Formal Specification of CORBA Services: Experience and Lessons Learned.
Rémi Bastide, Philippe Palanque, Ousmane Sy, David Navarre
LIHS, Université Toulouse 1, Place Anatole France, F-31042 Toulouse CEDEX, France
{bastide, palanque, sy, navarre}@univ-tlse1.fr

ABSTRACT
CORBA is now established as one of the main contenders in object-oriented middleware. Beyond the definition of this standard for distributed object systems, the Object Management Group (OMG) has specified several object services (Common Object Services, COS) that should foster the interoperability of distributed applications. Based on experiment, the goal of this paper is to show that the OMG's style of specification of the CORBA services is not suited to guarantee that implementers will produce interoperable and substitutable implementations. To illustrate our point, we give an account of an experiment based upon the formal specification of one COS, namely the CORBA Event Service. This formal specification highlights several ambiguities and under-specifications in the OMG document. We then test several commercial and public domain implementations of the CORBA Event Service, in order to assess how the implementers have dealt with these under-specifications. We show that the choices made by the implementers lead to incompatible implementations. We finally suggest a solution to overcome the problem of specification of object services, which satisfies the views of both implementers and users. Specifically, we suggest that the specification of such services be made using a formal description technique, and that implementers be provided with test cases derived from the formal specification.

Keywords
Distributed systems, behavioral specification, CORBA, high-level Petri nets.

1. INTRODUCTION
CORBA [19] (Common Object Request Broker Architecture) is a standard proposed by the Object Management Group (OMG) in order to promote interoperability between distributed object systems. CORBA mainly defines a common architecture for an Object Request Broker (ORB) and an Interface Definition Language (IDL). An ORB is a software component that provides the middleware necessary to locate server objects on a network and to route invocations between clients and servers regardless of both the programming language they have been written in and their actual location. To achieve this goal, the CORBA standard requires the use of IDL for describing the interface of remote objects. The IDL defines basic types (short, float…), structured types (struct, sequence, array…) and provides signatures of operations for interface types.

The OMG has specified a set of common object services (COS) in a document called the Common Object ServicesSpecifications (CROSS) [20]. CORBA services are designed to be the building blocks of large-scale CORBA applications and have the stated goal of promoting interoperability, as they give a standardized solution to commonly encountered problems. Among typical COS are the CORBA Naming Service (that allows retrieving a remote object reference by providing a symbolic name) and the CORBA Event Service (which simulates asynchronous multicast communication on top of the basic synchronous unicast communication scheme supported by CORBA).

The CORBA services specification is mainly provided by the OMG in the form of a mixture of IDL (for the definition of the interfaces) and English text (for the functional specification). The COSS do not integrate behavioral description and, as a result, are sometimes inaccurate and ambiguous.

Moreover, the COS are, in several places left deliberately underspecified. This has been a choice of the OMG to leave "quality of service" issues out of the specifications, in order to leave flexibility to the implementers of services. However, in our opinion, this form of flexibility has, in several occasions, gone so far as to hinder interoperability between distinct implementations.

Furthermore, the OMG does not provide test cases that would allow checking the conformity of implementations with respect to their specifications. Therefore, developers working in companies producing COS implementations have to overcome these inaccuracies and underspecifications using their own experience. To summarize, despite CORBA's clear objectives, the specification leads to potential interoperability problems.

The SERPICO research project aims at building an environment for the construction of CORBA systems and promotes the use of a formal behavioral specification language to complement CORBA-IDL. Therefore we have followed a development process in which formal and practical views go hand in hand. This has resulted in the elaboration of a formal notation (Cooperative Objects, CO) dedicated to CORBA specification and to the development of a tool (PetShop) which supports interactive edition, analysis, execution and debugging of specifications. The formal notation has been presented in [5], using a simple ad hoc case study, and the design and architecture of the PetShop tool have been presented in [6].

This paper presents in great detail neither the Cooperative Object formalism, nor the way it compares with other formal
approaches, since this has been done in previous work. Likewise, the paper does not give a complete formal description of the CORBA event service, since a separate paper is dedicated to it [4]. Rather, the goal of this paper is to prove by experiment that the OMG's style of specification of the CORBA services is not suited to guarantee that implementers will produce interoperable and substitutable implementations. Based on an experiments conducted on available implementations of the CORBA Event Service, we suggest that the specification be done with a formal description technique and that implementers be also provided with test cases.

The paper is organized as follows: section § 2 highlights the problems raised by the CORBA COSS based upon our analysis of the CORBA Event Service and makes a case for the use of formal methods. Section § 3 introduces the Cooperative Objects formalism, section §4 reports the testing of five implementations of the CORBA Event Service, section §5 presents our suggestion for providing implementers with a more reliable specification and tests cases, and finally section §6 concludes the paper.

