CO-OPN/2
A Specification Language for Distributed Systems Engineering *

Olivier Biberstein
CUI, University of Geneva
CH-1211 Genève 4, Switzerland
tel: (41)(22) 705-7619 fax:(41)(22) 705-7780
biberste@cui.unige.ch

Didier Buchs
LGL-DI, Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
tel: (41)(21) 693-2737 fax:(41)(21) 693-5079
buchs@di.epfl.ch

Nicolas Guelfi
GRIF, University of Paris Val de Marne
F-94000 Créteil, France
tel: (33)(1) 46 45 77 46 fax:(33)(1) 47 36 34 19
guelfi@univ-paris12.fr

Abstract

In this article we present the concurrent object-oriented specification language CO-OPN/2 which extends the CO-OPN (Concurrent Object Oriented Petri Nets) formalism, destined to support the development of large distributed systems. The development of distributed applications has recently been the subject of intensive research, due to potential benefits that can be expected to offer in increasing reliability and overall performance of computer processing. One way of improving the effectiveness of the distributed system construction is to adopt a rigorous approach for system development, based on structuring principles. The CO-OPN/2 approach proposes a specification language, based on the object-oriented paradigm, which includes a fine description of true concurrent behavior.

This hybrid approach (model and property-oriented) allows for a description of the concurrent aspects through the use of a high-level Petri nets formalism which includes data structures expressed with the algebraic abstract data types and a synchronization mechanism for building hierarchies of abstraction. This latter notion is the concept which is used in structuring applications.

Admirable properties of CO-OPN, such as the progressive refinement of specifications, allow for the building of systems in an incremental manner. In this article, we introduce CO-OPN/2 informally, by means of a typical example of distributed systems, the transit node, in order to introduce each useful and innovative mechanism of the language.

Keywords: software engineering, formal specification language, refinement, prototyping, distributed systems, object orientation, concurrency, Petri nets, algebraic specification.

*This work is partially supported by the FNRS (Fonds National de la Recherche Scientifique). Project 21.32286.91 and by the ESPRIT WG CALIBAN, no. 6007

1 Introduction

1.1 The Development of Distributed Systems

For important applications, distributed processing provides the most general, flexible and evolutionary approach in the provision of computer processing. Thus distributed computing offers advantages because of its potential for improvement in the following domains: availability and reliability through replication; performance through parallelism; sharing and interoperability through interconnection; and flexibility and incremental expansion through modularity.

However, distributed systems introduce several new considerations that must necessarily be taken into account. Interactions between concurrent components give rise to such issues as non-determinism, contention, and synchronization. This implies that with distributed systems particular care must be taken with the language used for expressing the requirements or the specifications. In the case of a rigorous process, including formal specification, it is necessary to have a sound mathematically-based formalism allowing for the expression of all the particularities of the distributed systems. Moreover, when we are faced with large problems it is also necessary to have complete structuring facilities. The objective of the specification phase is to clearly state the set of functions expected from the software. The ever-increasing complexity of software systems thus imposes a progressive procedure of adaptation based on abstraction, refinement and enrichment. It is preferable to have access to structuring primitives so as to effectively control this procedure. The possibility of organizing a formalism or a language in a modular way has become a necessary attribute in the specification of systems which are substantial in size. A method which is rigorous and which allows for the proof of certain characteristics of the specified system must be formal, i.e. entirely based on mathematical concepts. By formalizing the specification clearly, an operating environment is provided which supports for writing, refinement, enrichment, abstraction and analysis of structured specifications.

It is admitted that the only realistic way to construct a formal specification is by proceeding incrementally. Without being overly categorical, we demonstrate in the following sections that this is a very efficient way of working. Indeed, the initial perception of the system to be built may be very vague. As analysis and simultaneous validation progress, the visualization of the architecture, algorithms and associated data structures is improved right until final implementation take form.

In our proposition, we have chosen the object orientation as the structuring paradigm. We have given a general language which express both abstract and concrete aspects of a system, with emphasis on the structuring, concurrency and data description. This approach, called CO-OPN (Concurrent Object Oriented Petri Nets), has been presented in this paper in an intuitive way and the description is oriented around expressive aspects and methodological considerations rather than its theoretical dimensions that can be find in [BB94c, BB94b].

It must be noted that CO-OPN specification language focuses on the use of the Petri nets for the formal development, design and validation (through prototyping) of distributed systems rather than the qualitative or quantitative analysis of systems as it is generally made using Petri nets.

1.2 Object-Orientation

Object-Orientation is a principle of engineering used to create a model of the domain of real-world problem and to map it into a software solution domain [Eck94]. The object-orientation does not abandon all the sound principles of other methods but it incorporates new elements into these methods.

The object notion is the only concept common to all the object-oriented approaches, in which any perceived entity may be considered as an object. Object-orientation is traditionally viewed as being the result of three conceptual approaches:

- The object-based approach which supports the object notion and is the encapsulation of a set of operations and of a state altered by the operations,
- The class-based approach in which every object is the instance of a class. A class is a template from which objects with identical behavioral properties may be created,
- The object-oriented approach in which class hierarchies may be defined incrementally with an inheritance mechanism in which subclasses inherit their basic template from their superclass.

It seems, as Blair argues [BGHS91], that the view in which object-orientation corresponds to having objects, classes and inheritance is overly restrictive. The mechanisms are closely related to implementation problem and are not related to the problems solved by the object-orientation, which are more at encapsulation and classification levels. Encapsulation is the fundamental notion of object-orientation because it conveys the essential notion of objects, but nevertheless, large variations are possible according to potential properties. The information-hiding principle implies that only the operational interface is visible, while data representation and operation implementation remain concealed. This principle is usually broken with inheritance mechanisms, in which a sub-class is dependent on the internal representations and implementations of its super-class.

Classification is a principle in which various grouping of objects may be defined. All objects within a grouping share common properties. Both the intensional (the predicate) and extensional (the population) aspects of a classification must be considered. A characteristic of the classifications is that they must have potential point of intersection which allow them to share objects. Sub-typing is a mechanism used for sharing objects between sub-classes.

1.3 Concurrency and Object Orientation in Specification Languages

The introduction of object orientation into specification language is a important subject of research within the software engineering community. Programming languages have widely introduced such concepts in their definitions, in particular for the data aspect of the language Smalltalk, Ada 95, C++, etc. and, more recently, in concurrent languages. It seems that object orientation is not easily combined with the notion of concurrent systems [Ame87]. In our opinion, this is because concurrent aspects are orthogonally introduced (see [Pap89] for a deep analysis into most of these approaches) and interfere consequently with the inheritance mechanism, which is interpreted in an operational manner. In specification language, it is possible to abstract these mechanisms, and then to conceal a portion of the operational nature of the inheritance mechanism. In CO-OPN, concurrent aspects are not orthogonal to the behavioral description of the methods. Instead their behavior avoids any strange behavior or inconsistencies with respect to the inheritance mechanism.

1.4 The CO-OPN Approach

CO-OPN is a specification language destined for the specification and design of large concurrent systems. The last version of CO-OPN/2 extends the CO-OPN language by means of some object-oriented features such as class, inheritance and subtyping.

The object-orientation paradigm is introduced into CO-OPN/2 in order to solve the problem of large specification management. *Encapsulation* and *abstraction* are key notions included in the model. Each class or template can be used to instanciate the objects which compose the modelled system. A CO-OPN object can be viewed as a set of *encapsulated* state variables or attributes which can be used for implementing accessible methods. Moreover, object reaction is modelled by means of the definition of internal transitions. According to the common intuition, each object has its own behavior and evolves in concurrence with one another. As the purpose of CO-OPN is to capture the abstract concurrent behavior of each modelling entity, the concurrency granularity is not found in the objects but rather in the methods. Intra object concurrency can be obtained by expressing the causality relation between the behavior of the different methods. The relations between objects, such as object synchronization, must also be captured using method synchronization schemes allowing for the building of hierarchies of *abstraction*.

Our opinion is that the essence of the concurrent behavior of the objects must be represented operationally. We propose using Petri nets, which are an elegant model of true concurrency (true in opposition to interleaving semantics). Unfortunately, Petri nets have very weak modelling capabilities in term of their functional aspects and structuring capabilities. We add to Petri nets a synchronization scheme for each methods of the objects, modelling the possible concurrent processes needed with a method or an internal transition which might occur. This approach generalizes other approaches such as, the transition fusion [DHP91] and the hierarchical nets [Kie89].

Real systems must include other dimensions such as data structures with the associated operations acting upon these data. This is the reason why data structures are incorporated into Petri nets, forming a new class of Petri nets called high-level nets i.e. Petri nets with tokens modelled by data structures instead of anonymous tokens. Abstract modelling is imposed for considering the logical formalism for the description of data structures.

In our case we chose algebraic abstract data types which combine abstraction with a simple system of notation which is derived from the first order logic.

1.5 CO-OPN and other approaches

Our reflections and experiments correspond with the reflections of other research teams on formal methods and show that it is necessary to include the modelling of states into the formal languages under development which aim at modelling real (distributed) systems. Thus we propose using a combination of algebraic specifications and Petri nets [Vau87], called Algebraic Nets [BG92b].

Thus, the idea to extend algebraic specification leads to a large number of approaches that can be classified into two main groups. A pure algebraic approach, which includes the works of the Astesiano [AR93, Reg93] and Bernot [AB94] teams. A mixed approach (property and model oriented) which includes the High Level Petri Net approach (algebraic nets [Vau87, Rei91], LOOPN [Lak94], Cooperative Nets (CN) [SB94], the POT and POP's approaches [ELR90], OBJ-SA [BCM88], CLOWN [BC93], O-O-AHL [Rib94]), the process logic approach (with LOTOS) and the temporal logic approach (FOO in the table) [FM90]. We may mention also that the Z based approaches [LH93] can be included with into the model-oriented approach, but these methods do not encompass concurrency dimensions.

In the table 1, we have tried to express some particularities of the formalisms based on the object-orientation paradigm. We have taken into account their modelling power and the existence of tools, as well as how the object-orientation paradigm is interpreted.

A further problem is that large system development cannot be taken into account with respect to algebraic nets because, in such a model, the system is considered to be a flat collection of states and events. In the formalism CO-OPN [BG91, BG92a] we introduce objects as separate encapsulated set of states which are accessible by means of methods as well as a mechanism for synchronizing methods. This mechanism allows for expressing hierarchy of abstraction through synchronizations which use simultaneous, sequential and alternative operators. This earlier version of CO-OPN was object based and several concrete examples have shown that two kinds of modelling can be expressed by means of CO-OPN: object modelling [BBMM94] and class modelling in which CO-OPN objects are in fact a collection of object instances inside places [BMS93, BG93b].

