A Formal Specification Framework for Object-Oriented Distributed Systems

Didier Buchs and Nicolas Guelfi

Abstract—In this paper, we present the Concurrent Object-Oriented Petri Nets (CO-OPN/2) formalism devised to support the specification of large distributed systems. Our approach is based on two underlying formalisms: order-sorted algebra and algebraic Petri nets. With respect to the lack of structuring capabilities of Petri nets, CO-OPN/2 has adopted the object-oriented paradigm. In this hybrid approach (model- and property-oriented), classes of objects are described by means of algebraic Petri nets, while data structures are expressed by order-sorted algebraic specifications. An original feature is the sophisticated synchronization mechanism. This mechanism allows to involve many partners in a synchronization and to describe the synchronization policy. A typical example of distributed systems, namely the *Transit Node*, is used throughout this paper to introduce our formalism and the concrete specification language associated with it. By successive refinements of the components of the example, we present, informally, most of the notions of CO-OPN/2. We also give some insights about the coordination layer, Context and Objects Interface Language (COIL), which is built on top of CO-OPN/2. This coordination layer is used for the description of the concrete distributed architecture of the system. Together, CO-OPN/2 and COIL provide a complete formal framework for the specification of distributed systems.

Index Terms—Formal specifications, object-orientation, distributed systems, concurrency, algebraic Petri nets, refinement, subtyping, algebraic specifications.

1 Introduction

For important applications, distributed processing provides a general, flexible, and evolutionary approach. Distributed processing offers many advantages: availability and reliability through replication; performance through parallelism; sharing and interoperability through interconnection; flexibility and incremental expansion through modularity.

However, distributed systems introduce several new considerations that must be necessarily taken into account. Interactions between independent components give rise to new issues such as nondeterminism, contention, and synchronization.

In a formal and rigorous specification process, it is necessary to have a sound mathematically-based formalism which allows to express all of the characteristics of the systems. When we are faced with large problems, it is also necessary to have at one's disposal some structuring facilities. Moreover, the ever-increasing complexity of software systems imposes a progressive adaptation based on abstraction, refinement, and enrichment. Thus, specifying a system in a formal and incremental way offers several benefits. Indeed, the initial perception of the system to be built may be very vague. As the analysis and the simultaneous validation progress, the definition of the architecture, of the algorithms, and of the associated data structures gradually improve; finally, the implementation can take form.

In this paper, we present a formal framework for the development of distributed systems. The approach we propose has adopted the object-oriented paradigm as a structuring principle. On the one hand, we have devised a general formalism which can express both abstract and concrete aspects of systems, with emphasis on the description of concurrency and abstract data types. This approach, called Concurrent Object-Oriented Petri Nets (CO-OPN/2) extends its object-based predecessor COOPN [9]. On the other hand, a coordination layer, called Context and Objects Interface Language (COIL), has been developed on top of this formalism in order to be able to deal with distributed architectures. The objective of this paper is to introduce our approach in an intuitive way. Thus, the presentation is organized around expressiveness aspects and methodological considerations rather than theoretical matters which are described in [6] and [7], with respect to CO-OPN/2, and in [13] for COIL.

For the presentation of our approach, we use a typical example of distributed systems, namely the Transit Node (TNode), and progressively build a distributed communication system composed of TNodes. The presentation is divided into several steps. The first one deals with the abstract data types used in the example and constructs a basic version of the communication system in which all the TNodes can transmit messages to each other. This first simple version allows us to introduce most of the fundamental concepts of CO-OPN/2. The second step enriches the basic system regarding the data input/output ports. In the third step, the routes are added; the construction of this new system is based on inheritance.

- *D. Buchs is with the Software Engineering Laboratory, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland. E-mail: Didier.Buchs@di.epfl.ch.*
- *N. Guelfi is with the Department of Applied Computer Science, IST–Luxembourg University of Applied Sciences, 6 rue Richard Coudenhove–Kalergi, L-1359, Luxembourg-Kirchberg, Luxembourg. E-mail: Nicolas.Guelfi@ist.lu.*

Manuscript received May 1998; revised Feb. 1999; accepted Feb. 1999. Recommended for acceptance by E. Brinksma. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111432.

A more realistic system of a heterogeneous distributed system is then introduced. This system takes into account the timing information, the error detection algorithms, and the heterogeneous distributed systems gateway. A last evolution of the communication system introduces the coordination layer COIL to support the design of distributed architectures.

Finally, we discuss some semantic aspects of CO-OPN/2, and present the state of the art of various object-oriented formal methods based on a work that we did with other colleagues on this subject [16].

2 CO-OPN/2 Principles

In order to give the reader a quick overview of our approach, this section introduces the main characteristics of CO-OPN/2. More detailed explanations as well as illustrations of these features will be provided throughout the paper.

CO-OPN/2 is a formalism devised for the specification and the modeling of large concurrent systems. The two underlying formalisms of CO-OPN/2 are the algebraic specifications and the Petri nets which are combined in a way similar to algebraic nets [24]. The former is used to describe the data structures and the functional aspects of a system, while the latter is used to model its operational and concurrent characteristics. However, both these formalisms are not suitable to specify "in the large." To compensate for the lack of structuring capabilities of the Petri nets, the object paradigm has been adopted by the CO-OPN/2 approach. Thus, a system is considered as being a collection of independent entities which interact and collaborate with each other to accomplish the various tasks of the system.

In order to overcome some limitations of its predecessor, CO-OPN/2 introduces some notions specific to object-orientation such as the notions of class, inheritance, and subtyping. Moreover, order-sorted algebraic specifications [15] which support subsorting have been adopted for the description of data structures.

2.1 Object and Class

An object is considered as an independent entity composed of an internal state and which provides some services to the exterior. The only way to interact with an object is to request its services; the internal state is then protected against uncontrolled accesses. Our point of view is that encapsulation is an essential feature of object-orientation and there should not be any means to violate it.

CO-OPN/2 defines an object as being an encapsulated algebraic net in which the places compose the internal state and the transitions model the concurrent events of the object. A place consists of a multiset of algebraic values. The transitions are divided into two groups: The methods—transitions which may possess parameters—and internal transitions. The former correspond to the services provided to the outside, while the latter compose the internal behaviors of an object. Contrarily to the methods, the internal transitions are invisible to the exterior world and may be considered as being spontaneous events. Later on the term event is often used as synonym of method or internal transition.

An important characteristic of the systems we want to consider is their potential dynamic evolution in terms of the number of objects they may include. Thus, the dynamic creation of objects is a major objective. In order to describe similar dynamic evolving systems, the objects are grouped into classes. A class describes all the components of a set of objects and is considered as a template from which objects are instantiated. Thus, all the objects of one class have the same structure.

2.2 Object Interaction

In the CO-OPN/2 approach, the interaction between objects is synchronous. That is to say, when an object requires the service of another object, it requests a synchronization with the method of the object provider.

An innovative feature of CO-OPN/2 is the possibility of expressing the synchronization policy by means of a synchronization expression attached to an internal transition or a method. Such a synchronization expression[1] may involve the events of many partners joined by three synchronization operators (one for simultaneity, one for sequence, and one for alternative or nondeterminism). For example, a transition of an object can simultaneously request two different services from two different partners, followed by the service request to a third object.

From an operational point of view, the synchronization mechanism can be viewed as a generalization of the "rendezvous" or transaction mechanism found in other synchronous approaches. Informally, the event of a given object which requests a synchronization with events of several partners can occur, if and only if:

1. The conditions or guards[2] of the caller object transition are satisfied.
2. All the events of the partners involved in the synchronization can occur according to the policy of the synchronization expression.

It is important to note that a method should not be considered as a function which returns a value. Indeed values are exchanged between the partners involved in a synchronization by the unification of all the formal and actual parameters. Hence, the data flow of a synchronization is not explicitly expressed here at the specification level, but at a more concrete level, i.e., in the coordination layer. Several examples of synchronization as well as additional remarks will be provided throughout the paper.

Regarding the synchronous aspects, CO-OPN/2 can be assimilated to other strong synchronous approaches, like Esterel [5] and Statecharts [17], in the sense that any operation, event, synchronization, or change of state occur instantaneously. Nevertheless, the underlying semantics of CO-OPN/2 is not restricted to finite state automata.

2.3 Concurrency

Intuitively, each object possesses its own behavior and evolves independently from or, let us says concurrently with, the others. This is known as inter-object concurrency.

1. How synchronization expressions are established is explained in Section 4.3.

2. The global pre and postconditions of Section 4.3 can be viewed as guards.

When operations of an object can be performed independently, we call this intraconcurrency. The Petri net model introduces naturally both interobject and intraobject concurrency into CO-OPN/2, because objects are not restricted to sequential processes.

The CO-OPN/2 semantics is expressed in terms of transition systems whose the events are either atomic or composite. Atomic events are methods, while composite events are events corresponding to synchronization expressions. A composite event is used to describe the behavior of an object for a complex call by a client object of its methods.

The step semantics of CO-OPN/2 allows for the expression of true concurrency which is not the case of interleaving semantics. Nevertheless, the purpose of CO-OPN/2 consists in capturing the abstract concurrent behavior of each modeled entity, with the concurrency granularity associated to method invocations rather than to objects. A set of method calls can be concurrently performed on the same object. Furthermore, it must be noticed that as internal transitions are stabilized after a method call, then the overall concurrent behavior is seen atomic from the caller object point of view.

2.4 Object Identity

Within our framework, each class instance has an identity, also called an object identifier, that is used as a reference. Because a type is associated with each class, each object identifier belongs to at least one type (many in case of subtyping).