2. PROBLEMS OF CORBA COSS

This section presents the problems encountered in the formal specification of a particular CORBA service, the CORBA Event Service. The CORBA Event Service has been designed for maximal versatility, but its structure and behavior are far from being trivial or intuitive. Therefore we present the specification process we went through before giving its formal specification.

2.1. Related work

CORBA services have been the subject of both industrial and research interest. However the problems they raise have been denounced by researchers dealing with implementation issues. Firstly, they can be incredibly difficult to implement if one tries the guidelines of the OMG, that ask for reuse of existing specifications. For example, this point is reported by Kleindienst [16] who experienced the problem while implementing the CORBA Persistent Object Service. Secondly, they are too general. This allows for flexibility of implementation; but at the same time makes them harder to use for specific applications. Therefore, implementers tend to increase their functionality. For example Schmidt [13] discusses the specification of a real-time event service built on top of the existing CORBA Event service. Thirdly, they are imprecise, which leads to the need to clarify their behavior.

Voluntary or involuntary specification

The OMG's specification of the CORBA Event Service is often voluntarily incomplete: In particular, the OMG’s document does not state the quality of service that a conforming implementation should have. For example, the OMG specification does not impose a reliable transmission of events through the event channel: events may be lost or duplicated. For example, the OMG's specification states that “Clearly, an implementation of an event channel that discards all events is not a useful implementation”. This is an indication that an implementation of the Event Service could be built on top of unreliable, “best effort” transport protocols such as UDP.

As we wish to provide an accurate behavioral specification of the event service, we need to go further than the COSS does: with respect to the OMG document, we will provide an overspecification of the event service, in that we specify an event service that does not lose nor duplicate the events. Obviously, any implementer of the event service chooses to implement a given quality of service, and is supposed to document this choice to his customers. In the actual implementations we have experienced, however, this information is hardly ever provided.

2.2. Management of ambiguities in the initial specification

As a starting point to our specification we had a document in natural language, in which we were expecting to find underspecifications (involuntary this time), ambiguities and even contradictions. The objective of the formal specification is precisely to detect such flaws in the initial specification and to correct them. In the process of our specification, we effectively detected ambiguities and underspecifications, but no contradictions.

When problems were detected, our attitude was:

- To document precisely the problem: if we detect the problem during the formal specification, it is reasonable to believe that implementers of CORBA Event Service will also detect it during implementation, and that they

![Figure 4-9](image-url)

Figure 1. Excerpt of the OMG’s specification of the CORBA Event Service ([20], p. 4-14)
will solve it. The fact that we identified a problem will guide us during the tests of the real implementations;

• To complete and precise the specification, choosing among alternatives the one that seems to us the most logical, the easiest and the most in conformity with the philosophy of the CORBA Event Service. Of course, it is an arbitrary choice, and other specifiers might not share our views.

Figure 1 displays the kind of underspecification we have met at several points in the COSS. The figure shows an “informal” state diagram of a proxy as defined in [20], p. 4-14, along with comments in natural language. This specification is largely incomplete: the state diagram is not complete as many potential state transitions are ignored (for example, what happens when the operation “disconnect” is invoked when the proxy is in the state disconnected?). On the other hand, the comments made in natural language are ambiguous (“operations are only valid in the connected state”: what does “valid” mean for an operation?).

3. THE CO FORMALISM

The CO formalism has been presented in earlier publications ([5]) where its rationale is detailed along with relevant related work. Behavioral specification of object systems is a research field per se [15]. Among the various formalisms proposed, StateCharts [12] is the behavioral formalism for the popular Unified Modeling Language (UML) notation [25]. However, the integration of StateCharts with object-oriented features such as dynamic instanciation and polymorphism still suffers some theoretical problems. Several attempts have been made to use process algebraic techniques in this domain, notably the work of [10], which provides an algebraic semantics to the various invocation modes of CORBA. The need to specify formally behavior along with interfaces has been recognized by [23;24], and also the need to integrate them in a methodology [7;8;22].

3.1. Overview of CO definitions

The Cooperative Objects (CO) [3] formalism is a dialect of object-structured, high-level Petri nets. The object-oriented approach provides the concepts necessary to define the structure of objects and their relationships in order to specify the system according to the principles of strong cohesion and weak coupling; the theory of Petri nets provides the specification of the behavior of objects and of their inter communications, so that it is possible to express both the concurrency between different objects and concurrency internal to an object.

As shown in Figure 2, a CO class is the combination of a CORBA-IDL interface (syntactic viewpoint) and of a high-level Petri net (the Object Control Structure, ObCS) that specifies the behavior of the interface (behavioral viewpoint). In the ObCS, tokens are allowed to contain references to other objects of the system. As the behavior of the objects is defined in terms of Petri nets, we obtain a set of Petri nets that are mutually referenced.

