments

The authors would like to thank the anonymous referees for their insightful comments and suggestions, that greatly helped them improve the contents and readability of the paper. This work has been partially supported by Spanish CICYT Projects TIC99-1083-C02-01 and TIC01-2705-C03-02.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, "Abstracting Object Interactions Using Composition Filters," *Proc. Seventh European Conf. Object-Oriented Programming (ECOOP'93)*, pp. 152-184, 1993.

[2] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.

[3] R. Allen, D. Garlan, and J. Ivers, "Formal Model and Analysis of the HLA Component Integration Standard," *Proc. the Sixth Int'l Symp. Foundations of Software Eng.*, Nov. 1998.

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

[5] R. Bastide and O. Sy, "Towards Components that Plug AND Play," *Proc. European Conf. Object-Oriented Programming, Workshop Object Interoperability (WOI'00)*, A. Vallecillo, J. Hernandez, and J.M. Troya, eds., pp. 3-12, June 2000.

[6] R. Bastide, O. Sy, and P. Palanque, "Formal Specification and Prototyping of CORBA Systems," *Proc. 13th European Conf. Object-Oriented Programming (ECOOP'99)*, pp. 474-494, 1999.

[7] A. Bracciali, A. Brogi, and C. Canal, "Dynamically Adapting the Behaviour of Software Components," *Proc. Fifth Int'l Conf. Coordination Models and Languages—(Coordination'02)*, F. Arbab and C. Talcott, eds., pp. 88-95, 2002.

[8] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo, "Extending CORBA Interfaces with Protocols," *The Computer J.*, vol. 44, no. 5, pp. 448-462, Oct. 2001.

[9] C. Canal, E. Pimentel, and J.M. Troya, "Specification and Refinement of Dynamic Software Architectures," *Software Architecture*, pp. 107-125, P. Donohoe, ed., 1999.

[10] C. Canal, E. Pimentel, and J.M. Troya, "Compatibility and Inheritance in Software Architectures," *Science of Computer Programming*, vol. 41, pp. 105-138, 2001.

[11] I. Cho, J. McGregor, and L. Krause, "A Protocol-Based Approach to Specifying Interoperability between Objects," *Proc. 10th Int'l Conf. Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'98)*, pp. 84-96, 1998.

[12] D. Compare, P. Inverardi, and A.L. Wolf, "Uncovering Architectural Mismatch in Component Behavior," *Science of Computer Programming*, vol. 33, no. 2, pp. 101-131, Feb. 1999.

[13] L. Feijs, "Modelling Microsoft COM Using π-Calculus," *Proc. Formal Methods Europe (FME'99)*, pp. 1343–1363, Sept. 1999.

[14] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is So Hard," *IEEE Software*, vol. 12, no. 6, pp. 17-26, Nov. 1995.

[15] M. Gaspari and G. Zabattaro, "A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service," *Proc. 13th European Conf. Object-Oriented Programming*, pp. 495-518, 1999.

[16] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins, "Software Component Search," *J. Systems Integration*, vol. 6, pp. 93-134, Sept. 1996.

[17] J. Han, "Semantic and Usage Packaging for Software Components," *Proc. European Conf. Object-Oriented Programming, Workshop Object Interoperability (WOI'99)*, A. Vallecillo, J. Hernandez, and J. M. Troya, eds., pp. 25-34, June 1999.

[18] P. Henderson, "Formal Models of Process Components," *Proc. Sixth European Software Eng. Conf., FoCBS Workshop*, pp. 131-140, Sept. 1997.

[19] K. Honda, V.T. Vasconcelos, and M. Kubo, "Language Primitives and Type Disciplines for Structured Communication-Based Programming," *Proc. Seventh European Symp. Programming (ESOP'98)*, pp. 122-138, 1998, available at http://www.di.fc.ul.pt/vv/papers/lang-prim.pdf.