These case studies were developed using the CASE tool SANDS [BFR93] (Structured Algebraic Net Development System), which can graphically edit CO-OPN specifications, simulate the behavior of such specifications, or prove some basic temporal properties.

More recently, we have improved the semantics of CO-OPN by giving a generalized model called Hierarchical Algebraic Nets [Gue94]. This semantics serves to describe the semantics of the new version of CO-OPN , called CO-OPN/2 which enriches CO-OPN by adding class orientation, inheritance and subtyping dimensions [BB94c]. In particular, self-reference and distributed implementation are allowed.

Classes in CO-OPN/2 are templates that can be instanciated as often as necessary and, the subtyping notion is related to the ability to substitute certain objects with one which is more specialized. Furthermore, we consider inheritance to be a general syntactic model of property reusability.

1.6 Plan of the Paper

In the following sections the article describes the context for the development of distributed specifications, using refinement and the CO-OPN language, through the development of a comprehensive example which is the transit node. Beginning with a very abstract view of the system we introduce progressively several of its concrete dimensions in order to show how to use the CO-OPN/2 specification language in the development of real distributed systems. Several additional dimensions, concerning principally the semantics of CO-OPN/2 in their translation into hierarchical nets, can be found in [BB94b].

Formalism	Modelling				Object-orientation				Tools		
	Data	Abst.	State	Concur.	Obj.	Class	Inheri.	Encap.	Dev.	Proto.	Proof
Object-Z	$\star$		$\star$		$\star$	$\star$	$\star$	$\star$	$\star$		
OBJ-SA	$\star$		$\star$	$\star$	$\star$	$\star$		$\star$	$\star$	$\star$	$\star$
CLOWN	$\star$		$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$		
CO-OPN	$\star$	$\star$	$\star$	$\star$	$\star$			$\star$	$\star$	$\star$	$\star$
CO-OPN/2	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$		$\star$
FOO	$\star$		$\star$	$\star$	$\star$			$\star$			$\star$
LOOPN	$\star$		$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$	$\star$
OO-HAL	$\star$		$\star$	$\star$	$\star$	$\star$	$\star$				$\star$
CN	$\star$		$\star$	$\star$	$\star$	$\star$	$\star$				$\star$

Table 1: Comparison between different formalisms

2 The Development Process and the Refinement

The development process supported by the CO-OPN approach explains how we can progressively introduce new aspects in the specifications of the system in order to *derive* more concrete specifications. Two aspects can motivate this derivation: we can consider the introduction of new features in the modelled system or we can consider its specialization or *refinement*. These kinds of derivations are respectively motivated by horizontal or vertical derivations. The horizontal derivation focuses on the possibility for the developer to modify the system specification and this is taken into account with the loops of the software life cycle, with no regard for property preservation. The vertical derivation requires more extensive technical machinery in order to guarantee that the derivation will preserve the intended behavior of the specification. Within our theoretical framework [BG91, Gue94, BB94c], we deal with property preservation by means of a the definition of a preorder $\rightsquigarrow$ between the specifications based on bi-simulation principles. $Spec_1 \rightsquigarrow Spec_2$ is equivalent to $Spec_2|_{Spec_1}$ bi-similar to $Spec_1$ where $Spec_2|_{Spec_1}$ means that the observation of $Spec_2$ is limited to the element of $Spec_1$. As we are interested in separately considering the modular unit of the formalism, we examine only the separated refinement of a sub-part of the specifications, for instance the classes or the algebraic modules.

In CO-OPN we have the compositional property [BG91, Buc92] in which $Spec_1 \rightsquigarrow Spec'_1$ implies that $(Spec_1 + Spec_2) \rightsquigarrow (Spec'_1 + Spec_2)$ where $+$ is the union of specifications. This means that we are able to separately refine sub-part of the specification.

It must be noted that sub-typing represents also a preorder relationship between types which has a similar definition, with the exception that subtyping is used in a different direction of the development process. Due to the needed incrementality of the refinement process, the problem of proving the sub-type relationship remains intact even after expressing a refinement. In the case of a subtype relation $Spec_1 \leq Spec_2$, when $Spec_1$ and $Spec_2$ have been refined into, respectively, $Spec'_1$ and $Spec'_2$, the relation $Spec'_1 \leq Spec'_2$ cannot be guaranteed.

The previously introduced concepts shows that the proof of correct refinement is a important subject that must be taken into account. At the end of the paper we introduce the testing techniques that can be used in the proof of the validity of the refinement process.

3 Principles of CO-OPN/2

The purpose of this section is to introduce briefly and explain the CO-OPN/2 basis and to present some important aspects of the formalism. Among these aspects we may mention structuring capabilities, synchronizations of objets, inheritance, subtyping and object identity. Deeper explanations of the formalism and the associated methodology are set forth in the following section.

3.1 Abstract Data Types

An abstract data type is a representation of a data structure coupled with a set of operations which manipulate such a data structure (data type) independently of any particular implementation (abstract).

The algebraic approach [EM85] to specification considers a system in terms of abstract data types represented, in a mathematical discourse, by many-sorted algebras in combination with axioms or equations. An algebra is simply a domain provided with a set of operations. An algebraic specification $Spec = \langle S, F, X, Ax \rangle$ consists of a set S of sort names, a set F of operation names including their profile, a set X of variables and a set Ax of positive conditional axioms. With an algebraic approach the terms "sort" and "type" are distinguished. A type represents a domain and a set of operations whereas a sort is simply the name of a type. The axioms embody the operations properties and they are described as conditional equalities, as follows:

$$Cond \Rightarrow term = term'$$

in which the condition $Cond$ is the conjunction of equalities $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$.

Many-sorted algebras have been generalized to order-sorted algebras [GM89] which allow for a partial ordering relation on the set of sorts. On a semantic level, this ordering relationship induces the inclusion domains of ordered sorts. Order-sorted algebras support overloading, several forms of polymorphism and partial operations.

3.2 Petri Nets and Algebraic Nets

Petri nets [Rei85] are used for modelling and analyzing discrete systems in which notions such as interacting components, concurrency and synchronization all play a key role. Petri nets modelling embeds both the structural aspect of the system and its evolution. Systems are modelled by means the use of places and transitions.

The use of tokens along with a internal structures greatly increases the modelling power of the Petri nets. Structured tokens offer an insight into the way data structures evolve during the operation of the system and many aspects of the modelling also can be described in a more accurate and concise manner. The evolution of structured tokens are specified with rules of the underlying formalism used for representing these tokens. By means of their abstract and sound qualities, algebraic specifications reveal themselves to be well-suited for this purpose. The high-level nets obtained by describing structured tokens in the algebraic framework are called algebraic nets [BG93a].

The behavior of such algebraic nets is described by means of behavioral axioms. These axioms have the following textual form:

$$Cond \Rightarrow Transition :: Pre \rightarrow Post$$

in which $Cond$ is a condition of the algebraic values involved in the axiom, $Transition$ is the name of a net transition and Pre and $Post$ correspond, respectively, to what is removed and produced in the different places within the net.

3.3 The CO-OPN/2 Formalism

CO-OPN/2 is based on the algebraic specifications for the data structures aspects, on one hand, and on the algebraic net for behavioral and concurrent aspects of the systems, on the other hand. These two formalisms are quite useless when large problems are confronted and some structuring capabilities are then needed to overcome their limitations.

The CO-OPN/2 formalism has adopted the object orientation paradigm and indicates that a CO-OPN/2 specification is a collection of objects which interact concurrently. The notion of class allows one to specify a collection of objects which have the same properties. A class is viewed as an object template. Inheritance is a syntactic mechanism which avoid to develop classes from scratch and is mainly used for reusing specifications. A class may inherit all the properties of another one and may add some services or change the behaviors of some already defined services. However, a sub-typing relationship between objects implies a semantic relationship over class models. In other words, everywhere a super-type object is used, a sub-type object may be substituted. Thus, all the behaviors of a super-type have to be preserved in the sub-type.

A CO-OPN/2 specification consists of three different kinds of modules: the abstract data type modules, the class modules and the object modules. These three kinds of modules are composed of an interface, termed the module signature, and a body which primarily encapsulates the operations properties or the behavior of the objects. The abstract data type modules concern the data structure part of the specifications, and order-sorted algebraic specifications are used when describing these modules. Furthermore, the class modules correspond to a notion of encapsulated entities that possess an internal state and provide the outside with various services. For this second sort of module, algebraic net formalism has been adopted. The third kind of module, the object modules, corresponds to the static objects which are particularly important with respect to the specification simulation.

For the class or object modules, two types of transitions are provided. On one hand, there are the internal transitions which correspond to the object reaction and, on the other hand, are the parametrized external transitions, or methods, which are the events visible from the outside. Co-operation between the objects is realized by means of a synchronization mechanism, i.e. each object event may ask to be synchronized with the methods of one or a group of partners by means of a synchronization expression. Four synchronizations operators have been provided. '**with**' for abstraction, '//' for simultaneity, '..' for sequence and '$\oplus$' for alternative or non-determinism. Thus, a behavioral axiom is established in the form

$$Cond \Rightarrow Event\ Sync :: Pre \rightarrow Post$$

in which *Event* is an internal transition name or a method with parameters, and *Sync* is an optional synchronization expression.

Within the CO-OPN/2 framework, each class instance has an identity and it is used as the reference of an instance. Moreover, a type is explicitly associated with each class. Thus, each object identifier belongs to one type. In fact, the object identifiers are order-sorted algebraic values of a special order-sorted algebra which is provided in order to capture the notion of sub-typing . This object identifier algebra is built so as to reflect the subtype relation between the classes, i.e. two carrier sets are related by inclusion if and only if the two corresponding type are related by sub-typing.

Some properties are formalized within the CO-OPN framework [BG91, BG92a] which may be transposed to CO-OPN/2

- Object equivalence for a given algebra by means of the notion of concurrent bi-simulation.
- One object can be replaced in an object system with another equivalent object, producing a new object system bi-similar to the initial system.
- If two different algebras, observationally equivalent, are used in the same object system, then the two different object systems are equivalent due to bi-simulation.

