Towards a New Semantics for Mondel Specifications Based on the CO-Net Approach *

Nasreddine Aoumeur† Gunter Saake
Institut für Technische Informationssysteme
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg
E-mail: {aoumeur|Saake}@iti.cs.uni-magdeburg.de
Tel.: ++49-391-67-18659 Fax: ++49-391-67-12020

Abstract

The purpose of the present paper is twofolds. First we present our first results towards a tailored specification and validation object-oriented (OO) model for distributed systems. Referred to as CO-Nets, the model is a variant of object Petri nets characterized by: a complete integration of object-oriented concepts and constructions into an appropriate form of algebraic Petri nets named ECATNets; two communication patterns for intra- and inter-objects interaction that enhance modularity and concurrency without violating the encapsulated aspects of each class and last but not least, the interpretation of the behaviour of the constructed model into rewriting logic which allows some validation by rapid-prototyping using rewrite techniques. Second, as a significant case study, we assess the suitability of the CO-Nets approach by providing Mondel OO specification language with a formal semantics. Mondel language has been conceived for developing distributed applications with particularly: a state-oriented style of description, synchronous communication based on the rendez-vous mechanism and a formal semantics, based on coloured Petri nets, that we propose to improve using the CO-Nets approach.

1 Introduction and Motivation

The object orientation, with its powerfull abstraction mechanisms, is nowadays a widely accepted paradigm dealing with different phases i.e. analysis, specification/validation, design and implementation in the development life cycle of large software in general and distributed ones particularly. Indeed, the object paradigm allows for: an intrinsic concurrency and distribution through its message passing mechanism and information hiding; a natural conceptualization of systems as a community of objects and messages, which is very close to the intuitive perception of distributed systems; an incremental building of complex systems through its powerful abstraction mechanisms (including object composition, inheritance and interaction).

These advantages and appropriateness have been fully exploited and confirmed by a great number of OO approaches, languages and formalisms for developing large systems in general and distributed ones in particular. Among them we have especially: OOD [Boo91] and OMT [RBP+91] as OO approaches and methodologies; TROLL [JSHS96], Maude [Mes93] and Mondel [BBE+91] as OO specification/programming languages; and HOSA [GD93], rewriting logic [Mes92] and recently DTL [ECSD98] as (algebraic based) formalisms.

Proposed at Montreal University, the Mondel OO specification language is particularly distinguished by the following features: object description as type instance with a persistent identity;

*In Proc. of Modelierung'99, Karlsruhe, Germany, B. G. Teubner-Verlag, J. Desel and K. Pohl and P. Schuerr (eds.), pp. 107-122, 1999.

†On leave from Oran University(in Algeria), supported by a DAAD grant.

multiple inheritance based on subtyping; concurrency with synchronous communication based on the rendez-vous mechanism and a state-oriented methodology for developing applications. Beside that, Mondel specification has received a formal semantics [BB91], using coloured Petri nets, that allows verification of some crucial properties of the specification like absence of deadlock, executability of operations, etc, on the basis of coloured Petri nets analysis properties [Jen92].

However, in spite of the strengths of the proposed Mondel formal semantics, we claim that a more appropriate semantics can be obtained by overcoming the following shortcomings, that present this formalization, both at the translating ideas level as well at the model level.

- In the modeling of tokens in places as object instances of the form $< Id, attributes, stack >$, the Id(object identifier) and $attributes$(object attributes names with their actual values) components are very intuitive and sound translation ideas; however, the *stack component* — storing values of actual operation parameters, local variables and intermediate computations— is some how artificial, which make any straightforward mechanization of the translation difficult. Moreover, the messages (i.e. operations on objects), that have to be independent entities (i.e. may be created, deleted, etc), are rather conceived as fixed part of the stack content, which avoids any form of concurrent processing.

- The separation between object acquaintenances (i.e. attributes) and its 'controled' internal states, and the modeling of each object internal state as a place seems also to be non-intuitive translation ideas. Indeed, as will be shown later in our approach, the perception of such internal states *just like* the other (stateless) attributes is more natural. On the other hand, by avoiding these (state) places, the resulting net is more simple, manageable and understandable.

- CPNets have been suitably used for modelling object instances as 'coloured' tokens and object behaviour as appropriate transitions. On the other hand, some Mondel specification properties have been verified using CPNets properties analysis. However, CPNets need for an appropriate (OO) extensions, like the one given by C. Lakos [Lak95] for example, for dealing with the *inheritance* and the persistence of *object identities* that are the main concepts of the OO paradigm in general, and Mondel specification particularly.

With the aim to cope with these shortcomings, the present paper proposes to translate Mondel specifications into the CO-Nets approach that we are currently developing. The CO-Nets OO approach is intended to modeling and rapid-prototyping distributed systems with more adequacy to information systems specification. As modeling framework, CO-Nets model is based on a judicious integration of OO concepts and constructions (including classes as modules, inheritance and interaction) into the ECATNets [BM92] algebraic Petri nets, which is mainly distinguished by its capability of separating between the enabling tokens and the destroyed ones and by its true concurrent semantics expressed into rewriting logic. For rapid-prototyping, CO-Nets behaviour is expressed into rewriting logic, and hence rapid-prototypes can be directly generated using rewriting techniques.

The structure of the remaining sections is as follows: in section 2, the main concepts of the CO-Nets approach are described but without entering into technical details (a more formal presentation can be found in [AS99]). In section 3, we explain the basic features of the Mondel language using the vending machine example. In the main section, we illustrate by the vending machine example the main ideas of translating Mondel into CO-Nets, and we consolidate each idea with the corresponding general translation schema. Section 5 closes the paper by giving some concluding remarks and future work.

For the rest of the paper we assume the reader is more or less familiar with algebaric Petri nets and rewriting techniques and logic. Good references for these topics are respectively [EM85], [JR91] and [Rei91] for the algebraic setting and the algebraic Petri nets, [DJ90] and [Mes92] for rewriting techniques and logic. Throughout the paper, we use for the algebraic description of different structures an OBJ notation [GWM+92].

2 The CO-Nets Approach

As mentioned above, the CO-Nets is an OO approach for developing distributed systems. In this sense, CO-Nets aim to cover the important phases in developing such systems, particularly the formal specification and the verification/validation phases. In what follows, we present the main aspects of this approach by putting more emphasis on the modelling phase. First the ECATNets model is reviewed, second we present how structural aspects of a given system are captured using suitable OO signature, third we deal with the representation of objects and classes into the CO-Nets model. More complex systems are modeled next, using more advanced abstraction mechanisms, especially the specialization and the interaction between components.

