Structured Algebraic Nets with Object-Orientation

Olivier Biberstein

CUI, University of Geneva
CH-1211 Genève 4, Switzerland
biberste@cui.unige.ch

Didier Buchs

LGL-DI, Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
buchs@di.epfl.ch

Abstract

The aim of this paper is to briefly introduce the Concurrent Object-Oriented Petri Net specification language CO-OPN/2 which has overcome some limitations of its object-based ancestor CO-OPN. Classes are described by means of algebraic nets in which places play the role of attributes, and methods function as external parameterized transitions. Inheritance and sub-typing are distinguished, with the former is considered as a mechanism for reuse while the later is viewed on a semantic level, based on the substitutability principle with respect to the behavior of the objects. As in the CO-OPN language, cooperation between the objects is achieved by means of synchronization expressions which allow the designer to select object interaction policies.

1 Introduction

In recent years several formalisms based on P/T-nets with structuring capabilities have been developed, and many of these have adopted the object-orientation paradigm. Among these is included the CO-OPN (Concurrent Object-Oriented Petri Net) [BG91] which has been developed in the few years for the specification and design of large concurrent systems. Several examples developed with the SANDS tools (Structured Algebraic Net Development System) [BFR93], have indicated complex modeling must deal with structuring.

Recently, a new version of CO-OPN has been developed in order to enrich the earlier object-based version by including the notion of class, inheritance, along with the notion of sub-typing [BB94b]. This new version, called CO-OPN/2, is based upon two frameworks: order-sorted algebras (OSA) [GM89], for the data structure part, and the algebraic nets (AN), for the operational and concurrent aspects.

In keeping with the idea that the systems we wish to model are composed of different classes of independent entities (the objects) which concurrently collaborate to the systems' evolution, a CO-OPN/2 specification thus consists of a collection of two different kinds of modules: abstract data types (ADT) and classes. A class is viewed as a template and is used to (dynamically) instantiate the objects. The underlying formalisms of ADT and classes are the OSA and the AN. In other words, a class is described by means of an algebraic net in which the places correspond to the state variables or attributes, and the methods consist of special kinds of parameterized transitions. Moreover, both the inheritance and sub-typing hierarchies are clearly distinguished, so as to in order to avoid the typical conflicts.

Interactions between objects are accomplished by means of synchronizations. Each object may ask to be synchronized with one or several methods of another object or group of objects. Furthermore, the requiring object can choose the interaction policy by means of a synchronization expression and three operators (simultaneity, sequence, alternative).

In our approach, each object possesses an identity, also called object identifier. Semantically speaking, these object identifiers belong to a special algebra which encompasses the sub-typing hierarchy between the classes, i.e. two classes are related by sub-typing if and only if the carrier sets are related by inclusion. This characteristic corresponds to the Wegner's notion of inclusion polymorphism [Weg90].

The organization of this paper is as follows. Section 2 presents a general definition of object-orientation. Section 3 discusses the problems of the integration of concurrency in an object-oriented framework, section 4 covers different features of the CO-OPN/2 (such as class, inheritance, sub-typing and object identity), section 6 gives a summary of the semantic definitions, section 7 introduces the refinement mechanisms, section 8 presents a semantic model of sub-typing adapted to the CO-OPN/2 specifications which is able to take into account strong or weak sub-typing (even intermediate kind of sub-typing) and we finally offer some concluding remarks and a sketch of future research plans.

2 Object-Orientation

The traditional view of the object-orientation paradigm, according to Wegner's taxonomy [Weg90], involves the combination of the three notions: object, class and inheritance. This view, as Blair [Bla91] argues, seems to be overly restrictive.

Encapsulation is a fundamental dimension of object-orientation and it conveys the essential notion of the existence of objects. The principle of encapsulation consists of the concept of the separation of the interface of an object from its body. The interface describes the accessible behavior from the outside while its implementation remains concealed in the body.

Classification is another fundamental dimension which allows for the grouping of objects which share certain properties. Two important classification schemes are those of the type and of the class. In the first classification we say that objects belonging to a type, or having the same type, share the same external behavior. By external we designate what can be observed. from the exterior by means of message sending or method calls. In the second classification we consider objects of the same class as requiring the same implementation. Thus, a type is considered on a semantic level, while a class is viewed as an object template.

A third fundamental dimension of object-orientation is the ability of the objects to belong to more than one classification. The most important relationship created by classification is the hierarchy. From the point of view of class, this kind of relationship allows for reuse of implementation by means of the inheritance mechanism which is also called sub-classing, as well as for code sharing in the object-oriented programming languages. On the other hand, the type hierarchy corresponds to the substitutability principle, which is also termed sub-typing. The substitutability principle stands for the ability for an object of a type t to be substituted for another of a super-type t', without the behavior of the whole system being affected. This is written as $t \leq t'$.

It has been demonstrated in [Ame87, CHC90] that the identification of both these hierarchies creates several problems, and some object-oriented languages therefore provide a complete separation between them. If the notion of class includes the notion of type, it does not imply that the both hierarchies have been merged. Indeed, a class hierarchy may be defined by means of inheritance from the point of view of the designer, whereas a separate type hierarchy may be given. This offers the advantage of a compact description of the classes without loosing the advantage of having two different classification relationships. Unfortunately, the principle of encapsulation is usually violated whenever the inheritance mechanism is applied, in the sense that a sub-class depends on the internal representation of its super-class. A discussion of this problem can be found in [Sny86].

3 Concurrency and O-O

As the purpose of CO-OPN/2 is to capture the abstract concurrent behavior of each modeling entity, the concurrency granularity is not the objects but rather the methods. Intra-object concurrency can be obtained by expressing the causality relation between the behavior of the different methods. The relationships between objects, such as object synchronization, must also be captured by using method synchronization schemes which allow for the construction of hierarchies of *abstraction*.

If the object-orientation paradigm is well-suited with sequential programming, it seems that object-orientation cannot be easily combined with concurrent systems [Ame87]. A list of these inheritance anomalies is presented in [MY93]. In our opinion, this is because concurrent aspects are orthogonally introduced (see [Pap89] for an in-depth analysis of most of these approaches) and consequently interfere with the inheritance mechanism, which is interpreted, in turn, in an operational manner. In the specification language, it is possible to abstract these mechanisms, and to then conceal a portion of the operational nature of the inheritance mechanism. This is particularly true when the specification language is able to finely manage the concurrency, and then to eliminate the necessity to separate, for each method or group of methods, the behavioral part and the management of concurrent access.

This management is made for instance in concurrent programming languages through the definition of queue, monitors, etc. which are difficult to compose when dealing with inheritance. One way to solve this problem is to separately inherit both parts of the class definitions. An other way, adopted in CO-OPN, is to have concurrent aspects not orthogonal to the behavioral description of the methods. Consequently, the methods do not seem to have any strange behavior or inconsistencies with respect to the inheritance mechanism, because the concurrency control mechanism can be composed. The composition mechanism is performed in CO-OPN/2 through use of synchronization expressions which synchronize class extensions with their respective super-classes.

