CO-OPN/2 Applied to the Modeling of Cooperative Structured Editors*

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

Didier Buchs, Nicolas Guelfi
LGL-DI, Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
{buchs|guelfi}@di.epfl.ch

Abstract

In this paper we describe how to use the CO-OPN/2 (Concurrent Object-Oriented Petri Net) formalism, by developing a case study on groupware or, more specifically, on a cooperative editor of hierarchical diagrams. First of all, we present CO-OPN/2 and describe how some aspects specific to object orientation, such as the notions of class/object, of object reference and of the distinction between inheritance and sub-typing, are taken into account within the formalism. Afterwards, we show how these notions may be used for the modeling and the design of an editor of hierarchical Petri nets and discuss some points related to the concurrency. Finally, we show how flexible our modeling architecture is by giving some insight into to the design of other kinds of cooperative diagram editors.

Keywords: groupware, object-orientation, distributed systems, formal methods, Petri nets, algebraic specification, software engineering.

1 Introduction

The objective of this paper is to illustrate the modeling capabilities of the CO-OPN/2 specification language by means of a case study on groupware. The case study adopted consists of a Cooperative Diagram Editor (CDE) as has been proposed by R. Bastide, P. Palanque and C. Lakos. The aim of the case study has been to provide a common example for formal specification approaches that combine a formal model of concurrency with the object-oriented paradigm.

*This work has been sponsored partially by the Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) with the financial support of the OFES (Office Fédéral de l'Éducation et de la Science), and by the Swiss National Science Foundation project 2000.40583.94 "Formal Methods for Concurrency".

CO-OPN/2 is a Petri net based object-oriented formal specification language devoted to the specification of concurrent systems. We presented in the previous workshop on "Object-Oriented Programming and Models of Concurrency" the syntax and the semantics of the CO-OPN/2 language and some principles concerning inheritance, sub-typing and refinement [BB95]. During this workshop many fine proposals for object-oriented extensions of Petri nets were proposed. Although these proposals all have the same objectives, more or less, it is difficult to precisely delineate their similarities and differences.

The presentation of a common case study can help to classify the various formal modeling approaches. However, it seems that several points should be clarified in order to understand each formalism well. A non-exhaustive list of questions, the answers to which can help to understand and distinguish the various approaches, follows.

1. What kind of data structures are provided? How are they described?
2. What is an object?
3. How is a class considered?
4. How do the objects interact with each other?
5. How are the Petri nets and the object notion combined?
6. What kind of concurrency is promoted (intra/inter-object concurrency)?
7. What is the underlying semantics of concurrency?
8. How are the object identities managed?
9. What do the terms inheritance and sub-typing mean?

This paper is organized as follows. Section 2, the next section, briefly presents the inherent problems of cooperative systems. In Section 3, the principles of the CO-OPN/2 specification language, and the answers to the questions stated above are given. The actual case study is presented in Section 4. In Section 5 some general considerations about the modeling of the case study are given, while some aspects of its design are given in Section 6. The final section, Section 7, is devoted to a discussion of how this case study modeling might evolve, in order to permit the consideration of other diagram types.

2 Cooperative Software

This section is a brief presentation of the most important principles used in the design of cooperative software environments.

According to C. S. Ellis, a cooperative system is "a computer based system that supports groups of people engaged in a common task (or goal) and that provides an interface to a shared environment". Thus, in the cooperative system discussed here, we can say that the following computer science topics are involved:

- Distributed Systems: the design of a groupware application involves many users working on heterogeneous platforms which are linked by means of a computer network.
- Concurrency: the order in which the events in the cooperative system are related must be precisely studied so as to have a coherent and efficient shared environment.
- Networks and Telecommunications: the architecture and the functions of the cooperative systems are directly dependent upon concrete communication capabilities.
- Computer-Human Interaction: a multi-user interface design requires that new interaction modes be developed to permit complex synchronous, or asynchronous, group actions.

A basic notion of groupware is, above all, the shared context, which necessitates the description of a view of this context related to each user in interaction. Logically, this leads to concrete synchronism, or asynchronism, principles which are related to interaction time constraints. Other problems that must be solved by groupware are data granularity and strongly, or loosely, coupled works (which depend on the degree of cooperation between the users). Popular electronic mail or desktop conference systems are classic forms of groupware in the field of communication applications. For the shared editors we can quote systems such as Griffon [DQV92] or GroupDraw [GRWB92]), among others.

Several general multi-layer models have been given in order to have a precise framework for the development of groupware applications (see [Kar94] and [EW94]).

The architecture of synchronous groupware must be studied in order to build a robust, transparent and efficient system. The groupware must also allow for modifications of distributed data within some time limits that are satisfactory to the user and have no constraints upon the graphic interface. The two major approaches to groupware architectures are the centralized and the replicated architectures. The first has a central process which manages all the users' accesses along with the data integrity. The second introduces one process for each user and all these processes communicate together in order to preserve the integrity of the data which are replicated on each site.

We use a layered structure, in Section 5, for the specific cooperative editor model we wish to define, and we organize the system in a centralized way. It must be noted that the purpose of the Cooperative Distributed Editor modeling is to elaborate an initial abstract solution which respects the requirements.

3 CO-OPN/2 Principles

CO-OPN [BG91] (Concurrent Object-Oriented Petri Nets) is a specification language designed for the specification and the modeling of large concurrent systems. The two underlying formalisms of CO-OPN are the algebraic specifications and the Petri nets which are combined in a way that is similar to the combination of algebraic nets [Rei91]. The first formalism is used to describe the data structures and the functional aspects of a system, while the second serves to model its operational and concurrent features. However, both these formalisms are not suitable to specify "in the large". To compensate for the lack of structuring capabilities in the Petri nets, the object paradigm has been adopted. Thus, a system is considered as being a collection of independent entities which interact and collaborate together in order to accomplish the various tasks of the system. With respect to the data structures, the algebraic specifications are composed of modules which are organized into a hierarchy.