2.1 The ECATNets model

From a syntactical viewpoint, the ECATNets are a form of algebraic Petri nets [Rei91], where places are annotated by multisets of ground algebraic terms[1], w.r.t. a user defined signature [EM85]. Input arcs are annotated by two multisets of terms: the enabling tokens denoted by IC (i.e. input condition), and the destroyed tokens (DT); the output arcs are annotated by created tokens multisets (CT). Finally, boolean expressions can be associated with transitions denoted by TC (i.e. Transition Condition). All for all, the generic form of ECATNets [BM92] is depicted in figure 1(a).

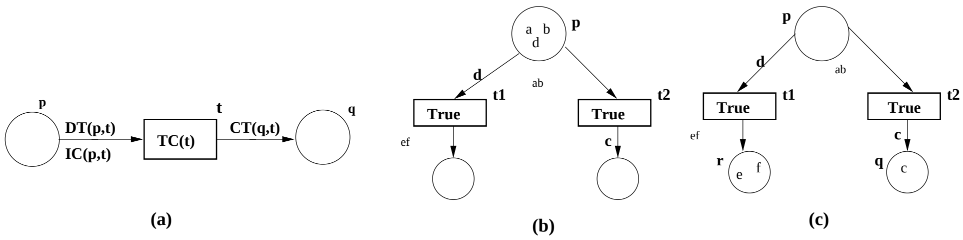


Figure 1: The Generic ECATNets and an illustative example.

Intuitively, the behaviour of a given ECATNet can be formulated as follows: For firing a transition t, $IC(p,t)$ must be included into $M(p)$ (for all input places p) and the condition $TC(t)$ is valuated to true. After firing t, there is simultaneously deletion of $DT(p,t)$ from each input places and addition of $CT(t,q)$ to the output places of t. The ECATNets *formal semantics* is expressed in rewriting logic [Mes92], where the effect of each transition is captured by an appropriate rewrite rule. The form of such rewrite rules depends essentially on the relation between the IC and DT, where three cases have been distinguished in [BM92]. We recall the corresponding rewrite rules for these cases after giving some necessary notations:

- As described above, the content of each place is a multiset of closed terms w.r.t. a given signature. We denote by $\oplus$ the union operation on this multiset.
- To capture rewrite rules and ECATNets states (i.e. tokens distribution), we use another multisets form, whose elements are pairs of the form: (*place_name, multiset of terms*). The

[1] denoted $M(p)$ for each place p.

union operation on this multiset is denoted by $\otimes$. In order to exhibit a full concurrency a distributed law, of the operator $\otimes$ over $\oplus$, had been defined as an axiom used as 'splitting' from left to right and as 'recombination' from right to left: $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$, where mt_1 and mt_2 are multisets of terms generated by $\oplus$.

- given a transition t, let $\{p_1, .., p_n\}$ its input places and $\{q_1, .., q_m\}$ its output places.

case1: $IC(p,t) = DT(p,t)$, for each $p \in \{p_1, ..., p_n\}$. In this case the rule takes the form: $t : (p_1, IC(p_1,t)) \otimes .. \otimes (p_n, IC(p_n,t)) \Rightarrow (q_1, CT(q_1,t)) \otimes .. \otimes (q_m, CT(q_m,t)\ if\ TC(t)$.

case2: $IC(p,t) \cap DT(p,t) = \emptyset$, for each $p \in \{\ p_1, ..., p_n\}$. In this case the rule takes the form: $t : (p_1, IC(p_1,t)) \otimes (p_1, DT(p_1,t)) \otimes .. \otimes (p_n, IC(p_n,t)) \otimes (p_n, DT(p_n,t) \Rightarrow$ $(p_1, IC(p_1,t)) \otimes (q_1, CT(q_1,t)) \otimes .. \otimes (q_m, IC(q_m,t)) \otimes (q_m, CT(q_m,t)\ if\ TC(t)$.

case3: $IC(p,t) \cap DT(p,t) \neq \emptyset$, for at least one $p \in \{p_1, .., p_n\}$. It has been shown in [BM92] that this case could be brought to the two already cases.

case4: The first three cases concern the CATNet version [BM92]: For the E(xtended) CATNets, two additional notations have been introduced [BMSB93]. Both notations concern the Input Condition. In the first one IC takes the form: $IC(p,t) =^{\sim}$ *multiset of terms*. Where $^{\sim}$ is a new symbol. The enabling condition in this case means that $IC(p,t)$ has not to be included in $M(p)$. Using the second notation we may have $IC(p,t) = \emptyset$ which simply means that the associated transition may be fired only when its input places are empty. The associated rewrite rules are detailed in [BMSB93].

Example: For the net depicted in figure1(b), by applying the first case, we obtain the following rules for the transitions $t1$ and $t2$. $t1 : (p,d) \rightarrow (r, e \oplus f)$ $t2 : (p, a \oplus b) \rightarrow (q,c)$. The intial state, represented in figure $1(b)$, is equivalent to: $(p, a \oplus b \oplus d) \otimes (q, \emptyset) \otimes (r, \emptyset)$.
Deduction of the next state depicted in figure 1(c) may be obtained by the following rewrite sequent: $(p, a \oplus b \oplus d) \otimes (q, \emptyset) \otimes (r, \emptyset) \Rightarrow (p, a \oplus b) \otimes (p,d) \otimes (q, \emptyset) \otimes (r, \emptyset)$ "By applying the decomposition rules [BM92]" $\Rightarrow (q,c) \otimes (r, e \oplus f) \otimes (q, \emptyset) \otimes (r, \emptyset)$ "By applying concurrently the rewrite rules associated with $t1$ and $t2$" $\Rightarrow (q,c) \otimes (r, e \oplus f) \otimes (p, \emptyset)$ "by applying concurrently the commutativity and the structural axioms of identity [BM92]". □

2.2 The CO-Nets: Structural Aspects

The first intuitive basic ideas for representing an OO system (i.e. a community of objects) into the ECATNets framework is to regard object states and message instances (i.e. method invocation) as algebraic terms— with adding logical identities to objects states and ensuring their uniqueness. Second, represent such message and state terms into associated places and their effect— as a result of interaction of messages with the objects to which they are sent— by corresponding transitions.