Our reflections and experiments correspond to those of other research teams which study formal methods and, who have shown that it is necessary to extend the model of concurrency, in association with object orientation, in order to model real (distributed) systems. Thus, the idea of extending concurrency models has generated many different approaches. The High Level Petri Net approach (algebraic nets [Vau87, Rei91], LOOPN++ [Lak94], the Cooperative Nets (CN) [SB94], the POT and POP's approaches [ELR90], OBJ-SA [BCM88], CLOWN [BC93], the O-O-AHL approach [Rib94]), the process logic approach (with LOTOS), the temporal logic approach [FM90] and our approach (structured algebraic nets) called CO-OPN [BG91].

4 CO-OPN/2 Language

As has been previously mentioned, the CO-OPN/2 formalism is based upon two separate frameworks, the order-sorted algebras for the data structures and the algebraic nets for the operational and concurrent aspects. These two formalisms are inadequate when large problems are confronted, and some structuring capabilities are then needed to overcome their limitations. A CO-OPN/2 specification is composed of two different kinds of entities (or modules). One copes with the notion of abstract algebraic data type, while the other with the notion of class. Each of these modules include a header (**ADT** or **Class**), a **Signature** and a **Body** part. All the components accessible from the outside are given in module's signature, while the body part includes all the local aspects such as the properties of the operations or the operational description of the objects. The module's name and the informations related to genericity or inheritance are grouped in the header section.

In order to explain the CO-OPN/2 language we will illustrate it concretely with the analogy of production and consumption in a Swiss chocolate factory. The factory produces two kinds of chocolate packages and a conveyor belt conveys the empty packagings to the packager which, in turn, fills in the empty packagings by means of two different kinds of chocolate. Figure 1 shows a natural, but partial, graphic representation of the three main devices of the factory. All the elements of this figure will be explained in more detail as the presentation develops, but we may mention that the dash arrow indicates the synchronization requests which are often also referred to as client-ship.

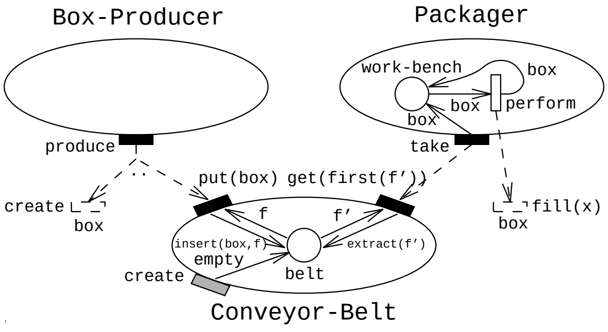

Figure 1: An outline of our Swiss chocolate factory

4.1 Abstract data types

As has been previously mentioned, data structures are described by means of OSA [GM89]. Sub-sorting, partial operations, polymorphism and overloading are notions which are well encompassed within the OSA framework. Figure 2 illustrates a generic first-in-first-out abstract data type by means of order-sorted algebras. The **Use** field lists all the required modules,

```
Generic ADT Fifo(Item);
Signature
 Use Item, Bool;
 Sorts nefifo, fifo;
 Subsorts nefifo < fifo;
 Generators
 empty : → fifo;
 insert _ _ : item fifo → nefifo;
 Operations
 first _     : nefifo → item;
 extract _   : nefifo → fifo;
Body
 Axioms
 first(insert(box,f)) = box;
 extract(insert(box,f)) = f;
 Where
 box : packaging;  f : fifo;
End Fifo;
```

Figure 2: First-In-First-Out ADT

such as the parameter module `Item` and the boolean ADT `Bool`, which are not given here. One way of using partial operations, such as `extract` and `first`, consists of partitioning the domains and associating a sort `nefifo` for the nonempty fifos and a "larger" one `fifo` for the empty or nonempty fifos. In this way, both the operations are partial across the entire domain but total on their sub-domains, as we can see in the **Axioms** field. Notice the underscore character which corresponds to the positions of the respective arguments.

4.2 Class

The class modules correspond to the notion of encapsulated entities that possess an internal state and furnish the outside with various services. The algebraic net formalism has been adopted for this second kind of module.

Two types of transitions have been provided. First of all, there are the internal transitions (usual transitions of P/T nets) which correspond to the object reaction and, secondly there are the parameterized external transitions, or methods, which are the visible events from the outside. Cooperation between the objects is accomplished by means of a synchronization mechanism, i.e. each object event may request synchronization with the methods of one or a group of partners by means of a synchronization expression. Three synchronizations operators have been provided: '//' for simultaneity, '..' for sequence and '$\oplus$' for alternative or non-determinism. Thus, a behavioral axiom is established as follows:

$$[\, Cond \Rightarrow]\, Event\, [\, \mathbf{with}\ Sync\,] :: Pre \rightarrow Post$$

in which *Cond* is a condition on the algebraic values, *Event* is an internal transition name or a method with parameters, and *Sync* is an optional synchronization expression in which the **with** keyword plays the role of a behavioral abstraction operator. *Pre* and *Post* correspond, respectively, to what is consumed and to what is produced at the different places within the net.

For instance, figure 3 shows the class of the regular packagings in our chocolate factory example and figure 4 illustrates the graphic representation of this class. The insides of the ellipses includes the encapsulated elements, the solid arrows represent the data flow, the circles indicate the places. In the case of the rectangles, the white ones correspond to the internal transitions, the dark ones indicates the methods, while the gray rectangles represent the special method **create** for coping with the creation of objects and their initialization.

Our example includes two kinds and two shapes of chocolates. The sorts `kind` and `shape`, with their

```
Class Packaging;
Signature
 Use Nat, Bool, Miscellaneaous;
 Type packaging;
 Methods
 fill _ : kind;
Body
 Places #square : {nat};
 Axioms
 x=praline ⇒ fill(x) ::
  #square{n+1} → #square{n};
 create :: → #square{nb-sqr};
 Where
 n : nat;  x : kind;
End Packaging;
```

Figure 3: The regular packaging class

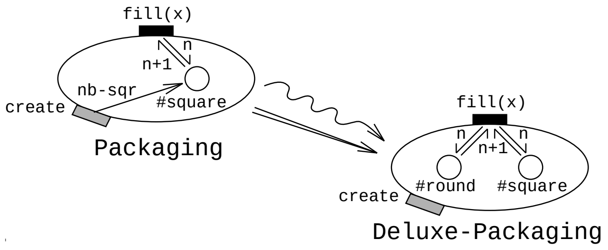

Figure 4: An outline of the packaging classes

respective generators, `square`, and `round`, and then `praline`, and `rum-truffle` as well as both the constants `nb-sqr` and `nb-rnd` are described in the `Miscellaneous` module not detailed here.

