A FORMAL SPECIFICATION OF THE CORBA EVENT SERVICE

Rémi Bastide, Ousmane Sy, David Navarre and Philippe Palanque
LIHS, Université Toulouse 1, Place Anatole France, F-31042 Toulouse CEDEX, France
{bastide, sy, navarre, palanque}@univ-tlse1.fr

Abstract: CORBA is a standard proposed by the Object Management Group (OMG) that promotes interoperability between distributed object systems. Following the standardization of this object-oriented middleware, the OMG has specified a set of Common Object Services (COS) that are meant to serve as the building blocks of distributed CORBA applications. The COSes are specified using CORBA Interface Definition Language (IDL), that describes the syntactic aspects of services supported by remote objects. However, CORBA-IDL does not support specification of the behaviour of objects in an abstract and formal way, and behavioural specification is mostly provided in plain English. To overcome this problem, we have proposed a formal description technique (Cooperative Objects) based on high-level Petri nets, and developed a software support environment. The goal of this paper is to demonstrate that our approach is suited to the formal specification of typical CORBA COS, and presents a Cooperative Object model of the CORBA event service, a COS that provides asynchronous, one-to-many communication between objects. The advantages of dealing with a tool-supported, executable formal notation are detailed, as well as the results that can be obtained through Petri net analysis techniques.

Key words: Distributed systems, behavioural specification, CORBA, high-level Petri nets.

1. INTRODUCTION

CORBA [15] (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 provides the middleware necessary to locate server objects on the network, and to route invocations between clients and servers regardless of the programming language they are written in. 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.

1.1 The CORBA Services

The OMG has specified a set of common object services (COS) in a huge document: the CORBA COSS (Common Object Services Specifications) [16]. The 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 standardised solutions to commonly encountered problems. Among typical COS are the CORBA Naming Service (that allows to retrieve a remote object reference by providing a symbolic name) and the CORBA Event Service (which allows event-based asynchronous communication using events).

The specification of the CORBA services 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 specification of the behaviour). The COSes do not contain an integrated behavioural description and, as a result, are sometimes inaccurate and ambiguous.

Moreover, the COSS are sometimes 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 flexibility has in several occasions gone so far as to hinder interoperability between distinct implementations.

Furthermore, the OMG does not provide test suites that would allow checking the conformity of implementations. Therefore, developers working in companies producing implementations of the COS have to overcome those inaccuracies and underspecifications using their own experience. As a summary, despite the clear objectives of CORBA, the specification leads to potential interoperability problems.

1.2 Formal specification of CORBA systems

In the framework of the SERPICO project, we have developed a formal notation suited to the behavioural specification of CORBA systems: Cooperative Objects (CO) [7], [8]. CO are an object-oriented dialect of high-level Petri nets that allow describing the behaviour of a collection of

distributed objects. The usefulness of combining Petri nets and objects has been recognised by several authors [1;2;10]. The CO notation is supported by a software tool called PetShop [9] that allows for the interactive editing, execution and analysis of specifications.

The present paper aims at showing that the Cooperative Objects formalism can provide a suitable solution to the problem of behavioural specification of distributed objects, in the context of CORBA. Moreover, we want to show that this formal specification technique is scalable enough to cope with large specifications such as the CORBA Event Service. The paper is organised as follows: we present in section 2 the Cooperative Objects formalism and how it fully supports the CORBA model. Section 3 gives an overview of the CORBA Event Service, that is formally specified in section 4. Section 4 also shortly describes the kind of model analysis and the PetShop environment that supports CORBA system modelling. Section 5 describes the ongoing work and concludes the paper.

2. THE CO FORMALISM

The Cooperative Objects (CO) [5] 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 behaviour of objects and of their inter communications, so that we can express both the concurrency between different objects and the internal concurrency of an object.

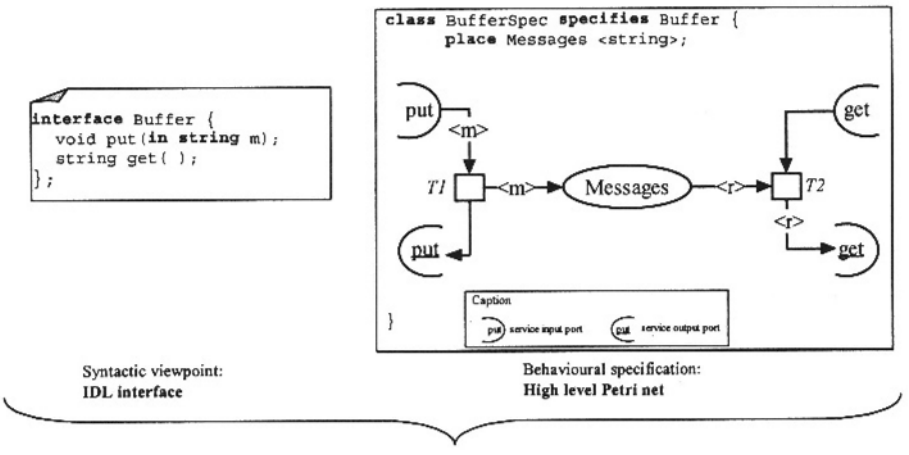


Figure 1. The Cooperative Object class BufferSpec

A CO class is the combination of a CORBA-IDL interface and of a high-level Petri net (the Object Control Structure, ObCS) that specifies the behaviour of the interface (*Figure 1*). In the ObCS, tokens are allowed to contain references to other objects of the system. As the behaviour of the objects is defined in terms of Petri nets, we obtain a set of Petri nets that are mutually referenced.

2.1 Overview of definitions of COs

The complete formal definition of CO has been provided in previous publications [8], [17], 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 labelled 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 [13]: test arcs and generalised inhibitor arcs.

