Extending IDLs with π-calculus for Protocol Compatibility

Carlos Canal, Lidia Fuentes, and Antonio Vallecillo

Dept. Lenguajes y Ciencias de la Computación
Universidad de Málaga. ETSI Informática.
Campus Teatinos, s/n. 29071 Málaga, Spain.
{canal,lff,av}@lcc.uma.es

Abstract. Traditional IDLs present some limitations, since they were defined for describing the services that objects offer, but not those services they require from other objects, nor the partial ordering in which their services are expected to be used. In this paper we propose an IDL extension that uses a subset of π-calculus for describing objects service protocols. Our approach can be also used in a component-oriented environment for automated checking of protocol interoperability between objects. This is supported by a platform that controls the dynamic composition of components.

1 Introduction

Traditional object *Interface Description Languages* (IDLs) were meant for describing object interfaces in closed client-server applications, and therefore they present some limitations when tried to be used in open component-based applications:

1. IDLs describe the services that objects offer, but not the services that they require from other objects in order to accomplish their tasks.
2. Typical IDLs provide just the *syntactic* descriptions of the objects public methods, i.e. their signatures and the types of their return values. However, nothing is said about the ordering in which the objects expect their methods to be called, or their blocking conditions.
3. IDLs are mainly used during compilation time but not during object execution, where other mechanisms need to be in place for dynamically knowing about the object interfaces and for checking their compatibility at run time.

Several authors have provided a number of proposals that try to overcome some of these limitations. In the first place, Doug Lea's PSL [3] proposes an extension of the CORBA IDL to describe the protocols (the partial ordering between the messages that two or more objects exchange, and their blocking conditions) associated to an object's methods. It is a very good and expressive approach, although it inherits one of the CORBA IDL limitations: it is a client-server approach based on RPCs only. Therefore, it does not account for the

services the object may need from other objects, and the RPC paradigm is embedded into the description language, which limits its applicability to other ways of object interaction.

Protocol Specifications [8] is a more general approach for describing protocols, although some of its limitations (as recognized by their own authors) make it too rigid for its general use in open and distributed environments.

Reuse Contracts [4] is another well known proposal, although it is based on textual annotations to facilitate the reuse by humans, not by computer programs during object execution.

Finally, we will also mention Architectural Description Languages (ADLs), since they usually include the descriptions of the protocols that determine the access to the components they define. It is also worth mentioning the use of standard notations for protocol description in some ADL proposals, that derive from process algebras (like CSP, CCS or π-calculus) for describing what they call the 'dynamic' aspects of the systems. One of the benefits of using standard calculi is that reasoning about system behavior and correctness can be done using appropriate tools. LEDA [1] is an example of an ADL that makes use of π-calculus for describing the behavior of the components of a system.

Regarding the extensions of IDLs that account for both the services offered and required by an object to accomplish its tasks, few of them have been actually defined. An example of those can be found in [6], where an IDL called CDL (Component Description Language) is defined for components in which both incoming and outgoing messages of an object can be specified. One of the benefits of CDL is that it can also be used by the components of a system while they are executing, hence providing dynamic and individual information about the services available in an open and evolving system during its lifetime. However, this sort of IDLs remain at the syntactic level, without offering any information about the protocols or the semantics of the services they describe.

On the other hand, component models are usually supported by a middleware platform. Distributed component platforms usually offer common services that provide the communication needed to dynamically assemble applications. This infrastructure allows coordination among components which have previously registered their IDLs in it. In this approach, the platform also has to deal with the protocols understood by the components that enter or leave the system dynamically. By finding out which components are currently connected, the platform is able to keep track of the dynamic attachments between them, and to verify their compatibility. The goal of the platform is to support the construction of applications with dynamic architectures, and also to manage the deadlock conditions that may arise during system execution.

2 A first extension to IDLs

Our main aim in this paper is to combine some of the present proposals to cover the three requirements mentioned at the beginning of the paper, namely describing at the IDL level the services that objects require during their exe-

cution, providing protocol information beyond the syntactic description of their methods, and making all this information available at run time.

As the base IDL we will use CDL, whose description can be found in [6], and the protocol information will be specified in textual π-calculus following the work by Canal et al. in [1]. MultiTEL [2] is a framework for building Web-based multimedia and collaborative applications that provides an ideal environment for building real applications, and that uses the latest component-based technology. MultiTEL will serve as the platform in which our proposal can be experimented and validated.

