Formal Specification and Prototyping of CORBA Systems

Rémi Bastide, Ousmane Sy, and Philippe Palanque

LIHS-FROGIS, Université Toulouse I, Place Anatole France,
F-31042 Toulouse CEDEX, France
{bastide, sy, palanque}@univ-tlse1.fr

Abstract. We propose to extend the CORBA interface definition of distributed objects by a behavioral specification based on high level Petri nets. This technique allows specifying in an abstract, concise and precise way the behavior of CORBA servers, including internal concurrency and synchronization. As the behavioral specification is fully executable, this approach also enables to early prototyping and testing of a distributed object system as soon as the behaviors of individual objects have been defined. The paper discusses several implementation issues of the multithreaded, distributed interpreter built for that purpose. The high level of formality of the chosen formalism allows for mathematical analysis of behavioral specifications.

Keywords. Formal methods, distributed object-oriented systems, CORBA, Petri nets, behavioral specification.

1 Introduction

CORBA [22], [32] (Common Object Request Broker Architecture) is a standard proposed by the Object Management Group (OMG) in order to promote interoperability between distributed object systems. The appearance of an industrial standard acknowledges the fact that the field of object-oriented distributed computing has moved, in the past few years, from experimental research projects to mainstream commercial products.

CORBA reaches several operational goals: standardize the best understood and best supported features of distributed object systems, promote interoperability between heterogeneous software systems, and provide a seamless integration with popular object-oriented and non-object-oriented programming languages such a C, C++, COBOL or Java.

CORBA defines an object model offering the following features:

- **Client/Server relationship**: a distributed object system consists of a set of objects that interact by invoking services to one another. The invocation is asymmetrical: one of the objects involved acts as a client, actively requesting the service, while the other acts as a server, passively waiting for requests and executing them as they come. Note that a given object usually acts both as a client and a server at different moments of its activity.

- **Access through references**: Before a client object can request a service from a server object, the client must acquire a reference to the server through some mechanism. Moreover, the invocation is performed through a well-known, strongly typed interface: the set of services that can be invoked and the signature of these services are known at compile time, and depend on the type of the reference held by the client. This is the basic type of interaction, although CORBA also supports the dynamic construction of invocations, to allow for contacting objects whose class is not known at compile time.
- **Synchronous invocation**: the invocation of a service results in synchronization between the client and the server: an invocation is a two-ways question/response interaction, where parameters are sent and results returned. The call is blocking on the client's side: it has to wait for the results before continuing its work. This is the basic invocation mode, although CORBA has provision for unidirectional message sending (the so-called *oneway* services).
- **Dynamic topology**: The fact that an object *A* holds a reference to an object *B* introduces a *reference relationship* between A and B. The topology of the reference relationship is dynamic. An object may transfer a reference it holds to another object; new objects can be dynamically introduced in the system.

This basic set of features comes as no surprise: it closely mimics the principles of mainstream sequential object-oriented languages such as C++, and allows extending such languages to the realm of distributed and concurrent systems.

CORBA has standardized the features described above by specifying an Interface Definition Language (IDL). CORBA-IDL is independent from any programming language (although closely patterned after C++) and object-oriented, supporting specialization of interfaces through inheritance. A CORBA-IDL interface specifies at a syntactic level the services that a client object can request from a server object that implements this interface. The interface details the services supported and their signature: a list of parameters with their IDL type and parameter-passing mode, the IDL type of the return value, the exceptions that may possibly be raised during the processing of the service.

1.1 CORBA and Behavioral Specification

A recognized limitation of CORBA is that it defines remote object classes in terms of their interface only. CORBA IDL covers only the syntactic aspects of the possible use of a remote object. IDL does not cover any semantic or behavioral description of the remote object, while this information is obviously of prime importance for the clients. By behavioral aspect, we mean:

- The constraints on the order of invocation of the services described in the interface.

- The concurrency constraints of the remote object: is it able to support concurrent access to its services, or does it force a serialization on the concurrent invocations?
- The conditions under which an exception might be raised during the processing of a service.

What CORBA lacks is an abstract way to specify the semantics of an IDL interface without constraining its implementation, much in the same way that an Abstract Data Type [11] (ADT) specification can be used for specifying the semantics of a sequential data type.

This limitation becomes evident when one considers the standardization of CORBA services (COS) [23] that is underway at the OMG. CORBA services are a standardization of the various basic services that any large-scale distributed application is expected to require. Example of these services are the naming service, allowing to retrieve a remote object reference by providing a symbolic name, or the event service, allowing to define one-to-many communications that go beyond the client-server paradigm that is basically supported by CORBA.

The specification of the CORBA services is 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 behavior). The specification document counts no less than one thousand pages of such "semi-formal" specification.

To illustrate the dire need for some form of behavioral specification, we may quote the OMG specification document for the event service: "Clearly, an implementation of an event channel that discards all events is *not a useful* implementation" (emphasis is from the original document). However, the rest of the document defines only informally what actually constitutes a useful implementation.

The present paper aims at providing a suitable solution to the problem of behavioral specification of distributed objects, in the context of CORBA. The paper is organized as follows: We first detail the requirements for behavioral specification formalism suited to CORBA. We then present how the Cooperative Objects formalism needs to be adapted in order to support fully the CORBA model. Section 4 presents a significant case study of specification using our approach. Section 5 explains how the formalism can be used to enable rapid prototyping of distributed systems.