2.2 Mapping CORBA-IDL to high-level Petri nets

We now illustrate the CO-CORBA integration by showing the CO class *BufferSpec* that specifies the behaviour of the interface *Buffer,* described in *Figure 1*.

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, labelled op), and a Service Output Port (SOP, labelled 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:
  a) the IDL type of the result returned by the service (if any);
  b) the list of the IDL types of all *out* and *inout* parameters of the service.

The invocation of one service results in one token holding all *in* and *inout* parameters being deposited in its SIP. The role of the ObCS net is to process this parameter token in some way, and eventually 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 modelled by a special exception transition.

Figure 1 shows the CO class *BufferSpec* that displays an ObCS and some textual annotations. These textual annotations are:

– 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 in *Figure 1* complements the CORBA IDL part by providing a sensible behavioural specification. The Petri net provided specifies an unbounded buffer, where the order of message extraction is done in a non-deterministic way. 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 behaviours (such as bounded buffer, FIFO extraction or non-blocking operations) could be specified just as easily.

The CO formalism distinguishes three kinds of transitions. All are used in the specification of the CORBA Event Service in §4:

- 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;
- Instantiation 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 [5], 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 [5]. This shows that object-oriented concepts such as instantiation, 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.

3. OVERVIEW OF THE CORBA EVENT SERVICE

Formal specification techniques are sometimes criticized on the grounds that they deal properly only with small-scale examples (such as the Buffer illustrated in *Figure 1*). The goal of this paper is to demonstrate that our approach scales well and allows one to tackle real-life specifications such as the COSS.

Among the fourteen services of the CORBA COSS [16], 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 COSes that rely on other COSes for their definition or operation. For example, the Life Cycle Service uses the Naming Service. Besides, the CORBA Event Service does not rely on functionalities defined by CORBA like 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 COSes: in fact the OMG has standardised some services for which there is not enough demand yet. Therefore the developers are not inclined to implementing all COSes.

3.1 Presentation of the CORBA Event Service

The CORBA Event Service ([16], Chap. 4 pp1-33) provides asynchronous, "one-to-many", event-based communication, using an event channel that allows 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 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 a self descriptive type 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 allows efficient processing for consumers that no more have to analyse a value of type *any*.

3.2 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"([16], 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, we detail only the generic event channel in push model. As such, the service is defined in two modules and seven interfaces. These interfaces are described in *Figure 2* and *Figure 3* using UML notation.

CORBA Event Service defines three main roles:

1. The channel administrators;
2. The event supplier;
3. The event consumer.

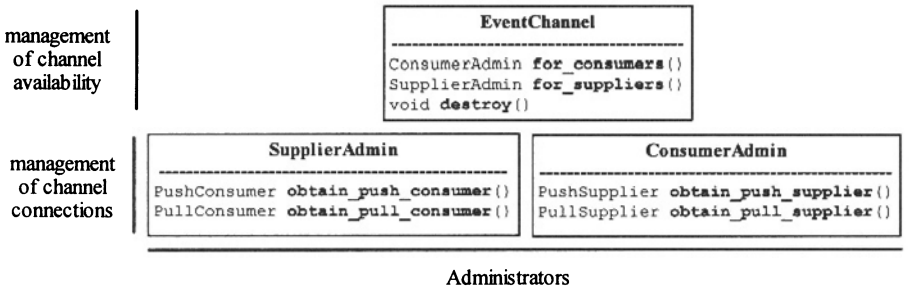


Figure 2. Interfaces of the channel administrators in the push model

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 customers of the event channel (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 the supplying customers of the channel to obtain a view of the channel that receives the events produced by the customers (operation *obtain_push_consumer*). The *ConsumerAdmin* interface allows the consuming customers to obtain a view of the event channel that supplies events to the customers (operation *obtain_push_supplier*).

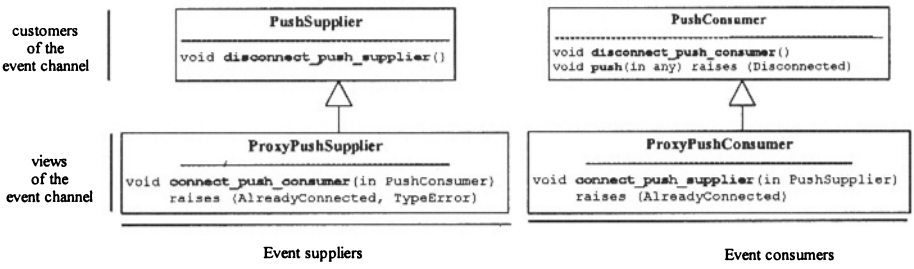


Figure 3. Interfaces of the event suppliers and consumers in the push model

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* specialises the *PushSupplier* and is the view presented to a customers of the event channel that is a consumer of event.