The complete formal definition of CO has been provided in previous papers [5], [28], and we will only recall informally their main features. The aim of this presentation is to allow the reader to understand the models in the following sections. CO can be considered as a mapping from CORBA-IDL to high-level Petri nets. More specifically:

• The type system (TypeSet) of the ObCS is defined in terms of CORBA-IDL. Variables having an IDL interface type are called references.

• Tokens are tuples of typed values. The arity of a token is the number of values it holds, and tokens of zero-arity are thus the “basic” tokens used in conventional Petri nets. We will call Token-type a tuple of types, describing the individual types of the values held by a token. Token-types are noted <Type_1 Type_n> or just <> to denote the Token-type of zero-arity tokens.

• Places are defined to hold tokens of a certain Token-type; thus all tokens stored in one place have the same Token-type and arity. A place holds a multiset of tokens; thus a given token may be present several times in the same place.

• Each arc is labeled by a tuple of variables, with a given multiplicity. The arity of an arc is the number of variables associated to it. The arity of an arc is necessarily the same as the arity of the Token-type of the place it is connected to, and the type of each variable is deduced from this Token-type. The multiplicity of an arc is the number of identical tokens that will be processed by the firing of a transition associated to this arc. The general form of an arc inscription is multiplicity*<v1… vn>.

• Transitions have a precondition (a Boolean expression of their input variables) and an action, which may use any operation allowed for the types of their input or output.
variables. The scope and type of each variable of an arc is local to the transition the arc connects to.

Enabling rule. A transition is enabled when:

• A substitution of its input variables to values stored in the tokens of its input places can be found
• The multiplicity of each substituted token in the input places is superior or equal to the multiplicity of the input arc,
• The precondition of the transition evaluates to true for the substitution.

Firing rule. The firing of a transition executes the transition’s action, computes new tokens and stores them in the output places of the transition. The formalism also supports two arc extensions [17]: test arcs and generalized inhibitor arcs.

3.2. Mapping CORBA-IDL to high-level Petri nets

We now illustrate an excerpt of the CO-CORBA integration by showing the CO class BufferSpec that specifies the behavior of the interface Buffer described in Figure 2. The first component of this integration is the mapping of services defined in the IDL to the ObCS of the CO class.

Mapping for services. Each service op defined in an IDL interface is mapped to two places in the ObCS net: a Service Input Port (SIP, labeled op), and a Service Output Port (SOP, labeled op). These two places are derived from the IDL, as follows:

• The Token-type of the SIP is the concatenation of the IDL types of all in and inout parameters of the service;
• The Token-type of the SOP is the concatenation of:
  • the IDL type of the result returned by the service (if any),
  • the list of the IDL types of all out and inout parameters of the service.

The CO formalism distinguishes three kinds of transitions:

• Invocation transitions, that are used to invoke CO instances. The action of an invocation transition is the invocation of a service offered by another object.
• Instanciation transitions that are used to create instances of a CO class;
• Exception transitions that interrupt the normal processing of a service.

The semantics of these three kinds of transitions is defined in terms of Petri nets, which allows to have a Petri net based semantics for a system of communicating objects, and not only for a single isolated object. A denotational semantics for the Cooperative Objects formalism is defined in [3]. This shows that object-oriented concepts such as instanciation, inheritance and dynamic binding can be formally represented within the framework of Petri nets theory. The basic principle of this denotational semantics is to construct a single, static high-level Petri net from the ObCS of all the classes involved in a specification.

The result of the invocation of one service is the deposit of one token (holding all in and inout parameters) in the SIP. The role of the ObCS net is to process this parameter token in some way, and eventually to deposit a result token (holding the result of the service, plus all out or inout parameters) in the SOP, thus completing the processing of the invocation. An invocation can be interrupted by the occurrence of an exception, which is modeled by an exception transition.

Figure 2 shows the CO class BufferSpec, which displays an ObCS and some textual annotations. These textual annotations are the following:

• The list of the interfaces that the CO class specifies (keyword specifies). In this case, the BufferSpec class specifies the Buffer interface;
• The description of the places’ token type: for example, the token type of Messages is string and corresponds to the data inserted in the buffer;
• The description of the transitions’ preconditions and actions, if any. Only the transitions with non-default precondition or action need to be stated in the textual part.

The ObCS complements the CORBA IDL part by providing a sensible behavioral specification. The Petri net provided specifies an unbounded buffer, where the order of message extraction is undeterministic. In this specification, operations put and get may occur concurrently (in terms of Petri nets, transition T1 and T2 are not conflicting). The net also describes a blocking semantics for the get operation: the client of the get operation is blocked until a message is available (i.e. until the transition T2 is enabled to fire). Alternative behaviors (such as bounded buffer, FIFO extraction or non-blocking operations) could be specified just as easily.