In order to overcome some limitations of CO-OPN regarding the object orientation, a new version, named CO-OPN/2 [BB95], has been developed which introduces some notions peculiar to object-orientation such as the notions of class, inheritance, and sub-typing. For the sake of homogeneity regarding the notion of sub-typing, order-sorted algebraic specifications [GM89] have been adopted for the description of the data structures. In these, many sorted algebraic specifications form a special case.

3.1 Object and Class

An object is considered as an independent entity composed of an internal state which provides some services to the exterior. The only way to interact with an object is to ask a service of it; the internal state is then protected from inadvertent accesses. Our point of view is that this protection mechanism, known as encapsulation, forms the essence of object-orientation and there should be no way of violating it.

CO-OPN/2 defines an object as begin an encapsulated algebraic net in which the places compose the internal state and the transitions model the concurrent behavior of the object. A place consists of a multi-set of algebraic values which are described by means of order-sorted algebraic values. The transitions are divided into two groups: the parameterized transitions, also called the methods, and the internal transitions. The former correspond to the services provided, while the latter composes the internal behaviors of an object. Contrary to what occurs with the methods, the internal transitions are invisible to the exterior world and may be considered as being spontaneous events.

An important characteristic of the systems we want to consider is that of their potential dynamic evolution in terms of the number of objects they may contain. Thus, the dynamic creation of objects is a major objective. In order to describe these dynamic evolving systems, the objects are grouped into classes. A class describes all the components of a set of objects and can be viewed as serving as an object template. All the objects of one class manifest the same structure.

3.2 Object Interaction

In our approach, the interaction with an object is synchronous, although asynchronous communications may be simulated. Thus, when an object requires a service it asks to be synchronized with the method (parameterized transition) of the object provider. The synchronization policy is expressed by means of a synchronization expression, which may involve many partners, as long as with three synchronization operators (one for simultaneity, one for sequence, and one for alternative or non-determinism). For example, an object may simultaneously request two different services of two different partners, followed by a request the service from a third object. This mechanism generalizes other approaches such as the transition fusion or the Petri net systems [Kie89].

3.3 Concurrency

From an intuitive point of view, each object possesses its own behavior and concurrently evolves with the others. The Petri net model naturally introduces both inter-object and intra-object concurrency into CO-OPN/2 because the objects are not restricted to sequential processes.

The step semantics of CO-OPN/2 allows for the expression expression of true concurrency in comparison with interleaving semantics.

Nevertheless, the purpose of CO-OPN/2 is that of capturing the abstract concurrent behavior of each entity modeled, with the concurrency granularity not found in the objects but rather in the invocations of the methods. A set of method calls can be concurrently exercised on the same object.

3.4 Object Identity

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

Since object identifiers are algebraic values, it is possible to define data structures which are build upon object identifiers, e.g. a stack or a queue of object identifiers. Evidently, the places of algebraic nets may contain object identifiers.

3.5 Inheritance and Sub-typing

We believe that inheritance and sub-typing are two different notions which are used for two different purposes. Inheritance is considered as being a syntactic mechanism which avoids the necessity of developing classes from scratch and mainly serves in re-use of parts of existing specifications. A class may inherit all the features of another and may also add some services or change the description of some already defined services.

The sub-typing relationship is based upon the strong version of the substitutability principle [Ame90, LW93]. This principle implies that, in any context, a class instance of a given type may be substituted with another instance of the sub-type while the behavior of the whole system remains unchanged. In other words, the instances of the sub-type have a strong semantic conformance relationship with the super-type definition. This conformance relationship is based, in CO-OPN/2, upon the bisimulation between the semantics of super-type and the semantics the sub-type restricted to the behavior of the super-type.

Both inheritance and sub-typing relationships must be explicitly given and both of the hierarchies generated by these relationships do not necessarily coincide. In other words, two classes related by inheritance are not necessarily related by sub-typing. Identifying both inheritance and sub-typing hierarchies leads to several problems as stated by Snyder [Sny86] and America [Ame87].

3.6 Answers to the Questions

Before proceeding to the treatment of the case study, we now furnish answers to the questions presented in the introduction. These short answers summarize the information contained in the preceding subsections and are destinated to give an overview of CO-OPN/2.

1. What kind of data structures are provided ? How are they described ?

   The data structures consist of order-sorted algebraic values which are described by means of hierarchical order-sorted algebraic specifications.

2. What is an object ?

   An object is an encapsulated entity which provide some services to the outside. An object is represented by a Petri net in which the tokens are algebraic values. The places (multi-set of algebraic values) of the net represent the state of the object. Two kinds of transitions are provided, the internal transitions or spontaneous events and the parameterized transition or methods.

3. How is a class considered ?

   A class describes the structure of the objects, and is simply considered to be an object template.

4. How do the objects interact with each other ?

   An object interacts with some other objects by asking to be synchronized with them. The synchronization policy is expressed by means of a synchronization expression involving three operators (simultaneity, sequence, and alternative).

5. How the Petri nets and the object notion are combined ?

   In the previous workshop [Bas95] summarized this by "Objects inside Petri nets, Petri nets inside objects, unifying approach". Because of its use of object references, CO-OPN/2 may be classified in the "unifying approach" category.

6. What kind of concurrency is promoted (intra/ inter-object concurrency) ?

   The Petri net model naturally implies the promotion of both intra/inter-object concurrency.

7. What is the underlying semantics of concurrency ?

   A true concurrency semantics (step semantics) has been adopted.