An original feature of CO-OPN/2 is the construction of a specific order-sorted algebra for the management of object identifiers. This order-sorted algebra is constructed in order to reflect the subtyping relationship between class types, i.e., two carrier sets of object identifiers are related by subsorting (or inclusion) if, and only if, the two corresponding types are related by subtyping. This order-sorted algebra is equipped with operations that are used to define membership predicates over types.

Since object identifiers result from an order-sorted algebra, they can be handled as algebraic values. Thus, it is possible, on the one hand, to store object identifiers in object attributes and, on the other hand, to define abstract data types built upon object identifiers, e.g., a stack or a queue of object identifiers.

2.5 Inheritance and Subtyping

We believe that inheritance and subtyping are two different notions which should be used for two different purposes. Inheritance is considered as being a syntactic mechanism which frees the specifier from the necessity of developing classes from scratch and is mainly employed to reuse parts of existing specifications. A class may inherit all the features of another and may also add some services, or change the description of some services already defined.

Our subtyping relationship is based on the strong version of the substitutability principle [2], [20]. This principle implies that, in any context, any class instance of a type may be substituted for a class instance of its supertype while the behavior of the whole system remains unchanged. In other words, the instances of the subtype have a strong semantic conformance relationship with the supertype definition. This conformance relationship is based, in CO-OPN/2 , upon the bisimulation between the semantics of the supertype and the semantics of the subtype restricted to the behavior of the supertype.

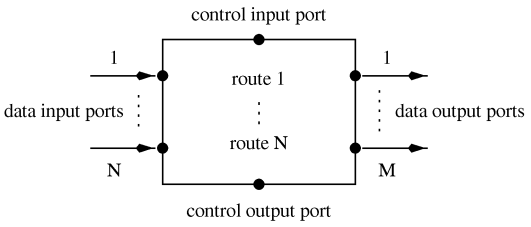


Fig. 1. The Transit Node.

Both inheritance and subtyping relationships must be explicitly given but the respective hierarchies generated by these relationships do not necessarily coincide. In other words, two classes related by inheritance are not necessarily related by subtyping. Identifying both inheritance and subtyping hierarchies leads to several limitations as pointed out by Snyder [26] and America [1], and as illustrated by our approach.

3 The Transit Node Case Study

This section introduces the CO-OPN/2 approach by means of a well-known case study, the Transit Node (TNode). A transit node is a node in a communication system which receives messages on various input ports and routes them toward various output ports. This case study was defined in the RACE Project 2039 and one may find assorted specifications of the transit node in [22], [23]. Slight changes have been made to the TNodes definition in the RACE project and an informal description of the transit node is given in Section 3.1.

Our aim is to develop a heterogeneous distributed communication system—which consists of different kinds of TNodes interconnected by means of wires—and to introduce progressively the syntactic and semantics aspects of CO-OPN/2. Among the concepts we are going to present, the concepts of data types and algebraic nets, as well as the object-oriented notions (including class definition, creation of dynamic objects, inheritance as well as subtyping) are particularly important. We begin by describing a basic and abstract version of the TNodes and the wires. Therefore, this allows us to present the main ideas of the language and to enrich and refine this version progressively.

3.1 Informal Description of the TNode

The RACE project has defined a Transit Node as being a node in a communication system which receives messages on its input ports, and then routes them onwards its output ports, according to some designated route.

A TNode, as depicted in Fig. 1, consists of N data input ports, M data output ports, N routes, one control input port, and one control output port. Each port is serialized and represents a specific entity which is independent from all others. The node is "fair," i.e., all messages are treated equally when a selection has to be made. Furthermore, all messages will eventually leave the node, or be placed within a collection of faulty messages. The control ports can

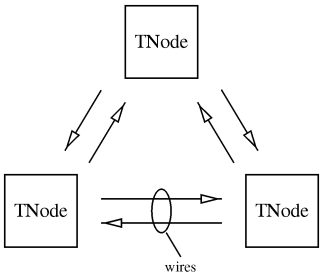


Fig. 2. A system of TNodes.

be used to configure the TNode or to get statistical informations such as the number of faulty messages, the average transit time, etc.

3.2 TNode-Based Distributed System

A system of interconnected TNodes is called a distributed communication system. The interconnection is realized by means of wires, each of which links an input port of one TNode to an output port of another TNode. Fig. 2 illustrates a distributed system composed of three interconnected TNodes which are identical.

However, a distributed communication system may involve different kinds of TNodes, and form what we call a heterogeneous distributed communication system. Fig. 3 shows such a communication system based on four different types of TNodes (i.e., TN_a, $TN_{a'}$, TN_b, $TN_{b'}$). Four types of links are used to ensure communications between them. In those distributed systems, TNodes, wires, and messages can be of different nature. Then, complex organizing notions such as communication protocol, gateways, etc. have to be defined.

3.3 Presentation Overview

Our informal introduction of CO-OPN/2 is divided into two stages, which are then split into different steps:

- In the first stage we begin to specify, on the one hand, the data types which are used in the example by means of algebraic specifications and, on the other hand, a basic version of the TNodes in which all TNodes can transmit messages to each other. The second step is concerned with the enrichment of the basic system regarding the data input/output ports. In the third step, the routes are added. The fourth step introduces the input/output control ports and a communication system which involves two different kinds of TNodes is finally developed.
- In the second stage, a more realistic example of a heterogeneous distributed communication system is added. It takes into account the timing informations, the error detection algorithms and the heterogeneous distributed systems gateway.

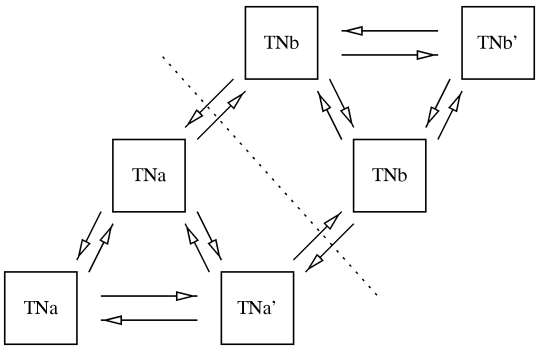


Fig. 3. A heterogeneous system of TNodes.

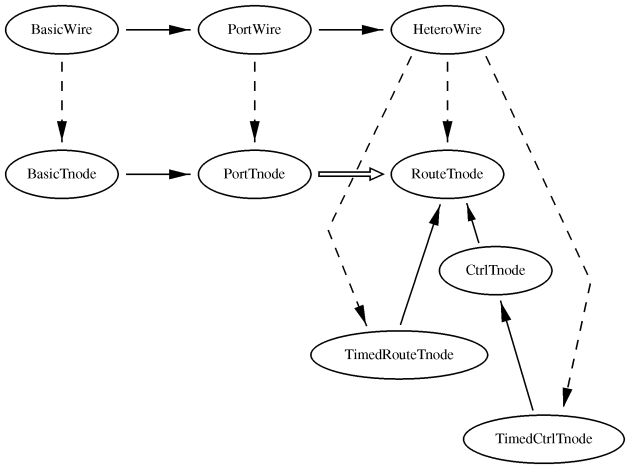


Fig. 4. Relation between the different kinds of TNodes and wires.

In Fig. 4, the organization scheme of all these steps is described by a graph of the entities of the different evolution phases using the following conventions:

- Dashed arrows indicate the synchronization requests between class instances (not the data flow).
- Thick white arrows indicate inheritance.
- Vertical solid arrows indicate subtyping.
- Horizontal solid arrows describe the evolution of classes through different development steps.

Since a TNode-based distributed system is viewed as composed of a communication layer (the wires) and a transmission layer (the nodes), we have two parts in the specification corresponding to these two layers.

4 ADT and Class

This section presents the first step of our progressive presentation of CO-OPN/2 using the concrete specification language associated with it. In this first step, we define the basic version of the TNode and the basic communication layer. These definitions require the description of the basic structure of a CO-OPN/2 specification, as well as the introduction of several fundamental concepts such as the concept of abstract data type, class, and cooperation between class instances using synchronization expressions.

4.1 Basic Structure

A CO-OPN/2 specification is composed of two kinds of modules: the abstract data type modules and the class modules; the underlying formalisms of these two kinds of modules are the order-sorted algebraic specifications and the algebraic Petri nets, respectively.

Both kinds of modules have the same basic structure, which is composed of three parts:[3] header, interface, and body.

1. The header section includes the information about inheritance and genericity; nongeneric modules developed from scratch begin with the **ADT** or **Class** keywords[4] followed by the module name.
2. The **Interface** section describes which components of the modules are accessible by other modules. The modules, which need components defined in other modules have to mention the module names where these components are defined.
3. The **Body** section primarily conceals the properties of the ADT operations, or the behavior, and the state of the class instances.

4.2 Abstract Data Types

Abstract data type modules are devoted to the data structures of the specification described algebraically.

In the case of ADT modules, the interface section includes usually the sorts, the generators, and the operations with their arity under the respective fields **Sorts**, **Generators**, and **Operations**. As for the body section, it includes the properties of the operations under the **Axioms** field and the required variable names are given under the **Where** field. The properties of the operations consist in positive conditional equations established as follows:

$$[Cond \ =>] \ Expr_1 = Expr_2.$$

Each equation relates two expressions $Expr_1$ and $Expr_2$, which states that both expressions denote the same value. An optional condition $Cond$ which determines the context in which an axiom holds true may be added.

Specification 1 groups three ADT modules. We can see the well-known modules of booleans and naturals with their operations and arities under the fields **Generators** and **Operations**. Note that the **Use** field in the `Natural` module is followed by the modules used by the module itself, here the module `Boolean`.

At this stage of the specification, all the messages transmitted through the communication system, consist simply (and deliberately) in the unspecified values of sort `message` as defined in the `Message` ADT module. At first glance such a specification does not seem pertinent, however, the details of the messages are not relevant for the moment. We will introduce later a more concrete kind of message.