The packaging of figure 3 can only contain a certain quantity of square chocolates. This quantity is stored in the place or attribute `#square`, which is updated by the method `fill`. The method `fill` models the action of filling an empty hole with a chocolate of the correct form (`praline`s are squares and `rum-truffle`s are rounds). Its behavior is shown in the **Axioms** field. The place `#square` contains the number of empty holes in the packaging and it is decreased by 1 if and only if the method parameter `x` is a praline. The value `n+1` is consumed at the place `#square`, imposing that the transition can be fired only if n is greater than zero, while the value `n` is produced at the same place. In order to avoid dealing with partial operations we did not use the substract operator, which would have been here more natural. Notice that the multi-set constructor is denoted with the round brackets. In order to clearly distinguish between the notion of class (or module) and the notion of type, the class' type must be given explicitly in the field **Type** and the special method **create** initializes the attribute with a constant value `nb-sqr`.

4.3 Inheritance and sub-typing

Our chocolate factory deals with two kinds of packagings, regular, as detailed above, and deluxe. Deluxe packaging is very similar to regular packagings, with the exception that it is bigger and that it includes both the chocolate forms. Since the deluxe packaging has the same features as the standard boxes, the new class associated with the deluxe packagings do not have to be developed from scratch and inheritance may be used. Figure 5 shows how the `Deluxe-Packaging` class reuses the `Packaging` template and adds the new attribute `#round`, the new behavior of the method `fill`, as well as the initialization of the new attribute. Figure 4 illustrates the inheritance and sub-typing relationships by means of, respectively, the curly arrow and the solid double arrow.

```
Class Deluxe-Packaging;
Inherit Packaging;
Renaming
 packaging Into deluxe-packaging;
Signature
 Subtypes deluxe-packaging < packaging;
Body
 Places #round : {nat};
 Axioms
 fill(rum-truffle) ::
  #round{n+1} → #round{n};
 create :: → #round{nb-rnd};
 Where
 n : nat;  x : kind;
End Deluxe-Packaging;
```

Figure 5: The deluxe packaging class

We observe that the external behaviors of the method `fill` in the `Deluxe-Packaging` class which are restricted to the behaviors of the parent class are uniformly identical. This means that, in any context, an instance of the `Packaging` class can be substituted with an instance of the `Deluxe-Packaging` class. In other words the sub-type hierarchy and the inheritance hierarchy may be superposed. This is accomplished by means of the **`Subtypes`** field.

4.4 Object identity and polymorphism

Sub-typing is closely related to polymorphism which is the ability to deal with values of different types. Indeed, sub-typing generates naturally a kind of polymorphism called *inclusion polymorphism*.

Let us illustrate in our example where inclusion polymorphism appears. Imagine a situation where, few years ago, the factory did not have an automatic packaging producer and the conveyor-belt in figure 1 was simply a heap composed of the two kinds of packagings. The modeling of such a heap is shown in figure 6.

```
Class Heap;
Signature
 Use Packaging;
 Type heap;
 Methods
 put _ , get _ : packaging;
Body
 Places
 storage : {packaging};
 Axioms
 put(box) :: → storage{box};
 get(box) :: storage{box} → ;
 Where
 box : packaging;
End Heap;
```

Figure 6: The heap modeling

In our example the type `deluxe-packaging` is a sub-type of the type `packaging`. Thus the place `storage` is a multi-set of object identifiers of the type `packaging` as well as of the type `deluxe-packaging`. In this sense, we can say that the attribute `storage` is polymorphic since values of both type can be stored. Moreover, both the methods, `put` and `get`, are polymorphic because different kinds of object references, belonging to the `packaging` hierarchy, can be handled by these methods.

It is necessary to recall that each object has an identity which is also called an object identifier. These object identifiers are values of a special order-sorted algebra which encompass the sub-typing hierarchy.

4.5 Inheritance without sub-typing

Whenever the factory acquired a packaging producer, the `Heap` class was reused to specify the `Conveyor-Belt` class. It was necessary also to redefine the implementation of the methods `put` and `get` as is shown in figure 7 and 8.

We can observe that the type `conveyor-belt` is not a sub-type of the type `heap` with respect to external behaviors (having strong typing in mind). Indeed, it is obvious that the method `put` and `get` of the sub-class cannot be fired concurrently any longer, as was the case in the super-class, because of the fifo data structure which is used in the heir class. This is the typical conflict which arises between reuse and specialization. Nevertheless, both the types `heap` and `conveyor-belt` might be "compatible" with respect to certain behaviors.

```
ADT Fifo-Packaging As Fifo(Packaging)
Morphism item → packaging;
End Fifo-Packaging;

Class Conveyor-Belt;
Inherit Heap;
Renaming heap Into conveyor-belt;
Signature
 Methods
 Redefine put _ , get _ : packaging;
Body
 Use Fifo-Packaging;
 Places
 belt : {fifo-packaging}
 Axioms
 put(box) ::
  belt{f} → belt{insert(box,f)};
 get(first(f')) ::
  belt{f'} → belt{extract(f')};
 create → belt{empty};
 Where
 f:fifo-packaging; f':nefifo-packaging;
 box : packaging;
End Conveyor-Belt;
```

Figure 7: The conveyor-belt modeling

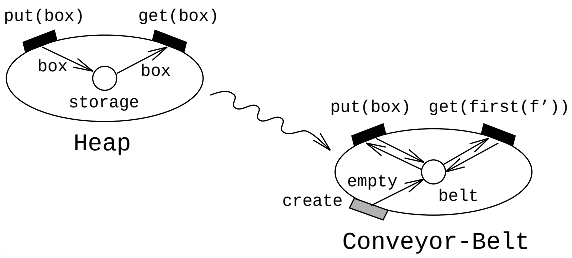

Figure 8: An outline of new conveyor belt

4.6 Object interaction

Both of the last classes of our chocolate factory concern the packaging producer and the packager which are given respectively in figures 9 and 10. An outline of these classes may be found in figure 1 on page 3, in which the dash lines represent synchronization requests.

The axiom of the `produce` method in class `Packaging-Producer` expresses the synchronization request in sequence **`with`** two methods of two distinct objects. The term `box.create` asks for the creation and the initialization of an object identified by the free variable `box` and `conveyor.put(box)` asks for the synchronization of the method `put` of an instance called `conveyor`. Hence the identity `box` of the previously created packaging is "transmitted" to the conveyor. Since the `box` variable is free, the axiom is satisfied for any value of `box` (regular packaging or deluxe packaging) and this holds for all free variables

```
Class Packaging-Producer;
Signature
 Use Packaging;
 Type packaging-producer;
 Methods produce;
Body
 Axioms
 produce with
  box.create . . conveyor.put(box);
 Where
 box : packaging;
End Packaging-Producer;
```

Figure 9: The packaging producer modeling

of the example. Note that we do not detail here how the `conveyor` instance is created.

```
Class Packager;
Signature
 Type Packager;
 Methods take;
Body
 Use Kind; Packaging, Conveyor;
 Transitions perform;
 Places
 work-bench : {packaging};
 Axioms
 take with conveyor.get(box) ::
  → work-bench{box};
 box = super(x) ⇒
  perform with box.fill(praline) ::
   work-bench{box} → work-bench{box};
 box = x ⇒
  perform with box.fill(choc) ::
   work-bench{box} → work-bench{box};
 Where
 choc : kind;
 box : packaging;  x : deluxe-packaging;
End Packager;
```