In MultiTEL, components receive input messages and propagate events to the environment, and the components descriptions in its IDL differentiate between incoming and outgoing messages, like in CDL. However, while MultiTEL determines how components are plugged together at run time according to some information about the system architecture, we are now more interested in considering components as isolated entities that are not part of a predefined or rigid architecture. In this way, we are extending MultiTEL with capabilities for checking protocol compatibility of unknown components at run time, and using CDL for component interface descriptions.

Let's use an example to illustrate our proposal. Suppose we have an electronic conference, where participants are modeled by objects with the following CDL simplified interface:

```
Component Participant {
    Parameters: String self; //its address
    Out: ?Connect(String self);
         !Connect(String self);
         !Disconnect(String self);
         !Talk(String speech);
    In:  Re:?Connect(String mngr, String info);
         Re:!Connect(String participants);
         !Talk(String speech);
}
```

This text describes the component's list of outgoing and incoming messages, identified by their selectors and arguments. Messages are information entities that model the way objects exchange information, and they are identified by their *selectors*, that determine the operation to be executed by the target object. For every method f implemented by an object, we define four different selectors: $!f$, $?f$, Re:$!f$ and Re:$?f$. The first one invokes the method, and Re:$!f$ is used for replying to it. Selector $?f$ asks the destination component whether it implements method f or not, and Re:$?f$ answers this question. Another interesting feature of this model is that messages can be broadcasted to domains, which are sets of interconnected machines.

The information provided on the `In` and `Out` parts of the CDL description of components has proved to be very useful, in particular for defining formally and reasoning at the syntactic level about concepts like equivalence, compatibility,

and substitutability of components in open and distributed environments [7]. The absence of this information in traditional IDLs, particularly the `Out` part, limits the sort of reasoning that can be made about the components described with them.

Coming back to the example, we can see in it how a participant can learn about a conference using message `?Connect(self)`, in which `self` is its own address, required for replying. The reply message is `Re:?Connect(mngr,info)`, whose arguments are the address of the conference manager and some information about it (e.g. the topics being discussed). A participant `Part` can also join a conference using the message `!Connect(Part)`, getting a reply given by `Re:!Connect(participants)` with the list of participants currently connected to it. Every time this list changes, all participants in the conference receive a message like this. Besides, participants can talk among themselves by sending message `!Talk` to other participants. This is why `!Talk` is both an outgoing and incoming message in the participant object interface. Finally, a participant can also leave the conference using message `!Disconnect(Part)`.

In addition to the participants in a conference, there should exist one conference manager, which is in charge of managing the list of participants. Its (simplified) CDL description is as follows:

```
Component Manager {
    Parameters: String self; //its address
                String info; //info about the conference
    Out: Re:?Connect(String self, String info);
         Re:!Connect(String participants);
    In:  ?Connect(String participant);
         !Connect(String participant);
         !Disconnect(String participant);
}
```

3 Extending IDLs with Protocols

There is a general problem with this sort of specifications, in which only the syntactic descriptions of the interoperation of objects are described. This problem is related to the partial order in which the objects expect their messages to be called, and the order they should invoke other objects' methods (i.e. their precise behavior). In the particular case of our example, this problem has effect in two different situations. The first one has to do with the fact that operation `Connect()` should be invoked for a conference before tapping into it. This is why protocols are needed in this case.

The second situation is related to the behavior of objects. Suppose that we have three different implementations of object `Participant`, each one reflecting different personalities. Let's call these participants *Mary*, *John*, and *Paul*. Suppose that *Mary* behaves in a civilized way, always talking and listening in an orderly manner; *Paul* is a shy participant that only listens; and *John* offers the

opposite behavior, he always talks but never listens. From this description it is obvious that a conference with several *Marys* and *Pauls* will progress with no problems, the same if we have several *Pauls* and *Johns*. However, other combinations will produce undesirable results: a conference with only two *Pauls* will be too quiet, and with two *Johns* it will be of no good to any of them. However, all three components conform to the previous interface description, which makes their behavior undistinguishable if we use only a traditional IDL, in which all three appear as equivalent. Protocols can also help in this situation, bringing to the interface level some aspects of the behavior of the objects they describe.