2 Requirements for a Behavioral Specification Formalism

Our goal is provide a notation suited to the behavioral specification of CORBA systems: we want to be able to describe the behavior of a collection of interacting objects, and not merely the behavior of a single object in isolation. A formalism aimed at serving this goal has to comply with several requirements:

- Cope with data flow as well as with control flow. The formalism needs to be able to deal with typed values, and not only with pure causal relationships. It will often be the case that the behavior of a CORBA system depends not only on the previous history of invocations between objects, but also on the values exchanged during these invocations. For a given state of an object, an invocation may succeed or fail according to the values of parameters of the invocation.

- Allow specifying internal concurrency for objects. This point is especially important for CORBA since a CORBA server object will often be a "large-grained", entity shared by a lot of clients, providing services whose processing will take some time. It is therefore unrealistic to enforce each service to be atomic, so that at most one service will be active at any time at the server. Actually all of current CORBA ORBs allow for "multi-threaded" server implementations, where a server object can serve several services at the same time.
- Serve the needs of the implementers of the server class, as well as those of the designers of systems that will be clients for this server. On the one hand, the behavioral specification must be complete and precise enough that the programmer implementing the server in some programming language knows in a precise and non-ambiguous way what behavior to implement; and it must be abstract enough not to constraint the implementation choices of the programmer. On the other hand, the potential clients of the class will use the specification to gain a non-ambiguous understanding of the semantics of each service.

In the above set of requirements for a CORBA specification formalism, we have purposefully not included the provision for object-oriented features such as inheritance, or encapsulation. Actually, the fact that CORBA defines mapping to non object-oriented languages such as C or COBOL is a demonstration that object-orientedness is conceptually not a requirement. Actually, the implementation of a CORBA interface in some programming language can be considered as a formal behavioral specification (albeit at a very low level of abstraction). However, object-oriented specification formalisms map more naturally to the object-oriented foundations of CORBA, which is beneficial for the overall usability of the specification documents.

2.1 Related Work

The behavioral specification of object systems is a field of research per se [15], [25]. The importance of providing behavioral specifications suited to the OMG object model as been recognized by several researchers. Sankar [30] argued that the introduction of formal methods could help maintaining the current level of software quality while the complexity of software increases due to the presence of distributed object. He also noted that formal methods are more likely to be accepted in the field of distributed object systems, where their overhead is considered acceptable.

It is tempting to try to adapt the "design by contract" paradigm popularized by the Eiffel language to CORBA systems. However, due to its roots in Abstract Data Types, such approaches often make the implicit assumption that the execution of a single operation is always atomic. As stated in [20]. "*Any client accessing an object through [an operation][1] must be guaranteed exclusive access to the object throughout the duration of the call*". *The smallest permissible level of granularity for exclusive access to an object is the execution of a call to an exported [operation]*). Although this limitation can be viable in the domain of *concurrent* systems, in our opinion it presents a serious drawback in the field of *distributed* systems such as CORBA. Most approaches based on pre and post-conditions implicitly consider an object as a monitor, allowing only one service to be active at a time, thus cannot be extended to the modeling of distributed CORBA servers with internal concurrency.

The work presented in [8] falls in this category: Bryan proposes the use of a language called ASL (Architecture Specification Language) which includes a behavioral specification part, as well as other extensions supporting the description of software architectures. However, the behavioral specification (based on pre-and post-conditions for operations) does not permit do describe internal concurrency for objects. The work of Sankar [30], who uses the textual language Borneo, also falls in the same category. Several proposals have been made to adapt the LARCH specification language [12] to CORBA IDL, e.g. [18], [33]. The former does not explicitly address intra-object concurrency, while the latter addresses it with the introduction of atomic and non-atomic operations. The Z language has also been proposed as a possible behavioral specification formalism [9], but it is not clear to what extend it supports the concurrency requirements of CORBA systems. Other work, such as [26], are more theoretical in nature, and do not meet the operational requirements of CORBA-based software engineering.

Among the various formalisms proposed, StateCharts [13] comply with the above requirements, and is actually the behavioral formalism for the popular Unified Modeling Language (UML) notation [28], [29].

We propose the use of high-level Petri nets as the behavioral specification medium for CORBA systems. The usefulness of combining Petri nets and objects has been recognized by several authors, as witnessed by the workshops held at previous editions of the ATPN conference [1], [2]. However, several proposals do not fit well with CORBA: A behavioral specification formalism for CORBA has to support its fundamental object model, and more precisely to allow remote objects to be designated by references, and these references to be transmitted as invocation parameters. Several object-oriented Petri net formalisms [34], [17], aimed at providing a solid theoretical basis to concurrent object-oriented computation, refrain from using references, and prefer considering tokens as objects, and not references to objects. Such formalism prevent a same object to be referenced by different tokens in different places, and thus would make the specification of a reference-based system such as CORBA less straightforward.

The approach proposed here uses an object-oriented dialect of high-level Petri nets called Cooperative Objects. This formalism complies with the requirement listed

[1] Meyer uses the Eiffel vocabulary of "feature" instead of operation.

above, and its Petri net roots provide is with a powerful basis for modeling and analyzing concurrent behaviors.

3 The Cooperative Object Formalism

Cooperative Objects [6] (CO) are a dialect of object-structured, high-level Petri nets. Their lengthy formal definition has been provided in previous publications [3], [31], and we will only recall informally their main features, through examples.