Figure 10: The Packager modeling

In the `Packager` class, two events ask for a synchronization. The method `take` is quite simple: "it removes a packaging from the conveyor and stores it on the work-bench". The `perform` internal transition is a little bit more sophisticated. In fact, the two axioms for the same event are mutually exclusive. The conditions `box=`**`super`**`(x)` and `box=x`, where `box` has the sort `packaging` and `x` has the sort `deluxe-packaging` are used for distinguishing the regular packagings from the deluxe packagings, and they take advantage of the inclusion domains of the order-sorted algebra, which encompasses the sub-typing hierarchy. The operation **`super`** returns an object identifier, which has the same super sort as its argument. Thus, the equation `box=`**`super`**`(x)` is valid if and only if the `box` identifier corresponds to a regular packaging. The contrary definition stands for the other equation. A more elaborate explanation of these aspects may be found in [BB94a].

5 CO-OPN/2 Syntax

The purpose of this section is to introduce the abstract syntax of CO-OPN/2 language.

Let us recall that an order-sorted algebraic specification is composed of a signature Σ, a set of variables V and a set of conditional positive axioms Ax. A signature Σ is a triple $\langle S, \leq_S, F\rangle$ where S denotes a set of sorts, $\leq_S \subseteq S \times S$ is a sub-sort relation and F is a set of operations coupled with their arity. Let *ASpec* be an order-sorted algebraic specification. The presentation of *ASpec* is noted $Pres(ASpec) = \langle \Sigma, V, Ax\rangle$. The semantics aspect of order-sorted algebraic specifications can be found in [GM89].

In the following discussion we consider the given universe $\mathcal{U}$ which includes the disjoint sets: *SORTS*, *FUNC*, *METHODS*, *PLACES*, *VAR*, *CLASSES* which correspond to sets of all sort, operation, method, place, variable and class names, respectively. The set *SORTS* is divided in two disjoint sets $SORTS_A$ and $SORTS_T$. The former is dedicated to the sorts of the algebraic part while the latter is related to the class' types and to the object identifier algebra A_{oid}.

A CO-OPN/2 specification corresponds to a set of classes which are described by means of algebraic nets. A class consists of three parts: a interface, a body, and an order-sorted algebraic specification. A interface corresponds to the visible part of a class. It is composed of the accessible methods and of the sorts of its parameters as well as of the class's type and the relationship of this type among the other type of the specification. A body encapsulates or conceal the state of the class instances and the implementation of their behavior. A body is composed of a set of places, a set of variables and a set of behavioral axioms which describes the behavior of the class. Finally, the data structure aspects are characterized by means of an order-sorted algebraic specification.

Definition 5.1 *Interface*
A triple $I = \langle M, t, \leq_t\rangle$ is a *interface* if

- M is an $SORTS_T \times SORTS^*$-sorted set of methods $(M_{t,w})_{t \in SORTS_T, w \in SORTS^*}$,
- $t \in SORTS_T$ also called the type of the class,
- $\leq_t \subset SORTS_T \times SORTS_T$ is a partial order. □

Definition 5.2 *Body*
A triple $B = \langle P, X, BAx\rangle$ is said a *body* if

- $P \subset PLACES$ is an *SORTS*-sorted set of place,
- $X \subset VAR$ is a *SORTS*-sorted set of variable divided into two disjoint sets X_A and X_T.
- BAx is a set of behavioral axioms (see below). □

Definition 5.3 *Class*
Let *ASpec* be an order-sorted algebraic specification. Let I and B be, respectively, a class interface and a class body. A *class* C is a triple $C = \langle ASpec, I, B\rangle$. □

Before defining what an behavioral axiom is, we have to discuss what types of events can occur. Our formalism provides two different categories of events: observable events and invisible events. On the one hand, the observable events correspond to the external stimuli of the objects; they are mainly composed of the method calls and the object synchronizations. On the other hand, the objects can react spontaneously. For example, an observable event can occur and modify the state of an object, this modification may induce a invisible and spontaneous reaction of the object. Events which describe these spontaneous reactions are called the invisible events, they correspond to the internal transitions of the formalism and are often referred by means of the τ symbol.

In the section dedicated to the semantics of the formalism we will show how these events are eliminated in order to express these spontaneous reactions. Obviously, this elimination does not prevent an object asking for a synchronization with other objects.

The set of the all events *EVENTS* is given in the next definition. Because this set is also used in the semantic part we give a generic definition. Recall that object synchronizations are expressed by means of synchronization expressions involving four operators: $\|$, $..$, $\oplus$ and **with** for concurrency, sequence, alternative and behavioral abstraction, respectively. Dot notation for methods calls has been adopted.

Definition 5.4 *Events*
The set of events over three sets A, M, X, written $EVENTS_{A,M,X}$, is generated by the following production rules:

$$\begin{array}{lcl} E & \rightarrow & x.\tau \mid x.\tau \;\mathtt{with}\; O \mid O \\ O & \rightarrow & x.m(a_1, \ldots, a_n) \mid x.\mathtt{create}(a_1, \ldots, a_n) \mid \\ & & O \,\|\, O \mid O \,..\, O \mid O \oplus O \end{array}$$

where $x \in X$, $m \in M$ and $a_1, \ldots, a_n$ are in A. □

Here we give the definition of the behavioral axioms. This definition is a little bit more general than the one given in section 4.2. $T_{SIG,X}$ denotes the set of all terms over the set of all signatures SIG and a set of variables X. A $SORTS$-P-sorted term t is a term such as $\forall s \in SORTS, \forall p \in P_s, t_p \in (T_{[\Sigma],X})_{[s]}$.

Definition 5.5 *Behavioral axiom*
Let P be a $SORTS$-sorted set of place names. A behavioral axiom is an expression

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

defined as follows for $X \subset VAR$:

- $Cond$ is an expression $t_1 = t'_1 \wedge t_2 = t'_2 \wedge \ldots \wedge t_n = t'_n$ where $t_i, t'_i \in (T_{SIG,X})_{s_i}, s_i \in SORTS$,
- $Event \in EVENTS_{T_{SIG,X}, METHODS, X_T}$
- Pre and $Post$ are $SORTS$-P-sorted terms. □

Finally, a CO-OPN/2 specification is a collection of classes. Recall that each class is contains the description of its data structures, an interface and a body.