Specification 1. Abstract Data Types

```
ADT Boolean;
Interface
   Sort boolean;
   Generators
     true, false: -> boolean;
   Operations
     not _ : boolean -> boolean;
     _ and _ : boolean boolean -> boolean;
Body
   Axioms
     not (true)      = false;
     not (false)     = true;
     true and      x = x;
     false and     x = false;
   Where x:boolean;
End Boolean;

ADT Natural;
Interface
   Use Boolean;
   Sort natural;
   Generators
     0 : natural;
     succ _ : natural -> natural;
   Operations
     _ + _ : natural natural -> natural;
     _ < _ : natural natural -> boolean;
     ; ; other operations
Body
   Axioms
     0+n = n; (succ n)+m = succ(n+m)
     0<0 = false; (succ n)<0 = false;
     0<(succ n) = true;
     (succ n)<(succ m) = n<m;
     ;; and their associated axioms
   Where n,m:natural;
End Natural;

ADT Message;
Interface
   Sort message;
End Message;
```

4.3 Basic Transit Node

The first entities of our example which have an internal state are the basic TNodes. A basic TNode consists of a simple, unstructured buffer in which messages arrive on an input port and remain inside the buffer until they are routed towards the output port.

We may notice that the **Use** field allows the `BasicTnode` class module to use the sort `message` defined in the `Message` ADT module. The **Type** field defines the `basictnode` type which will be used whenever object identifiers of basic TNodes will be required. This field has been introduced in order to avoid any confusion between the name of a module or a class and the name of its type, especially in cases where inheritance and subtyping are required. Both names are often very similar but address two different concepts. The **Methods** field lists all the parameterized events which are visible from the outside, here the `input` and `output` methods. A method must be interpreted as a predicate defined, similarly as operations, with an arity given by a list of sorts.

The last three fields remain in the **Body** section, ensuring encapsulation. In the field **Places**, the attribute `msg` consists in a multiset containing values of sort `message`. The two behavioral axioms under the **Axioms** field are quite simple. They do not entail neither conditions nor

3. Specifications 1 and 2 may help the reader to understand the meaning of the keywords introduced here.

4. Bold face keywords correspond to reserved words of the concrete language. These words are not case-sensitive and some of them can be singular or plural.

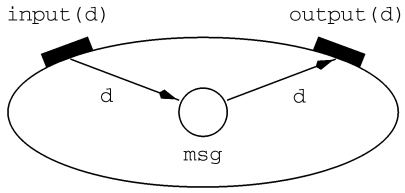


Fig. 5. Outline of the basic TNodes.

synchronization expressions. They simply express the fact that the formal argument d (defined in the **where** field) is added or removed from the msg multiset whenever the input or output method is invoked.

The properties of the methods and the internal transitions are described by means of behavioral axioms within the **Axioms** field. It is necessary to recall that an event (method or internal transition) may ask to be synchronized with other partners by means of a synchronization expression. The synchronization expressions are declared after the **With** keyword. The usual dot notation has been adopted and three synchronization operators are provided: '//' for simultaneity '..' for sequence, '+' for alternative. Behavioral axioms are established as follows:

$$[Cond =>]\ Event\ [\ \texttt{With}\ Sync\ ]\ ::\ Pre -> Post,$$

where *Cond* is an optional condition imposed on the algebraic values involved in the behavioral axiom, *Event* is either an internal transition name or a method along with its parameters, and *Sync* is an optional synchronization expression. *Pre* and *Post*, respectively, correspond to what is consumed and what is produced in the different places which compose the net.

Specification 2. The Basic TNodes

```
Class BasicTnode;
Interface
  Use Message
  Type basictnode;
  Methods
    input _ ,
    output _ : message;
Body
  Place msg _ : message;
  Axioms
    input (d) : : -> msg d ->;
    output (d): : msg d -> ;
  Where d:message;
End Basictnode;
```

Fig. 5 shows the graphic representation of the BasicTnode class module. Indeed, for each class module, a natural graphic representation can be depicted. This graphical representation is a partial view or an outline of the textual representation in which the following conventions are adopted:

- The inside of the ellipses represents what is encapsulated.
- The black rectangles represent the methods.
- The white rectangles correspond to the internal transitions;
- the gray rectangles correspond to the special method which takes charge of the dynamic creation and the initialization of the class instances, if no creation methods are specified, then the predefined **Create** method is not drawn.
- The circles identify the places or the attributes of the objects.
- The solid arrows indicate the data flow.

4.4 Basic Communication Layer

At this point, the communication layer is considered as the message passing from any TNode to any other TNode at any time. This is accomplished by synchronizing the events "message output" and "message input" of all TNodes. This abstract representation will be refined and enriched later.

The basic communication layer is represented by an instance of the class module given in Specification 3. This class is equipped with an internal transition msg-passing in the field **Transitions**. This internal transition requests the simultaneous synchronization between two TNodes. The behavioral axiom associated to the transition should be read "the internal event msg-passing behaves in the same way as both the simultaneous external events input and output of any different partners that can be identified by two different object identifiers of TNode tn1 and tn2."

Specification 3. Basic Communication Layer Classes

```
Class BasicWire;
Interface
  Use BasicTnode;
  Type basicwire;
Body
  Transition msg-passing;
  Axiom
    ! (tn1=tn2) -> msg-passing With
       tn1.input(d) // tn2.output(d):: -> ;
  Where  d:message;  tn1,tn2:basictnode;
End BasicWire;
```

Note the use of both the free object identifiers variables in the condition !(tn1=tn2) which imposes that the communication can be performed between any two different TNodes. The synchronization expression tn1.input(d) // tn2.output(d) involves the two partners tn1 and tn2 and their respective methods by means of the simultaneous operator "//." Finally, the pre- and post-conditions are left empty because the transition does not involve any attribute.

Fig. 6 depicts the outline of the BasicWire class. The dashed arrows correspond to the synchronization request between the communication layer and any two different TNodes. The direction of these arrows does not represent the data flow of the synchronization but the dependency relationship between the modules and, consequently, between the class instances.

4.5 Discussions

In this first step, we defined a simple version of the transit node which consists in a unstructured buffer equipped with an input and an output port. The basic communication layer is defined at a abstract level in the sense that only one instance of the class BasicWire models all the wires which link all the basic TNodes. At a first glance, this remark

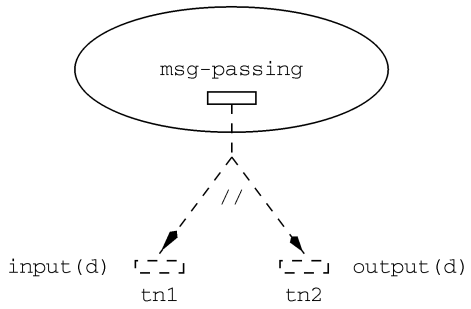


Fig. 6. Outline of the basic communication layer class.

seems trivial, but the use of free variables in the behavioral axiom of the `BasicWire` provides a nice abstract expression power. Actually, all the communications are the result of the concurrent occurrences of the same event.

Now, the questions that have to be considered are:

- How could we refine the basic TNode in order to obtain explicit distinct communication ports instead of a global and abstract mechanism?
- How could we specify the definition of a distributed system topology?

5 Enrichment and Object Identifiers

This step covers the enrichment of the basic TNodes and the basic communication layer. The notion of port is added, i.e., a message arrives at one of the N input ports of a TNode and leaves the TNode from one of the M output ports. As for the communication layer, the attribute of a wire contains the references of the two TNodes it connects.

5.1 TNodes and Communication Ports

A TNode has N input ports and M output ports. The messages arrive concurrently at the input ports, they are stored in the buffer and leave the node onwards from the output ports. A communication is viewed as the passing of a message through a wire which links one of the N input ports and one of the M output ports of two TNodes. A TNode can communicate with itself and several messages can be received or transmitted through the same port. However, the input and output ports are different, even if they use the same port number.

5.2 CO-OPN/2 Specifications

Both classes `BasicTnode` and `PortTnode` are similar. The introduction of the notion of port induces, of course, a new profile for the methods `input` and `output`, as well as a new behavior. These changes are given in Specification 5 and in Fig. 7. The specification of sort `port` (not detailed here) is built from the natural numbers and the constants `M` and `N`.

Specification 4. An Abstract Data Type of a Link

```
ADT Link;
Interface
  Use Port, PortTnode;
  Sort link;
  Generator
    <_ _ _ _> ; porttnode, port,
                porttnode, port - > link;
End Link;
```

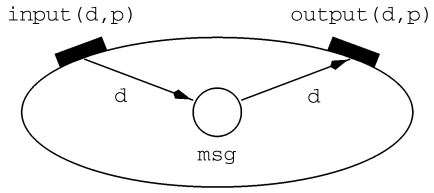


Fig. 7. Outline of TNodes with communication ports.

For the class `PortWire`, we need the specification of a link between two TNodes. This is realized by the module `Link` in Specification 4. The sort `link` represents a cartesian product of four components, the two TNodes references and their respective input/output port. Specification 6 outlined in Fig. 8, gives the specification of the class `PortWire`. We observe that the event `msg-passing` takes into account the attribute `linked`. Moreover, the special method `init` takes charge of the dynamic creation of the class instances and initializes the `linked` attribute with its parameter `ln` of sort `link`. This initialization ensures that only one quadruple is present in the place `linked`.