These properties allow us to refine an abstract CO-OPN/2 specification by progressively replacing abstract objects or classes with those which are more concrete.

4 Progressive Presentation of CO-OPN/2

This section is devoted to the introduction of the CO-OPN/2 language by means of a well-known case study, the Transit Node (TNode for short). A transit node is a node in a communication system in which messages arrive at various input ports and are routed onwards through various output ports, according a designated route. This case study was defined in the RACE project 2039, and one may find assorted specifications of the transit node in [MW91, MPM+91]. Slight changes have been made to the TNodes definition in the RACE project and a informal description of the transit node is given in section 4.1.

Our aim is to build an heterogeneous distributed communication system (i.e. which presents some interconnected TNodes with wires including different kinds of TNodes and wires) and to introduce progressively all the syntactic and semantics aspects of the CO-OPN/2 language. Among the concepts we are going to discuss, the concepts of data type and algebraic nets, as well as the object-oriented notions including those of class, of the creation of a dynamic object, of the inheritance mechanism as well as the the sub-typing notion, are cited as being particularly important.

Our approach starts with a basic and abstract version of the TNodes and the wires, which allows us to present the basic ideas of the language and to progressively enrich and refine the version in order to introduce all the concepts mentioned above as the presentation proceeds.

4.1 Informal Description of the Transit Node

The RACE project has defined a Transit Node as being a node in a communication system in which messages arrive at the input ports and are routed onwards from the ouput ports according a designated route. A TNode consists of N data input ports, M data output ports, one control input port and one control output port. Each port is serialized, concurrent all others and specified as separated concurrent entities. The node is "fair", i.e. all messages are likely to be treated equally when a selection must be made. Furthermore, all messages will eventually leave the node, or be placed within the collection of faulty messages.

The control ports can be used to configure the transit node or to get statistical informations such as faulty messages, average transit time ....

Such a transit node is illustrated in figure 1.

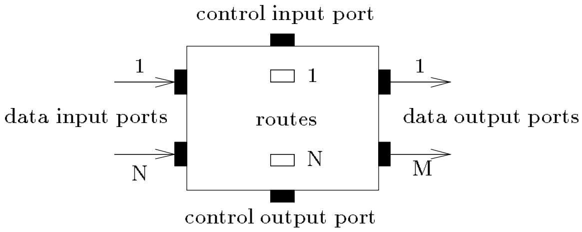

Fig. 1: The Transit Node

4.2 TNode Based Distributed System

A system of many interconnected TNodes is called a distributed communication system. The interconnection is realized by means of wires which link an input port and an output port of two TNodes. Figure 2 shows a distributed system composed of three interconnected TNodes.

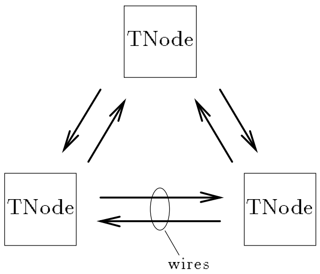

Fig. 2: A system of three transit nodes

All the TNodes involved in the above figure are identical. But, a distributed communication system may involve different kind of TNode and build what we call a heterogeneous distributed communication system. In figure 3 we have a distributed communication system based on four different type of TNodes (i.e. TNa, TNa', TNb, TNb'). Four kind of links are used to ensure communications between them. In these kind of complex distributed systems, TNodes, wires and messages can be of different nature and complex organizing notions have to be defined (such as communication protocol, gateways ...).

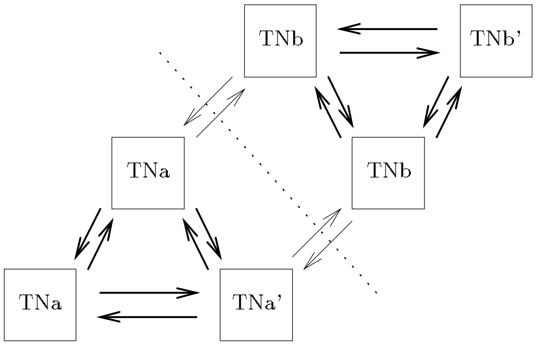

Fig. 3: A heterogeneous distributed system

4.3 Presentation Overview

As has already been mentioned, the CO-OPN/2 language will be introduced informally by means of the transit node case study. This informal introduction is divided into two stages, which are then split into different steps :

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

We describe in figure 4 the organization scheme of all these steps using a graph representation using the following convention :

- the dash arrows indicate the clientship (not the data flow)
- the curly arrows indicate inheritance, and
- the parallel arrows indicate sub-typing.

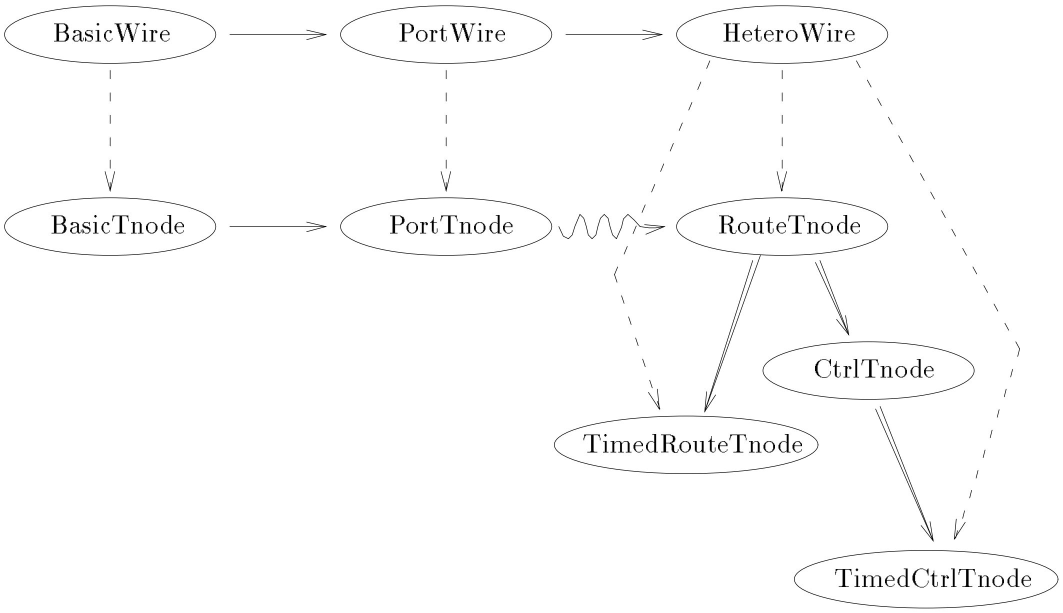

Fig. 4: Relation between the different kinds of TNodes and wires

The first steps of the running example use a simple development methodology composed of the following phases :

- Requirements : gives the principal properties of the system to be specified,
- Informal Design : indicates the CO-OPN/2 concepts which may be used to satisfy each requirement,
- CO-OPN/2 specification : gives the CO-OPN/2 description following the informations given in the informal design.

The requirements and the informal design stages are regrouped in a tabular diagram. As CO-OPN/2 is based on algebraic nets, the system features are presented in the tabular diagram using two concepts : the states and the events. Concerning the informal design part, we use the classical notions of pre and post conditions for the events design which is well suited for implementation in a Petri net frameworks thanks to their analogy with the flow relation. In addition to the pre and post conditions, we describe a synchronization constraint which gives informations on the way an event is synchronized with other events of the system. Data structures design is considered both in states and events informal design due to their use in these two parts.

As 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 corresponding to these two layers.

4.4 ADT, Class and Objects

Here we present the first step of the progressive presentation of the language. This first step which is composed of some basic TNodes and a basic communication layer, requires many basic concepts such as the concept of abstract data type, the concept of class and the notion of cooperation between class instances by means of synchronization expressions. All theses concepts as well as a graphic representation of the classes are introduced in the following.

4.4.1 Basic Transit Node

A basic TNode consists of a simple, unstructured buffer in which messages arrive at the input port and remain until they are routed onwards from the output port. At this step, communication is viewed as the passing of a message from one TNode to another by synchronizing the events “message output” and “message input”. Further information about the requirements and the informal design of the basic TNodes and communication is given in table 2.

Basic TNode		
	Requirements	Informal design
Notes	All TNodes share the same form.	• a class `BasicTnode`.
States	A bag which contains the messages in transit through the node. The initial state is an empty bag.	• a sort `message` for the data. • a place `msg` for values of sort `message`. • the initial marking is the empty multi-set.
Events	• "message input", the event can occur at any time and a message is stored in the buffer. • "message output", a message must be present in the buffer and it is then removed.	• a method `input` with a parameter `d` of sort `message`. *post:* add the message `d` to the place `msg`. • a method `output` with a parameter `d` of sort `message`. *pre:* check if a message `d` is in place `msg`. *post:* consume the message `d` in place `msg`.

Basic Communication		
	Requirements	Informal design
Notes	None.	• a class `BasicWire`.
States	Does not require a state.	None.
Events	• "message passing" which asks for the message transmission between to TNodes Remark : Both the TNodes can be the same and there is no constraint on the message passing order.	• a internal transition `msg-passing`. *sync:* ask for the simultaneous synchronization of methods `output` and `input` with the same parameter `d` of two objects identified by two free variables `tn1` and `tn2`.

Table 2: Requirements and informal design of basic TNodes and communications

4.4.2 CO-OPN/2 specifications

Having reached this point we may now develop the formal specifications of our basic transit node system. A CO-OPN/2 specification consists of a set of three different kinds of modules: the **ADT** modules (Abstract Data Type), the **Class** modules, and the **Object** module. All these three kinds of modules are composed of an interface, termed the module **Signature**, as well as a **Body** which primarily encapsulates the operations properties or the objects behavior.