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

3.3 A scenario of use of the event channel

Figure 4 shows a typical scenario of 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 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 steps, connection protocol:

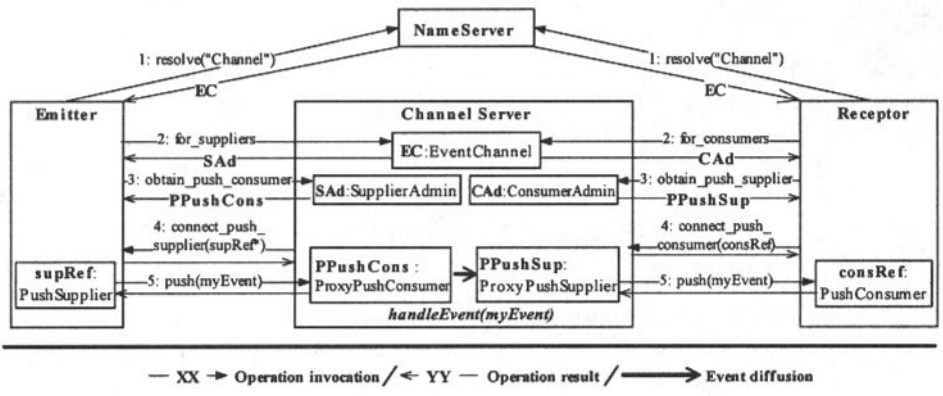


Figure 4. Scenario of use of the event channel

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 *CAd* returns a proxy supplier (*PPushSup*) to Receptor upon invocation of *obtain_push_supplier*;
4. The connection is established between the event channel and its customers after the invocation of the operations *connect_push_supplier* on *PPushCons* by Emitter and *connect_push_consummer* 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 modelled 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. CO-BASED FORMAL SPECIFICATION

The CORBA Event Service has been designed for maximal versatility, but its structure and behaviour are far from being trivial and intuitive. Therefore, we first present specification process we went through before giving its formal specification.

4.1 Voluntary or involuntary specification

The OMG's specification of the CORBA Event Service is voluntarily incomplete: In particular it does not state the quality of service 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. Paradoxically, the OMG's specification says that "*Clearly, an implementation of an event channel that discards all events is not a useful implementation*".

As we wish to provide an accurate behavioural 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.

4.2 Management of ambiguities in the initial specification

As the starting point of our specification is a document in natural language, we were expecting to find underspecifications (involuntary ones 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 we detected no contradiction.

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 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 5, below, displays the type of underspecification we have met in the COSS. The figure shows an "informal" state diagram of a proxy as defined in [16], 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?).

Proxies are in one of three states: *disconnected*, *connected* or *destroyed*. Figure 4-9 gives a state diagram for a proxy. The nodes of the diagram are the states and the edges are labelled with the operations that change the state of the proxy. `Push/pull` operations are only valid in the *connected* state.

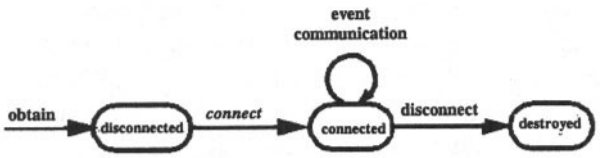


Figure 4-9 State diagram of a proxy

Figure 5. Excerpt of the OMG's specification of the CORBA Event Service ([16], p.4-14)

```
module CosEventChannelAdmin {
// original OMG specification of this module fits here
   interface Router{
      void handleEvent(in any data);
   };
   interface EventChannelRouter : EventChannel, SupplierAdmin,
      ConsumerAdmin, Router {};
   interface ProxyPushSupplierExt : ProxyPushSupplier{
      void deliver(in any data);
   };
   interface ProxyConsumerList{
      void add(in ProxyPushConsumer s);
      void remove(in ProxyPushConsumer s);
      void disconnectForAll();
   };
   interface ProxySupplierList{
      void add(in ProxyPushSupplier s);
      void remove(in ProxyPushSupplier s);
      void disconnectForAll();
      void pushForAll(in any data);
   };
};
```

Figure 6. Interfaces added to CORBA Event Service

In order to deal with the underspecifications stated above, we specify the behaviour of an extended CORBA Event Service in which new interfaces are added to the module *CosEventChannelAdmin* that defines the event channel (c.f. *Figure 6*). The new interfaces are defined using interface inheritance to extend the interfaces previously described in §3.1 in order to specify a *useful* CORBA Event Service.

We need essentially to specify how the channel routes the events from producers to consumers, and to describe how the service manages the proxy objects that it creates dynamically during its operation. We therefore introduce a *Router* interface and specialise *EventChannel* into *EventChannelRouter*. Thus the operation *handleEvent*, referred to in §3.3 is devoted to an *EventChannelRouter*. The *ProxyPushSupplier* is also specialised in *ProxyPushSupplierExt* that receives events (operation *deliver*) from the R*outer* and delivers them to its customer.

4.3 CO-based specification

The CORBA Event Service has been completely specified using CO, faithfully to the OMG specification, in [19]. This paper only presents the push model, for space reasons and also because the pull model is very symmetric to it, and would not add much to the presentation. We present the five CO classes that model the behaviour of:

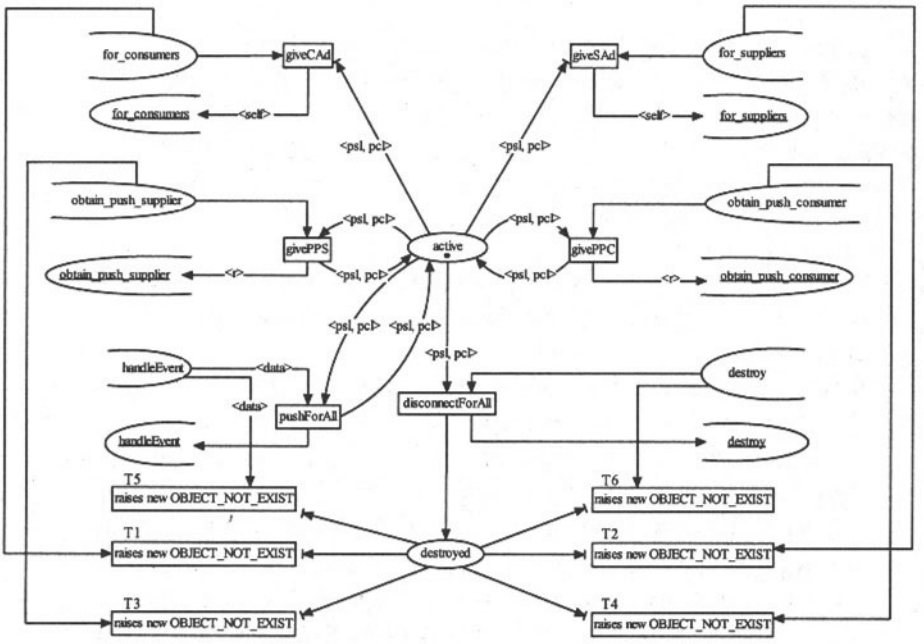
- The main interfaces of the event channel: *EventChannelRouterSpec*, *ProxyPushConsumerSpec* and *ProxyPushSupplierExtSpec*; and
- Two helper classes *ProxySupplierListSpec* and *ProxyConsumerListSpec*.