Specification 5. TNodes with Communication Ports

```
Class PortTnode;
Interface
  Use Message, Port, Natural;
  Type porttnode;
  Methods
    input  _ _,
    output _ _ : message port;
Body
  Place msg _ : message;
  Axioms
    p<N = true => input(d,p)   :: - > msg d;
    p<M = true => output (d,p) :: d - > ;
  Where d:message; p:port;
End PortTnode;
```

5.3 Discussions

Now, we have reached a more realistic TNode definition in which each TNode has several distinct communication ports. The communications are made through wires which are instances of the class `PortWire`, and which link explicitly input ports to output ports. Thus, it is possible to define the network topology since wire parameters are given at the instance creation. A wire does not support

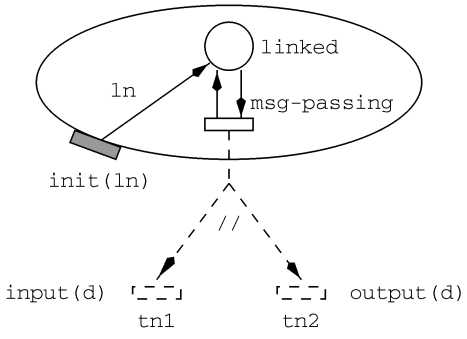


Fig. 8. Outline of wires between TNodes with ports.

simultaneous message passing and no routing technique is used inside a TNode.

The questions that have to be considered are:

- How could we easily serialize the access to the TNode ports and add routing features?
- What is the specification of an unreliable distributed system which could lose or corrupt messages?

Specification 6. Wires between Tnodes with Ports

```
Class PortWire;
Interface
  Use Message, Port, PortTnode, Link;
  Type portwire;
  Creation init _ : link;
Body
  Transition msg-passing;
  Place linked : link;
  Axioms
    msg-passing With
      tn.input(d,p)//tn'.output(d,p') ::
        linked <tn p tn' p'>
        -> linked <tn p tn' p'>;
    init (ln) :: -> linked ln;
  Where d:message; ln:link;
        p,p' :port; tn, tn' : porttnode;
End PortWire;
```

6 Inheritance

This enrichment step has the objective of introducing the routing part of the TNode. Inheritance is used in order to derive the new classes corresponding to the TNode and to the wire template. Both the previous classes are reused, some services as well as new attributes are added or redefined.

6.1 Transit Nodes and Routing

Each TNode includes information for the routing of the messages onward the output ports. This information associates each input port with a set of permissible output ports. It is essential that the routing information of a TNode can be modified if necessary.

Communication remains almost identical except that a wire now links two TNodes equipped with the new type of routing. The following sort `portset`, which is necessary to develop the specifications associated to the TNodes and the wires, is not detailed; it is obtained by instantiation of a generic algebraic sets specification with the module `Port` as actual parameter.

Specification 7. Reuse of the `PortWire` Class

```
Class PortWire;
Inherit PortWire;
  Rename portwire -> routewire;
  Rename PortTnode - > RouteTnode;
End RouteWire;
```

6.2 CO-OPN/2 Specifications

A new attribute and two new events are inserted. These are the route definition method `routedef` and the internal event `loss-msg`. Their respective behavioral axioms are

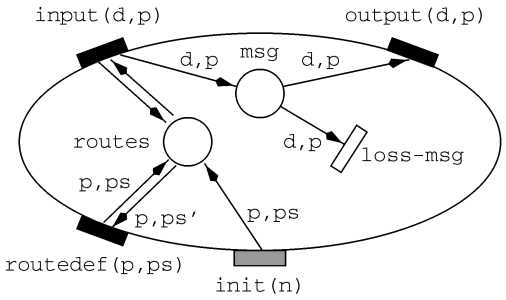


Fig. 9. Outline of the TNodes with routes.

added in the class `RouteTnode`. The class `RouteTnode` in Specification 8 (along with its graphic outline in Fig. 9) is not developed from scratch, it uses the inheritance mechanism which allows to reuse or redefine some components. Note that this example is rather poor in terms of reuse, because only the state definition is reused.

The keyword **`Redefine`** expresses that the behavioral axioms of inherited method are ignored and redefined by means of the new ones given in the current class. This is the case of methods `input` and `output`. Moreover, the principle may also be applied to inherited attributes but in this case we use the The keyword **`Undefine`** because we want to suppress everything concerning the inherited place `msg`. Routes are used to determine the message port destination. The new internal event `loss-msg` indicates that some messages can be lost in the TNode according to a given criterion. At this point the criterion is very abstract and represented by means of a function called `losscrit`. We assume that this function is defined in the module `LossCriteria`, which is not detailed here. The route wire is not changed, we only define a new class that inherits all the preceding properties (c.f. Specification 7).

The initialization `init` is defined recursively on the number of ports of the TNode. The free variable `ps` must be interpreted as a definition of a random choice of the routes. Note that the **`Use`** field restrict the visibility of the module `LossCriteria` to the body section.

Specification 8. Transit Nodes with Routes

```
Class RouteTnode;
Inherit PortTnode;
  Redefine input _ _, output _ _;
  Undefine msg _;
  Rename porttnode - > routetnode;
Interface
  Use Message, PortSet;
  Method routedef _ _ ; port portset;
  Creation init _ : natural ;
Body
  Use LossCriteria;
  Places routes _ _ : port portset;
         msg _ _ : message port;
  Transition loss-msg;
  Axioms
    p < N = true =>
    input(d,p) :: routes(p,ps)
      -> msg(d,select(ps)),routes(p,ps);
    p < M = true =>
```

```
    output(d,p) :: msg(d,p) -> ;
       routedef(p,ps) ::
       routes(p,ps') -> routes(p.ps);
    losscrit(d) = true =>
    loss-msg :: msg(d,p) -> ;
    init(0) :: -> routes(p,ps);
    p <N = true =>
    init (succ p) With Self.init (p) ::
      -> routes(p,ps);
  Where n:natural; d:message;
        p:port;    ps,ps':portset;
End RouteTnode;
```

6.3 Discussions

The routing and communication interferences have been easily obtained from a previous development step using CO-OPN/2 refinement techniques based on inheritance. The serialization of the input ports is performed through the routing mechanism, which is decided at message arrival, while no output order is imposed. The route initialization is implemented using a recursive method call. The questions that must be considered now are how to refine the TNodes in order to implement the communication interferences using an error detection code, and how we can collect messages that have to be recovered by the system manager. Moreover, we would like to achieve this specification without modifying the definition of the communication layer.

7 Inheritance Versus Subtyping

The previous versions of the running example led to a simple distributed system composed of transit nodes of class `RouteTnode` linked using basic wires. The `RouteTnode` class inherits from `PortTnode` without any subtype relation. Thus, it is not possible to define a distributed system made of `RouteTnode` and `PortTnode` objects linked by means of the wire class because they are not substitutable. In this version, we define a new class of transit node `CtrlTnode` in order to illustrate the notion of subtype, which is used to specialize the previous TNode into a particular error detection and recovery mechanism.

These new transit nodes filter erroneous messages as they did before, from the point of view of the output port i.e., messages that are corrupted when they are transiting inside the TNode. But these messages are redirected to a special port which can be used to collect faulty messages. This new class of TNode allows the client of the TNode to recover corrupt messages thanks to an error detection algorithm.

The class `CtrlTnode` is a subtype of `RouteTnode`. As previously explained, the subtyping relation is a semantic constraint. In our case, it means that each `CtrlTnode` behaves at least as a `RouteTnode`, in order to satisfy the substitutability principle. This notion of subtyping is strongly related to the notion of observational equivalence of objects and depends on the observers, which are generally the methods defined in the object interface. In our case, it is easy to show that the general class `RouteTnode` has a greater number of behaviors, due to the new method `collect-msg`. So, a `CtrlTnode` object can be substituted to a `RouteTnode` object without modifying the behavior of the system.

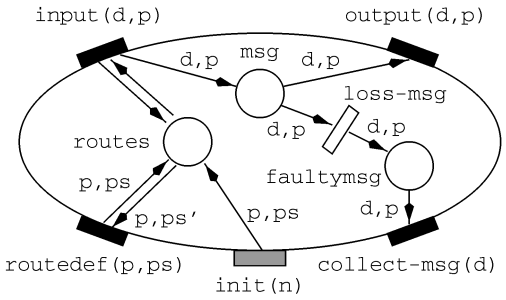


Fig. 10. Outline of TNodes with error message and recovering.

In our case, the objective is also to allow the use of polymorphic wires which can link `RouteTnode` as well as `CtrlTnode` because the message passing does not depend on the filtering semantics. Thus, the communication layer is exactly the same as in the previous part.

7.1 Design of the Loss of Messages

The state and event part is enriched from the previous version in order to collect all the faulty messages. Specification 9, as it's outlined in Fig. 10, shows the CO-OPN/2 specification of this class.

Specification 9. TNodes with Error Message and Recovering

```
Class CtrlTnode;
Inherit RouteTnode;
  Redefine loss-msg;
  Rename routetnode -> ctrltnode;
Interface
  Use Message;
  Subtype ctrltnode -> routetnode;
  Method collect-msg _ : message;
Body
  Use Message, PortSet, Port;
  Place faultymsg _ _ : message port;
  Axioms
    losscrit(d) = true =>
    loss-msg :: msg(d,p)-> faultymsg(d,p);
    collect-msg(d) :: faultymsg(d,p) ->;
  Where d:message;  p:port;
End CtrlTnode;
```

A class `CtrlTnode` inherits from the class `RouteTnode` and is extended with the following components: A place `faultymsg` contains values of the sort `messageport` which are the faulty messages; An internal transition `errdetect` selecting the erroneous messages; A method `collect-msg` with a parameter of sort `message` which is invoked to collect the faulty messages.

7.2 Discussions

Refinement using subtyping is exploited in order to create easily new kinds of TNodes which are compatible with their supertypes. This allows to modify the TNode without changing the communication layer. Moreover, timing considerations could be considered in order to control message contention. We may ask how to reach a complex heterogeneous distributed system in which time could be managed, and a gateway between heterogeneous subsystems established.