8. How are the object identities managed ?

   All the objects possess an identity or reference. These references are order-sorted algebraic values, established by a particular order-sorted algebra whose sub-sort relationship reflects the sub-typing relationship. Data structure of references may be constructed and may be contained in the places of an object.

9. What do the terms inheritance and sub-typing mean ?

   Inheritance is mainly considered to be a syntactic mechanism which serves in the re-use of parts of existing classes while sub-typing implies a strong semantic conformance relationship between the instances of the sub-type and the super-type instances. This conformance relationship is based on bisimulation. The inheritance process allows one to eliminate or re-define some previously defined components. Two classes related by inheritance are not necessarily related by sub-typing. Both of these relationships between classes must be explicitly expressed.

3.7 Other Approaches

The other approaches which may be compared to CO-OPN/2 may be categorized in the High Level Petri Net class of models. These include algebraic nets [Rei91], OPN [Lak95], Cooperative Nets [SB94], and CLOWN [BCC95] among others. All these models are based upon the Petri nets as well as upon some formalisms which vehicle the data structures aspects. Moreover, the object-oriented paradigm has been adopted by all these formalisms.

4 Cooperative Diagram Editor

We have chosen to adhere strictly to the original statement of the Cooperative Diagram Editor (CDE) case study, as has been suggested by Bastide Lakos and Palanque so as to correctly make a comparison between our modeling and other approaches. This section presents the original statement of the case study, equipped with some words that are highlighted by numbers which will be later clarified in Subsection 4.2. The details furnished by these later clarifications correspond to our specific interpretations of some freedom left by the authors of the case study.

4.1 Original Statement of the Case Study

The system to be studied is software allowing for cooperative editing of hierarchical diagrams. The diagrams may be the work products of some Object-Oriented Design methodology, Hardware Logic designs, *Petri Net diagrams*①, etc. (Note that if the diagrams happen to coincide with *the formalism you are proposing*②, be careful to distinguish clearly between the two.)

One key aspect of this problem is that the editor should cater for several users, working at different workstations, and cooperating in constructing the one diagram. In the Computer Supported Cooperative Work (CSCW) vocabulary, such a tool could be ranked amongst synchronous groupware (each user is informed in real time of the actions of the others) allowing for relaxed WYSIWIS (What You See Is What I See) : each user may have his own customized view of the diagram under design, viewing different parts of the drawing or examining it at a different level of detail.

A *second key aspect*③ of this problem is that the editor should cater for hierarchical diagrams, i.e. components of the diagram can be exploded to reveal subcomponents.

A simple *coordination protocol*④ is proposed to control the interactions between the various users:

1. (a) Users may join or leave the editing session at will, and may join with different levels of editing privileges. For example, a user may join the session merely to view the diagram, or perhaps to edit it as well (see below).
   (b) The current members of the editing session ought to be visible to all, together with their editing privileges.
2. (a) Graphical elements may be free or owned by a user.
   (b) Different levels of ownership should be supported, including ownership for deletion, encapsulation, modification, and inspection.
   (c) The ownership must be compatible with the user's editing privileges.
3. Ownership for deletion requires that no-one else has any ownership of the component – not even for inspection.
4. Ownership for encapsulation requires that only the owner can view the internal details of the component – all other users can only view the top level or *interface*⑤ to the component.
5. Ownership for modification allows the user to modify attributes, but not to delete the component.
6. Ownership for inspection only allows the user to view the *attributes*⑥.
7. Only ownership for encapsulation can persist between editing sessions. (Note that this ownership is tied to a particular user, not a particular workstation.) All other ownership must be surrendered between sessions.
8. Ownership for inspection is achieved simply by selecting the component.
9. Other forms of ownership (and release) are achieved by an appropriate command, having first selected the component.
10. The level of ownership of a component is visible to all other users, as is the identity of the owner.
11. The creator of an element owns it for deletion until it is explicitly released.

4.2 Specific Interpretations

As has been suggested in the statement of the case study, several kinds of cooperative editors may be considered and the designers may exercise a certain degree of freedom when making their interpretations. In order to present the informal specification with as much precision as possible, the following explain in details our specific interpretations of each highlighted words which appear the previous sub-section.

- ①,② We have decided to consider three kinds of CDE: hierarchical Petri nets similar to Petri nets systems [Kie89], SADT (Structured Analysis and Design Technique) [RS77] and CO-OPN/2 diagrams. The hierarchical Petri nets editor is used as basic case study and a complete modeling of this editor is presented. Nevertheless, both SADT and CO-OPN/2 diagrams editors are briefly presented as variants on the hierarchical Petri nets editor. We believe that both of these variant editors are relevant to the understanding of our presentation, seeing as the SADT diagrams have absolutely no connections to the Petri nets community, while CO-OPN/2 diagrams modeled by means of CO-OPN/2 introduce an elegant auto-reference exercise.
- ③ We have made the choice of representing a component by means of an object. In order to describe a hierarchical structure, two kinds of components have been introduced: the components which are hierarchical and the ones which are not. The former correspond to the nodes while the latter represent the leaves of the hierarchical structure.
- ④ The coordination protocol is only used to coordinate the diagram accesses of the users and must be distinguished from the synchronization protocol which, in turn, must fulfill the cooperative requirement which allow all users to act upon the same document. Moreover, it must ensure the integrity of the document, for example as described in Section 2.
- ⑤ With regard to the notion of interface, we have introduce the notion of anchor, the notion of link along with the notion of interface. An anchor of a component is a location to which another component may be attached. A link of a component is used to link two anchors, one of which is a being part of the component and the other part of another component. These two notions form the concept of the interface of a component corresponding to the set of sub-components which may be linked to any anchor of the component itself. Thus, the interface corresponds to a set of components. For example, the interface of the transition of the hierarchical Petri nets, which are under consideration consists of the input and output places.
- ⑥ The notion of attribute is present but we do not provide the specification associated with it. For example, the date of creation as well as the name of the creator of a component could be regarded as being attributes. Note that position and rotation are not considered as being attributes because they are integral parts of the component.