In order to overcome these problems, our proposal is based on extending the CDL descriptions of components with a new clause, `Protocols`, that describe the behavior of the objects they account for. In the particular case of the `Manager` we can have the following addition to its CDL description, which now states that one must connect to a conference before participating in it:

```
Component Manager {
 ....
 Protocols:
   Manager = * ?Connect(Part).Part Re :?Connect(self,info).Manager
           + * !Connect(Part).[Participants+=Part].
                   Participants Re :!Connect(Participants).Manager
           + Participants !Disconnect(Part).[Participants-=Part].
                   Participants Re :!Connect(Participants).Manager;
}
```

Protocols are described using the π-calculus, a process algebra specially suited for the specification of dynamic and distributed systems. Communication is performed using links that connect processes. Reception from a link named *message* is written `message(prmtrs)`, while the emission of messages is overlined, like in $\overline{\texttt{message}}$`(prmtrs)`. We name links using the convention `process method`, where `process` represents the sender of an incoming message or the recipient of an outgoing message, and it may be indicated by a component name, a list of components, or by '`*`', which stands for any component in the system. Thus, `* ?Connect` indicates the reception of a `?Connect` event (asking the receiver whether it implements the method `!Connect`), sent by any possible component. On the other hand, `Part Re` :!`Connect(Participants)` indicates that the list of participants of a conference is sent to a certain new participant `Part`, as a reply of the invocation of method `!Connect`. Finally, **0** represents the inactive process; τ represents an internal action; and the summation operator + indicates an alternative among two or more options. Thus, τ`.m1.P1 +` τ`.m2.P2` represents a process which is waiting either for an event `m1` or an event `m2`, being the process able to accept any of those events. However, by combining alternatives and internal actions, like in $\tau.\overline{\texttt{m1}}$`.P1 +` $\tau.\overline{\texttt{m2}}$`.P2` we can express *local choices*, since the process may choose any alternative with independence of its context, by performing an internal action τ, and then committing to one particular output action. Some internal actions of special interest (like the ones in

the example adding or removing participants from the list) are expressed inside square brackets. The π-calculus is a nice way for expressing protocols and it also allows the dynamic checking of protocol conformance and matching at run time.

Coming back to the example, the second of the problems previously mentioned was related to the necessity of distinguishing between different behaviors of objects that conform to the same syntactic interface. Now, we could define three different CDL descriptions for the three participant components, according to the way they behave. The differences would be only in their `Protocols` clause, since the rest of the CDL description is common to all three. Mary's protocol is described below, while Paul's and John's protocols can be obtained just suppressing the lines marked (1) and (2), respectively. It is worth mentioning that mechanisms of protocol inheritance and extension can be used to describe these protocols reusing what they have in common, as shown in [1].

```
....
Protocols:
    Mary = * ?Connect(self).Selecting;
    Selecting = * Re:?Connect(Mngr,info).
                  ( τ.Selecting
                  + τ.Mngr !Connect(self).
                            Mngr Re:!Connect(Participants).Talking );
    Talking = Participants !Talk(speech).Talking          (1)
            + Participants !Talk(speech).Talking          (2)
            + Mngr Re:!Connect(Participants).Talking
            + τ.Mngr !Disconnect.0;
```

4 Checking Protocol Compatibility at Run Time

The main aim of this section is to discuss about the possibility of having protocols checked during run time, and the mechanisms required for that purpose.

Once we have included protocol information into IDLs, we obtain several advantages: high level reasoning about software composability and behavioral subtyping, proving safety and liveness properties of the composed applications (e.g. no deadlocks), or even achieving process refinement. Thus, we are able to prove in an open component-based development environment, many of the results that were only available for software architects in their particular environments.

All those benefits are usually obtained during design or compile time. One approach to achieve them is based on using a component framework which contains architectural information about the system structure and its internal interconnections. Components are considered within a component framework context, and the information about the structure of the application is used at compile time to perform static analysis of protocol compatibility between the IDLs of the components that take part of the system, and also to guide the connection of components at run time.

However, there are many situations in which protocol compatibility has to be checked at run time. Typical cases are the applications developed in *independently extensible* systems [5], in which the evolution of the system and the components on it is unpredictable: new components may suddenly appear or disappear, while others are replaced without previous notification. In those cases the previous approach seems feasible, but unfortunately it is no longer valid since it cannot be used for dynamic attachments among unknown components.

Before continuing, we need to point out some issues that have to be considered in this sort of open environments:

- components are defined independently from their context and the applications they will be part of,
- the connections among the components are not explicitly stated anywhere, and
- components may dynamically join or leave the applications.

Under those circumstances, four main questions arise once we have enriched IDLs with protocols:

1. how could the platform decide which components should be analyzed for compatibility?
2. when the platform should check the compatibility of their protocols?
3. how to do it?
4. and finally, how do we deal with late-comer components that have to interact with other components that may be in an intermediate state?