Petri nets basics. Petri nets [24] have been studied for a long time as a mathematical formalism for the modeling of concurrent systems. A Petri net models a system as a set of state variables called *places* (represented as ellipses) and a set of state-changing operators called *transitions* (represented as rectangles). The state of the system is described as a distribution of information elements (called *tokens*) in the net's places; this distribution is called the *marking* of the net. In basic Petri nets, tokens are dimensionless entities modeling only conditions. Several dialects of Petri nets (called high-level Petri nets) allow tokens to carry information and to manipulate this information at the occurrence of transitions. Our own dialect is close to well-known high-level net models such as colored Petri nets [14] or Predicate-Transition nets [10]. They differ from these mainly by the nature of the inscriptions attached to the net elements, and by their object-oriented structure.

In Petri nets, the causality structure of the systems (stating under which condition a change of state may occur, and what will be its resulting state) is described by *arcs*, connecting places and transitions.

The input arcs of a transition (coming from places of the net) describe the preconditions of an occurrence of this transition. The transition may occur if the input places hold enough tokens. Conversely, the output arcs of a transition (going to places of the net) describe the postconditions of an occurrence of the transition, in terms of changes in the net's marking. After an occurrence, tokens are removed from the input places of the transition, and new tokens are set in its output places.

Petri nets allow for modeling very naturally systems with distributed state, and concurrent activities. Two transitions that do not share any input place may occur concurrently (marking permitting) and the resulting state will be the same whether they occur concurrently, or sequentially in any order.

A huge amount of work has been devoted to the study and analysis of Petri nets. Several important safety and liveness properties, as well as invariants in the dynamic behavior can be proved by mathematical means.

Structure of a CO class. A CO class specifies a class of objects by providing their interface (the set of services offered, along with their signature) and their dynamic behavior. The behavior of a CO class is called its *Object Control Structure* (ObCS), and is defined with a dialect of high-level Petri nets. More specifically:

- 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 will be 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 inscribed 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>`. A multiplicity of 1 can be omitted (thus `1*<v1,… vn>` can be abbreviated as `<v1,… vn>`) and an empty list of variables can also be omitted (thus `2*<>` can be abbreviated as `2`).
- Transitions have a precondition (a boolean expression of their input variables) and an action, which may use any service 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.

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.

The firing of a transition will execute the transition’s action, remove tokens from input places, compute new tokens and store them in the output places of the transition.

The formalism also supports two arc extensions [16]: test arcs (allowing to test for the presence of tokens in input places of a transition, without removing them at the occurrence of the transition) and inhibitor arcs (allowing to test for the absence of certain values in input places of a transition).

4 A Case Study in CORBA Behavioral Specification

Cooperative Objects were initially defined independently of CORBA, [4] but CO and CORBA happen to share the same object model, as described in §1. CO and CORBA complement each other nicely: The initial description of CO used an idiosyncratic language to describe interfaces, while CORBA provides an attractive, language

neutral IDL. On the other hand, CO can be used to provide the behavioral information that is lacking in CORBA-IDL.
This section presents the necessary syntactic adaptation that CO require to match more closely CORBA-IDL, and some additions to the formalism supporting CORBA-specific constructs such as exceptions and parameter-passing modes.

CORBA-IDL will be used as the data description language used by Cooperative Objects. Cooperative Objects will use IDL to describe:

- The system of data types used by the ObCS nets of Cooperative Objects. The Token-type of the places will thus be described in terms of CORBA IDL, and the variables on the arc will be of an IDL-defined type.
- The interface of a Cooperative Object class itself.

To achieve the integration of Cooperative Objects and CORBA IDL, we need to define a mapping from the constructs of CORBA IDL to those of Cooperative Objects, much in the same way that CORBA defines mappings from IDL to conventional programming languages such as C++, Smalltalk or Java.

4.1 Presentation of the Case Study

To illustrate our modeling approach, we now detail an example dealing with the banking business. This example is typical of the intended use of the CORBA technology (although much shorter than real-life examples) and is in fact used as a tutorial example for several commercial ORBs. Despite its small size, it illustrates all the main aspects of our technique, including exceptions, interface inheritance and realistic behavioral specification featuring intra-object concurrency.

The banking system is composed of two different kinds of entities: *bank accounts*, which keep a (positive or null) amount of money, and *banks*, which keep a set of bank accounts. This system is first described in terms of CORBA IDL. Cooperative Object class specifications are then provided, to specify the behavior of the system.

```
module Banking {
 exception noSuchAccount;
 exception nameAlreadyExists;
 exception insufficientFunds;
 exception accountIsClosed;

 interface Bank {
      Account newAccount(in string name,
            in float initialAmount )
            raises(nameAlreadyExists);
      Account findAccount(in string name )
            raises(noSuchAccount);
      void closeAccount(in Account account)
            raises(noSuchAccount);
 }
 interface Account {
      void transfer( in float amount )
            raises(accountIsClosed, insufficientFunds);
      float balance( );
}

 interface BankAccount : Account {
      void open();
      void close();
 }
}
```

Fig. 1. the IDL text for the banking system.

Fig. 1 shows the interface of the banking system, expressed in CORBA IDL. This IDL first defines data types for the exceptions that might be thrown during processing (*noSuchAccount, nameAlreadyExists, insufficientFunds, accountIsClosed*), and defines three interfaces:

- *Bank*, which the customers will use to create, close, and access their accounts. A bank is merely a repository for bank accounts, and allows associating an *Account* reference to a human-readable name. The service *findAccount* retrieves an *Account* reference for which only the name is known;
- *Account*, offering a service to obtain the current balance (*balance*) and a single service to credit or debit the account (*transfer*).
- *BankAccount*, which will be used internally by the bank to perform *open* and *close* operations on accounts. *BankAccount* is a specialization of the *Account* interface, meaning that is offers all the services of *Account*, in addition to the new ones it introduces (*open* and *close*).