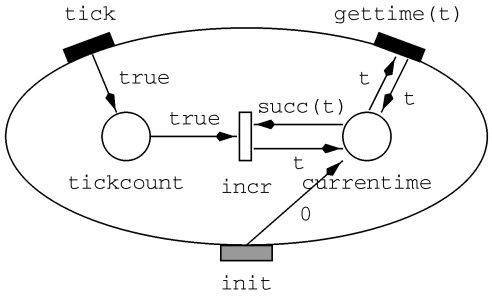


Fig. 11. Outline of the Clock Class.

8 Polymorphic References

In this part, we are interested in specifying the distributed systems which are made of two different kinds of nodes, and modeled by two different classes of TNodes and wires. The first kind of system is composed of TNodes and wires, as has been defined in the class `RouteTnode`, while the second is obtained from the first one by additionally managing timing information meant to model the timing latency inside the TNodes. This section introduces the ability of CO-OPN/2 to model heterogeneous systems.

8.1 Requirements and Design

Each new TNode has a local clock. The messages in transit across the system are composed of a timing information in addition to the useful data. The timing part indicates the duration of the total transit within the nodes having a clock. That is to say, if a message is a couple <t d> where `t` stands for the time information, then each TNode must increment `t` according to the number of time units between the message arrival and its time of departure. Furthermore, we must take into account the delay taken to pass through the physical links between TNodes.

We must transform the TNode of class `RouteTNode` by adding the clock features at the TNode level and by managing the transit delay. We do this for the TNode class `RouteTNode` and we suppose that it is done for the `CtrlTnode` class in the same way. Classes called `TimedRouteTnode` and `Clock` are defined. We suppose that there is also a class `TimedCtrlTnode` for the timed controlled TNodes.

8.2 The Clock Class

The time reference is given by one object of class `Clock`. This object provides two services. The time (described by the sort `time` as instance of naturals) is incremented after activating the method `tick`, and the current time value is given when activating the method `gettime`. Concurrent access to `gettime` and `tick` is allowed, although the method `gettime` is not self-concurrent and then sequentializes the access to the time reference.

We realize the clock (see Specification 10 and its outline in Fig. 11) using two places, the number of not yet registered `tick` which is equal to the number of token in the place `tickcount` and the current time value for the place `currenttime`. The internal transition `incr` immediately reacts after a `tick` event in order to adjust the `currenttime` value. This last event is necessary to make the methods `tick` and `gettime` concurrent.

Specification 10. The Clock Class

```
Class Clock;
Interface
   Use Time;
   Type clock;
   Methods
     gettime _ : time;
     tick;
   Creation init;
Body
  Use Boolean;
  Places
    currenttime _ : time;
    tickcount _ : boolean;
  Transition incr ;
  Axioms
    gettime(t) ::
      currenttime t -> currenttime t;
    tick :: -> tickcount true;
    incr :: tickcount true, currenttime t
            -> currenttime succ(t);
    init :: -> currenttime 0;
  Where t:time;
End Clock;
```

8.3 The TimedRouteTnode Class

Each `TimedRouteTnode` instance calls its associated local clock instance to get the time information needed to determine how long a message has been delayed inside. The new kind of messages are now couples of Time and Message in order to have an indication of the time spent inside the system.

The modeling of the messages has been made with sorts instead of classes for conceptual reasons. It implies that we cannot use subtyping to specialize `message` into `timedmessage` and consequently, we cannot have relations between those types. As defined before in `RouteTnode`, the data inside a TNode must be associated to an output port number, and furthermore with a time stamp. The main difference is that time stamping will be performed when entering the TNode and when leaving the TNodes. This allows for future versions of the TNodes to perform some latency control, in order to avoid excessive contention of messages inside a TNode. The new version of the TNodes, described in Specification 11, which includes a subtype relation, completely extends the previous one, including the synchronization with the clock `myclock` and the stamping of the messages. The clock is created at the initialisation of the nodes using the dedicated method `init` (this method has the number of routes as parameter).

Specification 11. The Timed TNode

```
Class TimedRouteTnode;
Inherit RouteTNode;
Interface
  Type timedtnode;
  Subtype timedtnode -> routetnode;
  Use TimedMessage, PortSet;
  Methods
    input _ _, output_ _ :timedmessage port;
```

```
    routedef _ _ : port portset;
  Creation init _ : natural;
Body
  Places
    routes : port portset;
    msg : time message time port;
    theclock : time;
  Axioms
    p < N => input(<t' d>,p)
      With myclock.getttime(t) ::
      routes(p,ps), theclock myclock
      -> msg(t',d,t,select(ps)),
        routes(p,ps), theclock myclock;

    p < M => output(<t - t''+ t' d>,p)
      With myclock.getttime(t) ::
      msg(t',d,t'',p), theclock myclock
      -> theclock my clock;

    routedef(p,ps) ::
      routes(p,ps') -> routes(p,ps);
    init (0) With myclock.init ::
      -> theclock my clock;
    init (succ(n)) With Self.init(n) ::
      -> routes(n,ps);
  Where n:natural; d:message;
        p:port; ps,ps':portset;
        t,t',t'':time; myclock:clock;
End TimedRouteTnode;
```

8.4 The Communication Layer

The idea is to give a description of how we can connect different TNodes expressed in specifications RouteTnode and TimedRouteTnode. The communication layer of class TimedRouteTnode integrates two new features: A transmission delay for the messages which are in transit between two TNodes of class TimedRouteTnode, and one for transmissions between TNodes of class RouteTnode and TNodes of class TimedRouteTnode. We have four kinds of wires defining the four different kinds of links between heterogeneous TNodes:

- The wires linking TNodes of class RouteTnode as it was presented before.
- The wires linking TNodes of class TimedRouteTnode. This is simply a rewriting of the class RouteTnode wire where message is replaced with timedmessage and the time stamp of the message entering a TNode is incremented by a constant representing the cost of the transit through the wires.
- The wires linking in an unidirectional way TNodes of class RouteTNode to TNodes of class TimedRouteTnode: The time stamp is set to zero when entering the TNodes of class TimedRouteTnode.
- The wire linking in an unidirectional way TNodes of class TimedRouteTnode to TNodes of class RouteTnode lose the time stamps attached to the messages.

The class HeteroWire described in Specification 12 shows the selection of the four kinds of transmissions between TNodes. The mechanism used to select the kind of TNode concerned by the transmission is based on the use of a fine discrimination of the reference hierarchy (routetnode is the supertype of timedtnode) using the following conditions:

- tn **isa** timedtnode is satisfied, if and only if, the value of tn, which is of type routetnode, belongs to the domain timedtnode.
- tn **isa** routetnode is satisfied, if and only if, the value of tn belongs to the domain routetnode but, not, to timedtnode.

This is a typing condition expressed on tn by means of the reference operations such as **isa** and **isany** which does not consider all the super types.

Specification 12. The wire for heterogeneous TNodes

```
Class HeteroWire;
Interface
  Use Link;
  Type wire;
  Creation init _ : link;
Body
  Use Message, Port, RouteTnode,
      TimedRouteTNode;
  Transition  msg-passing;
  Places linked : link;
  Axioms
    tn isa timedtnode
    & tn' isa timedtnode =>
    msg-passing
    With tn.input(<t+delay,d>,p)
        // tn'.output(<t d>,p')::
    linked <tn p tn' p'>
    -> linked <tn p tn' p'>;

    tn isa routenode
    & tn' isa timedtnode =>
    msg-passing
    With tn.input(d,p)
        // tn'.output(<t d>,p')::
    linked <tn p tn' p'>
    -> linked <tn p tn' p'>;

    tn isa timedtnode
    & tn' isa timedtnode =>
    msg-passing
    With tn.input(<0 d>,p)
        // tn'.output(d ,p')::
    linked <tn p tn' p'>
    -> linked <tn p tn' p'>;

    tn isa routetnode
    & tn' isa routetnode =>
    msg-passing
    With tn.input(d,p)
```

```
    // tn'.output(d ,p'):: 
    linked <tn p tn' p'>
    -> linked <tn p tn' p'>;

    init(ln) :: -> linked ln;
  Where d:message; ln:link;
        p,p':port; tn,tn':routetnode;
End HeteroWire;
```

8.5 Discussions

The local clock is modeled using reactive object instances, which could be controlled by a physical mechanism like a quartz system. Complex heterogeneous distributed systems are defined upon different message frames, transit time management schemes, network gateways, etc. In order to control access to different elements of the classes of the TNode hierarchy, fine reference control mechanisms are defined by the language to separate accesses to sub/superclass instances.

In the previous modeling, nothing was said about the way the whole system would be realized. It was naturally admitted that physical nodes and wires are modeled using the classes with the same names. This kind of implicit assumptions is difficult to deal with when we consider complex systems in which the boundaries of the system components are not known.

For instance, if we want to describe more elaborate messages, we can consider messages as objects. In this case, it appears that the use of object references as messages induces object localization problems.

In the following section, we will use an additional layer over CO-OPN/2 dealing with such types of information: Namely, the coordination layer that will be expressed with the language COIL.

9 Distributed Design of the Transit Node

This section introduces the management of concrete distribution using Context and Objects Interface Language (COIL) [13] coordination layer. COIL is a coordination model for distributed object systems. This model, namely the contextual coordination model, is based on execution contexts encompassing and coordinating objects. The configuration structures are given by means of hierarchies of contexts and objects, allowing to adapt the granularities of the processes and the objects, and providing localization information to the objects. Dynamic configurations, by means of object migrations, are also provided by the model and are usefull when the architecture of the distributed system changes during the system evolution.