4.3 Example of an Editing Session

Here is given an example of a hierarchical Petri net which could be edited by means of our cooperative diagram editor. Figure 1 depicts a two level Petri net involving some transitions, some places, some arcs, and one token. The figure shows that both the transitions and the places have been placed in a hierarchical relationship (a transition is associated with a sub-net and a place may contain some tokens). For instance, hierarchical transition t (level 0) is associated with a sub-net (level 1) composed of the component labeled p_i^t, t_i^t, a_j^t for $i = 1, 2$ and $j = 1, \ldots, 4$ and the place p_1 (level 0) contains one token labeled r (level 1) which is also displayed at level 0. The anchors are represented by small black squares placed at the border of each component. Recall that an anchor is a location to which another object may be linked and that a component is linked to another component by giving both the anchors and the identity the other component.

The three names 'Nicolas', 'Olivier' and 'Didier' represent the three users involved in the editing session. 'Nicolas' and 'Olivier' edit the Level 0 while 'Didier' acts upon the Level 1.

5 Modeling Structure

The purpose of this case study is to provide a CO-OPN/2 model of a cooperative editor of hierarchical diagrams. Among the two kinds of architectures which were introduced in Section 2 (centralized and replicated), the centralized architecture has been chosen because it encompasses all the requirements of the case study. Moreover, the design of applications seems to

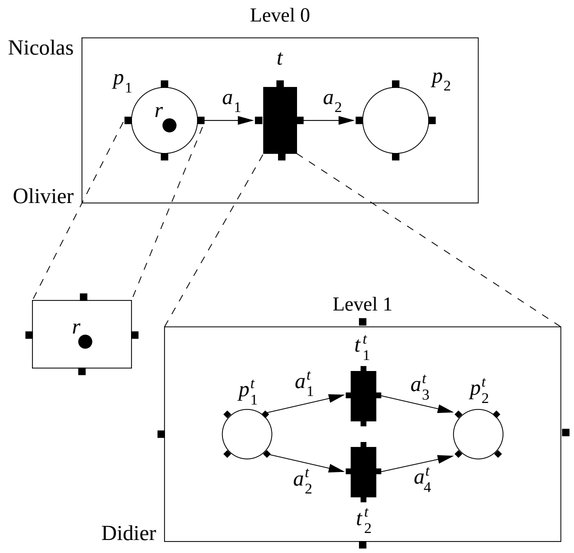

Figure 1: An Example of Editing Session.

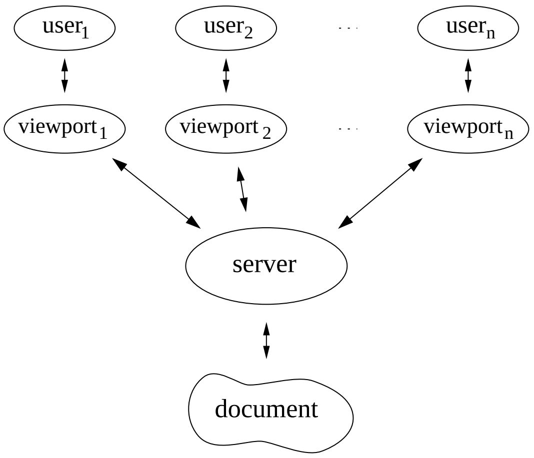

Figure 2: Overview of a CDE.

be easier with respect to centralized architectures in contrast to those which are replicated.

We use a layered abstract model of a cooperative software upon which to base our specification of a CDE. This model is a simplification of the one that has been proposed by Karsenty [Kar94] and has already been presented for the elaboration of a general cooperative system modeling which uses CO-OPN/2 [BBG96b].

We consider a Graphical Interface Layer (the viewports), a Centralized Synchronization Layer (the server) and an Abstract Document Representation Layer (the diagram document). This structure and its elements are described in this section.

5.1 Structure of a CDE

Four kinds of entities are involved in a centralized architecture of a CDE: users, viewports, a server and the document. Figure 2 depicts an overview of this structure in which one may see all four kinds of entities as well as the interaction between them, represented with the arrows.

All the users collaborate in the edition of a part of a hierarchical diagram by means of a viewport which provides some basic functions which permit that aspects of the document be modified. Viewports serve to collect the actions of its logged-in user, to transmit these actions to the server, and to display the document which pertains to the information sent by the server. The role of the server is to collect and handle the actions of the users, to update the document in accordance with the required coordination protocol, to inform every concerned viewport of the actions of the users and to allow for a simultaneous access to the document. Thus, when several users collaborate in the editing of one document, all of them see each others modifications.

5.2 Document Representation

As is required, a document has an organization that is hierarchical. In order to represent a tree structure, two kinds of components must be introduced : hierarchical components and atomic components. The former kind of components may include some subcomponents and correspond to the nodes of the hierarchical document, while the latter kind of components represent the leaves of the structure and thus may not include any sub-components. A document is, in itself, a hierarchical component which contains all the components of the document. Since each component is represented by means of an object, i.e. an entity which possesses some attributes and provides some services, a document is essentially a collection of objects which are organized into a hierarchy.