However, for the modelling of more complex systems as interacting components, we need further powerful mechanisms that allow for capturing OO contructions like: specialization, object composition and particularly the interaction between differents classes. For this aim, on the one hand, we propose to make clear distinction between local properties (i.e. objects attributes) that have to be hidden from the outside of the object and external ones that can be observed (and may be modified) by other objects classes. On the other hand, we propose as well to distinguish between the internal messages that allow for evolving the object states over the time and the external messages used for interacting different classes.

More precisely, each object state will be regarded as a term —with respect to the OO signature below— of the form $< I|atr_1 : val_1, ..., atr_k : val_k, atbs_1 : val'_1, ..., atbs_s : val'_s >$[2]; where I is an observed object identity taking its values from an appropriate abstract data type typed by OId; $atr_1, ..atr_k$ are the *local* attributes identifiers component that have as actual values respectively $val_1, .., val_k$. The observed part of an object state is identified by $atbs_1, .., atbs_s$ with their

[2]This structure is inspired by the Maude language [Mes93] but with an explicit distinction between hidden and observed —from the outside— attributes.

associated actual values $val'_1, .., val'_s$. Also, we assume that all the attributes identifiers(local or observed) range their value over a suitable sort denoted AID, and their associated values are ranged over the sort $Value$ with $OId < Value$ (i.e. OId as subsort of $Value$) to allow object valued attributes. Also, in order to have more flexibility of this object state and allow by the way to exhibit intra-object concurrency, we introduce an appropriate operator that we denote by $\oplus$ for splitting (resp. recombining), if need be, this state into part (resp. its part). This splitting/recombining operation, reflected by the axiom in the object-state signature as depicted below, is particularly important for the respect of the encapsulation property when interacting different classes.

Messages are viewed as operations with at least one of their arguments is of OId sort — in this sense a message should involve at least one object as sender or receiver. Each message generator ms_i will be typed by a sort denoted Ms_i. Moreover, as mentioned above, we distinguish between local and external messages.

All for all, hereafter the formal description of the object states as well as the classes structure using an OBJ notation:

obj Object-State **is**
sort AId .
subsort OId < Value , Attribute < Attributes , Id-Attributes < Object, Local-attributes < Attributes, External-attributes< Id-Attributes.
protecting Value OId AId .
op $\_ : \_$: AId Value $\rightarrow$ Attribute .
op $\_,\_$: Attribute Attributes $\rightarrow$ Attributes [associ. commu. Id:nil]
op $< \_|\_ >$: OId Attributes $\rightarrow$ Id-Attributes.
op $\_ \oplus \_$: Id-Attributes Id-Attributes $\rightarrow$ Id-Attributes [associ. commu. Id:nil] [3]
vars Attr: Attribute ; Attrs_1, Attrs_2: Attributes , I:OId.
ax $< I|attrs_1 > \oplus < I|attrs_2 > = < I|attrs_1, attrs_2 >$.
endo.

obj Class-Structure **is**
protecting Object-state, s-atr$_1$,...,s-atr$_n$, s-arg$_{11}$,.., s-arg$_{1k}$,...,s-arg$_{m1}$,...,s-arg$_{ml}$.
sort Id.Cl, Cl, Mes$_{l1}$, . . . ,Mes$_{ip}$
Local attributes
op $< \_|atr_1 : \_, ..., atr_n : \_ >$: Id.Cl s-atr$_1$ . . . s-atr$_n$ $\rightarrow$ Local-Attributes.
External attributes
op $< \_|atbs_1 :, ..., atbs_k : \_ >$: Id.Cl s-atbs$_1$ . . . s-atbs$_k$ $\rightarrow$ External-Attributes.
Local messages
op ms$_{l1}$: s-arg$_{l1,1}$ . . . s-arg$_{l1,l1}$ $\rightarrow$ Mes$_{l1}$.
. . . .
op ms$_{lk}$: s-arg$_{lk,1}$ . . . s-arg$_{lk,lk}$ $\rightarrow$ Mes$_{lk}$.
Export Messages
op ms$_{e1}$: s-arg$_{e1,1}$ . . . s-arg$_{e1,e1}$ $\rightarrow$ Mes$_{e1}$.
. . . .
op ms$_{er}$: s-arg$_{er,1}$ . . . s-arg$_{er,er}$ $\rightarrow$ Mes$_{er}$.
Import Messages
op ms$_{i1}$: s-arg$_{i1,1}$ . . . s-arg$_{i1,i1}$ $\rightarrow$ Mes$_{ip}$.
. . . .
op ms$_{ip}$: s-arg$_{ip,1}$ . . . s-arg$_{ip,ip}$ $\rightarrow$ Mes$_{ip}$.

[3] This important axiom allow for splitting and recombing the object state out of necessity.

vars Attr: Attribute ; Attrs-L : Local-Attributes, Attrs-E : External-Attributes, I: Id.Cl.
endo.

Comments:

- As explained above, we have three kind of messages: those that are local to the class; those declared as imported and then used as services provided by other classes and finally those declared as exported and are services produced by the concerned class and used by other classes.
- The state of a given object is splitted into local and then encapsulated part and an observed (and possibly modifiable by other classes) part.
- The local messages to a given class Cl have to include at least the two usual messages: the message destinated for creating a new object state and the message for the deletion of existing object that we denote respectively by Ad_{Cl} and Dl_{Cl}.

On the basis of the class description, we define informally the associated CO-Net structure as follows:

- The places of the CO-Net are precisely defined by associating with each message generator one place that we called message place. Therefore, each message place have to contain message instances, of a specific form, sent to the objects (and not yet performed). In addition to these message places, we associate with the object sort one (object) place that have to contain the current object states of this class.
- CO-Net transitions reflect the effect of messages on the object states to which they are sent. Also, we make distinction between local transitions that reflect the object states evolution and external transitions modeling the interaction between different classes. The conditions to be fullfilled for each kind of these transitions forms is given the subsection below.
- Conditions may be associated with transitions. They involve attribute and/or message parameters variables.

2.3 CO-Nets : Semantical Aspects

After giving how CO-Nets templates, as decription of classes, are constructed, we focus herein on the behavioural aspects of such classes. That is, how to construct a *coherent* object society as a community of object states and message instances, and how such a society evolves only into a *permissible* society. By coherence we mean the respecting of the system structure and the uniqueness of objects identities. And by permission we mainly understand the respecting of the encapsulation property.