4.3.1 Class EventChannelRouterSpec

Figure 7, below, shows the class *EventChannelRouterSpec* that specifies the behaviour of the interface *EventChannelRouter* defined in our extension of the OMG's specification. Place *active* is initially marked with one token {<psl, pcl>} that holds references to two lists of proxies: psl is a *ProxySupplierListSpec* (*Figure 9*) and pcl is a *ProxyConsumerListSpec* (*Figure 8*). The event channel is ready for operation and can distribute the reference to its administrators in accordance to the scenario described in §3.3. When the operations *for_consumers* (top left sub-net) and *for_suppliers* (top right sub-net) are invoked, transitions *giveCAd* and *giveSAd* return the event channel.

When a customer requests a *ProxyPushSupplier* through operation *obtain_push_supplier* (middle left), the **instantiation transition** *givePPS* returns a new *ProxyPushSupplierSpec* object. When a customer requests a *ProxyPushConsumer* through operation *obtain_push_consumer* (middle right), the instantiation transition *givePPC* returns a reference of a new *ProxyPushConsumerSpec* object. For the sake of readability, the ObCS of *Figure 7* does not represent the operations corresponding to the pull model: *obtain_pull_supplier* and *obtain_pull_supplier*. These operations return a *nil* object reference so that customers of the event channel cannot work in the pull model.

```
class EventChannelRouterSpec specifies EventChannelRouter{
place active <ProxySupplierList, ProxyConsumerList> =
     {new ProxySupplierListSpec(),new ProxyConsumerListSpec()};
place destroyed <>;
transition givePPS{
     action {
       r = new ProxyPushSupplierSpec(psl);
     }
}
transition givePPC{
     action {
       r = new ProxyPushConsumerSpec(self, pcl);
     }
}
transition pushForAll {
     action {
       psl.pushForAll();
     }
}
transition disconnectForAll {
     action {
       pcl.disconnectForAll();
       psl.disconnectForAll();
     }
}
transition t1, t2, t3, t4, t5, t6 {
     action {
       raise new OBJECT_NOT_EXIST();
     }
}
```




```
}
```

Figure 7. Class EventChannelRouterSpec

The ObCS of *Figure 7* also describes accurately how the operation *handleEvent* (bottom left sub-net) handles the routing of events to all customers of the event channel. Firing the **invocation transition** *pushForAll* does the following:

- It removes an event token from the SIP of *handleEvent*, that is the routed event;
- It removes the token in place *active*. Therefore, (1) two events cannot be routed simultaneously because the transition *pushForAll* is disabled; (2) no new proxies are added while routing because transitions *givePPS* and *givePPC* are also disabled; (3) the channel cannot be destroyed while routing because transition *disconnectForAll* is disabled;
- The action part is executed. The operation *pushForAll* (bottom left on *Figure 9*) is requested on the *ProxySupplierList,* psl.

The firing of the invocation transition *pushForAll* ends when the request on the *ProxySupplierList,* psl, is completed. A token <psl,pcl> is then deposited in the place *active* and a token <> is deposited in the SOP of *handleEvent*, which completes the invocation of operation *handleEvent*.

Finally, the ObCS of *Figure 7* precises the underspecified behaviour of the *EventChannel* when the operation *destroy* is requested. The invocation is completed when the administrators have been destroyed and all the proxies have been disconnected. This choice is implied by the fact that *destroy* is supposed to destroy the event channel; acting differently would allow to use the event channel after its destruction.

The event channel switches to inactivity by firing the invocation transition *disconnectForAll*, that requests the operation *disconnectForAll* from the two proxy lists, psl and pcl. As soon as these requests are completed, the proxy switches to the state *destroyed.*

The ObCS of *Figure 7* also precises the behaviour of all the operations once *destroy* is requested. All operations will raise the CORBA exception OBJECT_NOT-EXIST. Thus customers will realize that the event channel no more exists. The transitions *t1, t2, t3, t4, t5* and *t6* are **exception transitions**: they are labelled with the type of the exception they raise.