The provision for two different interfaces (*Account* and *BankAccount*) to describe the same concept of account enables to describe different access rights (or different views) of the same object, tailored for the needs of different clients. The signature of the services provided by the *Bank* interface only deal with *Account* references, which means that clients of a *Bank* object will only receive references of this type, and thus will be able to access only the services defined in *Account*.

4.2 CO-based Behavioral Specification

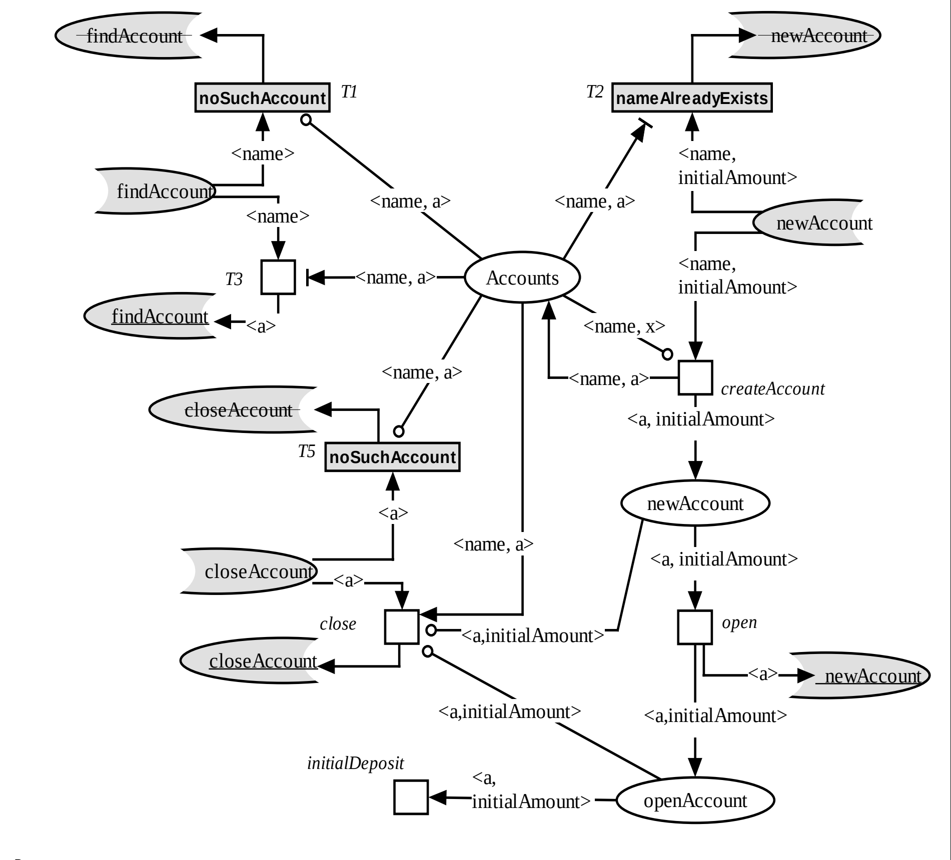
Obviously a lot more needs to be said, in addition to these interfaces, to have a complete specification of the banking system. The behavioral specification of the banking system is provided below, in terms of two Cooperative Object classes, *BankSpec* and *BankAccountSpec*.

IDL interfaces and CO classes. A CO class may specify one or several IDL interfaces, in the same way that a Java class, for example, may implement several Java interfaces. This is convenient since, very often, several interfaces are given for the same entity to allow providing different views of the same object, tailored for the needs of different clients. The CO class of Fig. 2 specifies only one interface, namely *Bank*. The keyword **specifies** is followed by the list of CORBA-IDL interfaces specified by the CO class.

Mapping for services. Each service *op* defined in an IDL interface is mapped to three places in the ObCS net: a Service Input Port (SIP, labeled op), a Service Output Port (SOP, labeled $\underline{op}$) and a Service Exception port (SEP, labeled ~~op~~). These three 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 Token-type of the SEP is `<Exception>`, where Exception is the super-type of all IDL exception types. The SEP is only used if the service is defined to raise an exception.

```
class BankSpec
specifies Bank {
      place Accounts <string, BankAccount>;
      place newAccount <string, BankAccount, float>;
      place openAccount <BankAccount, float >;
      transition createAccount {
            action {a = new bankAccountSpec();}
      }
      transition open {
            action { a.open(); }
      }
      transition initialDeposit {
            action { a.transfer(initialAmount); }
      }
      transition close {
            action { a.close(); }
      }
```



}

Fig. 2. Cooperative Object class specifying the *Bank* interface.

For instance, according to the IDL of Fig. 1, the service *findAccount* is mapped onto three places: findAccount for the SIP, findAccount for the SOP and ~~findAccount~~ for the SEP.

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 that raises an exception at some point will instead result in an exception token being deposited in the SEP.

Fig. 2 illustrates both the graphic syntax of a Cooperative Object class and the textual annotations that are necessary to complete its description. These textual annotations are:

- The list of the interfaces that the CO class specifies (keyword **specifies**). In this case, the *BankSpec* class specifies the *Bank* interface.

- The description of the places' type: for instance, the *Accounts* place holds tuples of the form `<string, BankAccount>`, i.e. the name of a bank account, and the associated reference to a *BankAccount* object. The type for the input, output and exception ports needs not be stated, since it is deduced from the signature of the associated service. For example, the type of the *newAccount* input port is `<string, float>`, and the type of its output port is <Account>.
- The description of the transitions' preconditions and actions: Empty actions default to no action, and empty preconditions default to true. Only the transitions with non-default precondition or action need to be stated in the textual part.