2.3.1 Objects creation and deletion

For ensurung the uniqueness of objects identities in a given class that we denote by Cl, we propose the following conceptualization:

1. Add to the associate (marked) CO-Net modeling such class, a new place of sort $Id.obj$ and denoted by $Id.Cl$ containing *actual objects identifiers* of objects of the place obj.
2. Objects creation is made through the net depicted in the left hand side of figure 2. Where the notation $\sim$ captures exactly the intended behaviour (i.e. the identifier Id should not already be in the place $Id.Cl$). After firing this transition, there is an addition of this new identifier to the place $Id.Cl$ and a creation of a new object, $< Id|atr_1 : in_1, ..., atr_k : in_k >$, with $in_1, ..., in_k$ as optional initial attributes values.

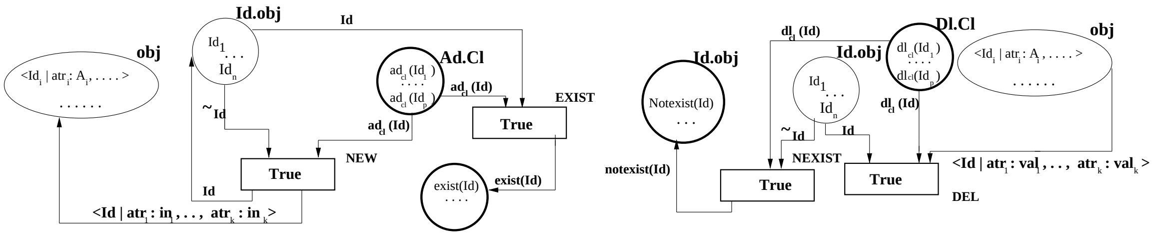


Figure 2: Objects Creation and Deletion Using OB-ECATNets

This objects creation model is very general, and can be adapted to specific cases. For example, when restriction conditions should be associated with initial values, the signature of the creation message can be modified to: $ad_{Cl}\ :\ Id.obj\ s\_atr_1\ ..\ s\_atr_k\ \rightarrow\ ad_{Cl}$ which allow to associate conditions with the transition NEW.

As depicted in the right hand of figure 2, the objects deletion is modelled following the same reasoning.

2.3.2 Evolution of Object States in Classes

For the evolution of object states in a given class, we propose a general pattern that have to be respected in order to ensure the encapsulation property—in the sense that no object states of other classes participate in the communication — as well as the preservation of the object identity uniqueness. Following such guidelines and in order to exhibit a maximal concurrency, this evolution schema is depicted in Figure 3, and it can be intuitively explained as follows: The contact of the only relevant parts[4] of some object states of a given Cl, —namely $< I_1|attrs_1 >$[5] ;..; $< I_k|attrs_k >$— with some messages $ms_{i1},..,ms_{ip},ms_{j1},..,ms_{jq}$—declared as *local or imported* in this class— and under some conditions on the invoked attributes and message parameters results results in the following effects:

- The messages $ms_{i1},..,ms_{ip},ms_{j1},..,ms_{iq}$ vanish;
- The state change of some (parts of) object states participating in the communication, namely $I_{s1},..,I_{st}$. Such change is symbolized by $attrs'_{s1},..,attrs'_{st}$ instead of $attrs_{s1},..,attrs_{st}$.
- Deletion of the some objects by sending explicitly delete messages for such objects.
- New messages are sent to objects of the class Cl , namely $ms'_{h1},..,ms'_{hr},ms'_{j1},..,ms'_{jq}$.

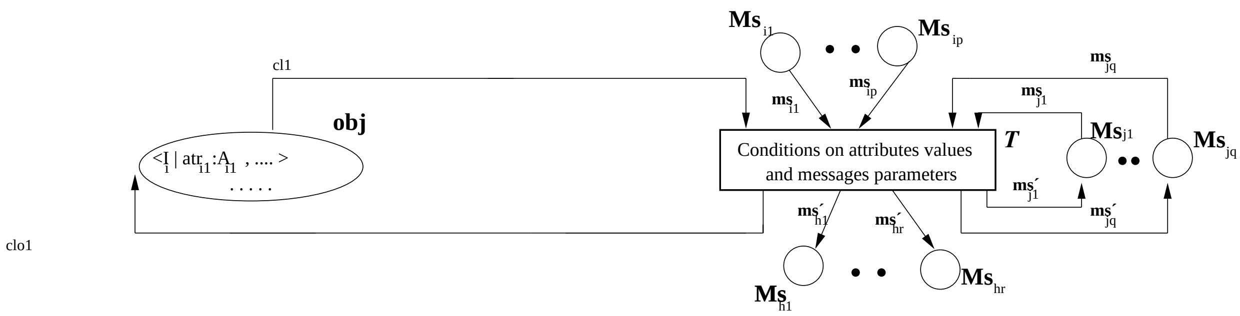


Figure 3: The Intra-Class CO-Nets Interaction Model.

[4]Such selection is possible due to the 'spliting/combination' axiom that have to be performed in front of each evolution depending on the invoked rewrite rule.

[5]$attrs_i$ is simplified notation of $atr_{i1} : val_{i1},..,atr_{ik} : val_{ik}$.

2.3.3 Rewriting rules gouverning the CO-Nets behaviour

In the same spirit of the ECATNets behaviour, each CO-Net transition is captured by an appropriate rewriting rule interpreted into rewrite logic. Following the communication pattern in figure 3, the general form of the rewrite rule associated with this intra-class interaction model takes the following form:

$$\begin{gathered}T : (Ms_{i1}, ms_{i1}) \otimes .. \otimes (Ms_{ip}, ms_{ip}) \otimes (Ms_{j1}, ms_{j1}) \otimes .. \otimes (Ms_{jq}, ms_{jq}) \otimes \\ (obj, < I_1|attrs_1 >) \otimes ... \otimes (obj, < I_k|attrs_k >) \Rightarrow (Ms_{h1}, ms'_{h1}) \otimes ... \otimes (Ms'_{hr}, ms'_{hr}) \otimes \\ (Ms_{j1}, ms'_{j1}) \otimes .. \otimes (Ms'_{jq}, ms'_{jq}) \otimes (obj, < I_{s1}|attrs'_{s1} >) \otimes ... \otimes (obj, < I_{st}|attrs'_{st} >) \\ if\ Conditions \text{ and } M(Ad_{Cl}) = \emptyset \text{ and } M(Dl_{Cl}) = \emptyset\ ^{6}\end{gathered}$$

2.4 CO-Nets : More Advanced Constructions

So far, we have presented only how the CO-Nets approach allows for conceiving independent classes. In what follows section, we give how more complex systems can be constructed using advanced abstraction mechanisms like the inheritance and the interaction between classes.