Our proposed approach is based on the assumption that the platform does not have any previous knowledge about the attachments among components, not being able to know when to verify protocol compatibility, nor among which components. Hence, the platform will have to collect at run time all the information needed for protocol checking. Following this approach, the information available for the platform is just the list of components that are bound for a particular application, together with the IDLs of these components, and a π-calculus interpreter. Besides, the platform is aware of the interchange of events among components.

In our proposal, the following is a list of the tasks that the platform should be able to do to achieve its goal:

- Component creation and deletion.
- IDL registration.
- Management of components input/output messages, including broadcast messages.
- Reproduction of components' behavior. Provided that the platform knows component protocols and the messages being interchanged, it will be able to reproduce the run time trace for every component in the system.
- Analysis of system behavior for deadlock detection.

With all this, the platform will be able to construct the system architecture at run time, so it can obtain information about the attachments that are dynamically established between system components.

On the other hand, protocol compatibility can be checked at run time by intercepting messages and verifying their correctness with regard to the current state of the destination component. In this way the platform will be able to detect a system crash before it happens, and take the appropriate actions beforehand. This sort of information is very useful for system debugging, and it may help components to make run time decisions about their behavior within an application. Components entering in a deadlock state would be notified by an error event and then they may decide to leave the application or just to cancel any pending request to other components. In the previous example, if the platform detects that only *John* components remain in a system, it can notify them about this situation with an error event. They can decide either to leave, or to remain in the system waiting for new *Mary* or *Paul* components to join the conference, in order to escape from deadlock.

On top of the benefits for open system debugging and prototyping that we can obtain checking compatibility during system execution, we want to go one step further and see whether we could perform static analysis at *connection time*, detecting protocol mismatch prior to execution. This implies the verification of collaborations among components before they begin. We have found that this problem can be solved by dividing protocols into different *roles*, each one describing the behavior (i.e. the rôle) a component plays in its interactions with other components. This could also allow the platform to check protocol compatibility between a source component, which is in its initial state, and a target component, that may be in an intermediate state.

In order to do this, we have to restrict the analysis to the interchange of messages that takes place through certain common channels. Thus, components must provide the platform with the list of different subprotocols or *roles* in which the component participate. In our example, the participants are involved into two different roles: connection/disconnection with the manager, and talking with other participants. To distinguish between both roles, we must indicate which messages are associated to each of them. Then, the platform will be able to perform protocol compatibility, but restricted just to a given role or subprotocol.

5 Discussion

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

However, our proposal also presents some limitations. In the first place, checking π-calculus specifications is in general an NP-hard problem, which makes it impractical for real heavy-weighed applications. Unfortunately, this is one of the most unavoidable drawbacks of checking object compatibility at the protocol or semantic levels at run time. One way of overcoming this problem is by doing it on the fly as the platform captures the messages exchanged by the components, checking the conformance to a given protocol message by message. This method delays analysis until run time, but has the advantage of making it tractable from a practical point of view, and it also allows the management of dynamic attachments in open environments in a natural way. The disadvantage is that detection of deadlock and other undesirable conditions is also delayed until just before they happen, which may be unacceptable in some specific situations. In any case, we are able to check and know about the occurrence of those problems, which is better than completely ignoring them.

Another limitation is that our proposal is currently defined on top of an specific IDL, namely CDL. Extending widely used IDLs (like CORBA or COM) with the features we propose here is something we are currently working on, as well as improving the automatic π-calculus checker so it can be packaged as a standard stand-alone component.

References

1. C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In Patrick Donohoe, editor, *Software Architecture (Proc. of WICSA'99)*, Kluwer Academic Publishers, pages 107–125, February 1999.
2. L. Fuentes and J. M. Troya. A Java framework for Web-based multimedia and collaborative applications. *IEEE Internet Computing*, 3(2):52–61, March/April 1999.
3. D. Lea. Interface-based protocol specification of open systems using PSL. In *Proc. of ECOOP'95*, number 1241 in LNCS. Springer-Verlag, 1995.
4. P. Steyaert, C. Lucas, K. Mens, and T. d'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.
5. C. Szyperski. Independently Extensible Systems – Software Engineering Potential and Challenges –. In *Proc. of 19th Australasian Computer Science Conference*, Melbourne, 1996.
6. J. M. Troya and A. Vallecillo. Software development from reusable components and controllers. In *Proc. of ASOO'98*, pages 47–58, Argentina, September 1998.
7. J. M. Troya and A. Vallecillo. Specifying Reusable Controllers for Software Components. In *Proc. of FMOODS'99*, Florence, February 1999.
8. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.