The picture also illustrate the notation for inhibitor arcs (ended with a circle instead of an arrow) and for test arcs (ended with a barred arrow).

The BankSpec class describes a sensible behavior for the bank entity. For example, it states that an Account reference can be retrieved (through the findAccount service) only after an account with the same name has been created (through the newAccount service) and before it has been closed (through the closeAccount service). It also specifies under which conditions exception might be raised (for instance a *nameAlreadyExists* exception is raised when the *newAccount* service is called with a name parameter that matches a previously registered name). Such behavioral constraints could have been modeled just as well using a conventional ADT-based specification. However, the description of more subtle behaviors requires a formalism expressive enough to encompass concurrency and synchronization constraints.

In order for the invocation of services to proceed in a sound way, the ObCS structure must respect a set of constraints. Informally, an object will provide either a result or an exception for each invocation, and will only provide results if it has been previously invoked. With respect to the ObCS, the arrival of one token in the SIP will eventually result later in exactly one token being deposited either in the SOP or in the SEP. These structural constraint on the ObCS are easily expressed in terms of Petri nets theory, and can be checked automatically. The formal description of these structural constraints can be found in [7], where the notion of Operation Control Structure (OpCS) is also defined. Informally, the OpCS of a service is a subnet encompassing it's SIP, SEP and SOP.

The OpCS of the *newAccount* service is especially noticeable: unlike the other two services (*findAccount* and *closeAccount*), it is not made of atomic transitions, but of a subnet encompassing the *createAccount* and *open* transitions. This feature allows for internal concurrency within a *BankSpec* instance: While an invocation of *newAccount* is being processed (i.e. when the *newAccount* place holds one token), other incoming invocations of *findAccount* or *closeAccount* can be serviced (if the marking of the *Account* place permits it). Moreover, several invocations of *newAccount* can be serviced concurrently (this will result in the *newAccount* place holding several tokens). Finally, when the service invocation returns (by setting the reference to the newly created *BankAccountSpec* object in the *newAccount* return port), the *BankSpec* object continues an internal processing, namely to initialize the *BankAccountSpec* object with its initial balance (transition *initialDeposit*). This faithfully models the way actual banks proceed in the creation of new customer accounts (at least in France): If a customer goes to a bank to create an account with an initial deposit, the account will be created immediately, but it will take some time before the account is actually credited with the initial deposit. If the account's balance is accessed in the meantime, it will be zero. The BankSpec also takes care that an account is not closed before the initial deposit is performed (inhibitor arc between *openAccount* and *close*).

Mapping for parameter-passing modes. The semantics for the three parameter-passing modes of CORBA IDL is clear.

- *in* parameters are values provided by the caller, that the service may use at its own will;
- *out* parameters are values computed by the service, and returned to the caller;
- *inout* parameters are values transformed by the service, i.e. provided by the caller and returned to it after being processed.

Inout parameters require a special treatment in the OpCS structure: in order to ensure that any inout parameter is properly transmitted from the Service Input Port to the Service Output Port, a sufficient condition is to check that the parameter name appears on every arc connected to each place of the OpCS.

Inter-object invocations in transitions. We want to be able to model CORBA systems, and not only isolated CORBA servers. Actually, the behavior of one class in isolation will often be of little interest, and will only become meaningful when we can describe how an instance of the class may interact with other instances of other classes in the system.

CORBA objects interact with one another by invoking services defined by the IDL interfaces they support. The Cooperative Object formalism supports this form of cooperation by allowing the action of a transition to be the invocation of a service. The operational semantics of such an invocation transition is a client / server protocol that can be formally defined in terms of Petri nets.

This client server protocol we use has been first presented in [27] for basic Petri nets, extended to object-oriented high-level Petri nets in [3], and is presented in [31]. The fact that not only the internal behavior of objects, but also their communication protocol is defined in terms of Petri nets allow us to reason about systems of cooperating objects, and not only on isolated instances. The theoretical details of how this client-server protocol is adapted to the semantics of CORBA inter-object communication are given in [7].

Mapping for exceptions. CORBA IDL allows specifying exceptions that may be raised during the processing of an invocation. An exception is an object of a specific data-type, and can hold information on the causes of its occurrence or other useful data.

When an exception is raised, the normal processing of the service is cancelled, the result, out and inout parameters of the service are undefined, an exception object is instantiated and only this object is transmitted to the client of the invocation.

In order to specify properly the behavior of a CORBA system, our formalism needs to address two concerns:

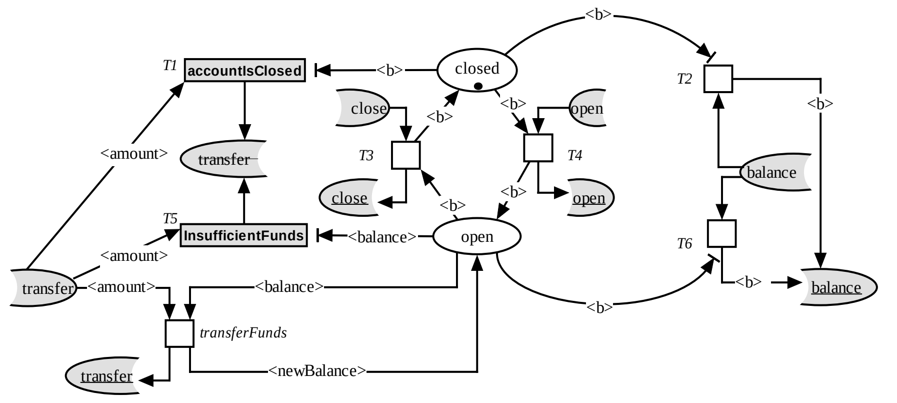
- Define under which conditions an exception may be raised during the processing of an invocation, and what corrective actions are eventually needed in the server object to restore a consistent state.
- Define what action a client object needs to take if a service invocation results in an exception instead of providing the expected result.