ADT modules are devoted to the data structure of the specification. Accepting the premise that all things defined in the module signature may be "used" from the outside, we usually give the sort and the operation definitions in the signature, whereas the axioms which describe the operations properties remain in the body part. In figure 5, one can see the well-known booleans' specification with its operations and their arity under the field **Operations**in which the underscore character gives the position of respective arguments position.

```
ADT Bool;
Signature
 Sorts bool;
 Generators
 true, false : bool;
 Operations
 not _    : bool → bool;
 _ or _   : bool bool → bool;
 _ and _  : bool bool → bool;
Body
 Axioms
 not(true)    = false;
 not(false)   = true;
 true  or x   = true;
 false or x   = x;
 true  and x  = x;
 false and x  = false;
 Where
 x : bool;
End Bool;
```

Fig. 5: Abstract data type of the booleans

The abstract data type of the natural numbers is given in figure 6. Note that the **Use** field is followed by the list of all the modules used in the module itself, which is here only the module `Bool`.

```
ADT Nat;
Signature
 Use Bool;
 Sorts nat;
 Generators
 0 : nat;
 succ _ : nat → nat;
 Operations
 _+_  : nat nat → nat;
  ... other operations ...
Body
 Axioms
 0+n = n;
 succ(n)+m = succ(n+m);
  ... and their associated axioms ...
 Where
 n, m : nat;
End Nat;
```

Fig. 6: Abstr. data type of the natural numbers

Throughout this paper our specifications will be as simple as possible so as to not hide the main ideas which we wish to express. In this way, the messages of the basic version have no data and only the sort `message` composes the messages as is shown in figure 7.

```
ADT Message;
Signature
 Sorts message;
End Message;
```

Fig. 7: Abstract data type of the messages

The first entities of our example which have an internal state are the basic TNodes. Figure 8 shows the specification of this kind of object. One notices the **Type** field followed by the type of the class. This explicit information avoids creating confusion between the name of the module and the type associated with the class, although these two identifiers are quite similar. Such type information will be used to define references variables with respect to the objects as well as for the subtyping hierarchy definition. The **Methods** field lists all the parametrized events which are visible from the outside. The underscore plays the same role as it did with the operations, but a method does not return any value seeing as it has participated in part in a rendezvous. The last three fields remain in the **Body** section and this ensures the principle of encapsulation. In the field **Places**, the surrounding curly brackets mean that the attribute `msg` is a multiset of values of sort `message`. The preceding remark concerns both of the behavioral axioms in the field **Axioms**. These axioms are quite simple and do not entail any conditions or synchronization expressions. They express the fact that the formal method argument `d` has been added to the place `msg` in the first case and has been removed in the second.

```
Class BasicTnode;
Signature
 Use Message;
 Type basictnode;
 Methods
 input  _ ,
 output _ : message;
Body
 Places msg : {message};
 Axioms
 input(d)  :: → msg{d};
 output(d) :: msg{d} → ;
 Where
 d : message;
End BasicTnode;
```

Fig. 8: The basic transit node

A natural but partial graphic representation of the classes is provided, according the following conventions:

- the inside of the ellipses represents what is encapsulated,
- the black rectangles represent the methods,
- the white rectangles correspond to the internal transitions,
- the grey rectangles correspond to a special method taking charge the dynamic creation and the initialization of the objects,
- the circles identify the places or the object attributes,
- the solid arrows indicate the data flow.

Figure 9 gives such a graphical representation for the class `basicTnode`.

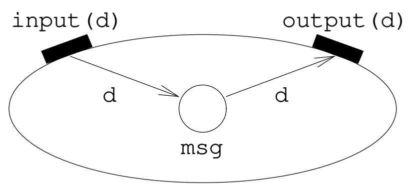

Fig. 9: An outline of the basic TNodes

The communication layer consists of an instance of the class `BasicWire` given in figure 10. This class must reflect that any TNode may exchange messages at any time with any other TNode. This is accomplished by means of the internal transition `msg-passing`, which asks for a simultaneous synchronization between two TNodes. The behavioral axiom should read "the event `msg-passing` behaves the same as both of the simultaneous external events, `input` and `output`, of both of the partners identified by `tn1` and `tn2`". We use the usual dot notation for method call. This axiom is satisfied for any values of both of the free variables `tn1` and `tn2`, and reflects that any TNode is free to exchange messages.

```
Class BasicWire;
Signature
 Use BasicTnode;
 Type basicwire;
Body
 Transitions
 msg-passing;
 Axioms
 msg-passing with
  tn1.input(d) // tn2.output(d);
 Where
 d : message;
 tn1, tn2 : basictnode;
End BasicWire;
```

Fig. 10: Basic communication layer class

Figure 11 gives the graphical illustration of the class `BasicWire`. Both of the dash arrows show the clientship between the communication layer and both the TNodes. However, the arrow direction does not express the data flow but the dependency relationship between the modules.

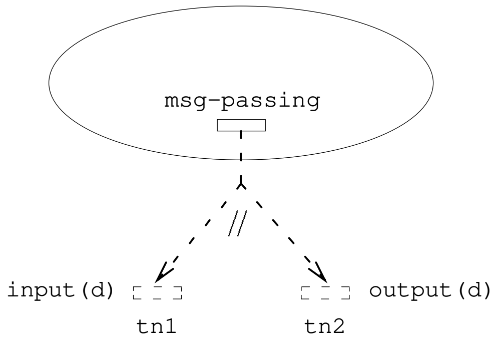

Fig. 11: Outline of the communication layer

4.4.3 Discussions

We have defined a very simple version of the transit node which is reduced to a buffer with several undistinguished communication ports. Unbounded concurrency between the communication ports is modelled by the auto-concurrence of the methods. The communication layer is defined at a very abstract level. All the wires are modelled using only one instance of the class `BasicWire`. All the communications are the result of the concurrent occurrences of the same event.

The questions that must be considered are how could we refine the TNode in order to have explicitly different communication ports instead of a global and abstract mechanism. How could we specify the definition of a distributed system topology.

4.5 Enrichment and Object Identifiers

This step takes charge of the enrichment of the basic TNodes and the basic communication. 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 onwards from one of the M output ports. Moreover, the use of object identifiers is improved in the senses that an attribute of a wire contains both the references of the TNodes it connects.

4.5.1 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 onward from the output ports.

A communication is viewed as the passing of a message through a wire linking one of the N input ports and one of the M output ports of two TNodes. Both the TNodes can be the same 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. Table 3 and 4 give more informations about the requirements and the informal design of the TNodes and the wire equipped with the notion of communication ports.

TNode with communication ports		
	Requirements	Informal design
Notes	none.	• a class `PortTnode`.
States	A bag which contains the message in transit through the node. The initial state is an empty bag.	• a sort `message` for the messages. • a place `msg` containing values of sort `message`. • the initial marking is the empty multi-set.
Events	• "message input" through a relevant port, the event can occur at any time and a message is put in the buffer. • "message output" through a relevant port, a message has to be present in the buffer and then removed.	• a method `input` with two parameters, `d` of sort `message` and `p` of sort `port`. *pre:* $p \leq N$. *post:* add the message `d` to the place `msg`. • a method `output` with two parameters, `d` of sort `message` and `p` of sort `port`. *pre:* $p \leq M$ and check if a message `d` is in place `msg`. *post:* consume the message `d` in place `msg`.

Table 3: Requirements and informal design of the TNodes with communication ports

Wires with communication ports		
	Requirements	Informal design
Notes	none.	• a class `PortWire`.
States	The linked TNodes with their respective input and output port.	• a sort `port` for the input/output ports. • a sort `link` which is a cartesian product of the sorts `porttnode` and `port` associates an input and output port of two TNodes. • a place `linked` of sort `link` with an empty initial marking.
Events	• "message passing", the "message output" and "message input" events for both the linked TNodes and their respective ports can be performed. Remark : both the TNnodes may be the same.	• an internal transition `msg-passing`. *pre:* the place `linked` contains only one value associating the output port `p` of the TNode `tn` to the input port `p'` of the TNode `tn'`. *sync:* simultaneous synchronization of the methods `output(p,d)` for `tn` and `input(p',d)` for `tn'`.

Table 4: Requirements and informal design of wires with communication ports

4.5.2 CO-OPN/2 Specifications

Both classes `BasicTnode` and `PortTnode` are similar, the introduction of the notion of port induces, of course, a new profile of both the methods `input` and `output` as well as a new behavior. These changes are reflected in figure 12. The specification of sort `port` (not detailed here) is built from the abstract data type of natural numbers in which both the constants `M` and `N` are defined.

```
Class PortTnode;
Signature
 Use  Message, Port;
 Type porttnode;
 Methods
 input  _ _ : message port;
 output _ _ : message port;
Body
 Places
 msg : {message};
 Axioms
 p < N ⇒ input(d,p)  :: → msg{d};
 p < M ⇒ output(d,p) :: msg{d} → ;
 Where
 d : message;
 p : port;
End PortTnode;
```

Fig. 12: The TNodes with communication ports

The outline of the class `PortTnode` in figure 13 is very similar to that one of the class `BasicTnode`. The structure does not change, only the methods' profiles are affected.

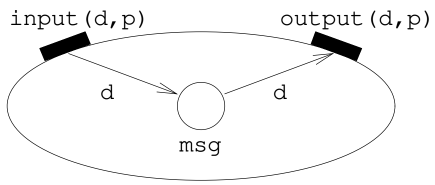

Fig. 13: An outline of the TNodes with ports

For the class `PortWire` we need the specification of a link between two TNodes. This is realized by the module `Link` in figure 14. The sort `link` represents a cartesian product of four components, the two Tnodes and their respective input/output port.

```
ADT Link;
Signature
 Use Port, PortTnode;
 Sorts link;
 Generators
 <_,_,_,_> : porttnode port
             porttnode port
             → link;
End Link;
```

Fig. 14: An abstract data type of a link

Figure 15 and 16 give, respectively, the specification and an outline of the class `PortWire`. One observes, on the one hand, the event `msg-passing` taking account of the attribute `linked` and, on the other hand, the special method **create**, which takes charge of the dynamic creation and initialization of the objects, and its parameter `ln` of sort `link` which is used to initialize the attribute. This initialization ensures that only one quadruplet is present in the place `linked`.

```
Class PortWire;
Signature
 Use Message, Port, PortTnode, Link;
 Type portwire;
 Methods
 create _ : link;
Body
 Transitions
 msg-passing;
 Places
 linked : link;
 Axioms
 msg-passing with
  tn.input(d,p) // tn'.output(d,p')::
  linked{<tn,p,tn',p'>} →
  linked{<tn,p,tn',p'>};
 create(ln) :: → linked{ln};
 Where
 d : message;
 ln : link;
 p, p' : port;
 tn, tn' : porttnode;
End PortWire;
```

Fig. 15: Wires between TNodes with ports

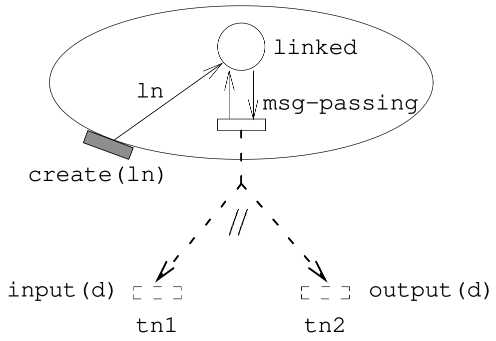

Fig. 16: Outline of the wires

4.5.3 Discussions

We have reached a more realistic TNode definition in which each TNode has now several distinguished communication ports. The communication 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 simultaneous message passing and no routing technique is used inside a TNode. A port can be auto-concurrently accessed inside a TNode.