5.3 Three Levels of Entities

In the interest of establishing generalizations and progressively refining our discussion, our modeling is organized in three main entity levels as is shown in Figure 3. Each level is, in fact, the result of a further step in the development of the specification which progressively enriches the simple specification of a single user editor. One may see the four entities which compose an editor presented horizontally, i.e. `User`,

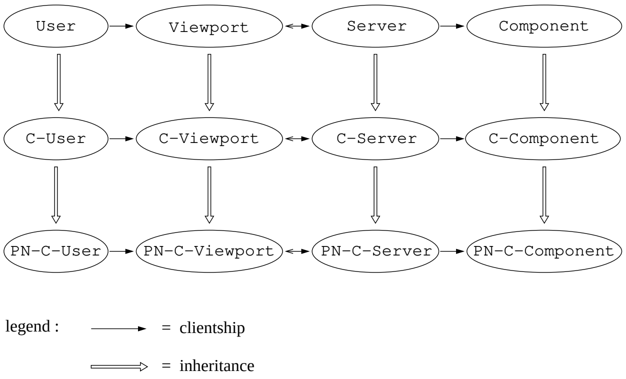

Figure 3: The Three Levels of Entities.

`Viewport`, `Server`, and `Component`. Vertically, the top level corresponds to classic editors, which does not permit any cooperation between more than one user. Thus, only one user at a time may edit a diagram. At the second level (`C-User`, `C-Server`, `C-Viewport`, `C-Component`), the notion of cooperation as well as the notions of attribute, ownership and interface are introduced. This second level, in fact, complies with all the proposed case study. The third level, depicted in Figure 3 shows an example of a concrete cooperative hierarchical Petri nets editor. It is composed of a `PN-C-Server`, `PN-C-Viewport` and its `PN-C-Components`. The effective components of a Petri nets, i.e. the arcs, the places, and the transitions, are not represented in this figure. Since they compose the document, they should be connected with the `PN-C-Component` entity. In the next sub-section these entities, which compose the document, are explained.

This structure makes possible to build various cooperative editors. At the third level of the Figure 3 we illustrate a cooperative Petri net editor but other kinds of CDE such as SADT diagrams or even CO-OPN/2 diagrams editor, could equally have been derived. These two variant editors will be briefly presented in Section 7.

5.4 Structure of the Classes

The classes which compose our modeling of a CDE, as well as the relationship between the classes, clearly arise from Figure 3. Thus, each oval of the figure corresponds to a class and the relationships between the classes are represented by the arrows. Three kinds of relationships between classes are especially relevant here : clientship, inheritance, and sub-typing.

Clientship

The clientship relationship between two classes indicates that an instance of one class requires for some services of an instance of another class by means of message passing or, in our case, of method synchronization. A clientship relationship is graphically represented by means of thin arrows. In Figure 3, for example, one may see, at the top level, that the `User` class uses the `Viewport` class because a user simply submits some requests to the viewport, e.g. to select a component or add a new component to the diagram, and no information are send back from the viewport to the user. A similar argument holds for the two other levels. Regarding the other entities (viewports, servers and components), the clientship relationship is symmetric. In the case of a cooperative editor, if we consider, for example, the user's action of adding a new sub-component to a given component will generate the following interaction between the other entities. First the viewport transmits the request to the server which asks for the creation of the new component and summons the concerned component to incorporate the new component. Then, the component involved communicates its new aspect to the server which must subsequently compute and send the information which must be transmitted to each viewport.

Inheritance

As mentioned in Section 3.5, we believe that inheritance and sub-typing are two different notions which are used for two different purposes. Inheritance is mainly considered to be a syntactic mechanism for reusing a part of an existing class while sub-typing pertains to the behavior of the instances and is a semantics concern.

In Figure 3, it clearly appears that the thick vertical arrows represent inheritance relationships. The `User`, `C-User`, `Server`, `Viewport` and `Component` classes have been built from scratch while the `C-Server`, `C-Viewport` and `C-Component` classes reuse or inherit from the initial classes and add what is needed in order to allow for cooperative edition. In a similar manner, the classes located at the lower level inherit from the higher level and add services required by a cooperative editor of hierarchical Petri nets, e.g services which are mainly related to the creation of new places, tokens, arcs or transitions. Note that the `User` and `C-User` classes have to be developed from scratch. Although both seem similar, too may services have to be redefined in the `C-User` class and inheritance does not furnish any benefit.

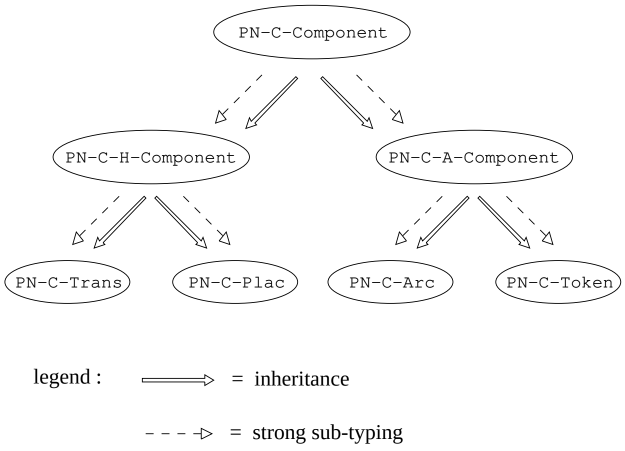

Figure 4: The Classes Related by Sub-typing.

Sub-typing

Recall that, in CO-OPN/2, sub-typing is based on the strong version of the substitutability principle. This principle implies that, in any context, a class instance of a given type may be substituted by another instance of the sub-type while the behavior of the whole system remains unchanged.

With respect to our modeling of a CDE, we shown that inheritance could be used without necessary imply a sub-typing relationship (c.f. Figure 3). Now, we introduce, in Figure 4, the classes which related, on the one hand, by sub-typing with the dashed arrows and, on the other hand, by inheritance. The classes at the lowest level represent the effective components involved in a cooperative hierarchical Petri net editor (i.e. transitions, places, arcs, and tokens), while the classes at the top and the second levels are used to classify and to make the distinction between the hierarchical components (i.e. the transitions and the places) and the atomic ones (i.e. the arcs and the tokens).