4. TESTING THE EVENT SERVICE

4.1. Overview

Among the fourteen services of the CORBA COSS [20], we have selected the CORBA Event Service for the following reasons:

• It is "self contained": it does not use any other service defined in the COSS, unlike many COS that rely on other COS for their definition or operation. For example, the Life Cycle Service uses the Naming Service. Besides, the CORBA Event Service does not rely on functionality defined by CORBA such as the Interface Repository. It is therefore possible to specify completely the CORBA Event Service without making any hypothesis on underlying services;
• It is complex due to its versatility. In particular, the connection to the event channel uses a non-trivial protocol between event consumers and event suppliers.
• Most ORB vendors implement it. This is not the case for the majority of COS: in fact the OMG has standardized some services for which there is not enough demand yet. Therefore the developers are not inclined to implementing all COS. Another interesting point about this service is that there is literature concerning its implementations ([9;26;27]) which gives insights on the design of implementations.

4.1.1. Presentation of the CORBA Event Service

The CORBA Event Service ([20], Chap. 4 pp. 1-33) provides "one-to-many", event-based communication, using an event channel that allows the decoupling event suppliers from event consumers. This goes beyond the synchronous and "one to one" client-server invocations supported by CORBA, where the client object must always hold a reference to the object to be invoked.
The CORBA Event Service defines two communication models according to who takes the initiative of the communication of events:

- **The push model** in which the consumer is passive and the supplier is active. The latter holds a reference to the consumer and invokes the consumer’s methods for event transmission;
- **The pull model** in which the consumer is active. It holds a reference to the supplier and requests events from the supplier.

Orthogonal to the two models of communication, two types of event channels are distinguished:

- the **generic event** channel that transports events of IDL type *any*. This type is self descriptive and allows the transportation of all kind of events by encapsulating them in a value of type IDL *any*;
- the **typed event channel** that transports events of a specific IDL type. That mode of transportation improves processing for consumers; it prevents from analyzing the type *any*.

4.1.2. Interfaces and roles

CORBA Event Service is “typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service”([20], p. 2-2). However, the term client is misleading in a client-server context because it infers that only the server is invoked. Therefore we use the term customer to refer to the clients of the CORBA Event Service because the event channel is a client or a server according to the interface it shows.

In this paper, for space reasons, we detail only the generic event channel in push model. Besides, the pull model is very symmetric to it, and would not add much to the presentation. As such, the service is defined in two modules and seven interfaces. These interfaces are described in Figure 3 in UML notation. CORBA Event Service defines three main roles:

- the channel administrators;
- the event supplier;
- the event consumer.

Channel administration occurs at two levels:

- **Management of event channel availability.** The EventChannel interface designates an event channel and is the lead administrator that decides on the availability of the event channel (operation *destroy*) and provides connection administrators for the event channel’s customers (operations *obtain_supplier_admin* and *obtain_consumer_admin*);
- **Management of connections to the event channel.** Administrators are subdivided in two groups according to the event channel customers they care about: the SupplierAdmin interface allows an event supplier to obtain a view of the channel that receives the events (operation *obtain_push_supplier*). The ConsumerAdmin interface allows an event consumer to obtain a view of the event channel that supplies events (operation *obtain_push_supplier*).

The PushSupplier interface designates a supplier of events. The operation *disconnect_push_supplier* allows the disconnection of the communication between a consumer and a supplier. The ProxyPushSupplier specializes the PushSupplier and is the view presented to a customer of the event channel that is a consumer of event.

The PushConsumer interface designates a consumer of events. The operation *disconnect_push_consumer* allows to terminate the communication between a consumer and a supplier. The ProxyPushConsumer specializes the PushConsumer interface and is the view of the channel presented to a customers of the event channel that is a supplier of events.

4.1.3. A scenario of the use of the event channel

Figure 4 shows a typical scenario of the use of the event channel in push model. An object (Receptor) wishes to subscribe to an event channel and play the role of a consumer of events. Another object (Emitter) wishes to subscribe and play the role of a supplier of events for the same event channel. The objects Emitter and Receptor connect to the event channel independently and without informing each other. The event channel plays the role of a buffer between the consumer and the supplier, and connects each of its customers (Emitter and Receptor) to a proxy, by following a well defined, four step, connection protocol:

1. Emitter and Receptor obtain a reference to the event channel (for example, by using a name Server to obtain a reference to an EventChannel object named “Channel”). The reference found is denoted EC;
2. Emitter and Receptor ask EC the references to an appropriate administrator and obtain respectively a reference to an administrator of suppliers (SAd, by invoking *for_suppliers*) and an administrator of consumers (CAd, by invoking *for_consumers*);
3. SAd returns a ProxyPushConsumer (PPushCons) to Emitter upon invocation of *obtain_push_consumer* and
4. The connection is established between the event channel and its customers after the invocation of connect_push_supplier on PPushCons by Emitter and connect_push_consumer on PPushSup by Receptor. The latter invocation requires one parameter that is the reference to a consumer (consRef), delegated by Receptor, so that the proxy supplier PPushSup can invoke the operation push when events are available. The communication takes place by invoking the operations push on the consumers at the entrance of the channel (left side of Figure 4) due to the supplier supRef, delegated by Emitter, and at the exit of the channel due to PPushSup (right side of Figure 4). The communication can be stopped by invoking the disconnection operations provided by the PushConsumer and PushSupplier interfaces of supRef, consRef, PPushCons and PPushSup, or by destroying the event channel EC.