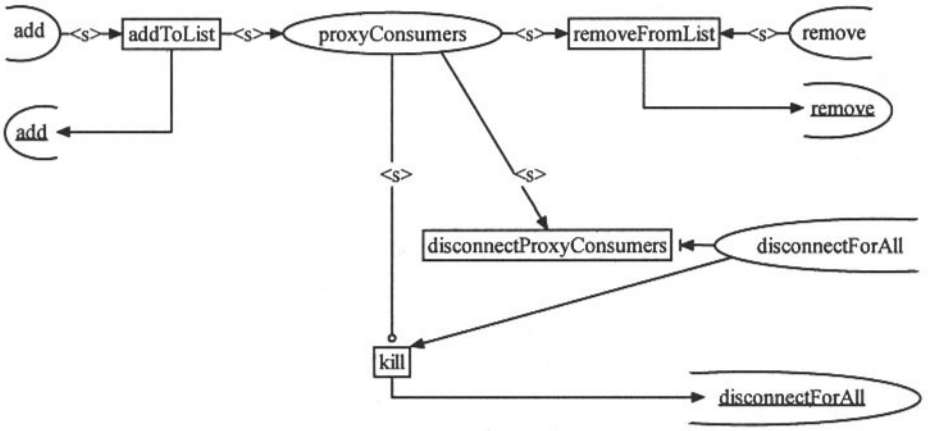
4.3.1.1 ProxyConsumerListSpec

ProxyConsumerListSpec (*Figure 8*) objects are responsible for managing the set of *ProxyPushConsumerSpec* objects given to customers of the event channel. Operations *add* and *remove* allow adding and removing proxies from place *proxyConsumers*.

Operation *disconnectForAll* disconnects and removes proxies from the set before returning. Firstly, transition *disconnectProxyConsumers* is linked to the SIP of *disconnectForAll* by a **test arc**. Test arcs are only involved in

the enabling rule, but, unlike input arcs, not in the firing rule; thus the tokens involved in the enabling substitution chosen for firing the transition are not removed. Therefore, firing *disconnectProxyConsumers* allows to empty the place *proxyConsumers*, and its action allows to disconnect the proxies.

```
class ProxyConsumerList specifies ProxyConsumerList{
place proxyConsumers <ProxyPushConsumer>;
transition disconnect {
     action {
        s.disconnect_push_consumer();
     }
}
```




```
}
```

Figure 8 Class ProxyConsumerListSpec

Secondly, transition *kill* is linked to the place *proxyConsumers* by an **inhibitor arc**. Inhibitor arcs are also only involved in the enabling rule and do not remove token during the firing: the enabling rule states that the inhibitor arc enables the transition *kill* only if the connected place is empty. The occurrence of *kill* removes the token from the SIP of *disconnectForAll* and puts a token in the SOP of *disconnectForAll* : the operation returns.

4.3.1.2 ProxySupplierListSpec

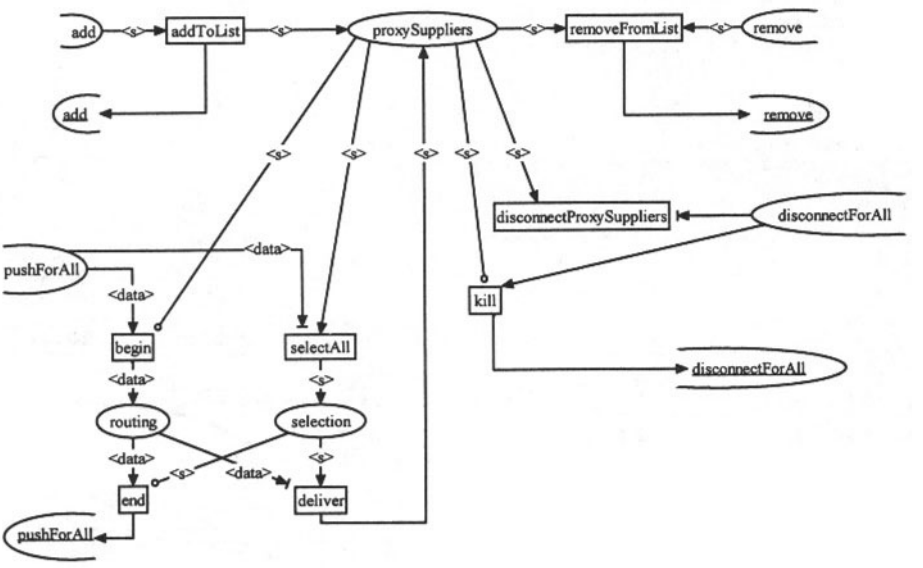
ProxySupplierListSpec objects play two roles (*Figure 9*):

- They manage the set of *ProxyPushSupplierExt* objects delivered by the event channel, exactly like *ProxyConsumerListSpec* objects do;
- They are responsible for the effective routing of events to customers of the event channel.

The ObCS of *Figure 9* describes how the routing is handled when operation *pushForAll* is invoked. Firstly, all proxy reference tokens are selected: they are removed from place *proxySuppliers* and deposited in place

selection (transition *selectAll*). Secondly, the event token is chosen in the SIP of *pushForAll* (transition *begin*) and one token is deposited in place *routing*. Thirdly, the operation *deliver* is invoked once on all the selected proxy references (invocation transition *deliver*). Finally, when all selected proxies have delivered the event the operation returns (*transition end*).

```
class ProxySupplierList specifies ProxySupplierList{
place proxySuppliers <ProxyPushSupplier>;
place routing <>;
place selection <ProxyPushSupplier>;
transition disconnectProxySupppliers {
     action {
       s.disconnect_push_supplier();
     }
}
transition deliver {
     action {
       s.deliver(data);
     }
}
```