Definition 5.6 *CO-OPN/2 specification*
Let $ASpec_i$ be a hierarchical order-sorted specification. A *CO-OPN/2 specification Spec* is a set of classes such as:

$$Spec = \{C_i \mid C_i = \langle ASpec_i, I_i, B_i\rangle, 1 \leq i \leq n\}$$

□

6 CO-OPN/2 Semantics

This section is dedicated to the semantic aspects of the CO-OPN/2 specifications which are mainly based on the order-sorted algebraic specifications and the notion of concurrent transition systems. Here we only focus our attention on the notion of concurrent systems, as the semantics aspects of order-sorted algebraic specification are presented in [GM89].

First we give the construction of the object identifier algebra. This algebra is an essential element of our formalism because it captures the sub-typing notion. In order to express concurrency we provide the multi-set extension of an algebraic specification as well as the extension of the object identifier algebra. This last extension allow us to deal with object identities. Before discussing the system of inference rules used to built the semantics of a CO-OPN/2 specification, we introduce the notion of object state as well as the notion of concurrent transition systems.

6.1 Object Identifier Algebra

As already mentioned, each object possesses an identity which is an algebraic value of an order-sorted algebra called object identifier algebra (A_{oid}). For a given CO-OPN/2 specification the associated object identifier algebra has to be built in order to reflect the sub-type relationship among the different classes of the specification. This is realized as follows: for each class type of the specification we associate a sort and the corresponding carrier sets are related by inclusion if the class types are related by sub-typing.

Since each class expresses its subtype relationship with its super-type by means of a partial order relation $\leq_t$ expressed in its interface, thus the partial order $\leq_T$ over all types has to be computed as follows:

Definition 6.1 *Partial order computation*
Let $Spec = \{C_i \mid C_i = \langle ASpec_i, I_i, B_i\rangle, 1 \leq i \leq n\}$ a CO-OPN/2 specification where $I_i = \langle M_i, t_i, \leq_{ti}\rangle$. The partial order over all types is

$$\leq_T = (\bigcup_{1 \leq i \leq n} \leq_{ti})^*$$

□

Now we explain, by means of an example, how the object identifier algebra is build for a given partial order $\leq_T$. We avoid the formal definition of this construction which can be found in [BB94b].

For example, let $\{t_1, t_2, t_3, t_4\}$ be a set of four types and $\leq_T$ be the sub-type relation over this set such that $t_2 \leq_T t_1$, $t_3 \leq_T t_1$ and $t_4 \leq_T t_3$. Then, the object identifier algebra is composed of the four corresponding carriers sets $A_{t_1}, A_{t_2}, A_{t_3}, A_{t_4}$ related as follows: $A_{t_2} \subseteq A_{t_1}, A_{t_3} \subseteq A_{t_1}$ and $A_{t_4} \subseteq A_{t_3}$. All these carrier sets are included in the carrier set of all oid's noted A_{t_0}. One can view the special type t_0 as the type of all oid's while each type of $\{t_1, \ldots, t_4\}$ corresponds to the type of a class. Moreover, the set of the actual oid's of the type t_i is noted $\overline{A}_{t_i}$ and is included in A_{t_i} $(i = 1, \ldots, 4)$. The following families of operations are coupled with these carrier sets $\overline{A}_{t_i}$ and A_{t_i} $(i = 1, \ldots, 4)$:

1) the generator $init_{t_i}$, the first value of the type t_i,
2) new_{t_i} that gives a new oid of the same type t_i,
3) sub_{t_i} that provides a new oid of the given subtype t_i,
4) $super_{t_i}$ that returns the oid of its ancestor.

In order to reflect the sub-typing relation by means of the carrier sets inclusion, we use an ordered set of prime numbers denoted by $\mathbb{P} = \{p_i \mid i \in \mathbb{N}\}$ with $p_i < p_j \Rightarrow i < j$ $(\forall i, j \in \mathbb{N})$ and any $n \in \{2, 3, \ldots, p_i - 1\}$ does not divide p_i.

The set A_{t_0} of all oid's is defined over an enumerable set:

$$A_{t_0} = \{oid_0, oid_1, oid_2, \ldots\} = \{oid_i \mid i \in \mathbb{N}\}$$

while all other carrier sets are subsets of A_{t_0}.

$$A_{t_1}=\{oid_k \mid k = p_1^i\} \qquad \overline{A}_{t_1}=\{oid_k \mid k = p_1^{(p_0^i)}\}$$
$$A_{t_2}=\{oid_k \mid k = p_1^{(p_1^i)}\} \qquad \overline{A}_{t_2}=\{oid_k \mid k = p_1^{(p_1^{(p_0^i)})}\}$$
$$A_{t_3}=\{oid_k \mid k = p_1^{(p_2^i)}\} \qquad \overline{A}_{t_3}=\{oid_k \mid k = p_1^{(p_2^{(p_0^i)})}\}$$
$$A_{t_4}=\{oid_k \mid k = p_1^{(p_1^{(p_2^i)})}\} \qquad \overline{A}_{t_4}=\{oid_k \mid k = p_1^{(p_2^{(p_1^{(p_0^i)})})}\}$$

Figure 11 depicts this construction. The sub-type hierarchy is on the left while the construction is on the right, some values have been added in order to illustrate the operations' semantics.

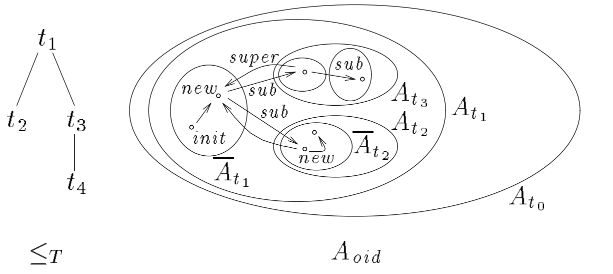

Figure 11: Carrier sets inclusion for a given $\leq_T$.

6.1.1 Multi-set and A_{oid} Extension

In our context, multi-sets are used for two purposes. The former is the need of indistinguishable copies of data and the latter is for the expression of true concurrency at the semantic level.

In order to be able to model a multi-set of values for each place, we syntactically extend the signature of the algebraic specification, i.e. we add for each sort the multi-set sort and its operations $[\_], +, \subseteq$. Then, instead of adding the corresponding axioms and consider the resulting models of the specification, we directly provide the multi-set operations' semantics.

Let *ASpec* an order-sorted algebraic specification. The syntactic multi-set extension is written $[ASpec]$, while the semantic extension of a model A of $Mod(ASpec)$ is denoted by $[A]$. Moreover, we write $Mod([ASpec])$ for the multi-set extension of all models of *ASpec*.

Moreover, in a similar manner we extend an algebraic specification with an object identifier algebra in order to be able to develop data structures of object identifiers and deal with this kind of values as in section 4.6. The syntactic extension is realized by adding for each class type the associated sort as well as the four operations *init, new, sub*, and *super*. On the other hand, each carrier set of the object identifier algebra and the four corresponding operation semantics are provided at the semantic level.

Note that this extension is dependent of the subtype relation. We write $Mod_{\leq_T}(ASpec)$ for both the object identifier and the multi-set extensions of all models of *ASpec*.