[20] R. Joshi, N. Vivekananda, and D.J. Ram, "Message Filters for Object-Oriented Systems," *Software-Practice and Experience,* vol. 17, no. 6, pp. 677-699, 1997.
[21] D. Lea, "Interface-Based Protocol Specification of Open Systems Using PSL," *Proc. Ninth European Conf. Object-Oriented Programming (ECOOP'95),* 1995.
[22] *Foundations of Component-Based Systems.* G.T. Leavens and M. Sitaraman, eds., Cambridge Univ. Press, 2000.
[23] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Trans. Software Eng.,* vol. 21, no. 4, pp. 336-355, Apr. 1995.
[24] J. Magee, J. Kramer, and D. Giannakopoulou, "Behaviour Analysis of Software Architectures," *Software Architecture,* pp. 35-49, 1999.
[25] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," *Proc. ACM SIGSOFT FSE'96: Fourth Symp. Foundations of Software Eng.,* pp. 24-32, Oct. 1996.
[26] R. Milner, "The Polyadic π-Calculus: A Tutorial," *Logic and Algebra of Specification,* pp. 203-246, 1993.
[27] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes," *J. Information and Computation,* vol. 100, pp. 1-77, 1992.
[28] O. Nierstrasz, "Regular Types for Active Objects," *Object-Oriented Software Composition,* O. Nierstrasz and D. Tsichritzis, eds., pp. 99-121, 1995.
[29] A. Olafsson and D. Bryan, "On the Need for Required Interfaces of Components," *Special Issues in Object-Oriented Programming, Workshop Reader of ECOOP'96,* pp. 159-165, 1996.
[30] OMG, *The CORBA Component Model.* Object Management Group, June 1999.
[31] OMG, *The C ommon Object Request Broker: Architecture and Specification,* 2.4 ed. Object Management Group, Nov. 2000, http://www.omg.org/technology/documents/formal/corbaiiop.htm.
[32] OMG, *The CORBA Event Service Specification.* Object Management Group, Jan. 2000.
[33] OMG, *The CORBA Transaction Service Specification.* Object Management Group, May 2000.
[34] D. Sangiorgi, "A Theory of Bisimulation for the π-Calculus," Technical Report ECS-LFCS-93-270, Univ. of Edinburgh, June 1993.
[35] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng.,* vol. 21, no. 4, pp. 314-335, Apr. 1995.
[36] J.P. Sousa and D. Garlan, "Formal Modeling of the Enterprise JavaBeans Component Integration Framework," *Proc. Formal Methods Europe,* pp. 1281-1300, 1999.
[37] C. Szyperski, *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
[38] *Proc. ECOOP'99 Workshop Object Interoperability,* A. Vallecillo, J. Hernandez, and J.M. Troya, eds., June 1999.
[39] B. Victor, "A Verification Tool for the Polyadic π-Calculus," Master's thesis, Uppsala Univ. (Sweden), May 1994.
[40] K.C. Wallnau, S.A. Hissam, and R.C. Seacord, *Building Systems from Commercial Components.* Addison-Wesley, 2002.
[41] D.M. Yellin and R.E. Strom, "Protocol Specifications and Components Adaptors," *ACM Trans. Progamming Language Systems,* vol. 19, no. 2, pp. 292-333, Mar. 1997.
[42] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 4, pp. 333-369, Oct. 1997.

Carlos Canal is an associate professor of software engineering at the University of Málaga (Spain), where he received the MSc and PhD degrees in computer science in 1993 and 2001, respectively. His research interests deal with software architecture and component-based software development, in particular with the application of formal methods to architectural specification, safe composition, and component adaptation issues. In addition to his participation as a speaker in international conferences on these subjects, his most significant publications can be found in *The Computer Journal* and *Science of Computer Programming.*

Lidia Fuentes received the MSc degree in computer science from the University of Málaga (Spain) in 1992 and the PhD degree in 1998 from the same university. She is an associate professor at the Department of Computer Science of the University of Málaga since 1993. Her research interests deal with component-based software development, frameworks, aspect orientation, software agents, and collaborative virtual environments. She has participated in several international conferences as a speaker, reviewer, and chairman. Her most significant publications can be found in *IEEE Internet Computing*, *ACM Computing Surveys*, or *Software Practice and Experience*. She has actively participated in several research projects and she has also done consulting activities for telecommunication companies such as Telefónica and Retevisión (Spain).

Ernesto Pimentel holds the BSc and MSc degrees in mathematics (1988), and the PhD degree in computer science (1993). He is currently an associate professor at the University of Málaga, and involved in a number of national and international research projects as responsible and senior researcher. Since 2001, he is part of the managing team of the Spanish National Research Program for Communication and Information Technologies. His research activity is focused on the application of formal methods to software engineering, including topics such as models and logics for concurrency, component-based software development, and abstract model checking. He has participated in many conferences as a speaker, member and president of the program committee, member of the organizing committee, member of the steering committee, and chairman. Most of his work has been published in international journals.

José M. Troya received the MSc (1975) and PhD (1980) degrees in physics from the Madrid Complutense University. From 1980 to 1988, he was an associate professor at that university, and, in 1988, became a full professor at the University of Málaga, where he has lead the Software Engineering Group. His research interests include parallel programming, distributed systems, and software architectures. He is very involved in several national and international research projects, has written articles for the most relevant computer conferences and journals, supervised numerous PhD thesis, and organized several workshops and international conferences.

Antonio Vallecillo holds the BSc and MSc degrees in mathematics, and the PhD degree in computer science. Most of his professional experience comes from the computer industry, where he has worked for more than 10 years for several international companies, both in Spain and in UK. Since 1996, he has worked at Málaga University, where he is currently an associate professor. His research interests include component-based software development, open distributed processing, and the industrial use of formal methods. He is the representative of the Málaga University at ISO and OMG, and a member of several professional organizations, including AENOR (the Spanish National Body for Standardization), the ACM, and the IEEE.