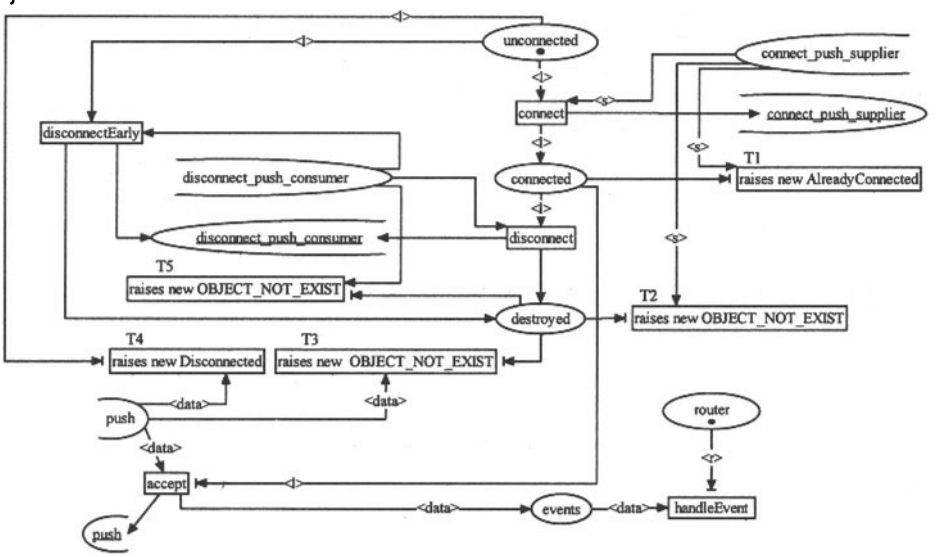

```
}
```

Figure 9 Class ProxySupplierListSpec

4.3.2 Class ProxyPushConsumerSpec

Figure 10 shows the class *ProxyPushConsumerSpec* that specifies the behaviour of the interface *ProxyPushConsumer* defined by the OMG's specification (*Figure 3*). Initially the place *unconnected* is marked with one token, which indicates that the proxy is not connected to a customer of the

```
class ProxyPushConsumerSpec specifies ProxyPushConsumer{
place unconnected <ProxyConsumerList> = {1*<proxy>};
place router <Router> = {1*<router>};
place connected <ProxyConsumerList>;
place destroyed <ProxyConsumerList>;
place events <any>;
transition connect {
     action {
       l.add(self);
     }
}
transition disconnect {
     action {
       l.remove(self);
       s.disconnect_push_consumer();
     }
}
transition disconnectEarly {
     action {
       l.remove(self);
     }
}
transition handleEvent {
     action {
       r.handleEvent(data);
     }
}
transition T1, T2, T3, T4, T5 {
     action {
       raise new OBJECT_NOT_EXIST();
     }
}
```




```
}
```

Figure 10. Class ProxyPushConsumerSpec

event channel; and has a parameterised marking {<*proxy*>} which holds a reference to a *ProxyConsumerList* given at instantiation. The place *router* has a parameterised marking {<*router*>}, where the variable *router* designates a *EventChannelRouter*.

Following the scenario of §3.3, the proxy evolves from the state unconnected to the state connected when the operation *connect_push_supplier* is invoked for the first time with a non *nil* reference and is destroyed when the operation *disconnect_push_consumer* is invoked. The ObCS details the behaviour and clarifies the inaccuracies of the original specification. When requested, operation *connect_push_supplier* terminates in one of the three following ways:

- a successful connection, transition *t5*;
- a failure raising the exception *AlreadyConnected* if the proxy is already connected, invocation transition *t8*;
- a failure if the proxy has already been destroyed by a call to operation *disconnect_push_consumer*.

When the operation *push* is invoked, and the proxy is connected, the event is first stored locally and the operation terminates (transition *t1*). The *EventChannelRouter* object is then requested to route the event by a call to its operation handleEvent (invocation transition *t11*). If the proxy is not yet connected, the operation raises the exception *Disconnected* (transition *t8*).

Once the proxy is destroyed (transition *t3*) all operations raise the CORBA exception *OBJECT_NOT_EXIST*. When the operation *disconnect_push_consumer* is invoked and the proxy is not yet connected the proxy is however destroyed (transition *t9*).

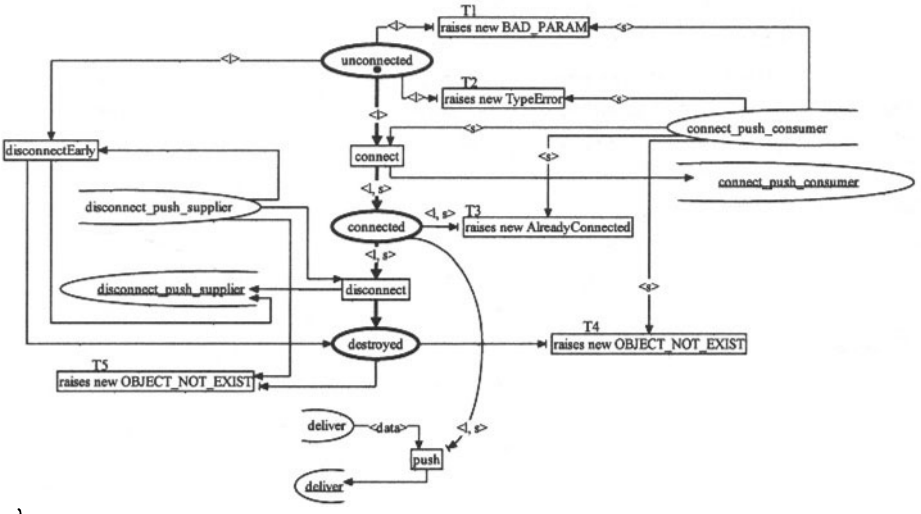
4.3.3 Class ProxyPushSupplierExt

Figure 11, below, shows the class *ProxyPushSupplierExtSpec* that specifies the behaviour of the interface *ProxyPushSupplierExt* defined in *Figure 6*. Initially, the proxy is not connected to a customer of the event channel. An invocation of the operation *connect_push_consumer* results in one of the five following ways:

1. a successful connection if the parameter of the call is a non *nil* reference (transition *t3*) and the deposit of the reference of the *PushConsumer* of the customer (*Receptor*) in place *consumer*;
2. a failure raising the exception *TypeError* if the customer's type does not suit the proxy (transition *t4*);
3. a failure raising exception *BAD_PARAM* if a *nil* object reference is given (transition *t5*);
4. a failure if the proxy is already connected to a customer (transition *t6*);
5. a failure if the proxy has already been destroyed (transition *t8*).