The questions that must be considered are how could we easily serialize the access to the TNode ports and add routing features and what could be a specification of unreliable distributed systems which could loss or corrupt messages.

4.6 Inheritance

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

4.6.1 Transit Nodes and Routing

Each TNode includes an information for the routing of its messages to the output ports. This information associates each input port with a set of permissible output ports which may be used for routing the data messages. It is essential that the routing information of a TNode may be modified if necessary. The table 5 gives more informations about the requirements and the informal design of this kind of entity.

Communication is almost the same as the preceding one except that a wire links two TNodes equipped with routing. of the new type.

The following sorts which are necessary to develop the specifications associated to the TNodes and the wires are not detailed, only an informal description is given:

- a sort `portset`, obtained by the instanciation of the the generic algebraic specification of the sets with the effective module parameter `Port`.
- a sort `route`, which is cartesian product of the sorts `port` and `portset` which associates a set of output ports with an input.
- a sort `messageport`, which is a cartesian product of the sorts `message` and `port` and which is associated with an output port in choosing the route to a data message.

TNode with routing		
	Requirements	Informal design
Notes	none.	• Each TNode is an instance of a class `RouteTnode`.
States	The state is an enrichment of the one of the TNodes with communication ports. The additional information concern the sets of the output ports associated to each input port.	• a place `msg` that contains values of sort `messageport`. • a place `routes` that contains only one value of sort `route`. • the initial marking of `msg` and `routes` is the empty multi-set.
Events	• "message input" through a relevant port, a message is put in the buffer. • "message output" through a relevant port, a message has to be present in the buffer and then removed. • "Route definition", a new set of output ports is associated to an input port.	• a method `input` with two parameters, `p` of sort `port` and `d` of sort `message`. *pre:* `p < N`, there exists a pair `<p,ps>` in the place `routes`. *post:* the message `d` is added to place `msg`. • a method `output` with two parameters, `d` of sort `message` and `p` of sort `port` *pre:* `p < M` and check if a message `d` is present in place `msg`. *post:* consume the message `d` in place `msg`. • a method `routedef` with two parameters, `p` of sort `port` and `ps` of sort `portset`. *pre:* check if only the couple `<p,ps>` is present in the place `routes`. *post:* substitute the couple `<p,ps'>` by `<p,ps>` in `routes`.

Table 5: The requirements and informal design of the TNodes equipped with route

4.6.2 CO-OPN/2 specifications

The class RouteTnode in figure 17 (with its graphic outline version in figure 18) is not developed from scratch, it uses the inheritance mechanism which allows to reuse an already defined specification and add or redefine some components.

```
Class RouteTnode;
Inherit PortTnode;
Renaming porttnode Into routetnode;
Signature
 Use Message, PortSet, Route;
 Methods
 Redefine input _ _ : message port;
 Redefine output _ _ : message port;
 routedef _ _ : port portset;
 Redefine create _ : nat;
Body
 Use MessagePort, LossCriteria;
 Places
 routes : {route};
 Redefine msg : {messageport};
 Transitions
 lossmsg;
 Axioms
 p < N ⇒ input(d,p) ::
  routes{<p,ps>} →
  msg{<d,select(ps)>},routes{<p,ps>};

 p < M ⇒ output(d,p) ::
  msg{<d,p>} → ;

 routedef(p,ps) ::
  routes{p,ps'} → routes{<p,ps>};

 losscrit(d) = true ⇒ lossmsg ::
  msg{d} → ;

 create(0) :: → route{<p,ps>};
 p < N ⇒ create(succ(p)) with
  self.create(p) :: → route{<p,ps>};
 Where
 n : nat;
 d : message;
 p : port;
 ps, ps' : portset;
End RouteTnode;
```

Fig. 17: Transit nodes with routes

A new attributes and two new external events are added, these are the route definition method routedef and the internal events. Their respective behavioral axioms are added in the class RouteTnode. The keyword **Redefine** expresses that the inherited method and its behavioral axioms are ignored and redefined by means of the new ones given in the current class. This the case of both the methods input and output. Moreover, the principle may be applied to inherited attributes.

The new internal event loss-msg indicate that some messages can be lost in the TNode according a given criteria. At this step the criteria is very abstract and represented by means of a function called loss-crit. We assume that this function is defined in the module LossCriteria which is not detailed here.

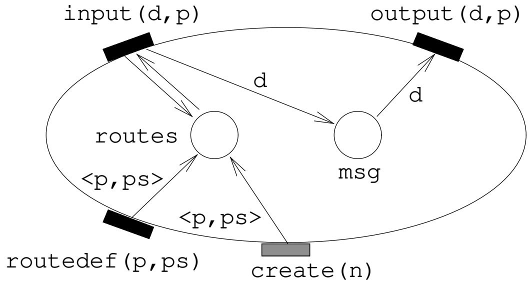

Fig. 18: Outline of the TNodes with routes

The route wire is not changed we only defined a new class which inherits all the preceding properties (cf. figure 19).

```
Class RouteWire;
Inherit PortWire;
Renaming portwire Into routewire;
End RouteWire;
```

Fig. 19: Reuse of the PortWire class

4.6.3 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 the message arrival. The route initialization is implemented using recursive call of method.

The questions that must be considered are how refining the TNodes in order to implement the communication interferences using an error detection code and how we can collect messages to be recovered by the system manager. Moreover we would like to reach this specification without modifying the definition of the communication layer.

4.7 Inheritance versus Subtyping

The previous versions of the running example led to a simple distributed systems made of transit nodes of class RouteTnode linked using basic wires. The RouteTnode class inherits from the PortTnode without sub-type relation. Thus, it is not possible to define a distributed system made of RouteTnode and PortTnode objects linked with the use of wire class. In this version, we define a new class of transit node CtrlTnode in order to illustrate the notion of sub-type, which is used to specialize the previous Tnode into a particular kind of error detection mechanism.

These new transit nodes filter the erroneous messages as before, for the point of view of the output port. But these messages are redirected to a special port which can be used to collect the faulty messages. This new class of TNode allows the customers to recover messages which include errors detected by using an error coding algorithm such as Hamming detection code.

We define a new class CtrlTnode which is a sub-type of RouteTnode. As previously explained, the subtyping relation is a semantic constraint. In our case, it should say that each CtrlTnode behaves at least as a RouteTnode, in order to satisfy the substituability 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 behavior, due to the weak constraint imposed to the loss function, than the particular hamming function.

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 it was defined in the previous part.

4.7.1 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. A class CtrlTnode inherits from the class RouteTnode and is extended with the following components:

- The specification of the sort message is enriched by an operation hamming which takes a data message as parameter and returns a boolean telling whether an error as been detected or not.
- A place faultymsg containing values of the sort messageport which are the faulty messages.
- an internal transition errdetect selecting the erroneous messages.
- a method recover with a parameter of sort message which can be used to collect the faulty messages.

The figure 20 and 21 show the CO-OPN/2 specification of this class.

```
ADT HMessage;
Signature
Use Message , Bool;
Operations
hamming _ : message → bool ;
Axioms
 ...
End HMessage;
```

Fig. 20: Hamming error detection on messages

```
Class CtrlTnode;
Inherit RouteTnode;
Renaming routetnode Into ctrltnode;
Signature
 Use Message;
 Subtype
 Methods
  recover _ : message ;
 Body
 Use MessagePort, HMessage,
      PortSet, Route;
 Places
  faultymsg : {messageport};
 Transitions
  errdetect ;
 Axioms
  hamming(d) = true ⇒ errdetect ::
   msg{<d,p>} → faultymsg{<d,p>};

  recover(d) :: faultymsg{<d,p>} → ;
 Where
 d : message;
 p : port;
End CtrlTnode;
```

Fig. 21: Transit nodes with routes and control

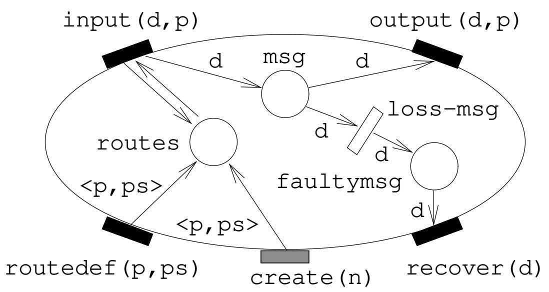

Fig. 22: Outline of the TNodes with error message filtering and recovering

4.7.2 Discussions

The refinement using sub-typing is used in order to easily create a new kind of TNode which are compatible with respect to their super-types. This allow to change the TNode without changing the communication layer. This new Tnode model a fault rejection mechanism.

We would like to know what kind of distributed system we can build and how we can we built it. Moreover, timing considerations would be considered in order to control the message contention.

4.8 Static Objects

The purpose of this section is to show how to build a concrete system from the previously presented definitions of classes. CO-OPN/2 includes the notion of static object of its predecessor CO-OPN. A static object is an object which is ever accessible from the beginning of the system life time to its end. In order to start a system composed of dynamic entities, static objects can be used to initialize some usefull objects as it is presented in our example. An other alternative, not presented here, is to use class internal transitions to start the system. But for the sake of simplicity we prefer to introduce the static objects.

4.8.1 Requirements and informal design of the static object `root`

The TNode layer of the distributed system is composed of two `routetnode` and one `ctrltnode` numbered one, two and three. Remember that `ctrltnode` type is a subtype of the type `routenode`. Each of these TNodes possess two input ports and two output ports.

For the communication part there is three wires linking the port one of the TNode one to the port one of the TNode two, etc.... Note that the wires are polymorphic in the sense that a wire links `routetnode` as well as `ctrltnode` as a result of the subtype relationship. Two messages are sent to the different input ports of each transit node.