6.1.2 Management of object identifiers

In order to be able to create new class instances we have to know, at any time, the last used object identifier. This is achieved by the set of functions $Loid = \{f : Ty \rightarrow A_{oid} \mid f(t) \in \overline{A}_t\}$ (*Loid*: Last object identifier). Here we rather use a set of functions than a value of the algebra A_{oid}, because each type has its own last used identifier which have to be memorized.

Moreover, to compute new identifiers for each type t of a set of type T, the following family of functions $newloid_t : Loid \rightarrow Loid$ (*newloid*: new last object identifier) is defined for this purpose $\forall t' \in T, newloid_t(loid) = loid'$ such that

$$\begin{cases} loid'(t') = loid(t) & \text{if } t' \neq t \\ loid'(t') = new_t(loid(t)) & \text{otherwise} \end{cases}$$

This family of functions $newloid_t$ will be used in the inference rule regarding the creation of new instances.

6.1.3 Object states

The state of an object is based on the notion of marking, i.e. a mapping which returns the multi-sets of algebraic values of each place of the object. However, this usual notion of marking does not well capture the states of a set of class instances. Thus, we extend the domain of the markings as follows: a marking $m : A_{oid} \times PLACES \rightarrow A$ where A is a multi-set of algebraic values. We denote by $Mark_{A_{oid}, PLACES, A}$ the set of all markings.

Definition 6.2 *Concurrent transition system*
Let *ASpec* be an order-sorted algebraic specification

and A a model of $Mod_{\leq_T}(ASpec)$. Given a set of events $E = EVENTS_{A,METHODS,SORTS_T}$, a *concurrent transition system* over E, $PLACES$ and A is defined as

$$CTS_A = \langle Mark_{A_{oid},PLACES,A} \times E \times Mark_{A_{oid},PLACES,A} \rangle$$

We note a transition e between two states m and m' by $m \xrightarrow{e} m'$.

We introduce two basic operators on markings: $\bowtie$ and $\trianglelefteq$. Informally, $m_1 \bowtie m_2$ means that two markings are equal on their common places, $m_1 \trianglelefteq m_2$ considers two markings and returns a marking with the values of the marking m_1 plus the values of the places of m_2 not in m_1.

Definition 6.3 *Marking domain*
Let $m \in Mark_{A_{oid},PLACES,A}$ be a marking. The *domain marking* of a marking m is given by

$$Dom(m) = \{\langle oid, p\rangle \mid m(oid,p) \text{ is defined}, p \in PLACES, oid \in A_{oid}\}$$

□

Definition 6.4 *Common markings, marking fusion*
Given two markings $m_1, m_2 \in Mark_{PLACES,A}$

$$m_1 \bowtie m_2 \iff \forall \langle oid,p\rangle \in Dom(m_1) \cap Dom(m_2)\quad m_1(oid,p) = m_2(oid,p)$$

$$m_1 \trianglelefteq m_2 = m_3 \iff \forall \langle oid,p\rangle\ m_3(oid,p) = \begin{cases} m_1(oid,p) & \text{if } \langle oid,p\rangle \in Dom(m_1) \\ m_2(oid,p) & \text{if } \langle oid,p\rangle \in Dom(m_2) \setminus Dom(m_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$ □

When a new object is dynamically created, it is necessary to give it an initial state. The function *Init* takes charge of this initialization. We denote by P_t the set of places names of a class C_t (which has the type t) and by $init_p$ the algebraic value of the initial marking related to the place p.

$$Init : A_{oid} \times SORTS_T \to Mark_{A_{oid},PLACES,A}$$

$$Init(oid,t) = m \text{ such that } \begin{cases} m(oid,p) = init_p \forall p \in P_t & \text{if } oid \in \overline{A}_t \\ \text{not defined} & \text{otherwise} \end{cases}$$

6.2 Inference Rules

In order to construct the semantics, mainly a concurrent transition system, of a CO-OPN/2 specification we provide a set of six inference rules which are given in figure 12 at the end of the paper. These rules realizes the following tasks:

- Class-Sem builds, for a given class, its the partial transition system according to its methods, places, and behavioral axioms,
- Beh-seq computes all deductible sequential behaviors,
- Beh-con computes all deductible concurrent behaviors,
- Sync solves all the synchronizations between the transition systems built by the rule Class-Sem,
- Stab "eliminates" all invisible or spontaneous events which correspond to the internal transition of the classes,
- Create takes charge of the dynamic creation and the initialization of new class instances.

The last five rules induce a function over transition systems, called *@Rules*. This function computes a fixed point which corresponds to the concurrent transition system resulting from the application of the rules Beh-seq, Beh-con, Sync, Stab, and Create. This fixed point exists because the monotonicity of the rules.

Finally, the semantics of a class and the semantics of a CO-OPN/2 specification are given by the following definitions.

Definition 6.5 *Semantics of a class*
Let $ASpec$ be an order-sorted algebraic specification such that $Pres(ASpec) = \langle \Sigma, V, Ax\rangle$ with the signature $\Sigma = \langle S, \leq_S, F\rangle$ and a class $C = \langle ASpec, I, B\rangle$ in which $I = \langle M, t, \leq_T\rangle$ and $B = \langle P, X, BAx\rangle$. Given a model A of $Mod_{\leq_T}(ASpec)$, the semantics of a class C is the mapping

$$Sem_A(C) = CTS_A$$

where CTS_A is the concurrent transition system built up from the inference rule Class-Sem. □

Definition 6.6 *Semantics of a CO-OPN/2 specification*
Let $Spec = \{C_i \mid C_i = \langle ASpec_i, I_i, B_i\rangle, 1 \leq i \leq n\}$ be a CO-OPN/2 specification and A be a model of $Mod_{\leq_T}(\cup_{1\leq i \leq n} ASpec)$ with the restriction the common parts are equivalent. The semantics of $Spec$ is

given by the mapping

$$\begin{aligned} Sem(Spec) &= \{\langle A, CTS_A\rangle \mid A \in Mod_{\leq_T}(ASpec), \\ &\quad CTS_A = @Rules(\bigcup_{C_i \in Spec} Sem_A(C_i))\} \end{aligned}$$

□

6.3 Semantic Discussion about the Example

Since we have introduced operators dealing with the concurrent behavior of the objects (operator '$\|$'), the transition systems express the evolution steps of concurrent events, i.e. $m \xrightarrow{t_1 \,\|\, t_2 \,\|\, \cdots \,\|\, t_n} m'$ where m, m' denote two markings and $t_1 \,\|\, t_2 \,\|\, \cdots \,\|\, t_n \equiv [t_1, t_2, \ldots, t_n]$ is the multi-set of the events t_i ($i = 1, \ldots, n$) as previously presented.

For example, in the introduction of the CO-OPN/2 language we introduced three classes where synchronizations may occur (see figure 1). Thus, we have, for instance, the following behaviors for the isolated objects after application of rule Class-Sem:

`Conveyor-belt` $m_1 \xrightarrow{\texttt{put(box)}} m_1' \xrightarrow{\texttt{get(box')}} m_1''$

`Packer` $m_2 \xrightarrow{\texttt{take } \textbf{with} \texttt{ c.get(box')}} m_2'$.