We have left some indeterminism between point 1 and point 2 that was initially contained in the original specification: specific implementations must decide on which conditions the exception *TypeError* will be raised.

```
class ProxyPushSupplierExtSpec specifies ProxyPushSupplierExt{
place unconnected <ProxySupplierList> = {1*<proxy>};
place connected <ProxySupplierList, PushConsumer>;
place destroyed <>;
transition connect {
     action {
       l.add(self);
     }
}
transition disconnect, disconnectEarly {
     action {
       l.remove(self);
     }
}
transition push {
     action {
       s.push(data);
     }
}
transition T1, T2, T3, T4, T5 {
     action {
       raise new OBJECT_NOT_EXIST();
     }
}
```




```
}
```

Figure 11. Class ProxyPushSupplierExt

The operation *deliver* (bottom center), called by the *EventChannelRouter* that created the *ProxyPushSupplierExt*, will call the *push* operation of the

customer that is a consumer of events if the proxy is connected (transition *t9*), or will fail silently raising no exception (transitions *t10* and *t11*).

4.4 Model analysis and tool support

The use of Petri nets as the formal behavioural notation enables us to perform several kinds of analysis on the specifications. We give a short overview of the kind properties that are obtained with our tool and the way they can be interpreted. In *Figure 11*, we illustrate the P-invariant for the *ProxyPushSupplierExtSpec*:

Marking(unconnected) + Marking(connected) + Marking(destroyed) = 1.

It is obtained by mathematical analysis of the Petri net. It means that whatever the actions performed by a *ProxyPushSupplierExtSpec* object, there will be one token in mutual exclusion 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 *t3*) and from *connected* to *destroyed* (transition *t1*).

The PetShop environment [9] is designed to integrate seamlessly in the development life cycle of a CORBA system. It takes into account the following steps:

- **Editing of the behavioural specification**: the environment includes a syntactical editor of Petri nets that supports the editing of the ObCS and the declarative part.
- **Generation from IDL**: based on the mapping between CO and IDL, a skeleton of the behavioural specification is then built. The IDL is directly read at runtime from an interface repository.
- **Mathematical analysis of the models**: PetShop includes classical analysis algorithms that check properties on the ObCS, such as P and Tinvariants or liveness properties of the ObCS.
- **Interpretation and debugging of models**: in PetShop, each instance of a CO class is executed by an interpreter described in [6]. The execution is tightly coupled with the editing in order to interactively test the behaviour of the model under design without any recompilation. Each instance of a CO class is a live CORBA object that can invoke or be invoked by any other CORBA object.
- **Distributed execution**: PetShop is itself a CORBA server. So it is possible from one PetShop to request the instantiation of a class on another distant PetShop.

- **Generation of executable standalone prototypes**: It is possible to generate standalone prototypes, that can be executed outside the environment.

Using the PetShop environment, we have been able to produce an executable formal specification of the CORBA Event Service. The tool support has been particularly useful as it provided assistance to detect mistakes thanks to invariants analysis and to be more rigorous on the expressions used on arc expressions as the tool is unforgiving where a human user understands the meaning. In turn, the interpretation 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.

5. ONGOING WORK

Generation of test cases. As our experiments show [18], in the current state of industrial practice, it is difficult to rely on informal specifications to develop reliable and interoperable CORBA servers. A promising research direction consists in combining formal specifications 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 [3;4;12]. 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.

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 analyse constructs specific to object-oriented systems. For example, when two IDL interfaces are related through inheritance, some form of behavioural inheritance needs to be respected for CO classes that specify these interfaces. The work presented in [20] is a useful starting point for us, the most relevant notion in the context of CORBA appearing to be the one based on the hiding of new methods introduced in subclasses.

6. CONCLUSION

The approach presented here is motivated by the momentum gained by CORBA as a standard for distributed object systems, and by the evidence that some form of abstract behavioural modelling supporting the CORBA

object model can be of great help in the development life-cycle of such systems. We have shown that our approach is suitable for the specification of services having a high degree of complexity and corresponding to real problems. The Cooperative Objects formalism allowed us to specify a real service due to its expressiveness in describing complex behaviours, including concurrency and synchronisation. Not surprisingly, the resulting formal specification is complex: this is quite natural, with regards to the complexity of the problem to model. However, due to the object-oriented nature of the CO notation, individual models (CO classes) remain simple and easy to understand (provided the basic notation, high-level Petri nets, is known to the specifier). The printed version of the specification is not fair to the usability of the CO notation: the PetShop tool, with its ability to execute the specification and to provide analysis results, is of great help during the construction of the models. Moreover, the tool integrates an elaborate debugger that allows to inspect and to change the marking of the places, and even the net structure at run-time. It is therefore very easy to correct design flaws and to investigate different specification scenarios interactively.

During the process of building the formal specification of the event service, we have detected several incompleteness or ambiguities in the COSS document. We wanted to know if the actual available implementations of the service had suffered the same problem, and if they had solved it in a consistent fashion. The experiment and the tests conducted on four implementations of the CORBA Event Service justify the need for formalisms that describe the behaviour so that implementers can refer to complete, precise and non-ambiguous specifications. The tested vendors' implementations, based on the OMG's specifications, exhibit many behavioural incompatibilities. We summarize the main results of our test experiments as reported in [18]:

- **Incompatibility with respect to Liskov's ([14]) substitutability principle:** if a customers 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); but also we can point out that the OMG does not give a set of minimal test cases to test implementations.
- **No explicit indication of the behaviour of the implementations:** the specification 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 behaviour of the CORBA Event Service they want to use. We therefore produced some reverse specifications based on our test campaign.