The first point is tackled by *exception transitions*: Exception transitions are labeled by the name of the exception data-type that is raised. They can have input and output arcs from any place of the OpCS of one service, but necessarily have exactly one output arc connected to the SEP of this service. The occurrence of an exception transition models the fact that an exceptional condition has occurred during the processing of an invocation, and that this processing cannot be carried any further. The *T5* transition in Fig. 2 is an exception transition. It models the fact that an account needs to be known by the bank before being closed.

The second point is tackled by:

- a simple syntactic extension of the graphic syntax of invocation transitions (called emission rules), not illustrated in our case study,
- a straightforward extension of the client-server protocol described in [7], acknowledging the fact that an invocation can have two different outcomes: a normal outcome, providing the expected results, or an exception outcome, providing no result other than the exception raised by the server.

```
class BankAccountSpec
specifies BankAccount {
      place closed <float> = { <0> };
      place open <float>;
      transition transferFunds {
            precondition {
                  (amount + balance) >= 0;
            }
            action {
                  newBalance = amount + balance;
            }
      }
      transition t5 {
            precondition {
                  (amount + balance) < 0;
            }
```



}

Fig. 3. CO class specifying the *BankAccount* interface.

The behavioral specification of bank accounts is shown in Fig. 3. This CO class specifies the *BankAccount* interface and implicitly specifies the *Account* interface since the two are related through inheritance. Several new syntactic constructs are illustrated in this class:

- Places can be provided with an initial marking, stating their content right after an instance of this class is created. In this case, an account is created in the closed state, with an initial balance of 0, as stated by the definition of the *closed* place.
- The *transferFunds* and *T5* transitions are in structural conflict. This conflict is deterministically solved by the preconditions of these two transitions, which are mutually exclusive. Thus, the *InsufficientFunds* exception is raised when the current balance is lower than the amount that one tries to withdraw.

Note that (for the sake of brevity), the specification detailed in the *BankSpec* and *BankAccountSpec* classes offers no provision for reopening a closed account. This could be achieved, for instance, by providing a subclass of *BankSpec* with additional services to transfer the balance from a closed account to a newly created account.

5 Prototyping CORBA Systems

A well-known advantage of Petri nets is their executability. This is highly beneficial to our approach, since as soon as a behavioral specification is provided in terms of CO classes, this specification can be interpreted to provide additional insights on the possible evolutions of the system.

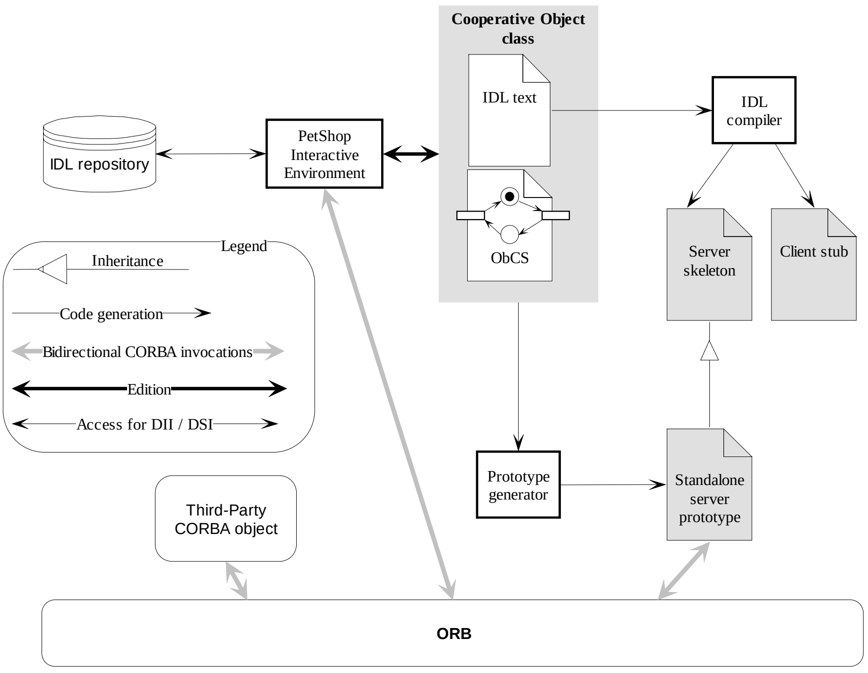


Fig. 4. Architecture of the PetShop environment

Our approach is supported by a tool called PetShop (Fig. 4), which includes a distributed implementation of the high-level Petri net interpreter described in [5], rewritten in Java. The implementation is such that:

- An interpreted ObCS can invoke (from an invocation transition) a service of a "third party" CORBA server running outside of the environment;
- Conversely, an external CORBA client can call a service of a Cooperative Object, regardless of the fact that this service invocation will be performed by interpretation of the CO ObCS.

This offers complete interoperability between the CORBA world and our formalism, and enables us to work in realistic settings, where we possess the complete behavioral specification of some objects only, but where we can nonetheless access objects provided by other sources, of which we know only the CORBA IDL.

In this section, we describe how the PetShop CASE tool fits into the CORBA program development cycle. As a whole, PetShop completely supports the following phases

- Editing of the behavioral specification as a Cooperative Object class;
- IDL generation;
- Cooperative object class analysis;
- Interactive interpretation of models;

- Prototype generation.