2.4.1 Inheritance in CO-Nets

For space limitation, we restrict ourselve only to the simple case of inheritance without overriding. Giving a (super)class modelled as a CO-Net, for constructing a subclass—with additional attributes and behaviour— that inherit the structure as well as the behaviour of its superclass and that it may exhibit new behaviour involving the additional attributes, we propose the following modelization. First, for allowing the inheritance of the structure and the behaviour of the superclass, we propose, on the one hand, to gather the object states of both classes (i.e. the super as wellas the subclass objects) into the same object place (of the superclass). On the other hand, we add an *attribute variable* (i.e. variable of sort *Attributes* as declared in the object-state signature) to all arcs going to or from this place. Finally, the proper behaviour of the subclass is constructed in the usual way (i.e. by adding new places for each message and constructing the associated transitions reflecting the new behaviour that should take into consideration the additional attributes).

In a more clear way, if the IC, DT and CT associated with the arcs related to the object place contain terms of the form $< Id|atr_1 : V_1, ..., atr_k : V_k >$, we propose to transform them into $< Id|atr_1 : V_1, ..., atr_k : val_k, ATS >$ where ATS can be instantiated either by *nil* whenever the objects of the superclass are involved by the associated transition or by the additional attributes(i.e. by $at\_ad_1 : Var_1, .., at\_ad_q : Var_q$)[7] whenever the subclass objects are involved. Noting that, $Var_1, .., Var_q$ should be declared as variables of *Attributes* sorts (see, the object-state signature). Semantically, each instantiation of the ATS variable is captured by a corresponding rewriting rule. Therefore, *two rewrite rules* are associated with each transition of the superclass.

2.4.2 Interaction in CO-Nets

Our conceptualization of the inter-class communication is inspired by the general communication schema proposed by J.Meseguer for concurrent OO systems [Mes93], but with taking into account the fact that intra-class objects evolution in each class is ensured by the model proposed in Figure 3. Therefore, as depicted in figure 4, the inter-class communication that we propose can be characterized as follows: The contact of some *imported or exported* messages namely

[6] This condition requires that the creation and the deletion of objects have to performed at first. In other words, before performing this rewrite rules the marking in the Ad_{Cl} as well of the Dl_{Cl} places have to be empty.

[7] Where $at\_ad_q, ..., at\_ad_q$ are assumed to be the additional attributes of the subclass.

$ms_{i1},..,ms_{ip},ms_{j1},..,ms_{jq}$, that may belong to different classes denoted by $Cl_1,...,Cl_m$, with exclusively the *external* parts of some objects states included in such classes, namely $< I_1|attrs_1 >$,..., $< I_k|attrs_k >$. Such contact, under some conditions that may involve objects attribute values and parameter messages, results in the following:

- The messages $ms_{i1},..,ms_{ip},ms_{j1},..,ms_{iq}$ vanish;
- The state change of some (external parts of) object states participating in the communication, namely $I_{s1},..,I_{st}$. Such change is symbolized by $attrs'_{s1},..,attrs'_{st}$ instead of $attrs_{s1},..,attrs_{st}$. The other objects components remain inchanged (i.e. there is no delection of parts of objects states).
- new external messages (that may involve delection/creaction ones) are sent to objects of different classes, namely $ms'_{h1},..,ms'_{hr},ms'_{j1},..,ms'_{jq}$.

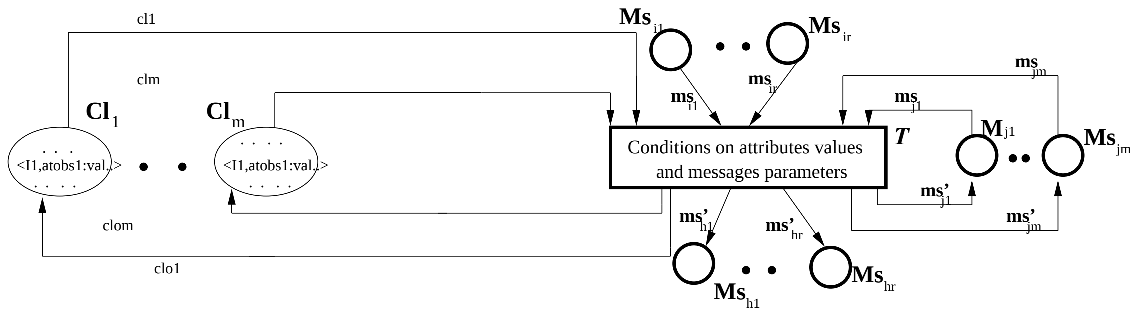


Figure 4: The Interaction General Model Between classes

3 Mondel Overview

The Mondel language has been designed particularly for modelling and specifying distributed applications into an object oriented setting. The main features of Mondel specification include: object persistence, multiple inheritance, rendez-vous communication between objects. An object, in Mondel, is an instance of a type (i.e., a class in other OO languages) that specifies the properties that characterize all its instances and the operations that can be accepted by the object following their specific behaviour. More precisely, each object type is described through three sections:

The Attributes Section: that may be empty, describes the attributes identifiers and their types (i.e., their sortes) that are often object-valued. Moreover, as mentioned above, one of the specific feature of Mondel specification is that some attributes, those modeling *internal objects states*, are implicit and their value depend on the actual procedure being executed (on the object).

The Operations Section: The definition of a given type include the declaration of operations (called methods in other OO languages) that may be invoked by objects of other types. Each operation has a fixed number of parameters(which may be of objects and/or predefined types).

The Behaviour Section: It describes, through some procedures, the execution of each operation . Noting that, the Mondel specification philosophy favours *communication* over computation. In other words, types behaviour is specified using a state-oriented style, where internal states are modeled as Mondel procedures. Such communication is made through a rendez-vous between the caller and the callee (*accept* for receiving and ! for sending), where the caller must *wait* until the callee has issued the corresponding *return* statement.