9.1 The Transit Node with Active Messages

In this section, the evolution of the transit node model is continued in the direction of a distributed system in which the messages are not only data values but more complex entities possessing their own behavior. Other entities of the transit Node are taken from the simple model of Section 5.

9.2 Messages are Objects

Activities of messages are not fully detailed here, but they can be of various natures: check consistency of message, signal an alarm, etc. In order to illustrate more concretely these actions, we provide two operations which are supposed to model the possible alteration of messages and the correction of possible inconsistencies.

The active concrete messages are not precisely defined in this paper, they should be derived by subtyping from an abstract class given in Specification 13.

Specification 13. Messages as Objects

```
Abstract Class Message;
Interface
  Use Natural;
  Type message;
  Methods
    crc: natural;
    iserror;
Body
End Message;
```

9.3 Boundaries and Synchronizations

The components of the distributed system are obviously nodes and wires. The boundaries of these components are modeled by methods and gates.

In COIL, methods have the same meaning as in COOPN/2 , they provides services from a context while gates describes services *requested* by a context. The methods for input of TNodes and gates for outputs are described in Specification 14. Graphically a component is a gray circle with black and white boxes at its border, the black part being at the exterior for the methods and the white part at the exterior for the gates. Contexts have the same structure but they have a rectangular shape. This context encapsulates the object `tnode`, limiting the access to this object from outside of this context to the use of `DistributedTNode` context. The wires which are illustrating the physical connectors that exist in the reality are described in Specification 15. We transform the wire object previously defined into a component in which the methods that can be called are explicitely described by gates. All these contexts encapsulate the behavior of the entities `tnode` and `wire` and manage the object mobility. The whole system is modelled by synchronizing at the system context level (Fig. 12) the gate and methods belonging to the contexts.

9.4 Object Mobility

The intrinsic complexity of active messages as well as their localization need to introduce additional informations in the connection points as in Specification 14. The keywords **`Take`** and **`Give`** are used for that purpose in COIL, indicating a permanent localization change in a chosen direction. Other COIL keywords **`Lend`** and **`Borrow`** have the same meaning, but the move is valid for the duration of the transactions. It must be noted that only the move that can be observed after firing methods needs to be managed by **`Take/Give`** keywords as in the `DistributedWire` context.

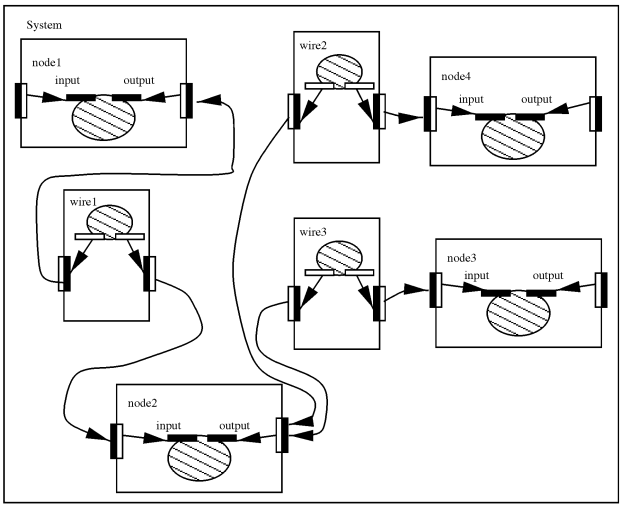


Fig. 12. A distributed system of TNodes.

Specification 14. TNodes Expressed by Contexts

```
Context DistributedTNode;
Interface
  Use Message, PortTNode;
  Object tnode:porttnode;
  Methods
    input _ _ : Give message, port;
  Gate
    output _ _ : Take message, port;
Body
  Axioms
    input(d,p) with tnode.input(d,p);
    Output(d,p) with tnode.output(d,p);
  Where d:message; p:port;
End DistributedTNode;
```

Specification 15. Wires Expressed by Contexts

```
Class ComponentPortWire;
Interface
  Use Message, Port;
  Type portwire;
Gates
  out: message port;
  inp: message port;
Body
  Transition msg-passing;
  Axioms
    msg-passing with
      inp(d,p) // out(d,p') ::
        -> ;
  Where d:message;
        p:port;
End ComponentPortWire;

Context DistributedWire;
Interface
  Use Message, ComponentPortwire;
  Object wire:portwire;
  Gate
    input _ _ : message, port;
    output _ _ : message, port;
Body
  Axioms
    wire.out(d,p) With input(d,p);
    wire.inp(d,p) With output(d,p);
  Where d:message; p:port;
End DistributedWire;
```

9.5 Construction of a Distributed System

The creation of a particular system is performed by instanciating the various TNodes and by interconnecting them in the system context as shown in Specification 16. This model reflects that wires can represent physical entities with TNode independent behavior.

Specification 16. System of TNodes Expressed by Contexts

```
Context node1 as DistributedTNode();
  End node1;
Context node2 as DistributedTNode();
  End node2;
Context node3 as DistributedTNode();
  End node3;
Context node4 as DistributedTNode();
  End node4;
Context wire1 as DistributedWire();
  End wire1;
Context wire2 as DistributedTWire();
  End wire2;
Context wire3 as DistributedWire();
  End wire3;
Context System;
Interface
Body
  Use message, Port;
  Use Context node1, node2, node3, node4,
                  wire1, wire2, wire3;
  Axioms
    wire1.input(m,p) With
      node1.output(m,p);
    wire1.output(m,p) With
      node2.input(m,p);
    wire2.input(m,p) With
      node2.output(m,p);
    wire2.output(m,p) With
      node3.input(m,p);
    wire3.input(m,p) With
      node2.output(m,p);
    wire3.output(m,p) With
      node4.input(m,p);
  Where m:message; p:port;
End System
```

This simple example shows how the context modules can be used in order to take into account physical characteristics of the modeled distributed system. Mapping into a real implementation is another level of use of context diagram in which contexts are assign to physical entities. The mapping must preserve localization principles in which subcontexts are assigned to the same context as their enclosing contexts,

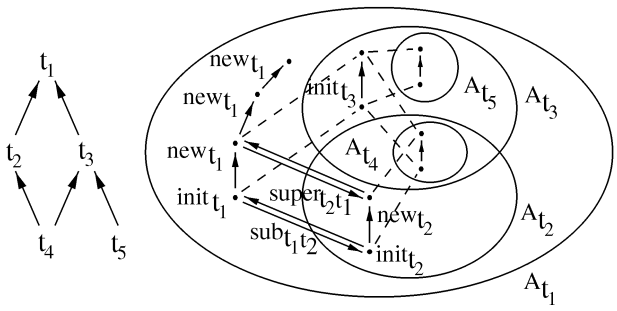


Fig. 13. Model for the management of object identitiers.

with the opportunity that some larger context could not be assigned to a specific physical entity.

10 Semantic Aspects of CO-OPN/2

This section is aimed at giving some insights about the semantic aspects of CO-OPN/2 and at answering the following questions:

- What kind of semantics is defined?
- How is the semantics constructed?
- How object identifiers are managed?

We answer these questions in an informal way, but a formal treatment of all these notions can be found in [6] and [7] .

As already mentioned, our approach is based on two underlying formalisms: order-sorted algebra and algebraic nets. From a static semantics point of view, a CO-OPN/2 specification is a set of ADT modules and class modules. A specification is said well-formed when it is complete,[5] when the syntactic conditions over subsorting and subtyping (contravariance of method arguments) are satisfied, and when there is no cycle in the dependency graphs induced by the modules of the specification.

10.1 Abstract Data Types

With regard to the dynamics semantics, both the abstract data types and algebraic nets dimensions of the formalism are obviously considered differently. The former is based on order-sorted algebra as described in [15]. For the sake of simplicity, we have adopted the initial approach. Note that, for a given CO-OPN/2 specification, the initial model of the abstract data type dimension is composed of two disjoint parts. The first one consists of all the carrier sets defined by the ADT modules of the specification, while the second one deals with the management of the object identifiers. In fact, this second part, called *object identifier algebra*, is induced by the information defined in the class modules, namely the types and the subtypes. The object identifier algebra forms an order-sorted algebra of object identifiers, organized to reflect the subtyping relationship between class types, i.e., two carrier sets of object identifiers are related by subsorting (or inclusion) if, and only if, the two corresponding types are related by subtyping. Fig. 13 illustrates an example of such an object identifier algebra based on five classes of type[6] t_i $(1 \leq i \leq 5)$ such that $t_4 < t_2$, $t_4 < t_3$, $t_5 < t_3$, $t_2 < t_1$, and $t_3 < t_1$. This subtyping relation is depicted on the left side of the figure. On the right side, we can observe the five carrier sets of object identifiers, denoted A_{t_i} $(1 \leq i \leq 5)$, which are related by subsorting (inclusion) according to the subtyping relation given on the left. Operations over these carrier sets are provided. The generators "init_{t_i}" and "new_{t_i}" constructs the values, while the operations "$\mathrm{super}_{t,t'}$" and "$\mathrm{sub}_{t,t'}$" are mainly used to determine the type of a given object identifier. These semantic operations are used to define the semantics of the keywords **`isa`** and **`isany`**.

10.2 Algebraic Nets

The semantics of the algebraic nets dimension is given in terms of transition systems, a widespread basic formalism used for expressing the evolution a system. A transition system is based on the set of all possible states of the system, one of which is the initial state, and all the events which can occur.