The transportation of events from proxy to proxy is not specified by the OMG and is left out to implementers. We modeled that transportation with an operation handleEvent in the case of the push model. When supRef transmits the event myEvent as parameter of the push operation to PPushCons, the latter invokes handleEvent on ChannelServer in order to route the message to all ProxyPushSupplier objects, like PPushSup. Lastly, PPushSup invokes the operation push on consRef with the parameter myEvent. We precise the object which supports handleEvent later.

4.2. CO-based specification of the ProxyPushConsumer interface

The CORBA Event Service has been completely specified using CO, faithfully to the OMG specification, in [4], and only a short excerpt is provided here, in order to give the flavor of CO-based specification. This specification was performed using the PetShop environment [6], which allowed us to produce an executable formal specification of the CORBA Event Service.

This section presents the CO class ProxyPushConsumerSpec that models the behavior of the ProxyPushConsumer interface.

```co
module CosEventChannelAdmin {
    interface ProxyPushConsumer{
        void push( in any data);
        void disconnect_push_consumer()
            raises Disconnected;
        void connect_push_supplier
            ( in PushSupplier s);
    }
}
```

Figure 5. ProxyPushConsumer interface

Figure 5 shows the interface ProxyPushConsumer defined by the OMG’s document. The proxy has three operations, which provide connection to a supplier (connect_push_supplier), entrance of events into the event channel (push) and disconnection of the proxy (disconnect_push_consumer).

```co
class ProxyPushConsumerSpec
    specifies ProxyPushConsumer{
        place unconnected <ProxyConsumerList> = (1*<proxy>);
        place router <Router> = (1*<router>);
        place connected <ProxyConsumerList>;
        place destroyed <>;
        place events <any>;
        transition connect {
            action {
                1.add(self);
            }
        }
    }
```
by a call to its operation handleEvent (invocation transition handleEvent). If the proxy has not been connected yet, the operation raises the exception Disconnected (transition t4).

Once the proxy has been destroyed (transition disconnect) all operations raise the exception OBJECT_NOT_EXIST. When the operation disconnect_push_consumer is invoked and the proxy has not been connected yet the proxy is nonetheless destroyed (transition disconnectEarly).

4.3. Functional test cases derivation
Following the specification of the CORBA Event Service, we have decided to compare a few implementations. The goal of these tests needs some explanation: we do not want to test whether these implementations conform to our specification; but to check whether independently made implementations are compatible.

In fact, during our formal specification, we have noticed and removed ambiguities and underspecifications in the OMG’s document. Therefore, we did expect implementers to have also noticed them. However, the fact that we did a formal specification led us to notice exhaustively the underspecifications. Then we were able to define test cases that aim at determining how the implementers solved the same underspecifications.

4.3.1. Test cases
The inaccuracies that we detected in the OMG’s document often concerned boundary conditions for objects, and more specially the relation between service invocation and the life cycle of objects. In this paper, we limit our discussion to some of the problems that concern the proxy objects (ProxyPushConsumer), and for which the OMG’s document give no answer.

One difficulty in testing characteristics of the implementations is that we are testing a system and not individual units in isolation: we are involved in higher order testing. Therefore, we concentrated on testing the functionalities of the interfaces as obtained using the scenario of §4.1.3.

4.3.1.1. Test cases
For the scope of this paper we selected four test cases of the ProxyPushConsumer. The first three tests defined in this section are compatibility tests, the last one tests conformance with the OMG’s document:

- Proxies have operations for connection (e.g. connect_push_supplier) and disconnection (e.g. disconnect_push_consumer). What is the effect of the sequence of invocations (disconnect, connect) on the proxy? In our formal specification, we decided that the disconnect operation would systematically put the proxy in the destroyed state unless it is already in that state. The test made to evaluate the behavior of the implementations and is referred to as Test 1.

- Is it possible to invoke more than once the disconnect operation on a proxy? Will the invocation sequence (connect, disconnect*) raise exceptions or succeed? In our formal specification, we chose that only the first call to disconnect would succeed and subsequent invocations of disconnect would raise the exception OBJECT_NOT_EXIST. The test made to evaluate this behavior in implementations is referred to as Test 2.