Example: Let us consider a vending machine [EDB93]which receives a coin and delivers candies to its users. The specification of the vending machine system consists of one module composed of two types: the type *Machine* and the type *User*, as described below using Mondel syntax. The relation between the types *User* and *Machine* is represented by the attribute m of type *Machine*, defined within the *User* type. The operations, *InsertCoin* and *PushAndGetCandy*, are specified within the operation clause (as shown in lines 2 and 4). Note that these operations are without parameters. The user is initially in a *Thinking* state, and when he decides to buy a candy he inserts a coin. After the coin has been accepted, the user enters the *GetCandy* state, ready to accept a coin. Once a coin is inserted, the machine accepts the coin and then it enters the *DeliverCandy* state. After the user has pushed the button of the machine, it delivers him/her a candy and it becomes *Ready* again.

```
0  unit VMsystem =
1  type Machine = object with
2  operation
3   InserCoin ;
4   PushAndGetCandy ;
5  behaviour
6   Ready
7  where
8   Procedure Ready =
9   accept InsertCoin do
10    return;
11   end;
12   DeliverCandy ;
13  endproc Ready

14  Procedure DeliverCandy =
15  accept PushAndGetCandy do
16   return;
17  end;
18   Ready
19  endproc DeliverCandy
20  endtype Machine

21  type User = object with
22  m : Machine ;
23   behaviour;
24    Thinking ;
25   where;
26   procedure Thinking =
27    m!InserCoin ;
28    GetCandy ;
29   endproc Thinking ;

30   procedure GetCandy
31    m!InserCoin ;
32    Thinking ;
33   endproc GetCandy ;
34   endtype User ;
{the vending machine system behaviour }
35   behaviour
36   define Amachine new(Machine) in
37    eval new(User(Amachine)) ;
38   end ;
39   endunit VMsystem
```

4 Translating Mondel Specifications into CO-Nets

In this section, we provide Mondel specifications with a clean semantics based on CO-Nets. For this aim, first, we present how object types inluding attributes, operations and behaviour are modeled using CO-Nets. Also, for the behaviour section, more emphasis will be done to the communication aspects; because we estimate that the modelling of the computational aspects, using the *define* statement, are more trivial. Second, we shortly comment how inheritance can be taken into account. Our translation ideas are illustrated using the vending machine example.

4.1 Translating Mondel Specifications into CO-Nets

According to the CO-Nets approach as presented in section 2, a Mondel specification may be translated into CO-Nets in straigthforward manner. More precisely, with each type, we associate an CO-Net (i.e. a class) composed of:

- **One object states place:** containing object states having as attributes those declared explicitly in the attribute section plus *the implicit ones* modeling the states of the object, and which are usually initialized in the beginning of the behaviour section.

Example For the machine type, we have no explicit attributes. However, each machine is characterized by its state (shortly, *st*) initialized by the value *Ready* that can become *DeliverCandy*. Then, object instances of type machine, modeled by a corresponding *place*, should be declared as follows:

op $< \_|st : \_ >: Id.machime\ state \rightarrow Machine.$

Where $Id.machine$ stands for Machine identities sort, and *state* have two values {*Rd, Dv*} (i.e. *Rd* stands for *Ready* and *Dv* for *DeliverCandy*). The same reasoning may be applied to the *User* type, where the machine m is an explicit attribute and the *User* state (shortly, *su*), that can be in *Thinking*(shortly *Th*) or *GetCandy* (shortly, *Gt*), is an internal state.

- **Operations places:** For each declared operation, we associate a corresponding place. If an operation is delared as $opr(arg_1, .., arg_n)$, where $arg_1, ..., arg_n$ are types parameters, we associate to it the following OBJ declaration:

  op *opr* : $arg_1\ \ ...\ \ arg_n\ Id.callee\ Id.caller \rightarrow Msg_i$

  The addition of the identifier sorts of the caller and the callee is necessary. In fact, it allows us to know, when a message is accepted by an object (identified by *Id.callee*), which object is concerned by the associated *return* (i.e. which object receive the permission to continue its execution here the *Id.callee*).

  According to this, we should also add to each object type that send messages (i.e. containing the primitive '!') a place containing *return* messages declared as:

  op *ret* : $Id.callee \rightarrow Msg_i$

  Example: Following the above translation ideas, the CO-Net associated with the vending machine is decribed below. For instance, the two machine operations, modelled by two corresponding places, are declared as follows. The *InserCoint* operation is declared as:

  op *ins* : $Id.machine\ Id.User \rightarrow Msg_1.$

  Hence, *ins(u,m)*, means that the *User u* inserts the button of the machine m. The *Pushand-DeliverCandy* operation is declared as:

  op *push* : $Id.machine\ Id.User \rightarrow Msg_2.$

  For the *User* type, we have no explicit operation and then no operations places. However, it contains (two occurrences of) a sent primitive, and therefore we have to conceive a 'return' place that store messages instance of the form:

  op *ret* : $Id.User \rightarrow Msg_3.$

- **A Transition is associated with each procedure:** According to the two CO-Nets communications patterns, each procedure describing state change of an object can be easily captured by a transition. Particularly, the rendez-vous communication is modeled as follows:

  - An object changes its state by accepting some operation. Following the accept general form [BB91]: *accept OpName[***Provided** *PureExpr* **do** *stat* **end**. This behaviour is captured by a suitable transition that express the contact of *OpName* with the associated *object state*—both modeled as tokens into their associated places— under the (transition) condition *PureExpr*. The result of firing such transition (i.e. the created tokens) is described by the statement *stat*.

Example: If we take, for example, the ready procedure from the machine type:

```
Procedure Ready =
 accept InsertCoin do
   return;
  end;
  DeliverCandy ;
 endproc Ready
```

This procedure is captured by the transition **RD**, as depicted in the associated net below, that takes as input places the **ins**(sertCoin) and the **Mc**(i.e., machine) places and as output ones the **Mc** and the **Ret**(i.e. return) places. More precisely, the contact of *ins(u,m)* with $< m|st : Rd >$ results in the change of the machine state to $< m|st : Dv >$ and a sent of a knowledge to the *User u* that the message is accepted, expressed by *ret(u)*.

– The same reasoning may be applied to the statment of the form: *PureExpr ! OpName[PureExprList)]*. More explicitly, here the caller *Id.caller* send the message *OpName(Id.Callee, PureExpr, PureExprList)* to the object place associated with *OpName*. This fact is captured by a transition that takes as input place the caller object place and as output place *OpName*.

Example: The procedure Thinking described in the User type as:

```
procedure Thinking =
 m!InserCoin ;
 GetCandy ;
endproc Thinking ;
```

is captured by the transition **Th**, having as input place the **User** place which send the message *ins(u,m)* to the machine. But according to the *rendez-vous communication principle*, the **User** must *wait* until the acceptance of such a message (i.e. the firing of the transition **Ac-Is** : accept-insert).