In the Petri nets community, the state of a system corresponds to the notion of *marking*, that is to say a mapping which returns, for each place of the net, a multiset of algebraic values. However, this current notion of marking is not suitable in the CO-OPN/2 context. It is worthwhile to recall that CO-OPN/2 is a structured formalism which allows the description of a system by means of a collection of entities. Moreover, this collection can dynamically increase or decrease in terms of number of entities. This implies that the system has to retain, throughout its evolution, two additional informations:

1. A function which returns, for each class type, the last object identifier used so as to be able to compute a new object identifier, whenever, a new object is dynamically created.
2. A function which returns, for each class, the set of all the active object identifiers, i.e., the object which have already been created.

Thus, in our approach, a state consists of both the functions as above, as well a the usual marking. An event is either a transition or a method found in the classes. Then, we define a transition system as a ternary relation over the states and the events.

On this basis, we are now able to describe how, and by means of which tools, the semantics (i.e., the transition system) of a given CO-OPN/2 specification is built.

The idea behind the construction of the semantics of a specification composed of several class modules, is to build the semantics of each individual class modules first, and compose them subsequently by means of synchronizations. The semantics of an individual class module is called a *partial semantics* in the sense that it is not yet composed with other partial semantics.

The distinction between the methods (observable events) and the internal transitions (invisible and spontaneous events) implies a *stabilization process*. This process is necessary so that the methods are performed only when all transitions have occurred. A system in which no more invisible event can occur is said to be in a *stable* state.

5. A CO-OPN/2 specification is complete when every symbol is defined either in the module, which uses it, or in another module of the specification.

6. The notation $t < t'$ means that t is a subtype of t', graphically we represent this by $t \longrightarrow t'$.

Another operation called the *closure operation* is necessary to take into account the synchronization requests, as well as the three operators (sequence, simultaneity, and alternative). Such a closure operation determines all the sequential, concurrent, and nondeterministic behaviors of a given semantics and composes the different parts of the semantics by means of synchronization requests.

The successive composition of both the stabilization process and the closure operation over all the class modules of the specification will provide a transition system in which:

- All the sequential, concurrent, and nondeterministic behavior will have been inferred;
- all the synchronization requests will have been solved; and
- all the invisible or spontaneous events will have been eliminated; in other words, every state of the transition system is stable.

Finally, we define the *step semantics* of a CO-OPN/2 specification from the above semantics in which we only retain the events that are atomic or simultaneous. Moreover, we only consider the transitions from states that are reachable from the initial state.

In order to perform the three steps mentioned above, three different sets of inference rules are provided. The first set copes with the construction of the partial semantics of an individual class. It is composed of two rules which generate the events (methods and internal transitions) that follow from the behavioral axiom of the class, the first rule for the dynamic creation of objects, and the second rule for their destruction. The second set of inference rules aims at performing the internal transitions by collapsing them with non-internal events. The third set is composed of three rules for the three operators (sequence, simultaneity, and alternative), as well as one rule which takes into account the synchronization requests, that is to say which composes the various partial semantics established previously. The subsequent application of these inference rules induces three functions, let us say $PSem$ (partial semantics), $Closure$, $Stab$. The semantics is calculated starting from the partial semantics of the least class module (according a total order over the class modules of the specification), and repeatedly adding the partial semantics of a new class module. Whenever the partial semantics of a class module is introduced, it is necessary to apply the function $Stab$, and subsequently the function $Closure$, over the semantics which includes the new class module.

Fig. 14 illustrates the step semantics of an hypothetical CO-OPN/2 specification composed of only one basic TNode (see Specification 2 and Fig. 5). For the sake of clarity, only a few events are represented, and only two messages a and b are considered. Moreover, the figure shows the transitions system for a single object o in which the states are simply represented as multisets of messages. One observes that the black arrows represent atomic events which were generated by the $PSem$ partial semantics function. The grey arrows, which denotes the simultaneous events, were generated by the $Closure$ operation. The $Closure$ operation generates not only all the simultaneous events, but also all the sequential and alternative events. However, being not essential for the step semantics, these latter events have been removed during the last process.

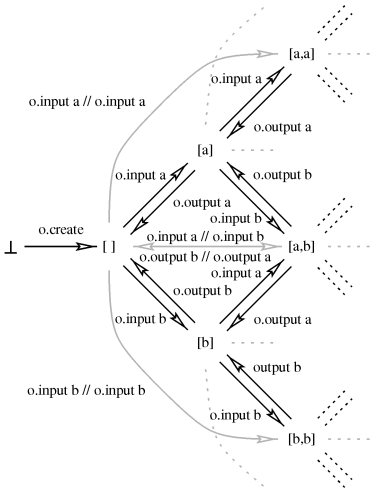


Fig. 14. Step semantics of the basic TNode.

11 State of the Art

CO-OPN/2 is considered as a formal method for system specification, since it allows to state the system properties using language with a precise syntax and a semantics given using mathematical concepts.

In the field of formal methods for computer science and more especially for software specification, we can classify the existing approaches into two classes: the property oriented one where the system properties are given explicitly and the model oriented approach where they are implicitly given.

In the property oriented class, we found approaches which are naturally based on mathematical logic. There are classical approaches like pure logic and temporal logic, or approaches which combines logic and set theory like Z and VDM, or logic and algebra like algebraic specifications. In all these approches the specifier must write logical formula which define the properties that must be verified by the future program.

The model oriented specification formalisms are always based on a metamodel which provides basic elements and composition operators. Thus, a specifier builds a model of its system by composing basic element or already composed elements. In this class, we found an approach where the metamodel is Petri nets, process algebra or state charts.

The property oriented approaches are better suited for engineering process where reasoning on the system properties is important. For example if deductive mathematical proofs are important, or if a logical implementation language is used. They are also usefull since they force the specifier to think explicitly at its system's properties, thus trying to avoid hidden properties.

The model oriented approaches are better suited for system design in the sense that they often build operational models of the software to be developed. These approaches are also interesting since they can provide a metamodel

having rich elements thus reducing the size complexity of the specification.

CO-OPN/2 is a specification formalism using a Petri net like approach for modeling concurrent behaviors, and algebraic specifications for describing the state values of the modeled system. That is why CO-OPN/2 must be seen more as a model oriented specification formalism in the spirit of Petri nets, than as a property-oriented specification formalism like algebraic specification or temporal logic.

Furthermore, CO-OPN/2 provides an object-oriented way of describing its specifications. Many of the classical specification formalisms have been extended in order to provide object orientation. A study of some of these approaches can be found in [16]. Concerning approaches close to CO-OPN/2, we have studied more precisely Cooperative Objects (CO), Object Petri Nets (OPN), and CLass Orientation With Nets (CLOWN). We provide below a comparison of these approaches which will allow the reader to understand precisely the main differences between these formalisms.

11.1 Comparison with CO-OPN/2

In this section, we describe the main specification formalisms which are based on Petri nets and which intend to achieve the same objectives than CO-OPN/2. These formalisms are: CLass Orientation With Nets (CLOWN) [4], Cooperative Objects (CO) [25], Object Petri Nets (OPN) [19], [18]. A general comparison of these approaches, with respect to CO-OPN/2, is provided at the end of this section.

We discuss in this paragraph the similarities and differences of the formalisms presented above with respect to CO-OPN/2 . We address four main categories of object oriented specification formalisms which are: object based aspects, object-oriented aspects, semantics foundations, and process model supported.

Object-Based Aspects: All the four formalisms possess a notion of object and each object has a persistent and unique identity, but in CLOWN, the behavior of each class of objects is governed by a net and the tokens (data structures) that circulate in the net represent the objects, while in the other approaches, each object corresponds to a net and the places play the role of the attributes. Only CO does not ensure encapsulation.

Concerning data structures, only CO-OPN/2 and CLOWN have the ability to describe abstract data types, as distinct from individual objects and only CLOWN cannot define data structures of objects (e.g., stack of objects).

Concerning concurrency, they all have the ability to express concurrent events inside an object (intraobject concurrency) and concurrent progress of objects (interobject concurrency) apart form CLOWN which have only interobject concurrency.

Concerning communications, they all provide synchronous service request and only CO and CO-OPN provide a direct way for asynchronous communications (nonblocking message passing). The synchronization mechanism offered by CO-OPN/2 (i.e., synchronization expressions) appears to be the most flexible with respect to concurrency.

Object-Oriented Aspects: They all consider a class as a template which describes the common aspects of objects and each class defines a type which is associated with all instances of the class (statically or dynamically created). No notion of class as a collection exists in these formalisms (i.e., extensional description of an homogeneous collection of existing objects).

Inheritance and subtyping is proposed in all these formalisms apart from OPN which does not provide subtyping. It must be noticed that, only CO-OPN/2 makes the difference between inheritance and subtyping (as explained in this paper). CLOWN provides multiple inheritance but with some limitations due to semantic aspects.

They all provide in some way some mechanisms for genericity or parameterization, but for CO and OPN it is provided in a limited way only, for data structures and not classes.

Process Model: The integration of the specification formalism in a well-defined process model is mandatory for industrial software development. Currently, only CO-OPN/2 covers this aspect. It does it by providing a development methodology based on formal refinement of CO-OPN/2 specification as presented in [21]. A process model starts with an abstract specification of the system of interest, and then several refinements steps are made according to a precise notion of refinement. The objective is to achieve a specification as concrete as possible which integrates all the design decisions and which can be directly translated into a programming language. The programming language considered in [21] is Java for which a important work has been provided in order to control the implementation phase (from the more concrete CO-OPN/2 specification to the Java program). A way of verifying properties during all the development process is provided based on a notion of contract (associated to each specification phase), which is a set of temporal logical formulae that must be satisfied at all the following refinement levels.

Furthermore, other aspects of the development process are covered like testing [3] which is integrated with our refinement methodology, and distributed system architecture modeling [13] presented in this paper.

Future work will consist in providing a way of addressing requirement specification using a semiformal method Fusion.