- What is the effect of the sequence of invocations (connect, disconnect, push*), on a ProxyPushConsumer? In our formal specification, we chose to systematically raise the exception OBJECT_NOT_EXIST for all invocations of the
push operation following a call to disconnect operation. The test made to evaluate this behavior in implementations is referred to as Test 3.

- We also decided to test certain points of the specification for conformity because their interpretation had seemed difficult during the first reading of the OMG's specification. For instance, when a PushSupplier connects to a ProxyPushConsumer (by invoking operation connect_push_supplier), the OMG's document specifies that the PushSupplier does not need to give its reference as parameter and can supply a nil reference. In theory, the reference supplied at connection is not necessary for the propagation of events. We wanted to know whether the implementation accepts the nil reference. The test made to evaluate this behavior in implementations is referred to as Test 4. As this is a conformance test, we define the acceptable results for the oracle: the test is passed if the proxy accepts the data that is pushed.

4.3.1.2. Test sequences
Table 1 presents the test sequences corresponding to the four tests defined in the previous section. They are expressed using the variable names defined in the scenario of §4.1.3.

4.3.1.3. Test programs
We have manually programmed the test cases that allow us to determine how the implementers of the CORBA Event Service had solved the ambiguities discussed above. The objects involved in the test conform to the scenario of §4.1.3. The objects used are the following:

Receptor. This is a consumer customer of the event channel that allows us to check that the channel outputs events.

Test1. It is the class responsible for executing Test1. Its instances play the role of Emitter in §4.1.3. Figure 8 shows excerpts of the main method of class Test1. At line 3 the Emitter gets a reference to an object that can find the event channel, at line 6, the Emitter gets the reference of the proxy, at line 9, the proxy is disconnected, and at line 10 the Emitter tries to connect to the proxy. Finally after line 12, the emitter tries to send events into the event channel at the users command.

Test2, Test3 and Test 4 are the classes responsible for executing Test2, Test3 and Test4.

CorbaEnvironment. It is the class that is responsible for getting the reference of the EventChannel object and that encapsulates the access to the proxies. This class allows us to hide the technical characteristics relating to each implementation.

Table 1. Test sequences on the proxy PPushCons

<table>
<thead>
<tr>
<th>Test 1</th>
<th>disconnect_push_consumer(); connect_push_supplier(supRef)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 2</td>
<td>connect_push_supplier(supRef); disconnect_push_consumer(); disconnect_push_consumer()</td>
</tr>
<tr>
<td>Test 3</td>
<td>connect_push_supplier(supRef); disconnect_push_consumer(); push(myEvent)</td>
</tr>
<tr>
<td>Test 4</td>
<td>connect_push_supplier(nil); push(myEvent)</td>
</tr>
</tbody>
</table>

Figure 7. ObCS of class ProxyPushConsumerSpec
4.3.2. Selected implementations

We made two testing campaigns corresponding to a major change in CORBA (the introduction of the Portable Object Adapter):

1. Inprise's Event Service (http://www.inprise.com), bundled in Visibroker 3.4;
2. Exemplar Development's Event Service (http://www.exemplardev.com) that no longer exists;
3. Prism technologies' Event Service (http://www.prismtechnologies.com), included in Open Fusion version beta;
4. DSTC's COS Notification version 1.0.2 (http://www.dstc.com). DSTC is an Australian research center that developed an implementation of COS Notification [21] that works with Visibroker. The COS Notification is an extended version of the CORBA Event Service that contains all its functionalities.

In the second campaign we updated our previous study for Visibroker 4.0 and added another implementation, Orbacus 4.0, (http://www.ooc.com).

Table 2. Conditions of test

<table>
<thead>
<tr>
<th></th>
<th>First campaign</th>
<th>Second campaign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Intel Pentium II 350 MHz</td>
<td>Intel Pentium II 350 MHz</td>
</tr>
<tr>
<td>Language</td>
<td>Java (JDK 1.1.8)</td>
<td>Java (JDK-1.2-V)</td>
</tr>
<tr>
<td>OS</td>
<td>Windows NT 4.0</td>
<td>Windows NT 4.0</td>
</tr>
</tbody>
</table>

Table 2 presents the conditions of tests. Each implementation was tested in isolation of the other implementations. The test programs are exactly the same for all tested implementations.

4.3.3. Test results

The tests made on the four selected implementations of the CORBA Event Service are not exhaustive. However the results, summarized in Figure 9, are striking: especially, there is no agreement among the four implementations concerning the four selected tests.

Concerning Test1 and Test2 some implementations accept the sequence of invocation and function normally afterwards. Other implementations raise an exception, which is the choice made in our formal specification.