`Producer` $m_3 \xrightarrow{\texttt{produce } \textbf{with} \texttt{ c.put(box)}} m_3'$.

If we compose these transition systems and reduce the synchronization requests by means of semantic inference rules, we obtain

$$(m_1 + m_2 + m_3) \xrightarrow{\texttt{produce..take}} (m_1'' + m_2' + m_3')$$

This transition shows that the events `produce` and `take` can occur in sequence. Using the semantic inference rules Sync and Beh-seq we are able to determine that `produce` and `take` cannot occur simultaneously in a system based upon the `Conveyor-belt` class, but can concurrently occur in a system based upon the `Heap` class.

7 Refinement

An important notion which takes part in the software development process from the formal specification up to the actual implementation is the stepwise refinement. Intuitively, a refinement is the substitution of a part of a specification by an another part which is, in general, more concrete. Obviously, we expect that such a substitution preserves the semantics of the original specification with respect to the meaning of the term "semantics preservation" we adopt. Indeed, various levels of semantics preservation may given. For example, one might be too restrictive and requires that both semantics have to be isomorphic. On the other hand, someone else could have a weaker point of view and choose bisimulation or observational equivalence.

In other words, saying that one part of a specification is a refinement of another one means that one is able to compare the semantics of these two parts and deduce the semantic preservation.

More formally, but still in an abstract way, we could define what a "good" refinement is as follows. Let *Spec* be a specification and Cl, Cl' be two sets of classes such that $Cl \subseteq Spec$. Cl' is a refinement of Cl, written $Cl' \preceq Cl$, if there exists a relation $\preceq_{\text{Sem}}$ such that

$$\begin{aligned} &Sem(Cl') \preceq_{\text{Sem}} Sem(Cl) \Rightarrow \\ &\quad Sem((Spec \setminus Cl) \cup Cl') \preceq_{\text{Sem}} Sem(Spec) \end{aligned}$$

The definition we gave describes a class of refinements because the semantic comparison criterion is free. This freedom allows one to select the criterion according to its needs. Thus, one could choose bisimulation, observational equivalence or trace equivalence, for example.

7.1 Strong Concurrent Bisimulation

Here we give the definition of the strong concurrent bisimulation and we put the previous definition in a concrete form. In other words, we are going to establish that bisimulation stands for our previous refinement definition by means of the theorem 7.1

Definition 7.1 *Strong concurrent bisimulation* Given a model A and two concurrent transition systems $CTS_{A,1}$ and $CTS_{A,2}$. A bisimulation between $CTS_{A,1}$ and $CTS_{A,2}$ is the greatest relation $\mathrm{R} \subseteq Mark(CTS_{A,1}) \times Mark(CTS_{A,2})$ such that

1) if $m_1 \mathrel{\mathrm{R}} m_2$ and $m_1 \xrightarrow{e} m_1' \in CTS_{A,1}$ then there is $m_2 \xrightarrow{e} m_2' \in CTS_{A,2}$ such that $m_1' \mathrel{\mathrm{R}} m_2'$.

2) if $m_2 \mathrel{\mathrm{R}} m_1$ and $m_2 \xrightarrow{e} m_2' \in CTS_{A,2}$ then there is $m_1 \xrightarrow{e} m_1' \in CTS_{A,1}$ such that $m_1' \mathrel{\mathrm{R}} m_2'$.

We say that $CTS_{A,1}$ and $CTS_{A,2}$ are bisimilar if there exists a non empty relation R between $CTS_{A,1}$ and $CTS_{A,2}$, we denote this by $CTS_{A,1} \leftrightarrows_{\mathrm{c}} CTS_{A,2}$. □

For a given model A, the notation $CTS_A(Cl)$ denotes the concurrent system associated to the semantics of the set of classes Cl. Note that in the following theorem we consider a unique algebraic specification and transition system build up from the same algebraic model.

Theorem 7.1
Let *Spec* be a CO-OPN/2 specification and Cl and Cl' be two sets of classes such that $Cl \subseteq Spec$.

$$CTS_A(Cl')|_{Cl} \leftrightarrows_{\mathrm{c}} CTS_A(Cl) \Rightarrow$$
$$CTS_A((Spec \setminus Cl) \cup Cl')|_{Spec} \leftrightarrows_{\mathrm{c}} CTS_A(Spec)$$

□

The notation $Sem\,Cl'|_{Cl}$ means that the concurrent transition system associated to the semantics of Cl' is limited to the elements of Cl.

Proof
An obvious extension of the proof of theorem "replacement of objects" presented in [BG90]. □

We have the following correspondence with the previous general definition of refinement:

$$\_|_{Cl} \leftrightarrows_{\mathrm{c}} \_ \iff \_ \leq_{\mathrm{Sem}} \_$$

8 Sub-typing

We believe, like many other researchers [Ame90, LW93], that sub-type relationship relates the behavior of the objects and therefore is a semantic concern while inheritance is used for implementation convenience.

The sub-type relationship we are going to set out is based on the substitutability principle. Two forms of the principle have been established: the *weak* form and the *strong* form.

The *weak* form of the principle says that "s is a sub-type of t if an instance of type s can be substituted when an instance of type t is expected and no type error will occur". In other words, an instance of type t "understands" at least all messages of an instance of type s, and no "message not understood" will occur.

The other form, the *strong* form of the principle, requires not only that no type error occurs but also that the behavior of the whole system is preserved. This principle says that "for a system S, t is a sub-type of s if an instance of type s can be substituted when an instance of type t is expected and the behavior of S will be unchanged".

Programming languages often relate sub-typing with the weak form of the principle because it does not involve semantic constraints. However, our point of view is that more safety is needed, especially for specification languages. That is the reason we are more interested in the strong form of the principle than the weak one.

We can formalize the notion of sub-type in a similar way we did for the refinement. While refinement deals with class substitution, here we are concerned with class instance substitution.

Let *Spec* be a specification and s, t be two types associated to both the classes C_s and C_t respectively. Our sub-type definition is as follows: we say that the type s is a sub-type of t, written $s \leq_T t$ for a given criteria, if there exists a relation $\leq_{\mathrm{Sem}}$ such that

$$Sem(C_s) \leq_{\mathrm{Sem}} Sem(C_t) \Rightarrow$$
$$Sem(Spec \setminus \{C_t\}) \leftarrow_{C_t \leftarrow C_s} Sem(C_s) \leq_{\mathrm{Sem}} Sem(Spec)$$

This definition means that for classes having a semantics subtype relation ($\leq_{\mathrm{Sem}}$), their use in a specification implies a similar behavior if we substitute a class C_t by a class C_s whenever $s \leq_T t$. This substitution is realized by the operator $\leftarrow_{C_t \leftarrow C_s}$ which renames all the object identifiers and then inserts the new semantics.