12 Conclusion

In this paper, we have presented a formal approach called CO-OPN/2 which can be used for the development of large distributed systems. This approach is flexible enough to be adapted to the modeling of many different kinds of applications (including the development of distributed algorithms [12], the development of parallel algorithms [8]) as well as to the semantics of concurrent language (by modeling the actor languages [10]). The transit node has been used to describe completely and informally particular aspects of the CO-OPN/2 language.

CO-OPN/2 supports modularity and the expression of various kinds of concurrencies. In this paper, we have shown how CO-OPN/2 can handle concurrency issues and classification requirements. The following major points of our approach have been presented: We express intraobject and interobject true concurrency; We differentiate

inheritance as a syntactic mechanism from subtyping which is a semantic concern; We give a way to express hierarchies of abstraction; We allow modular construction of specifications; we give a formal model of specification refinement.

Nevertheless, important research topics around CO-OPN/2 have not been presented here, such as verification issues that can be found in [3], validation issues that are presented in [11] and refinement for distributed system development and verification based on temporal logic that are presented in [14]. Complete CO-OPN/2 semantics is presented in [6]and [7].

With the progressive development of the Transit node, we have shown the possible ways of refining specifications, starting from an abstract description and progressively introducing concrete aspects. The validation of each step of the refinement process was possible through the use of prototyping or proving tools available in the SANDS environment [8].

Future studies will be conducted along three directions: Tools to support the new features of the CO-OPN/2 language for the already developed environment SANDS will be developed (you can download the tool at: http:/lglwww.epfl.ch). A distribution model of CO-OPN/2 objects will be defined for heterogeneous systems with its operational semantics; Experiment will be performed with CO-OPN/2 on different practical case studies.

ACKNOWLEDGMENTS

The authors would like to thank their collegues who have participated to the development of the CO-OPN/2 formalism. They would also like to thank Mathieu Buffo for his help on writting the COIL related part and Olivier Biberstein for his contributions to a first version of this paper.

This work has been partially sponsored by the Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) with the financial support of the Office Fédéral de l'Education et de la Science, and by the Swiss National Science Foundation Projects 20.47030.96 and 20.53554.98.

REFERENCES

[1] P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," *Proc. European Conf. Object-Oriented Programming,* vol. 276, pp. 234–242, June 1987.

[2] P. America, "A Behavioral Approach to Subtyping in Object-Oriented Programming Languages," Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B.V., Apr. 1989, revised Jan. 1989.

[3] S. Barbey, D. Buchs, and C. Péraire, "A Theory of Specification-Based Testing for Object-Oriented Software," *Proc. European Dependable Computing Conf.,* Technical Report (EPFL-DI-LGL no. 96/163), Oct. 1996.

[4] E. Battiston, A. Chizzoni, and F. De Cindio, "Modeling a Cooperative Environment with CLOWN," *Proc. Second Int'l Workshop Object-Oriented Programming and Models of Concurrency within the 16th Int'l Conf. Application and Theory of Petri Nets,* pp. 12–24, June 1996.

[5] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," technical report, INRIA, 1988.

[6] O. Biberstein, "CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems," PhD thesis, University of Geneva, July 1997.

[7] O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," *Advances in Petri Nets on Object-Orientation,* G. Agha and F. De Cindio, eds., Springer-Verlag, 2000.

[8] D. Buchs, P. Racloz, M. Buffo, J. Flumet, and E. Urland, "Deriving Parallel Programs Using SANDS Tools," *Transputer Comm.,* vol. 3, no. 1, pp. 23–32, Jan. 1996.

[9] D. Buchs and N. Guelfi, "CO-OPN: A Concurrent Object-Oriented Petri Nets Approach for System Specification," *Proc. 12th Int'l Conf. Application and Theory of Petri Nets,* pp. 432–454, June 1991.

[10] D. Buchs and N. Guelfi, "Formal Development of Actor Programs Using Structured Algebraic Petri Nets," *Proc. Int'l Conf. Parallel Architectures and Languages Europe (PARLE),* vol. 694, pp 353–366, June 1993.

[11] D. Buchs and J. Hulaas, "Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets," *Proc. Int'l Conf. Systems, Man and Cybernetics,* Oct. 1996.

[12] D. Buchs, F. Mourlin, and L. Safavi, "CO-OPN for Specifying a Distributed Termination Detection Algorithm," *World Transputer Congress, Workshop on Software Engineering for Parallel Systems,* vol. 3, pp. 25–29, Sept. 1993.

[13] M. Buffo and D. Buchs, "A Coordination Model for Distributed Object Systems," *Proc. Second Int'l Conf. Coordination Models and Languages (COORDINATION),* vol. 1,282, pp. 410–413, 1997.

[14] G. Di Marzo Serugendo and N. Guelfi, "Using Object-Oriented Algebraic Nets for the Reverse Engineering of Java Programs: A Case Study," *Proc. Int'l Conf. Application of Concurrency to System Design (CSD),* Technical Report (EPFL-DI no. 98/267), IEEE CS Press, 1998.

[15] J.A. Goguen and J. Meseguer, "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions, and Partial Operations," *Theoretical Computer Science,* Technical Report SRI-CSL-89-10 (1989), Computer Science Lab., SRI International, vol. 105, no. 2, pp. 217–273, 1992.

[16] N. Guelfi, O. Biberstein, D. Buchs, E. Canver, M-C. Gaudel, F. von Henke, and D. Schwier, *Comparison of Object-Oriented Formal Methods,* technical report of the Esprit Long Term Research Project 20072, "Design For Validation," University of Newcastle Upon Tyne, Department of Computing Science, 1997.

[17] D. Harel, "Statecharts: A Visual Formalism for Complex System," *Science of Computer Programming,* vol. 8, no. 3, pp. 231–274, 1987.

[18] C.A. Lakos, "The Object Orientation of Object Petri Nets," *Proc. First Int'l Workshop Object-Oriented Programming and Models of Concurrency within the 16th Int'l Conf. Application and Theory of Petri Nets,* pp. 1–14, June 1995.

[19] C.A. Lakos, "The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets," *Proc. Application and Theory of Petri Nets,* vol. 1,091, pp. 380–399, June. 1996

[20] B. Liskov and J.M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 6, pp. 1,811–1,841, Nov. 1994.

[21] G. Di Marzo, "Stepwise Refinement of Formal Specifications Based on Logical Formulae: From CO-OPN/2 Specifications to Java Programs," PhD thesis, no. 1931, Ecole Polytechnique Fdrale de Lausanne, Dpartement d'Informatique, 1015 Lausanne, Switzerland, Jan. 1999.

[22] S. Mauw and F. Wiedijk, "Specification of the Transit Node in PSF_d," *Algebraic Methods II: Theory, Tools and Applications,* J.A. Bergstra and L.M.G. Feijs, eds., vol. 490, Vrije Univ., Springer Verlag, pp. 340–361, 1991.

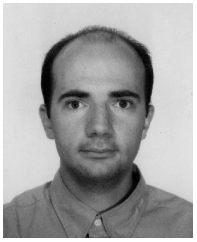
[23] M. Bidoit, M.-C. Gaudel, J. Hagelstein, A. Mauboussin, and H. Perdrix, "From An ERAE Requirements Specification to a PLUSS Algebraic Specification: A Case Study," *Algebraic Methods II: Theory, Tools and Applications,* J.A. Bergstra and L.M.G. Feijs, eds., Vrije Univ., Springer Verlag, vol. 490, pp. 395–397, 1991.

[24] W. Reisig, "Petri Nets and Algebraic Specifications," *Theoretical Computer Science,* vol. 80, pp. 1–34, 1991.

[25] C. Sibertin-Blanc, "Cooperative Nets," *Proc. 15th Int'l Conf., Application and Theory of Petri Nets,* R. Valette, ed., vol. 815, pp. 471–490, June 1994.

[26] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. Object-Oriented Programming Languages and Applications, ACM SIGPLAN Notices,* vol. 21, no 11, pp. 38–45, Nov. 1986.

Didier Buchs received the PhD degree in Computer Science from the University of Geneva in 1989. He was a researcher at Laboratoire de Recherche en Informatique of the University of Paris Sud (Orsay) from 1989 to 1991. Also, he was the leader of the research team working on formal methods and concurrency at the University of Geneva from 1991 to 1993. He was a research director ("Adjoint scientifique") at the Software Engineering Laboratory of Swiss Federal Institute of Technology (EPFL). Since 1993, he has been the leader of the Concurrency and Formal Methods (ConForm) group. He supervised eight PhD candidates at the University of Geneva and Swiss Federal Institute of Technology, respectively. He participated at several national and international conferences. He directed the project Formal Methods for Concurrent Systems and the Swiss participation to the DeVa (Design for Validation) Esprit project. His current research interests include formal specification methods and testing techniques for real size distributed systems using object-orientation.



Nicolas Guelfi received the PhD degree in 1994 from the University of Paris XI-Orsay in France in the field of formal specification of concurrent systems. Since 1999, he has been a full professor at the Applied Computer Science Department at the Luxembourg University of Applied Sciences. In 1994 and 1995, he worked as a research and teaching assistant at the University of Paris XII-Creteil and then joined, for three years, the Software Engineering Laboratory at the Lausanne Swiss Federal Institute of Technology in Switzerland where he has participated to teaching, research projects, and PhD thesis supervision. Since 1990, he has been involved in three European ESPRIT BRA projects (DEMON, CALIBAN, and DEVA), two national research projects in Switzerland and one technology transfer project. All these research activities have been done in the field of formal software engineering methods and tools for distributed systems and have contributed to the CO-OPN framework. Currently, he is the head of the Software Engineering Competence Center and director of the Computer Sciences Studies where he is in charge of managing national and international research projects on scientific methodologies applied to the engineering of distributed systems and databases.