We must define an internal transition `init` for the creation/initialization of the TNnodes and the wire associated to the free variables `tn1`, `tn2`, `ctn3` of sort `routetnode` and `ctn3` of sort `ctrltnode` and `w_11_21`, `w_21_31`, `w_31_11` of sort `wire`.

This is made by the internal transition `init` in figure 23 that it is fired at the beginning of the life of the system. The initial marking for static objects is defined using the field **`Initial`**.

```
Object root;
Body
 Use Bool;
 Places begin:{Bool}
 Transitions init;
 Axioms
 init with
  (tn1.create ..
   tn2.create ..
   ctn3.create ..
   w_11_21.create(tn1,1,tn2,1) ..
   w_21_31.create(tn2,1,ctn3,1) ..
   w_31_11.create(ctn3,1,tn1,1)
  ) ..
  (
   tn1.msg_input(1,msg1) //
   tn1.msg_input(2,msg2) //
   tn2.msg_input(1,msg3) //
   tn2.msg_input(2,msg4) //
   ctn3.msg_input(1,msg5) //
   ctn3.msg_input(2,msg6)
  ) :: begin{true}→;
 Initial
  begin{true};
 Where
 tn1,tn2 : routetnode;
 ctn3 : ctrltnode;
 w_11_21,w_21_31,w_31_11 : wire;
 msg1, msg2, msg3,
 msg4, msg5, msg6 : message;
End root;
```

Fig. 23: Abstract data type of the ports

4.8.2 Discussions

The static CO-OPN/2 objects are a usefull way to define always executable objects which can be used for system initialisation. Some particularities, such as the sequentialization of the **`create`** class methods, appears in the initialization process.

We may ask how to reach a complex heterogenous distributed systems where time could be managed and gateway between heterogeneous subsystems established.

4.9 Polymorphic references

4.9.1 Merging Heterogeneous Distributed Systems

In this part we are interested in specifying the distributed systems which are made of two different kinds of networks, modelled by two different classes of TNodes and wires. The first is composed of TNodes and wires, as has been defined in the class `RouteTnode`, while the second is obtained from the first one by managing timing information destinated to model the timing latency into the TNodes.

We have not used our tabular description because we have supposed that the reader has been trained enough in the CO-OPN/2 concepts and in the development methodology. This section introduces the modelling ability of CO-OPN/2 to describe heterogeneous systems.

4.9.2 Requirements and Design

We consider that new systems are composed of the TNode of class `RouteTNode` and class `CtrlTNode` and the new TNodes that are developed in this section. Each new TNode has a local clock. The messages in transit across all of the system have a frame composed of a timing information and a data component. The timing part indicates the time of total transit. 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 TNode of class `RouteTNode` by adding the clock's 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.

The TNode layer is split in two parts : the clock part and the Tnode part and a class is defined for each of them, called `TimedRouteTnode` and `Clock`. We suppose that there is also the class `TimedCtrlTnode` for the controllable TNodes with time.

4.9.3 The Clock Class

The reference of time is given by one object of class `Clock`. This object provides two services, the time (described by the sort `time` in figure 24) 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 auto-concurrent and then sequentialize the access to the reference of time.

```
ADT Time As Nat;
Signature
 Renaming nat Into time;
End Time;
```

Fig. 24: The time sort

We realize the clock (see figure 25 and 26) using two states, the number of not registered `tick` events into the place `tickcount` and the place `currenttime` containing the current time value. The internal transition `incr` immediately react after a `tick` event in order to adjust the `currenttime` value, this last event is necessary to let the methods `tick` and `gettime` concurrent.

```
Class Clock;
Signature
  Use Time ;
  Type clock;
  Methods
   gettime _: time ;
   tick ;
Body
 Use Bool;
 Places
   currenttime :{time};
   tickcount : {bool};
 Transitions
   incr ;
 Axioms
  gettime(t) ::
    currenttime{t} → currenttime{t} ;

  tick ::
    → tickcount{true} ;

  incr ::
    tickcount{true}, currenttime{t}
       → currenttime{succ(t)} ;

 create :: → currenttime{zero} ;
 Where
  t : time;
End Clock;
```

Fig. 25: The clock class

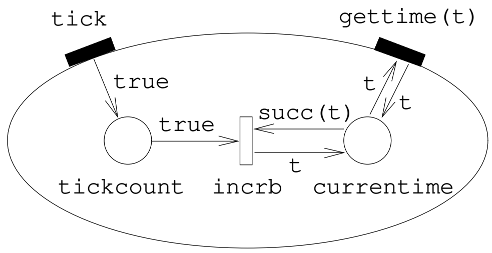

Fig. 26: An outline of the clock class

4.9.4 The TimedRouteTnode class

Each `TimedRouteTnode` instance calls its associated local clock instance (see its creation in the base case `create(0)`) in order to get the time information to determine how long a message has been standing inside. The new messages are now couple of Time and Message (see figure 27) in order to have an indication of the time spends into the system. The modelling of the messages has been made using sorts instead of classes for conceptual reasons, it implies that we cannot use subtyping scheme to specialize `message` into `timedmessage` and consequently, we cannot have relation between that types (sub-sorting cannot be used for such sorts).

```
ADT TimedMessage
    As CP2(Time, Message);
Signature
 Renaming cp2 Into timedmessage;
End TimedMessage;
```

Fig. 27: The timed message class

As it was made before in `RouteTnode`, the data inside a TNode must be labelled by an output port number as in figure 28 in which we instanciate the generic algebraic specification module `CP3`, and furthermore with a time stamp in order to measure the time used to transit through the node.

```
ADT PortTimedMessage
    As CP3(TimedMessage, Time, Port);
Signature
 Renaming cp3 Into porttimedmessage;
End PortTimedMessage;
```

Fig. 28: The message inside the timed TNode

The main difference is that time stamping will be performed when entering the TNode and when leaving the Tnode. This allow for future versions of the Tnode to perform some latency control in order to avoid the too long contention of messages into a TNode. The new version of the Tnode described in figure 29, which has a subtype relation completely extend the previous one, including the synchronization to the clock class and the stamping of the messages.

```
Class TimedRouteTnode;
Inherit RouteTNode;
Signature
 Type timedtnode;
 Subtype timedtnode < routetnode;
 Use TimedMessage, PortSet,
      PortTimedMessage;
 Methods
 input   _ _ : timedmessage port;
 output _ _ : timedmessage port;
 routedef _ _ : port portset;
 create _ : nat;
Body
 Use Route;
 Places
  routes : {route};
  msg : {porttimedmessage};
  theclock : {time};
 Axioms
  p < N ⇒
   input(<t',d>,p)
    with myclock.gettime(t) ::
     route{<p,ps>},
     theclock{myclock} →
     msg{<<t',d>,t,select(ps)>},
     route{<p,ps>},
     theclock{myclock};

  p < M ⇒
   output(<t - t''+ t',d>,p)
    with myclock.gettime(t) ::
     msg{<<t',d>,t'',p>},
     theclock{myclock} →
     theclock{myclock};

   routedef(p,ps) ::
     routes{p,ps'} →
     routes{<p,ps>};

   create(0) with myclock.create ::
     → theclock{myclock};
   create(succ(n))
    with self.create(n) ::
     → route{<n,ps>};
 Where
  n : nat;
  d : message;
  p : port;
  ps, ps' : portset;
  t,t':time;
  myclock:clock;
End TimedRouteTnode;
```

Fig. 29: The timed TNode

4.9.5 The communication layer

The idea is to give a description of how we can connect different TNode expressed in specification `RouteTnode` and `TimedRouteTnode`. We suppose, as for the TNode layer, that we have the class `RouteTnode` communication layer features available in this version. The communication layer of class `TimedRouteTnode` integrates two new features : adding of a transmission delay to the message time information of the messages which are in transit between to Tnodes of class TimedRouteTnode and 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 wire 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 wire linking in an unidirectional way TNodes of class RouteTNode to TNodes of class TimedRouteTnode, the time stamp is set to zero when entering into the TNodes of class TimedRouteTnode.
- The wire linking in an unidirectional way TNodes of class TimedRouteTnode to TNodes of class RouteTnode looses the time stamps attached to the messages.

The class HeteroWire described in figure 30 contains the description of the selection of the four kind of transmission between TNodes. The mechanism used to select the kind of TNode that are concerned by the transmission is based on the use of a fine discrimination of the reference hierarchy (routetnode is the super-type of timedtnode) using the following conditions:

- tn = ref is satisfied if and only if the value of tn which is of type routetnode belongs to the domain timedtnode.
- tn = **super**(ref) is satisfied if and only if the value of tn belongs to the domain routetnode but not to timedtnode.

```
Class HeteroWire;
Signature
 Use Link;
 Type wire;
 Methods
 create _ : link;
Body
 Use Message, Port, RouteTnode,
 TimedRouteTNode;
 Transitions
  msg-passing;
 Places
  linked : link;
 Axioms
  tn = ref and tn' = ref ⇒
   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 = super(ref) and tn' = ref ⇒
   msg-passing with
    tn.input(d,p) //
       tn'.output(<t,d>,p')::
    linked{<tn,p,tn',p'>} →
    linked{<tn,p,tn',p'>}

  tn = ref and tn' = super(ref) ⇒
   msg-passing with
    tn.input(<0,d>,p) //
       tn'.output(d,p')::
    linked{<tn,p,tn',p'>} →
    linked{<tn,p,tn',p'>}

  tn = super(ref)
          and tn' = super(ref) ⇒
   msg-passing with
    tn.input(d,p) //
       tn'.output(d,p')::
    linked{<tn,p,tn',p'>} →
    linked{<tn,p,tn',p'>}

   create(ln) :: → linked{ln};
 Where
 d : message;
 ln : link;
 p, p' : port;
 tn, tn' : routetnode;
 ref: timedtnode;
End HeteroWire;
```

Fig. 30: The wire for heterogeneous TNodes

4.9.6 Discussions

The local clock is modelled using reactive object instance which could be controlled by a physical mechanism like a quartz system. Complex heterogeneous distributed system are defined including different message frame, transit time management, network gateways .... In order to control access to different element of the class of the TNode hierarchy, fine reference control mechanisms are used to separate access to sub or super class instances.

4.10 A Heterogeneous Distributed System Example

4.10.1 Requirements and Design

The TNode part of the distributed system is composed of two different sub-distributed systems constructed with standard transit nodes and timed transit nodes. The first distributed system is made of two `RouteTnode` and one `CtrlTnode`. The second distributed system is made of three timed Transit nodes `TimedRouteTnode` which can also be controlled (with their associated clocks).