It is not surprising that the depicted classes in Figure 4 alone are related by sub-typing. In fact, we will see in the next section that contra-variant rule is violated between `Component`, `C-Component`, and `PN-Component`. A similar argument will be given for the viewport classes as well as for the server and user classes.

6 CO-OPN/2 and CDE Design

In this section we describe the CO-OPN/2 design of a cooperative hierarchical Petri net editor. In order to explain some relevant parts of the specification, we first describe the major syntactic aspects of the CO-OPN/2 language before introducing our design. Unfortunately, due to the size of most modules we do not provide a complete specification of these but rather a specification of the relevant parts. Nevertheless, the complete specification of the case study can be found in [BBG96a].

6.1 Syntactic Aspects of CO-OPN/2

A CO-OPN/2 specification consists of two kinds of modules: the algebraic abstract data type modules and the class modules. These two kinds of modules are composed of three parts: a header, which includes the information about inheritance and genericity; an interface, which describes what is accessible when another module uses it; and a body, which primarily conceals the properties of the operation, the behavior and the state of the objects.

When a non-generic class is developed from scratch, its header comes down to the keyword **Class**[1] followed by the name of the class. When a class is used only for classification, or when it is not completely implemented and the creation of some of its instances makes no sense, we preface the keyword **Class** by the keyword **Abstract**. In the **Interface** section, the field **Use** declares all the modules used by the current class. The **Type** field declares the name of the type of the instances which is used whenever an object identity has to be defined. This field has been introduced in order to avoid that the name of a module or a class and the name of its type are confused, especially in cases when inheritance or sub-typing are required. Both names are often very similar but address two different concepts. Usually classes are used to dynamically create new instances but it is also possible to declare static instances by means of the **Objects** field. All the services provided by the class instances are declared within the field **Methods**. Note that the mixfix and the applicative notation has been adopted for the profile of the methods. The final field **Creation** included in the interface section concerns the dynamic creation of the class instances. Within this field are listed the particular methods which create and initialize the objects; these methods may be used only once. A pre-defined creation method `create` is provided when the **Creation** field is empty or absent. The **Body** section includes some internal or spontaneous transitions declared under the **Transitions**

[1] Specification 1 may help the reader to understand the meaning of the various keywords introduced in this sub-section.

field as well as the attributes of the instances within the **Places** field. The **Initial** field describes the initial marking or the static initialization of each instance while the properties of the methods and internal transitions are described by means of behavioral axioms within the **Axioms** field. It is necessary to recall that a transition (parameterized transition or internal transition) may ask to be synchronized with other partners by means of a synchronization expression. The synchronization expressions takes place after the **with** keyword. The usual dot notation has been adopted and three synchronization operators has been provided: '//' for simultaneity '..' for sequence, '+' for alternative.

A behavioral axiom is established as follows

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

in which *Cond* is an optional condition imposed upon the algebraic values involved in the axiom, *Event* is either an internal transition name or a method with parameters, and *Sync* is an optional synchronization expression. *Pre* and *Post*, respectively, correspond to what is consumed and what is produced at the different places within the net. Finally, the variables used within the **Axioms** field are grouped together under the **where** field.

6.2 Design of the Entities

After this short introduction of the CO-OPN/2 syntactic aspects, we can now describe the design of the various entities presented in Figure 3. Other concepts of the language will be explained as required.

Note that our design takes into account three levels of classes related by inheritance but not by sub-typing. Sub-typing is only present between the `PN-Component` class and the Petri net components classes. One may find all the entities and their relationships presented here depicted in Figure 3 or 4. However, for the sake of clarity, we have only provided the complete hierarchy of the classes which concern the components. In this paper, the `C-User`, `C-Viewport`, and `C-Server` classes are build from scratch and not derived from the top level as shown in Figure 3.

Users

The users involved in a CDE are modeled by means of the `C-User` class. This is a small and simple class, built from scratch, which encompasses some relevant aspects easy to explain. In order to give a global view

```
Abstract Class C-User;
Interface
 Use Coord, Size, Angle,
     Privilege, Ownership;
 Type c-user;
 Methods
  join _ : privilege;
  leave;
  select  _ : coord;
  unselect;
  go-down;
  go-back;
  delete;
  move    _ : coord;
  resize  _ : size;
  rotate  _ : angle;
  own-for _ : ownership;
Body
 Use Unique, C-Viewport;
 Places
  idle   _ ,
  active _ : unique;
  cur-vp _ : c-viewport;
 Initial
  idle @;
 Axioms
  join pr with v.(self join pr) ::
   idle @ -> active @, cur-vp v;
  leave with v.leave ::
   active @, cur-vp v -> idle @;
  select p with v.select p ::
   active @, cur-vp v -> active @, cur-vp v;
  move p with v.move p ::
   active @, cur-vp v -> active @, cur-vp v;
  own-for os with v.own-for os ::
   active @, cur-vp v -> active @, cur-vp v;
  where
   p  : coord;      v  : c-viewport;
   pr : privilege;  os : ownership;
 ;; ... and more axioms ...
End C-User;
```

Specification 1: The Cooperative Users.

of a CO-OPN/2 class, we provide the textual form of the `C-User` class in Specification 1 as well as its partial graphic outline in Figure 5.

Within the **Use** field in the **Interface** section of Specification 1, one may see which algebraic abstract data types modules are used by the `C-User` class. These algebraic modules may be found in appendix. Nevertheless, we mention that the `Coord` ADT module defines the sort `coord`, which is pair of naturals and that the `Privilege` module defines both the `view` and `edit` user privileges as required. The four ownerships required by the proposal are defined in the ADT module `Ownership`. As for the `Unique` module, it only defines the generator '`@`' of sort `unique`, which plays the role of a black token.