In order to make this waiting state *explicit*, we propose to add to the *User* states two other values, namely $wt1$ (i.e. wait1) and $wt2$ (wait2). Where, $wt1$ expresses that the *User*, after he send a $ins(u, m)$ message, is in a waiting state before he enters the *GetCandy*(Gc) state (i.e. before receiving the associated *return* expressed by the firing of the transition **Ac-Is**). The $wt2$ expresses the waiting to enter the *Thinking(Th)* state as modeled by the transition **Ac-Ps**(short for accept-push) (i.e. waiting for the acceptation of the message *push(u,m)*).

Note that these intuitive suggestions should be taken into account in each similar case. In other words, for each send primitive occurrence (i.e. **!** operation) one should associate a waiting value. We can also, as done in [BB91], modelize all wait occurrence as one and associate with *each return* a corresponding place. However, in this case the number of places may become unmanageable.

All for all, the structural as well as the behavioural (i.e. the associated CO-Nets) aspects of the vending machine are described as follows:

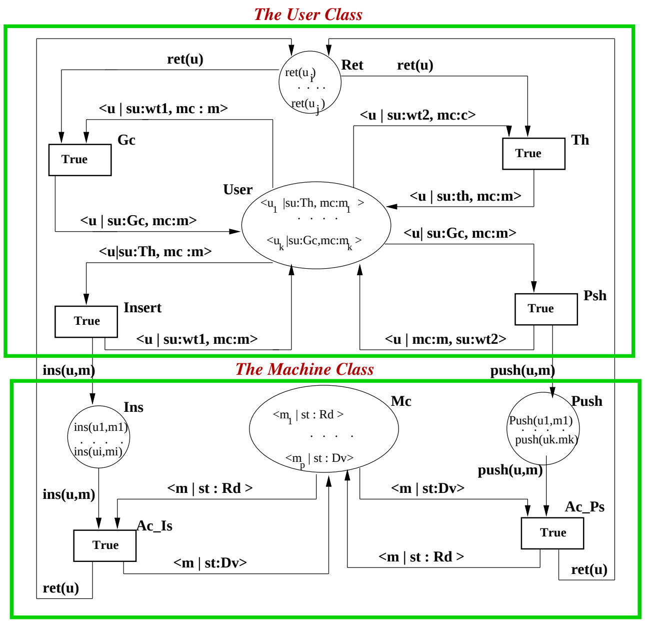
- The machine and user states Specification:

obj MU-state **is**
 protecting string .
 sort st-user, st-mc, Id.machine, Id.User .
 subsort Id.machine, Id.User $<$ String
 op Rd, Dv : $\rightarrow$ St-mc .
 op Th, Gc, wt1, wt2 : $\rightarrow$ St-user .
endo

- Specification of the Machine and User states

obj Machine **is** .
 protecting MU-state .
 sort Machine Msg1 Msg2 .
 op $<\_|st : \_>$ Id.machine St-mc $\rightarrow$ Machine .
 Imported Messages .
 op ins : Id.User Id.machine $\rightarrow$ Msg1 .
 op push : Id.User Id.machine $\rightarrow$ Msg2 .
 vars m:Id.machine
endo

obj User **is**
 protecting MU-state .
 sort User return .
 op $<\_|su : \_, mc : \_>$ St-user Id.machine $\rightarrow$ User.
 Imported Messages .
 op ret : Id.User $\rightarrow$ return .
 vars u:Id.user
endo



4.2 Translating Mondel subclasses into CO-Nets

Hereafter we explain how the proposed CO-Net inheritance allows for capturing the Mondel subtyping-based inheritance. For this aim, we propose to extend our running vending machine example. We assume that, beside the already specified machine, we may have a particular machine that before becoming ready it should be initialized. Moreover, we suppose that such intialization (operation) is possible only if the corresponding buttom is one-line. For capturing this informal description, we propose to add to the machine type new attribute named signal(shortly, *sg*), that have to be initialized by the value one(1), and a new state denoted by *init*. On the other hand, we assume that the user may send a message to initialize (shortly, *initialize*) such (special) machine. More precisely, this special vending machine can specified with Mondel as follows:

```
unit SVMsystem =
type SpMachine = Machine with
 sg : VAR[integer]
  operation ;
  initialize ;
newbehaviour
 init
where
 Procedure init =
  accept initialize do
    if sg = 1 then
    Ready
    endproc init
  endtype Machine

type User = object with
m : Machine ;
 newbehaviour;
  initial ;
 where;
 procedure initial =
  m!Initialize ;
  Thinking ;
 endproc initial ;
 endunit SpVMsystem
```

Following the CO-Nets inheritance principles, this vending machine extended behaviour can be easily captured as depicted in figure 5. Where the **Machine** place have to include now the usual machine states as well as the special machine states. The behaviour of the usual machine remains inchanged except, as required by the inheritance principle, the introduction of the ATS variable(that can take nil or $sg : Var$ as value) in the arcs related to the **Mc** place. For capturing the new behaviour in the machine subclass, a new place associated with the message *initialize* is constructed with its associated transition. Also, a new transition in the **User** class is added that reflect the sent of an intialization.

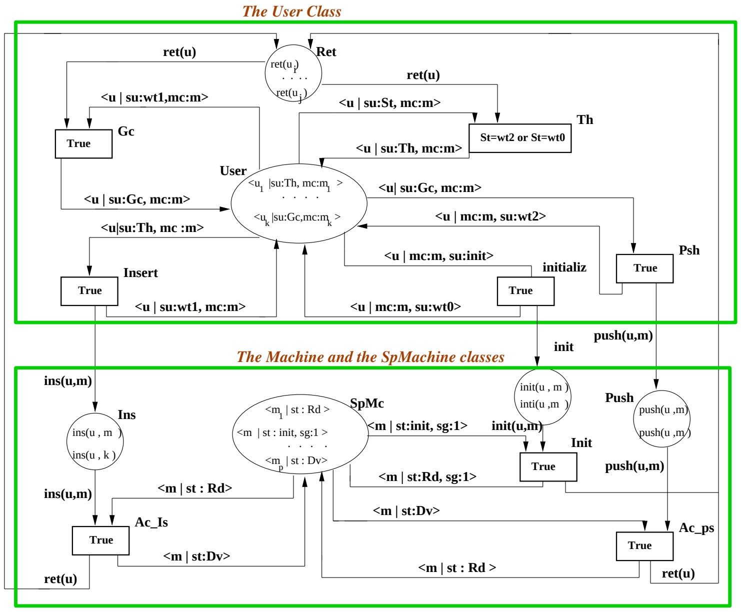


Figure 5: Mondeling the SpVMachine Using CO-Nets

4.3 Rapid-Prototyping Mondel Specification using CO-Nets

As pointed out in the introduction, for proving crucial properties of a given specification using the CO-Nets approach rapid-prototypes can be generated using rewriting techniques and specifically the current implementation of the Maude language [MOM96]. Here we restrict ourselves to present the associated rewrite rules that reflect the CO-Nets behaviour associated with the vending machine system.