Cooperative object class editing. The Petri net editor provides the graphical interface for both the editing of the textual part and the ObCS part of each CO class.

As the ObCS can specify several IDL interfaces, the tool also provides access to existing IDL interfaces and allows for the creation of new IDL interfaces.

IDL generation. The original goal of our approach is to add behavior to IDL. A frequent design activity is to start from pre-defined IDLs (such as the ones provided in the definition of the CorbaServices [23]) and to write a behavioral specification for it. The designer can also start from scratch and generate CORBA IDL matching a given behavioral specification.

Cooperative object class analysis. As Cooperative Objects are a subclass of Petri nets, the analysis module allows proving some interesting properties of the nets such as invariants, conflicts (structural properties), liveness, boundedness, home state (behavioral properties).

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 [35]. We also want to analyze the co-operation between several CO instances, to check properties of a system of interacting objects. This is possible since the client-server protocol presented in [7] is described in terms of Petri nets, which allows generating a single static Petri net from the ObCS of classes in a system [3], [31]. This global net can be used to prove properties of the system as a whole.

Interactive interpretation of models. One of the well-known advantages of Petri nets is their executability. This feature is exploited in the PetShop tool, that offers the well known advantages of interpreted environments in terms of flexibility, interactivity and ease of use ([19], [21])[21]. PetShop allows an interpreted object to seamlessly access "real" third-party CORBA servers, or to be accessed as a server from other CORBA objects developed independently and maybe in a different programming language.

In our implementation, each edited net is executed using the ObCS interpreter described in [5]: the editor itself acts as a client/server program. As a result, as soon as a net is edited, an instance of the ObCS starts executing. The interpreted instance is accessed using the dynamic CORBA mechanisms of Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI). This allows us to test the behavior in a CORBA runtime environment and make real distributed experiments between CORBA objects hosted by the same or different computers.

The environment allows for graphic debugging of the interpreted net, by examining or modifying its marking, or even modifying its control structure dynamically.

Prototyping Distributed System. In the last phase of development, the programmer is interested in shipping a finished stand-alone CORBA object that can be run without the support of the CASE tool. For that reason PetShop can generate a prototype and launch instances of the prototypes. Those instances are “almost functional” implementations, in the sense that only behavioral requirements are dealt with. Other “quality of service” requirements, such as performance, persistence, replication or fault-tolerance, have to be taken care of in a completely functional implementation. The generated prototype is still an interpreted high-level Petri net, but does not have graphical features, like graphic debugging or live editing capabilities, which allows it to perform faster, and to function on machines that lack graphic abilities.

6 Conclusion and Future Work

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 behavioral modeling supporting the CORBA object model can be of great help in the development life-cycle of such systems.

Several aspects of the CORBA/Cooperative Objects integration are not presented here, for space reasons: among these are the support for unidirectional service invocation (oneway services in IDL), or the provision for IDL constructs such as constants and context information.

Among the works in progress on our approach are the following:

- Use of Petri net analysis techniques: Our approach has a strong operational bias: we accept to trade full analysis possibilities in favor of modeling power. However, we do not give up the potential for analysis potential of Petri nets altogether. Currently, our tool includes the usual Petri net analysis algorithms (such as P and T-Invariants, liveness, boundedness, siphons and traps) that operate on the ObCS *underlying net*, i.e. the ObCS where all variables and data types are removed. We are currently in the process of assessing which of the analysis result for basic Petri net are preserved in the high-level ObCS. We are also investigating analysis techniques especially devised for high-level nets.
- Analysis of object-oriented features: we wish to use Petri net analysis techniques not only 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 [35]. We also want to analyze the cooperation between several CO instances, to check properties of a system of interacting objects.
- Test generation: Another ongoing research is the ability to generate test suites from the Cooperative Object class definition. The tool supporting our can generate a prototypes that are “almost functional” implementations or CORBA-IDL interfaces, in the sense that only behavioral requirements are dealt with. Other “quality of service” requirements, such as performance, persistence, replication or fault-tolerance, have to be taken care of in a fully functional implementation. The idea is that the CO class can serve as the formal specification of the class, and that programmers will implement the

specification in some programming language, implementing the other quality-of service requirements. This implementation needs to be tested for conformance against the original CO-based specification and the CO specification can be used to generate a test suite for the implementation.

Acknowledgements

The work presented here is funded by CNET, the National Research Center for the French telecommunications agency (France Telecom) under grant number 98 1B 059. The final goal of the project is to provide a fully integrated CASE tool supporting our approach of behavioral specification of distributed object systems using high-level Petri nets.

References

1. Agha, Gul, and De Cindio, Fiorella. "Workshop on Object-Oriented Programming and Models of Concurrency." *16th International Conference on Applications and Theory of Petri Nets, ATPN'95,* Torino, Italy, June 26-30, 1995. 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 Applications and Theory of Petri Nets, ATPN'96*, Osaka, Japan, June 24, 1996. Gul Agha, Fiorella De Cindio, and Akinori Yonezawa, editors. (1996)
3. Bastide, Rémi. "Objets Coopératifs : Un Formalisme Pour La Modélisation Des Systèmes Concurrents." Ph.D. thesis, Université Toulouse III (1992).
4. Bastide, Rémi, and Palanque, Philippe. "Cooperative Objects: a Concurrent, Petri-Net Based Object-Oriented Language." *Systems Engineering in the Service of Humans, IEEE-SMC'93,* Le Touquet, France, October 15-20, 1993. IEEE Press (1993)
5. 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, ATPN'95,* Torino, Italy, June 1995 . Giorgio De Michelis, and Michel Diaz, Volume editors. Lecture Notes in Computer Science, no. 935. Springer (1995) 66-83.
6. Bastide, Rémi, and Palanque, Philippe. "Modeling a Groupware Editing Tool With Cooperative Objects." *Concurrent Object-Oriented Programming and Petri Nets.* Gul A. Agha, and Fiorella De Cindio, editors. Wien: Springer-Verlag (1998)
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, ATPN'99,* Williamsburg, VA, USA, June 21-25, 1999. (1999)