Under the field **Type**, one finds the type name of the users.

Every name within the **Methods** field models an action that a user would like to accomplish by means of the viewport he is connected to. Among these methods we may mention the `join` and `leave` methods which correspond to the will of a user to join or to leave an editing session. The `select` method is used to select a component of the diagram at a given location, the `go-down` allows a hierarchical component, previously selected, to be visited, and the `go-back` rises again within in the hierarchy, `delete` removes a selected component. The `move`, `resize`, and `rotate` methods are used to modify aspects of selected components. The `own-for` method represents the user's will to own a component for a given ownership.

The `Viewport` and `Unique` modules are only used by in the **Body** section, this is why both of these modules are established within the **Body** section and not in the **Interface** section.

The instance variables are grouped under the **Places** field and are initialized as has been described in the **Initial** field (with the empty multi-set being the default value). These instance variables are involved in the behavioral axioms which describe the properties of the methods. For instance, the behavioral axiom

```
join pr with v.(self join pr) ::
  idle @ -> active @, cur-vp v;
```

indicates that the `join` method is asking to be synchronized with the `join` method for the privilege `pr` of an available existing viewport `v`. Recall that method synchronization (a method is a parameterized transition) corresponds to a unification of both formal and effective parameters and may occur if and only if pre- and post-conditions have been satisfied. In other words, the user must be idle (the value `@` in the place `idle` can be consumed) and will become active (the value `@` will be produced in the place `active`). Moreover, the free variable `v` will be unified with an available existing viewport and memorized by the current viewport attribute (`cur-vp v`). The self-reference **`self`** informs the viewport of the identity of the user who wants to be connected to it. The viewport will store this object identifier in order to know what connection has been established.

In a second example, we consider the axiom associated with the `select` method:

```
select p with v.select p ::
  active @, cur-vp v -> active @, cur-vp v;
```

The role of this action is the transmission of the user's will of selecting an existing component at a given position `p` to the current viewport `v`. The same remark holds for the other methods.

Figure 5 provides a partial graphic view of the `C-User` class. The inside of the ellipse includes the encapsulated elements, the solid arrows represent the data flow, as described by the axioms, and the circles indicate the places. As for the rectangles, those which are white correspond to the internal or spontaneous transitions (which are not present in this class) and those which are dark indicate the methods. Note that for the sake of clarity not all the methods have been represented.

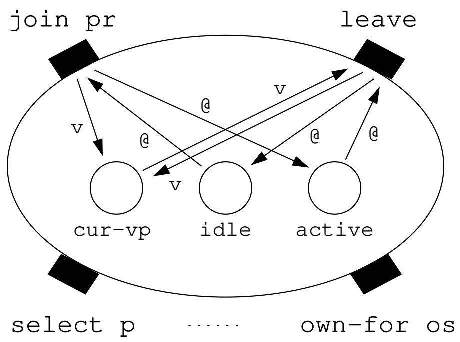

Figure 5: Outline of the `C-User` Class.

```
Class PN-C-User;
Inherit C-User;
 Rename c-user -> pn-c-user;
        C-Viewport -> PN-C-Viewport;
        c-viewport -> pn-c-viewport;
Interface
 Objects Didier, Nicolas, Olivier : c-user;
 Methods
  new-trans;
  new-plac;
  new-token;
  new-arc;
Body
 Axioms
  new-trans with v.new-trans ::
   cur-vp v -> cur-vp v;
  where
   v : pn-c-viewport;
 ;; ... and more axioms ...
End PN-C-User;
```

Specification 2: Cooperative Petri Net Users.

In order to obtain an effective class of users which may be involved in a hierarchical Petri net editor we make use of inheritance. In Specification 2 one may see the inheriting class `PN-C-User` which inherits from the class `C-User` as declared under the **Inherit** field. In this situation some renamings, declared under the **Rename** field, are necessary. As expected, the profiles and the properties of the new services provided by the inheriting class have been introduced within the **Methods** and **Axioms** fields, respectively. The new methods introduced in this class model the user's will to create the effective components involved in the hierarchical Petri net diagrams at a given location, i.e.

places, transitions, arcs, and tokens. Moreover, one observes that three static objects have been declared under the **Objects** field.

Viewport

A viewport has two main responsibilities. First a viewport has to transmit the user's actions to the server. Its second role is to receive the information of the server and to redisplay the document whenever it is necessary.

```
Abstract Class C-Viewport;
Interface
  Use Coord, Size, Angle, Picture,
      Privilege, Ownership, C-User;
  Type c-viewport;
  Methods
    _ join _ : c-user privilege;
    leave;
    select _ : coord;
    unselect;
    go-down;
    go-back;
    delete;
    move    _ : coord;
    resize  _ : size;
    rotate  _ : angle;
    own-for _ : ownership;
    display _ : picture;
    clear-screen;
Body
  Use C-Server, C-User;
  Places
    priv _ : privilege;
    cur-user _ : c-user;
  Axioms
    u join pr
      with the-server.(u login self) ::
      -> priv pr, cur-user u;
    leave with the-server.(u logout) ::
      cur-user u, priv pr -> ;
    select p with
      the-server.(u select p wth pr) ::
      cur-user u, priv pr ->
      cur-user u, priv pr;
    move p
      with the-server.(u move p wth pr) ::
      cur-user u, priv pr ->
      cur-user u, priv pr;
    own-for os
      with the-server.(u own-for os) ::
      cur-user u -> cur-user u;
    display pic  :: -> ;
    where
      p  : coord;
      pr : privilege;   os  : ownership;
      u  : c-user;      pic : picture;
  ;; ... and more axioms ...
End C-Viewport;
```

Specification 3: General Class of Viewports.