The communication layer is used for linking the transit nodes within each sub-distributed systems and also to link the heterogeneous sub-distributed systems in both directions.

Two messages are sent to the different input ports of each transit node of the overall distributed system.

4.10.2 CO-OPN/2 specification

We use the notion of static object of CO-OPN/2 as in section 4.8 which are immediately active at the beginning of the system. We define only one static objet `distribSysInit` (figure 31) in order to create the heterogeneous distributed system. This object has only one internal methods `init` with a complex synchronization expression doing all the needed object creation.

```
Object distribSysInit;
Body
 Use Bool, HeteroWire, PortTn-
ode, RouteTnode,
               TimedRouteTn-
ode, TimedCtrlTnode;
 Places begin:{Bool}
 Transitions init;
 Axioms
 init with
(
tn1.create. .
tn2.create. .
c_tn3.create. .
w_11_21.create(tn1,1,tn2,1). .
w_21_31.create(tn2,1,c_tn3,1). .
w_31_11.create(c_tn3,1,tn1,1). .

t_tn1.create. .
cl1.create. .
t_tn2.create. .
cl2.create. .
t_c_tn3.create. .
cl3.create. .
w_11_21.create(t_tn1,1,t_tn2,1). .
w_21_31.create(t_tn2,1,t_c_tn3,1). .
w_31_11.create(t_c_tn3,1,t_tn1,1). .

w_12_11.create(t_tn1,2,tn1,1). .

w_32_22.create(c_tn3,2,t_tn2,2). .

  )
. .
  (
tn1.input(1,msg1)//
tn1.input(2,msg2)//
tn2.input(1,msg3)//
tn2.input(2,msg4)//
c_tn3.input(1,msg5)//
c_tn3.input(2,msg6)//

t_tn1.input(1,t_msg1)//
t_tn1.input(2,t_msg2)//
t_tn2.input(1,t_msg3)//
t_tn2.input(2,t_msg4)//
t_c_tn3.input(1,t_msg5)//
t_c_tn3.input(2,t_msg6)//
);
 Initial
  begin{true}
 Where
 tn1,tn2 : routetnode;
 ctn3 : ctrltnode;
 t_tn1, t_tn2: timedtnode;
 c_t_tn3  : ctrltimedtnode;
 w_11_21,w_21_31,w_31_11 : wire;
 t_w_21_31 and t_w_31_11 : wire;
 msg1, msg2, msg3,
 msg4, msg5, msg6 : message;
 t_msg1, t_msg2, t_msg3,
t_msg4, t_msg5, t_msg6 : message;
End distribSysInit;
```

Fig. 31: Static object for initialization of the heterogeneous distributed system

5 Simulation, Prototyping

The development of applications in CO-OPN is supported by the SANDS environment which includes simulation tools and prototyping tools, the simulation tool is based on same techniques except that computation is performed symbolically on an higher level of abstraction than that of the prototyping.

5.1 Prototyping

Prototyping with different objectives can be applied at different levels in the development process in order to :

- Determine the feasibility of the product.
- Evaluate of the realization cost.
- Establish and validating requirements.
- Validate the functional specifications.
- Supply information about the performance of the system and to evaluate execution costs.
- Serve as a model, or as a dynamic representation, especially during the testing of the system.

Prototyping is beneficial with respect to the general knowhow acquired when building a given system.

5.2 Kinds of Prototyping Methods

Starting with from the initial requirements, there are three main possible ways in which prototyping may be used in the software development process:

Exploratory programming is the most primitive among these three ways. It is very close to traditional application development, in that there is no specification. Modules are added to the system, one after another, and those not yet implemented are simulated to allow testing. This approach may lead to unexpected results when integrating a new module due to its unpredictable side-effects. This is particularly the case within parallel environments.

Throw-away prototyping, in which the prototype serves mainly to validate the requirements or a part of them, after which the operational system must be reconstructed from scratch. This technique should be used when the primary goal is rapid and flexible system evaluation, and when the prototyping environment lacks reliability or execution efficiency. In this case we differentiate between the simulated and the stand-alone prototypes. In the former, execution is possible only within the development environment, whereas in the latter, the resulting prototype is autonomous.

Incremental prototyping, in which the prototype is developed through various steps leads to the ultimate goal which is with respect to user application. This implies that the prototyping tool has provided the necessary characteristics with respect to completeness, speed and robustness.

Our proposal consists of a combination of incremental and throw-away prototyping as it can evolve by virtue of refinement from the early stages in the software life cycle to an almost fully implemented version. The code generated by our compiler can form the skeleton of the application, or of a particular module in the application, and should be efficient and legible enough to be incorporated into a complete production-quality environment.

One can also distinguish different functional purposes for the prototype:

In vertical prototyping only certain critical functions are represented. They are considered as being independent of the rest of the system. In horizontal prototyping, all the functions of the final product are accounted for, but this is sometimes only at the draft level which leaves aside certain aspects that are not mandatory within the current aims.

These two approaches can easily be combined within any given prototype, especially within prototyping environments such as ours, where emphasis is given to modularity.

5.3 Prototyping with SANDS

The SANDS environment provides a general way for producing incremental prototyping from CO-OPN specification in which each unit can be automatically implemented by means of a canonical, semantical correct, code generation [BFR92]. This is possible if the specification adheres to certain rules that that may be expressed syntactically. This is the well known 'gracious presentation' in the algebraic specification [Bid87, Bid89] which guaranties the construction of a notherian rewriting system based upon the axiomatic specification. With respect to the Petri net, the satisfaction of the free flow condition [Buc92] allows for the finding of an ordering in the evaluation of the local net condition and in its synchronization. The incremental construction of the prototype production allows the user to progressively replace automatically generated codes with user written codes and then to assist the user with the true size evaluation of the product under development.

5.4 Techniques for Prototype and Simulation Implementation

We present here some tools that we are considering for use in our automatic code generation. The state of the art in prototype implementation is based upon the definition of operational semantics for the algebraic specification languages. Two kinds of semantics are available: those based on rewriting [NJ90] which are valid only for closed terms and those which are based on resolution [NJ90]. Useful methodological techniques, such as the integration of already implemented codes into rewriting are proposed in [Cho87]. Different algorithms have been developed to evaluate efficiently high level nets, the small grain approaches such as in [BEM90] or the coarse grain approach [BFR92].

6 Testing and Analysis

Writing programs from specifications is necessary in order to create a realistic software systems. Nevertheless, because of the potential for producing unsound or incomplete realizations it is necessary to provide for a way of identifying possible fallacies created during writing. Testing methods are essentially proposed to detect potential errors without concretely proving the completeness of the programs with respect to the specifications.

We introduce in [Buc95] a methodology for testing concurrent programs against their formal specifications, in particular CO-OPN specifications. In this case, we explore the testing of units against their formal specifications [SM94]. This seems to be one of the most promising approaches for improving confidence in the implementation of critical systems. Our framework presents the principles that must be used for selecting valid tests which one to be applied automatically to the concurrent programs. It also helps to determine how well the tests succeed. It is obvious that these tests must be finite and that, consequently it is not possible to prove completely the adequation of the programs with respect to their specifications. This may be done only in the case where certain hypotheses are verified. These hypotheses must represent current testing practices. Previous work on testing methods has been described using abstract data types [BB94a] and we extended this work to concurrent programs, in order to be able to test full Ada programs specified using CO-OPN .

Starting with the well defined semantics of the formalism used for expressing the formal specifications, we have elaborated the correct theoretical testing framework. This has been accomplished by adopting the theory of testing from Bernot [BGM91] and the operational techniques which take into account testing strategies, from Marre [Mar88, Mar91]. Moreover, the validity of a program with respect to its specification has been introduced by means of proper satisfaction relation which is based upon the bi-simulation equivalence relation. Such equivalence is characterized with temporal logic formulae, in particular the process logic of Hennessy-Milner [Mil89]. The CO-OPN semantics and the definition of test sets through their logical characterization lead to the design of tools that can be used to select test sets automatically. These are based mainly on symbolic computations which one in turn based upon extensions of the resolution principle [Mar91] for equational logic.

After generating test sets, a test must be designated as being valid or not in the program under consideration, which is the purpose of the Oracle[Wey80]. The principle applied is that partial computations of the programs to be compared to the temporal process logic which characterizes the specification behavior. Thus the oracle may try to determine if observational equivalence exists between the program and the specification by well-known decision procedure for finite systems [Kor91]. The test generation is under the full control of the user and this person must employ competence and ingenuity to select the most appropriate tests. Moreover, a procedure is applied for checking that the program validates the test which is produced.

7 Conclusion

In this paper we have presented a formal approach called CO-OPN 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 algorithm, the vision system, the development of parallel algorithms etc. , as well as for the semantics of concurrent language by the modelling of the actor languages. The transit node is used to describe completely and in an informal way particular notions of the CO-OPN/2 language.

The development of distributed systems needs languages which pride for the possible description of desired properties in a structured fashion. CO-OPN/2 support modularity and the expression of different kind of concurrency. In this paper we have shown how the concepts of CO-OPN/2 can handle concurrency issues and classification requirements. The following major points of our approach have been presented:

- we express intra-object and inter-object true concurrency,
- we differentiate inheritance as a syntactic mechanism and subtyping which is a semantic principle,
- we give a way to express hierarchies of abstraction,
- we allow modular construction of specifications,
- we give a formal model of specification refinement.

With the progressive development of the Transit node, we have shown the possible way of refining specifications, starting from 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.

Future studies will be conducted along three lines:

- the description of the semantics of CO-OPN/2 which concerns the subtyping and dynamic binding will be improved and the multiple inheritance notions will be presented with more precision,
- tools to support the new features of the CO-OPN/2 language for the already developed environment SANDS will be constructed,
- a distribution model of CO-OPN/2 objects will be defined for heterogeneous systems with its operational semantics,
- experiment will be made with CO-OPN/2 on different practical case studies.

Acknowledgements

We wish to thank S. Barbey, J. Hulaas and P. Racloz for their numerous and valuable comments during the evolution of this study.

References

[AB94] Marc Aiguier and Gilles Bernot. Algebraic semantics of object type specifications. In R. Wieringa and R. Feenstra, editors, *Working papers of the international Workshop on Information System Correctness and Reusability IS-CORE'94*, pages 286–295, Amsterdam, The Netherlands, September 1994. vrije University.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bézivin, editor, *ECOOP '87 Conference*, volume 276 of *Lecture Notes in Computer Sciences*, pages 234–242, Paris, France, June 1987. Springer Verlag.