8. Bryan, Doug. "Exactness and Clarity in a Component-Based Specification Language." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 1-15.
9. Bryant, Antony, and Evans, Andy. "A Formal Basis for Specifying Object Behaviour." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 17-30.
10. Genrich, H. J., and Lautenbach, K. "System Modelling With High-Level Petri Nets." *Theoretical Computer Science.* Vol. 13. North-Holland (1981) 109-36.
11. Goguen, J. A., Thatcher, J. W., and Wagner, E. G. "Current Trends in Programming Methodology." *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.*Prentice-Hall (1978) 80-149.
12. Guttag, John V., James J. Horning, S. J. Garland, A. Jones, and J. M. Wing. *Larch: Languages and Tools for Formal Specification* Springer-Verlag, New-York (1993).
13. Harel, David, and Gery, Eran. "Executable Object Modeling With Statecharts." *IEEE Computer* 30, no. 7 (1997) 31-42.
14. Jensen, Kurt. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* 2nd edition ed., Vol. 2 Springer-Verlag (1996).
15. Kilov, Haïm, and William Harvey, editors. *Object-Oriented Behavioral Specifications* Kluwer Academic Publishers (1996).
16. Lakos, Charles. "A General Systematic Approach to Arc Extensions for Coloured Petri Nets." *15th International Conference on Applications and Theory of Petri Nets, ATPN'94*, June 1995. Lecture Notes in Computer Science, no. 815. Springer (1995) 338-57.
17. Lakos, Charles, and Keen, C. D. "LOOPN++: a New Language for Object-Oriented Petri Nets." *European Simulation Multiconference*, Barcelona, Spain, June 1994. (1994)
18. Leavens, Gary T., and Yoonsik Cheon. *Extending CORBA IDL to Specify Behavior With LARCH,* TR #93-20. Iowa State University, Department of Computer Science, 1995.
19. Merle, Philippe, Gransart, Christophe, Geib, Jean-Marc, and Laukien, Marc. "The CorbaScript Language." *ORBOS OMG Meeting on Scripting Languages*, Helsinki, Finland, July 27-31 1998. (1998)
20. Meyer, Bertrand. "Systematic Concurrent Object-Oriented Programming." *Communications of the ACM* 36, no. 9 (1993) 56-80.
21. Netscape Communications Inc. *CORBA Component Scripting OMG TC Revised Joint Submission. Orbos/98-07-02*, Framingham, MA (1998).
22. Object Management Group. *The Common Object Request Broker: Architecture and Specification. CORBA IIOP 2.2 /98-02-01*, Framingham, MA (1998).
23. ———. *Common Object Services Specification /98-07-05*, Framingham, MA (1998).
24. Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems* Prentice-Hall (1981).
25. Prinz, Andreas. "Describing Behaviour in Interfaces." *Formal Methods for Open Object-Based Distributed Systems.* Elie Najm, and Jean Bernard Stefani, editors. Chapman & Hall (1997) 36-43.

26. Puntigam, F. "Types for Active Objects Based on Trace Semantics." *Formal Methods for Open Object-Based Distributed Systems.* Elie Najm, and Jean Bernard Stefani, editors. Chapman & Hall (1997) 4-19.
27. Ramamoorthy, C. V., and Ho, G. S. "Performance Evaluation of Asynchronous Concurrent Systems." *IEEE Transactions of Software Enginnering* 6, no. 5 (1980) 440-449.
28. Rational Software Corporation. *UML Notation Guide*. 1.1 ed.1997.
29. ———. *UML Semantics*. 1.1 ed.1997.
30. Sankar, Sriram. "Introducing Formal Methods to Software Engineers Through OMG's CORBA Environment and Interface Definition Language." *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, Munich, Germany, July 1-5, 1996. Martin Wirsing, and Maurice Nivat, Editors. ed. Lecture Notes In Computer Science, no. 1101. Springer (1996) 52-61.
31. Sibertin-Blanc, Christophe. "Cooperative Nets." *15th International Conference on Applications and Theory of Petri Nets, ATPN'94*, June 1995. Lecture Notes in Computer Science, no. 815. Springer (1995) 471-90.
32. Siegel, Jon. "OMG Overview: CORBA and the OMA in Enterprise Computing." *Communications of the ACM* 41, no. 10 (1998) 37-43.
33. Sivaprasad, Gowri Sankar. *Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces*, TR #95-27a. Iowa State University, Department of Computer Science, 1995.
34. Valk, Rüdiger. "Petri Nets As Token Objects: an Introduction to Elementary Object Nets." *19th International Conference on Applications and Theory of Petri Nets, ATPN'98*, Lissabon, Portugal, June 1998. Springer (1998)
35. van der Aalst, W. M. P., and Basten, T. "Life-Cycle Inheritance, a Petri-Net Based Approach." *18th International Conference on Applications and Theory of Petri Nets, ATPN'97*, Toulouse, France, June 1997. Pierre Azéma, and Gianfranco Balbo, editors. Lecture Notes in Computer Science, no. 1248. Springer (1997) 62-81.