8.1 Strong Form of the Substitutability

Put in a concrete form, the previous definition allows us to give in the context of CO-OPN/2

Let *Spec* be a CO-OPN/2 specification. Let s and t be two types of $SORTS_T$ and the corresponding classes C_s and C_t of $CLASSES$. The following result is expected.

$$CTS_A(C_s)\|_{C_t \leftarrow C_s} \leftrightarrows_{\mathrm{c}} CTS_A(C_t) \Rightarrow$$
$$@Rules(CTS_A(Spec \setminus \{C_t\}) \cup CTS_A(C_s)\|_{C_t \leftarrow C_s})$$
$$\leftrightarrows_{\mathrm{c}} CTS_A(Spec)$$

As previously mentioned in the refinement section we have correspondence with the general sub-type relation:

$$\_ \cup (\_)\|_{C_t \leftarrow C_s} \iff \_ \leftarrow_{C_t \leftarrow C_s} \_$$
$$@Rules(\_) \leftrightarrows_{\mathrm{c}} \_ \iff \_ \leq_{\mathrm{Sem}} \_$$

For example, in the section 4.3 we introduced two classes related by sub-typing: the `Deluxe-packaging` class and `Packaging`. Since the new behaviors of the class `Deluxe-packaging` does not affect at all the behaviors of its super-class `Packaging` our definition holds. Indeed, in this special case both the semantics of the classes C_s and C_t are isomorphic with respect to the restrictions. The same remark holds for the semantics of the specification *Spec* and the semantics of *Spec* where the substitution and the renamings have been realized.

On the other hand, in the semantics discussion we mentioned that both the classes `Conveyor-belt` and `Heap` do not have the same concurrent behaviors. Another way to show this result is to use our sub-type relation. In the `Heap` class concurrent access to the boxes is possible (simultaneous firing of `get` method is allowed) while this kind of event does not appear in the `Conveyor-belt` class. Therefore, both the classes are not bisimilar and consequently not related by sub-typing.

9 Summary and Conclusions

In this paper we have attempted to describe briefly the concurrent specification language CO-OPN/2 which has improved the previous CO-OPN language.

The concrete analogy of a Swiss chocolate factory has permitted the illustration of new features of the language, such as the introduction of the notion of class by means of algebraic nets, the separation between inheritance and sub-typing hierarchies, and the elegant handling of object identity by means of order-sorted algebras. In this paper, the notion of behavioral sub-typing has been shortly presented with a formal description. Our central research concerns of the future will therefore be oriented around this crucial point, in particular the introduction of the concept of observation.

$$\textsc{Class-Sem}\quad \frac{\begin{array}{l}\exists bax = Cond \Rightarrow Event :: Pre \rightarrow Post \in bax \in BAx \text{ and an assignment } \sigma : X \rightarrow A \\ e = [\![Event]\!]_\sigma^{EVENTS_{A,METHODS}} \\ \langle A, \sigma\rangle \models Cond, [\![Pre]\!]_\sigma^P \subseteq m, m + [\![Post]\!]_\sigma^P = m' + [\![Pre]\!]_\sigma^P\end{array}}{\langle loid, m\rangle \xrightarrow{e} \langle loid, m'\rangle}$$

$$\textsc{Beh-seq}\quad \frac{m'_1 \bowtie m_2, \langle loid, m_1\rangle \xrightarrow{e_1} \langle loid, m'_1\rangle, \langle loid, m_2\rangle \xrightarrow{e_2} \langle loid, m'_2\rangle}{\langle loid, m_1 \trianglelefteq m_2\rangle \xrightarrow{e_1 \,..\, e_2} \langle loid, m'_2 \trianglelefteq m'_1\rangle}$$

$$\textsc{Beh-con}\quad \frac{\langle loid, m_1\rangle \xrightarrow{e_1} \langle loid, m'_1\rangle, \langle loid, m_2\rangle \xrightarrow{e_2} \langle loid, m'_2\rangle}{\langle loid, m_1 + m_2\rangle \xrightarrow{e_1 \,\|\, e_2} \langle loid, m'_1 + m'_2\rangle}$$

$$\textsc{Sync}\quad \frac{\langle loid, m_1\rangle \xrightarrow{e_1 \text{ with } e_2} \langle loid, m'_1\rangle, \langle loid, m_2\rangle \xrightarrow{e_2} \langle loid, m'_2\rangle}{\langle loid, m_1 + m_2\rangle \xrightarrow{e_1} \langle loid, m'_2 + m'_1\rangle}$$

$$\textsc{Stab}\quad \frac{m'_1 \bowtie m_2, \langle loid, m_1\rangle \xrightarrow{e_1} \langle loid, m'_1\rangle, \langle loid, m_2\rangle \xrightarrow{\tau} \langle loid, m'_2\rangle}{\langle loid, m_1 \trianglelefteq m_2\rangle \xrightarrow{e_1} \langle loid, m'_2 \trianglelefteq m'_1\rangle}$$

$$\textsc{Create}\quad \frac{oid \in \overline{A_t},\ oid = loid(t)}{\langle loid, m\rangle \xrightarrow{oid.\text{create}} \langle newloid_t(loid), m + Init(oid, t)\rangle}$$

Figure 12: Inference rules for CO-OPN/2 semantics.

References

[Ame87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP'88: European conference on object-oriented programming: proceedings*, volume 276 of *LNCS*, pages 234–242, Paris, France, June 1987.

[Ame90] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. Bakker, W. P. Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX school proc.*, volume 489 of *LNCS*, pages 60–90, Noordwijkerhout, The Netherlands, May/June 1990.

[BB94a] Olivier Biberstein and Didier Buchs. CO-OPN/2 an Object-Oriented specification language based on hierarchical algebraic nets. Tech. Report 76, Ecole Polytechnique Fédérale de Lausanne, 1994.

[BB94b] 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.

[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.

[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.

[BG90] Didier Buchs and Nicolas Guelfi. CO-OPN: A concurrent object oriented Petri nets model. Rapports de Recherche, LRI 616, University of Paris-Sud, 1990.

[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.

[Bla91] Gordon S. Blair. What are object-oriented systems? — chapter 5. In Gordon Blair, John Gallagher, David Hutchinson, and Doug Shepherd, editors, *Object-oriented languages, systems and applications*, pages 108–135. Pitman Publishing, London, GB, 1991.

[CHC90] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Proceedings POPL '90*, San Francisco, Jan 17-19 1990.

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

[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.

[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.

[LW93] Barbara Liskov and Jeanette M. Wing. Specifications and their use in defining subtypes. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 16–28, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Pap89] Michael Papathomas. Concurrency Issues in Object-Oriented Languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–246. CUI, University of Geneva, July 1989.

[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 Systems Correctness and Reusability IS-CORE'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.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86*, volume 21, 11 of *ACM SIGPLAN Notices*, pages 38–45, November 1986.

[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.

[Weg90] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *Oops Messenger*, 1(1):7–87, June 1990.