Concerning Test3, all implementations raise exceptions, but they diverge on the exception that is raised. The implementers of DSTC chose to raise the exception DISCONNECTED. We decided to raise the exception OBJECT_NOT_EXIST, because it matched the spirit of the OMG's specification that states that the Event Channel can free the resources allocated at the instanciation of the proxy when it has been disconnected.

Test4 highlights the most striking dysfunction: two implementations clearly do not conform to the OMG's specification. Moreover these two implementations do not fail in the same manner.

Dysfunction in a composition scenario. The OMG's document gives an event channel composition scenario in ([20], Chap. 4 p. 14) in which "an external agent [...] would compose two channels by obtaining a proxy supplier from one and a proxy consumer from the other, and passing each of them a reference to the other as part of their connect operation." Using Orbacus and Visibroker, for example, the "external agent" will experience failure because the events propagated by the Orbacus channel server never reach the consumer customers of the Visibroker event channel.

What can we conclude from this experiment?

- For instance, a client that invokes repeatedly the operation disconnect_push_consumer (for instance during an uncontrolled disconnection phase) can function perfectly well with DSTC's implementation, but will fail with VisiBroker's implementation.
- None of these implementations makes explicit the additional specification introduced by the implementers to resolve the underspecifications of the OMG's document. In a practice, this means that a CORBA Event Service user should make tests similar to ours to know how an implementation functions.
- Some of the implementations violate the most explicit specifications of the OMG. One explanation might be the complexity of the specification, but another reason is the absence of test cases for validation that are not provided by the OMG together with the specification. These test cases would allow establishing experimentally the conformity of an implementation.

This study has given encouraging results for the use of formal description techniques and the applicability of the Cooperative Objects formalism:

- During the formal specification phase, we were able to detect ambiguities and inaccuracies in the OMG's document. This is not surprising: the discipline imposed by the use of a formal notation forces the specifier to question the OMG's document and answer these questions precisely and non-ambiguously. Our formal specification deserves no special credit for that point; using another formal notation would probably have been as effective for the purpose of detection. The interesting part is that the problems are not theoretical ones: all implementers faced them, but, unfortunately, they produced incompatible implementations. This latter point justifies the use of a formal notation adapted to CORBA.
The tested vendors’ implementations, based on the OMG’s specifications, exhibit many behavioral incompatibilities. We summarize the main results of our test experiments:

- **Incompatibility with respect to Liskov’s ([18]) substitutability principle:** if a customer works correctly with one implementation, it may well work incorrectly with another implementation because servers do not react in the same way to invocations or sequences of invocations;

- **Violations of OMG’s specifications:** some implementations do not respect explicit specifications. It may result from the complexity of the specification (overlooking a point). We can point out that the OMG does not give a set of minimal test cases to test implementations.

- **No explicit indication of the behavior of the implementations:** the implementation choices that were made to complete and precise the OMG’s specifications are not documented and result in the customers having to test the implementation like we did to determine the behavior of the CORBA Event Service they want to use. We therefore produced some reverse specifications based on our test campaign.

5. PROPOSAL

The previous sections clearly show that the current practice of using informal specifications leads to untrustable and non-interoperable implementations of CORBA services. A promising research direction would be to combine formal description techniques and testing techniques in order to be able to test the functionality of CORBA systems [11]: two distinct roles can be defined in the development process of a CORBA based system:

- The “principal contractor”, who specifies the functionality of the system to be built. The principal contractor should provide the description of the system in CORBA-IDL, along with a formal description suited to CORBA (see [5] for the detailed requirements). Moreover, the principal contractor should provide functional test cases that will allow to validate the implementation. These test cases should be generated automatically or semi-automatically from the formal specification (Figure 10).

- The “implementer” who develops the software that must conform to the specification. The implementer should be provided with a formal specification that is non-ambiguous, and also with test cases that allow to validate the conformity of the implementation.

Currently, the OMG plays the role of the principal contractor when it publishes the CORBA COSS, and the ORB vendors play the role of the implementers when they provide with the implementations of the services described in the CORBA COSS. Unfortunately, the test cases are missing, and this leads to serious dysfunction as shown in §4.3.3.

Our point of view is that an approach based on test case generation seems more realistic in the short run than approaches based on code generation form a formal specification. As far as our approach is concerned, code generation seems neither possible nor desirable: the CO-based specification is abstract and devoid of implementation contingencies. Implementers are left free to choose, and must express their creativity, while developing an implementation. Moreover, the specification does not include requirements such as robustness, performance, persistence, and also it does not take into account technological constraints such as the choice of a database system. All these contingencies must be taken into account by the implementers along with the behavioral specification while constructing the implementation.
5.1. Benefits

5.1.1. Model analysis

Some formal description techniques, as is the case with Petri nets, allow us to perform several kinds of analyses on the specifications. We give a short overview of the kind of properties obtained for a CO class with our tool and the way they can be interpreted.