ACKNOWLEDGEMENTS

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

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

REFERENCES

1. Agha, Gul, and De Cindio, Fiorella. "Workshop on Object-Oriented Programming and Models of Concurrency." *16th International Conference on Application and Theory of Petri Nets, ICATPN'95*, Torino, Italy. Gul Agha, and Fiorella De Cindio, organizers. (1995)

2. Agha, Gul, De Cindio, Fiorella and Yonezawa, Akinori. "2nd International Workshop on Object-Oriented Programming and Models of Concurrency." *17th International Conference on Application and Theory of Petri Nets, ICATPN'96*, Osaka, Japan. Gul Agha, Fiorella De Cindio, and Akinori Yonezawa, editors. (1996)

3. Barbey, Stéphane, Buchs, Didier and Péraire, Cécile. "Overview and Theory for Unit Testing of Object-Oriented Software." *Tagungsband "Qualitätsmanagement Der Objektorientierten Software-Entwicklung"*, Basel, Switzerland. (1996) 73-112.

4. Barbey, Stéphane, Buchs, Didier and Péraire, Cécile. "A Theory of Specification-Based Testing for Object-Oriented Software." *European Dependable Computing Conference (EDDC2)*, Taormina, Italy. Lecture Notes in Computer Science, no. 1150. Springer-Verlag (1996) 303-20.

5. Bastide, Rémi. "Objets Coopératifs : Un Formalisme Pour La Modélisation Des Systèmes Concurrents." Ph.D. thesis, Université Toulouse III (1992).

6. Bastide, Rémi, and Palanque, Philippe. "A Petri-Net Based Environment for the Design of Event-Driven Interfaces ." *16th International Conference on Applications and Theory of Petri Nets, ICATPN'95*, Torino, Italy. Giorgio De Michelis, and Michel Diaz, Volume editors. Lecture Notes in Computer Science, no. 935. Springer (1995) 66-83.

7. Bastide, Rémi, Palanque, Philippe, Sy, Ousmane, Le, Duc-Hoa and Navarre, David. "Petri-Net Based Behavioural Specification of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ICATPN'99*, Williamsburg, VA, USA. Susanna Donatelli, and Jetty Kleijn, Volume editors. Lecture Notes in Computer Science, no. 1639. Springer (1999) 66-85.

8. Bastide, Rémi, Sy, Ousmane and Palanque, Philippe. "Formal Specification and Prototyping of CORBA Systems." *13th European Conference on Object-Oriented Programming, ECOOP'99*, Lisbon, Portugal. Rachid Guerraoui, Volume editor. Lecture Notes in Computer Science, no. 1628. Springer (1999) 474-94.

9. Bastide, Rémi, Sy, Ousmane and Palanque, Philippe. "Formal Support for the Engineering of CORBA-Based Distributed Object Systems." *IEEE International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland. IEEE Computer Society (1999) 264-72.

10. Diagne, Alioune, and Estraillier, Pascal. "Formal Specification and Design of Distributed Systems." *IFIP TC6/WG6.1 First International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, Paris, France . Elie Najm, and Jean-Bernard Stefani. Chapman & Hall, UK (1997)

11. Gaudel, Marie-Claude. "Testing Can Be Formal, Too." *TAPSOFT '95: Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE*, Aarhus, Denmark. Lecture Notes In Computer Science, eds. G. Goos, J. Hartmanis, and J. van Leeuwen, no. 915. Springer Verlag, Heidelberg (1995) 82-126.
12. James, P. R., and Gaudel, Marie-Claude. "Testing Algebraic Data Types and Processes: a Unifying Theory." *Formal Aspects of Computing* 10, no. Special issue. Best papers of FMICS98 (1999) 436-51.
13. Lakos, Charles. "A General Systematic Approach to Arc Extensions for Coloured Petri Nets." *15th International Conference on Application and Theory of Petri Nets, ICATPN'94*. Lecture Notes in Computer Science, no. 815. Springer (1994) 338-57.
14. Liskov, Barbara, and Wing, Jeannette M. "A Behavioral Notion of Subtyping." *ACM TOPLAS* 16, no. 6 (1994) 1811-41.
15. Object Management Group. *The Common Object Request Broker: Architecture and Specification. CORBA IIOP 2.2 /98-02-01*, Framingham, MA (1998).
16. ———. *Common Object Services Specification /98-07-05*, Framingham, MA (1998).
17. Sibertin-Blanc, Christophe. "Cooperative Nets." *15th International Conference on Application and Theory of Petri Nets, ICATPN'94*. Lecture Notes in Computer Science, no. 815. Springer (1994) 471-90.
18. Sy, Ousmane, and Rémi Bastide. *Compte Rendu De Tests D'Une Sélection D'Implémentations Du COS Event Service*. SERPICO/Lot 1/FOR3. Laboratoire LIHS - FROGIS, Université Toulouse I, 1999.
19. ———. *Correction De La Spécification Du COS Event Pour Le Modèle Push*. SERPICO/Lot 1/FOR4. Laboratoire LIHS - FROGIS, Université Toulouse I, 1999.
20. van der Aalst, W. M. P., and Basten, T. "Life-Cycle Inheritance, a Petri-Net Based Approach." *18th International Conference on Application and Theory of Petri Nets, ICATPN'97*, Toulouse, France. Pierre Azéma, and Gianfranco Balbo, editors. Lecture Notes in Computer Science, no. 1248. Springer (1997) 62-81.