[AR93] Egidio Astesiano and Gianna Reggio. Algebraic specification of concurrency. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, selected papers*, volume 655 of *LNCS*, pages 1–39. Springer-Verlag, 1993.

[BB94a] Stéphane Barbey and Didier Buchs. Testing Ada abstract data types using formal specifications. In Marcel Toussaint, editor, *Ada Europe '94 Conference Proceedings*, volume 887 of *LNCS*, pages 76–89, Copenhagen, Danemark, September 26-30 1994. Springer-Verlag. (to appear).

[BB94b] Olivier Biberstein and Didier Buchs. CO-OPN an object- oriented specification language based on hierarchical algebraic nets. Tech. Report 76, EPFL, 1994.

[BB94c] Olivier Biberstein and Didier Buchs. An object oriented specification language based on hierarchical algebraic petri nets. In R. Wieringa and R. Feenstra, editors, *Working papers of the international Workshop on Information System Correctness and Reusability IS-CORE'94*, pages 47–62, Amsterdam, The Netherlands, September 1994. vrije University.

[BBMM94] Denis Brunet, Didier Buchs, Daniel Monteiro, and Fabrice Mourlin. Parallel distributed algorithm design through specification transformation: The asynchronous vision system. In *FIP WG10.3 Working Conference on Programming Environments for Massively Parrallel Distributed Systems*, Monte Verita, Ascona, April 25-29 1994.

[BC93] Eugenio Battiston and Fiorella De Cindio. Class orientation and inheritance in modular algebraic nets. In *Proceedings International Conference on Systems, Man and Cybernetics*, volume 2, pages 717–723, Palais de L'Europe Hôtel Westminster, Le Touquet- France, October 1993. Ecole Centrale de Lille.

[BCM88] Eugenio Battiston, Fiorella De Cindio, and G. Mauri. OBJSA nets: a class of high level nets having objects as domains. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1988*, volume 340, pages 20–43. Springer-Verlag, 1988.

[BEM90] B. Butler, R. Esser, and R. Mattmann. A distributed simulator for high order petri nets. In *Advances in Petri Nets 1990*, 1990. Lecture Notes in Computer Science, Springer Verlag.

[BFR92] Didier Buchs, Jacques Flumet, and Pascal Racloz. Producing prototypes from CO-OPN specifications. In *3rd International Workshop on Rapid System Prototyping*, pages 77–93. Research Triangle Park, North Carolina, USA, IEEE, June 1992.

[BFR93] Didier Buchs, Jacques Flumet, and Pascal Racloz. SANDS: Structured algebraic net development system. In Buy Ugo, editor, *14th International Conference on Application and Theory of Petri Nets, Tool presentation abstracts*, pages 25–29. Chicago, USA, June 1993.

[BG91] Didier Buchs and Nicolas Guelfi. A concurrent object oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[BG92a] Didier Buchs and Nicolas Guelfi. Distributed system specification using CO-OPN. In Yau S. Stephen and Yuen Daniel, editors, *3rd Workshop on Future Trends of Distributed Computing Systems*, volume 3, pages 26–33. Tapei, Taiwan, IEEE, June 1992.

[BG92b] Didier Buchs and Nicolas Guelfi. High level nets. Cahiers du CUI 58, University of Geneva, 1992.

[BG93a] D. Buchs and N. Guelfi. Modularity, compositionality and hierarchies. In E.BEST, editor, *ESPRIT DEMON no3148 Final Report*, volume 217 of *GMD-studies*, pages 218–234. vrije University, 1993.

[BG93b] Didier Buchs and Nicolas Guelfi. Formal development of actor programs using structured algebraic Petri nets. In Bode Arndt, Reeve Mike, and Wolf Gottfried, editors, *Int. Conference on parallel Architectures and Languages Europe PARLE'93*, volume 694 of *LNCS*, pages 353–366. ECRC, Munich Germany, Springer-Verlag, June 1993.

[BGHS91] G. Blair, J. Gallagher, D. Hutchinson, and D. Sheperd, editors. *Object-Oriented Languages, Systems and Applications*. Pitman, 1991.

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[Bid87] M. Bidoit. The stratified loose approach: a generalization of initial and loose semantics. In D. Sanella and A Tarlecki, editors, *Recent Trends in Data type Specification, Selected papers of the 5th workshop on specification of abstract data types*, volume 332 of *LNCS*, pages 1–22, Gullane, Scotland, September 1987. Springer-Verlag.

[Bid89] M. Bidoit. *Pluss, un langage pour le developpement de spécifications algébriques modulaires.* PhD thesis, LRI Université de Paris-Sud, Centre d'Orsay, May 1989. Thèse de doctorat d'Etat.

[BMS93] Didier Buchs, Fabrice Mourlin, and Lorens Safavi. CO-OPN for specifying a distributed termination detection algorithm. In *World Transputer Congress, Workshop on Software Engineering for Parallel Systems*, volume 3, pages 25–29. Aachen, Germany, IOS Press, September 1993.

[Buc92] Didier Buchs. Observational equivalence of algebraic Petri nets. Cahiers du CUI 57, University of Geneva, 1992.

[Buc95] Didier Buchs. Test selection method to validate concurrent programs against their specification. In M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM '95 Third Conference on Software Quality Management*, Seville, Spain, April 1995.

[Cho87] C. Choppy. Formal specification, prototyping and integration tests. In *Proc. 1st European Software Engineering Conference, Strasbourg*, pages 185–192, 1987.

[DHP91] C. Dimitrovici, U. Hummert, and L. Petrucci. Semantics, composition, and properties of algebraic high-level nets. In *Advances in Petri Nets*, volume 524 of *LNCS*, pages 93–117. Springer-Verlag, 1991.

[Eck94] Gabriel Eckert. Types, classes and collections in object-oriented analysis. In *IEEE International Conference on Requirements Engineering*, pages 32–39, Colorado Springs (USA), April 1994. Also available as Technical Report (EPFL-DI-LGL No 93/33).

[ELR90] J. Engelfriet, G. Leih, and G. Rozenberg. Parallel object-based systems and petri nets. Tech. Report 04, 05, Leiden University, 1990.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6 of *EATC Monographs*. Springer-Verlag, 1985.

[FM90] J. Fiadero and T. Maibaum. Describing, structuring and implementing objects. In J. W. Bakker, W. P. Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX school proc.*, volume 489 of *LNCS*, pages 274–310, Noordwijkerhout, The Netherlands, May/June 1990.

[GM89] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and patial operations. Technical Report SRI-CSL-89-10, Computer Science Lab, SRI International, July 1989.

[Gue94] Nicolas Guelfi. *Les réseaux algébriques hiérachiques: un formalisme de spécification structurée pour le développement de systèmes concurrents.* PhD thesis, Université Paris-Sud, LRI, September 1994.

[Kie89] A. Kiehn. Petri net systems and their closure properties. In Grzegorz Rozenberg, editor, *Advances in Petri nets*, volume 424 of *LNCS*, pages 306–328, Berlin, 1989.

[Kor91] H. Korver. The current states of bisimulation tools. Master's thesis, University of Amsterdam, January 1991. Internal Report P9101.

[Lak94] C. A. Lakos. LOOPN++: A new language for object oriented Petri nets. R94-4, University of Tasmania, Dept. Comp. Science, Networking Research Group, April 1994.

[LH93] K. Lano and H. Haughton. *Object-Oriented Specification Case Studies.* Prentice Hall, 1993.

[Mar88] Bruno Marre. Vers une génération automatique de jeux de tests en utilisant les spécifications algébriques et la programmation logique (towards automatic test sets generation using algebraic specifications and logic programming). In *CGL4 - 4th Conference on Software Engineering*, pages 19–29, Paris, France, October 18-21 1988. AFCET.

[Mar91] Bruno Marre. *Sélection automatique de jeux de tests à partir de spécifications algébriques en utilisant la programmation logique.* PhD thesis, LRI, Université de Paris XI, Orsay, France, January 1991.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPM+91] A. Mauboussin, H. Perdrix, M.Bidoit, M.C. Gaudel, and J. Hagelstein. From an erae requirements specification to a pluss algebraic specification: a case study. In J.A Bergstra and L.M.G. Feijs, editors, *Algebraic methods II: Theory, Tools and Applications*, volume 490 of *LNCS*. vrije University, Springer Verlag, 1991.

[MW91] S. Mauw and F. Wiedijk. Specification of the transit node in psfd. In J.A Bergstra and L.M.G. Feijs, editors, *Algebraic methods II: Theory, Tools and Applications*, volume 490 of *LNCS*. vrije University, Springer Verlag, 1991.

[NJ90] N. Dershowitz N. and J.-P. Jouannaud. *Rewrite Systems*, pages 243–309. North-Holland, 1990.

[Pap89] Michael Papathomas. *Concurrency issues in object-oriented languages*, pages 207–246. University of Geneva, 1989.

[Reg93] G. Reggio. Event logic for specifying abstract dynamic data types. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification: selected papers*, volume 655 of *LNCS*, pages 292–309. Springer-Verlag, 1993.

[Rei85] Wolfgang Reisig. Petri nets. an introduction. In W. Brauer, G.Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer-Verlag, 1985.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. In *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.

[Rib94] Leila Ribeiro. OO-AHL-NETS object oriented algebraic high level nest. In R. Wieringa and R. Feenstra, editors, *Working papers of the international Workshop on Information System Correctness and Reusability ISCORE'94*, pages 146–162, Amsterdam, The Netherlands, September 1994. vrije University.

[SB94] C. Sibertin-Blanc. Cooperative nets. In Robert Valette, editor, *Application and Theory of Petri Nets 1994*, volume 815 of *LNCS*, pages 471–490, 15th International Conference, Zaragoza, Spain, June 1994. Springer-Verlag.

[SM94] I. Spence and C. Meudec. Generation of software tests from specifications. In M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM '94 Second Conference on Software Quality Management*, volume 2, pages 517–530, Edinburgh, Scotland, UK, July 26-28 1994.

[Vau87] Jacques Vautherin. Parallel systems specification with colored Petri nets and algebraic specification. In Grzegorz Rozenberg, editor, *Advances in Petri Nets*, volume 266 of *LNCS*, pages 294–308. Springer-Verlag, 1987.

[Wey80] Elaine J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.