- **The User Class**
  - **Insert:** $(User, < u|su : Th, mc : m >) \Rightarrow (User, < u|su : wt1, mc : m >) \otimes (Ins, ins(u, m))$
  - **Psh:** $(User, < u|su : Gc, mc : m >) \Rightarrow (User, < u|su : wt2, mc : m >) \otimes (Push, push(u, m))$
  - **Gc:** $(Ret, ret(u)) \otimes (User, < u|su : wt1, mc : m >) \Rightarrow (User, < u|su : Gc, mc : m >)$
  - **Th:** $(Ret, ret(u)) \otimes (User, < u|su : wt2, mc : m >) \Rightarrow (User, < u|su : Th, mc : m >)$

- **The Machine Class**
  - **Ac-Is:** $(Ins, ins(u, m)) \otimes (Mc, < m|St : Rd >) \Rightarrow (Mc, < m|st : Dv >) \otimes (Ret, ret(u)$
  - **Ac-Ps:** $(Push, push(u, m)) \otimes (Mc, < m|st : Dv >) \Rightarrow (Mc, < m|st : Rd >) \otimes (Ret, ret(u)$

For the special Machine depicted in figure 5, the associated rewrite rules can be splitted in two classes: The rewrite rules that allows the inheritance of the behaviour of the machine and the rules that specify the new behaviour. The inherited part is captured by the following two rewrite rules

- **Ac-Is:** $(Ins, ins(u, m)) \otimes (Mc, < m|St : Rd, sg : V >)$
  $\Rightarrow (Mc, < m|st : Dv, sg : V >) \otimes (Ret, ret(u))$[8]

- **Ac-Ps:** $(Push, push(u, m)) \otimes (Mc, < m|st : Dv, sg : V >)$
  $\Rightarrow (Mc, < m|st : Rd, sg : V >) \otimes (Ret, ret(u))$

5 Conclusion

In this paper, we proposed in a more intuitive way the CO-Nets approach as a formal model for concurrent OO systems. The approach is based on a complete integration of OO concepts and constructions into the ECATNets model, which is a form of high level Petri nets combining the strengths of Petri nets with those of abstract data type. Some key advantages of CO-Nets include: the modelling of simple and multiple inheritance in a straightforward way; the characterization of two communication patterns for intra-class as well as inter-class evolution promoting intra- and inter-objects concurrency and preserving the encapsulation property; the interpretation of the model into rewriting logic which allows the generation of rapid-prototypes using rewriting techniques and particularly the Maude language [Mes93].

As a significant case study for the assessment of the adequacy of the proposed model, for specifying and validating distributed systems in an object-oriented setting, we have shown how Mondel specifications can be easily and naturally translated into the CO-Nets framework. Moreover, due to the CO-Nets semantics, some of the Mondel properties can be verified either through graphical animation or by symbolic deduction using rewriting techniques. However, on the one hand, we have to investigate in a more detail which properties are particularly verifiable using these techniques. On the other hand, we plan to adapt the analysis techniques developed for coloured Petri nets and its object-oriented extensions developed in [Lak95].

Besides that, we are working for tailoring the CO-Net approach for the specification and validation of distributed information systems that present more difficulties at the structural level, which necessitate more advanced structuring mechanims like the aggregation and the notion of role, as well as at the behaviour level, where notion of events composition have to be taken into account.

[8] Here the ATS attribute variable is instantiated by $sg : V$, where V is a variable that can take any variable of sg.

References

[AS99] N. Aoumeur and G. Saake. CO-Nets : A Formal OO Framework for Modelling and Validating Distributed (Information) Systems. Technical report, university of Magdeburg, 1999.

[BB91] M. Barbeau and G.v. Bochmann. Formal Verification of Mondel Specification Using a Coloured Petri Nets Techniques. Technical report, publication n 784, Département d'informatique et de Recherche Opérationnelle, Université de Montréal, 1991.

[BBE+91] G.v. Bochmann, M. Barbeau, M. Erradi, M. Lecome, P. Mondain-Mondel, and N. Williams. Mondel : An Object Oriented Specification Language. Publication N 748, Département d'informatique et de Recherche Opérationnelle, Université de Montréal, 1991.

[BM92] M. Bettaz and M. Maouche. How to Specify Non Determinism and True Concurrency with Algebraic Term Nets. In M. Bidoit, and C. Choppy, editor, *Proc. of 8th Workshop on Abstract Data Types*, volume 655 of *Lecture Notes in Computer Science*, pages 164–180, 1992.

[BMSB93] M. Bettaz, M. Maouche, M. Soualmi, and S. Boukebeche. Protocol Specification using ECATNets. *Reséaux et Informatique Répartie*, 3(1):7–35, 1993.

[Boo91] G. Booch, editor. *Object Oriented Design With Applications*. The Banjamin/Cummings Publishing Compagny, inc., 1991.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 6, pages 167–198. Kluwer Academic Publishers, Boston, 1998.

[EDB93] M. Erradi, R. Dssouli, and G.v. Bochmann. A Framework for Dynamic Evolution of Distributed Systems Specifications. *Reséaux et Informatique Répartie(Networking and Distributed Computing)*, 3(1), 1993.

[EM85] H. Ehrig and B. Mahr. Fundamentals of algebraic specifications 1 : Equations and initial semantics. *EATCS Monographs on Theoretical Computer Science*, 21, 1985.

[GD93] J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In *Proc. of 10th Workshop on Abstract Data types*, 1993.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.

[Jen92] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and practical Use - Volume 1 : Basic Concepts. *EATCS Monographs in Computer Science*, 26, 1992.

[JR91] K. Jensen and G. Rozenberg. In *High-level Petri Nets*. Springer-Verlag, 1991.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[Lak95] C. Lakos. From Coloured Petri Nets to Object Petri nets. In *Proc. of 16th Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 278–287. Springer-Verlag, 1995.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[MOM96] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. of First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 189–224, 1996.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[Rei91] W. Reisig. Petri Nets and Abstract Data Types. *Theoretical Computer Science*, 80:1–30, 1991.