Figure 7 highlights in bold a P-invariant for the ProxyPushConsumerSpec.

\[
\text{Marking(unconnected)} + \text{Marking(connected)} + \text{Marking(destroyed)} = 1
\]

It is obtained by mathematical analysis of the Petri net. It means that whatever the actions performed by a ProxyPushConsumerSpec object, there will be one and only one token in the three places. From the point of view of the specifier, the invariant is interpreted as a desirable liveness property. The system has an initial state (unconnected is marked), then evolves to an operational state (connected is marked) and finally terminates (destroyed is marked): the tool indicates the transitions responsible for the flow of tokens from unconnected to connected (transition connect) and from connected to destroyed (transition disconnect). In the model we can see that the three transitions (connect, T1 and T2) are all related to the same SIP, connect_push_consumer. Due to the place invariant these three transitions are in mutual exclusion thus guaranteeing both the fact that at least one transition is available (and thus able to process the service invocation) and that only one of these transition is available at a time (the choice is deterministic).

5.1.2. Test cases generation

The benefit of some formal description techniques is the fact that they allow generation of tests, which allow for the immediate testing of implementations. A promising research direction consists in combining formal specification techniques and testing techniques to derive functional tests of components under development [11]. We are working at integrating the works already done at LRI and EPFL [1;2;14]. The theoretical problems stem from the inherent non-determinism in the models, the concurrency of objects, the definition of a parallel test driver that emulates the distributed environment, the oracle problem… We have so far outlined the methodology and hope to reach results in the time frame of the SERPICO project.

5.1.3. Analysis of object-oriented features.

We wish to use Petri net analysis techniques not only to prove properties on an isolated object, but also to analyze constructs specific to object-oriented systems. For example, when two IDL interfaces are related through inheritance, some form of behavioral inheritance needs to be respected for CO classes that specify these interfaces. The work presented in [29] is a useful starting-point for us, the most relevant notion in the context of CORBA appearing to be the one based on hiding the new methods introduced in subclasses.

5.2. Drawbacks

The main drawback is gaining acceptance for a formal description technique (preferably CO of course). Therefore, the usability of the formalism has to be considered a top priority. In the case of CO, we have built a tool called PetShop [6] that emphasizes interactivity and that seamlessly integrates into the CORBA development process.

6. CONCLUSION

This paper has shown that the OMG's style of specification is not appropriate for specifying object services intended to provide interoperable and substitutable implementations. Although a number of researchers and practitioners were already suspecting this fact, this is to our knowledge the first time that a formal and experimental proof of this is provided.
overcome the lack of formal specifications. Our experiment shows that such practices become less and less viable considering the level of complexity reached by current middleware. We have underlined the need to provide some behavioral formal description technique that suits the CORBA object model. We have shown through the CORBA Event Service example that the use of a behavioral modeling language can improve the life-cycle development of CORBA systems.

Our proposal is based on a formal description technique called Cooperative Objects. Although formal specification techniques are sometimes criticized on the grounds that they only deal properly with small-scale examples, we believe that this criticism does not apply to our formalism. The Cooperative Objects-based formal specification of the CORBA Event Service demonstrates that our approach scales well and allows one to tackle real-life specifications such as the ones of the CORBA COSS. The Cooperative Objects formalism allowed us to specify a real service due to its expressiveness in describing complex behaviors, including concurrency and synchronization.

During our formal specification of the CORBA Event Service, we have detected several inaccuracies and ambiguities in the OMG's document. Besides, the tests conducted on five implementations of this service confirm our remarks since the tests conclude to incompatibilities between these implementations. These observations justify the need for formal description techniques that describe the behavior so that implementers can refer to complete, precise and non-ambiguous specifications.

Furthermore, we make a general suggestion that a specification approach intended for implementers should contain two parts: on the one hand, a formal behavioral specification describing precisely, completely and non-ambiguously the expected behavior of implementations, and on the other hand test cases allowing to validate the implementations produced by implementers.

Finally, our tool, supporting the Cooperative Objects formalism, has been particularly useful as it provided assistance to detect mistakes. In turn, the execution of the formal specification was very important to get a feel of what was happening and allowed us to clarify notions like exception handling and to clarify the frontier between concepts relating to the formalism and those relating to tooling the formalism.

7. ACKNOWLEDGEMENTS
The work of David Navarre is funded by ESPRIT Reactive LTR project n° 24963, MEFISTO.

The work of Ousmane Sy is funded by FT R&D (formerly CNET), the Research Center of France Telecom, under the SERPICO project, grant number 98 1B 059.

We would also like to thank for her grammatical and spelling corrections, Aïssatou Sy-Wonyu, and especially for their constructive remarks Chihab Hanachi, Eric Andonoff and Lotfi Admane.

8. REFERENCES