In Specification 3 we give a partial textual form of the `C-Viewport` class. One may see that the majority of the services concern the requests of the users. For example, the behavioral axiom of the `_join_` method informs the server that the user `u` wishes to be logged-in with the privilege `pr`. Thus, the viewport stores both the parameters.

The `select_` method asks the server to select a component which would be in location `p`, the current user and its privilege are also transmitted to the server.

The `display` method (abstractly modeled here) is activated by the server sends back the picture to be displayed on the screen.

We deliberately have not provided the specification of the effective class of the viewports involved in a hierarchical Petri net editor. We simply mention that the `PN-C-Viewport` class is based on the `C-Viewport` class and introduces the four services mandatory in the creation of the Petri net components.

```
Abstract Class Component;
Interface
 Use Coord, Size, Angle, String,
     Link, Set-Of-Links,
     Anchor, Set-Of-Anchors;
 Type component;
 Methods
  move       _ : coord;
  resize     _ : size;
  rotate     _ : angle;
  get-pos    _ : coord;
  put-label  _ : string;
  add-link   _ ,
  del-link   _ : link;
  get-links  _ : set-of-links;
  add-anchor  _ ,
  del-anchor  _ : anchor;
  get-anchors _ : set-of-anchors;
Body
 Places
  position  _ : coord;
  dimension _ : size;
  rotation  _ : angle;
  label     _ : string;
  links   _ : set-of-links;
  anchors _ : set-of-anchors;
 Axioms
  move p    :: position p' -> position p;
  resize s  :: size s'     -> size s;
  rotate r  :: rotation r' -> rotation r;
  get-pos p :: position p  -> position p;
  put-label lb  :: label lb' -> label lb;
  add-link l :: links lks -> links lks + l;
  del-link l :: links lks -> links lks-l;
  get-links lks :: links lks -> links lks;
  where
   p, p' : coord;     s, s'   : size;
   r, r' : angle;     lb, lb' : string;
   l : link;          lks : set-of-links;
 ;; ... and more axioms ...
End Component;
```

Specification 4: General Class of Components.

Components

A document is composed of components organized into a hierarchy . Some of these components are said to be hierarchical in the sense that they may contain sub-components and some are said to be atomic because they represent the leaves of the tree structure of the document and, consequently, cannot contain any sub-components. In fact a document is a hierarchical component in itself.

The `Component` class given in Specification 4 models the simplest components of our class structure. This kind of components does not include any notion of cooperation specific to the case study. The component state consists of six places, four of them concern the graphic aspects of the component itself, i.e. `position`, `dimension`, `rotation` and `label`. The `anchors` and `links` places contain, respectively, the set of the locations at which another component may be attached, and which components are linked to (a link is a triple of two anchors and one component).

The notions of cooperation between several users required by the case study are introduced in the `C-Component` class which inherits from the previous class as illustrated in Specification 5. The particular `authorized-for` place associates a set of users with each ownership. The associations user-ownership are surrendered by the `surrender` service as required. The `authorized-for?` method is used by the server to determine if a given user owns the component for a given ownership, while the `own-for` method allows a user to modify its ownerships. The behavioral axiom of the `own-for` event ensures that nobody owns the component for `deletion` and adds the user `u` to the users who already own the component for `modification`.

The `PN-C-Component` class in Specification 6 models the components in relation with a Petri net editor. As usual, this class inherits from the `C-Component` with some necessary renamings. From this class we have derived the `PC-C-H-Component` class, related to the hierarchical Petri net components in Specification 7, and the `PN-C-A-Component` class given in appendix.

The `PN-C-H-Component` class introduces three new instance variables as well as some associated methods. The `parent` place represents the identity of the ancestor component, while the `children` place corresponds to the set of sub-components. As for the `disp-depth` instance variable, it corresponds to how many levels have to be display by the `component-pic` method which returns the picture of the component. Thus, some components may be opaque or not (e.g. the places are not opaque since their tokens must be displayed). Opaqueness is represented by the value 0.

```
Abstract Class C-Component;
Inherit Component;
 Rename component -> c-component;
        Link -> C-Link;
        link -> c-link;
Interface
 Use Attributes,Ownership,Picture,C-User;
 Methods
  get-attrib _, put-attrib _ : attributes;
  _ own-for _ ,
  _ authorized-for? _ : c-user ownership;
  surrender;
  component-pic _ : picture;
Body
 Use Set-Of-C-Users;
 Places
  attrib _ : attributes;
  _ authorized-for _ : set-of-c-users
                       ownership;
 Initial
  [] authorized-for deletion;
  [] authorized-for inspection;
  [] authorized-for modification;
  [] authorized-for encapsulation;
 Axioms
  u own-for deletion ::
   (((users1 = []) or (users1 = []+u)) and
    ((users2 = []) or (users2 = []+u)) and
    ((users3 = []) or (users3 = []+u)) and
    ((users4 = []) or (users4 = []+u))) =>
   users1 authorized-for deletion,
   users2 authorized-for inspection,
   users3 authorized-for modification,
   users4 authorized-for encapsulation
   ->
   ([] + u) authorized-for deletion,
   users2   authorized-for inspection,
   users3   authorized-for modification,
   users4   authorized-for encapsulation;
  u own-for os ::
   not (os = deletion) =>
    [] authorized-for deletion,
    users authorized-for os
    ->
    [] authorized-for deletion,
    (users + u) authorized-for os;
  u authorized-for? os ::
   (users + u) authorized-for os
   ->
   (users + u) authorized-for os;
  where
   os : ownership;   u : c-user;
   users, users1, users2,
   users3, users4 : set-of-c-users;
End C-component;
```

Specification 5: Cooperative Components.

```
Abstract Class PN-C-Component;
Inherit C-Component;
  Rename
    c-component -> pn-c-component;
    C-Link -> PN-C-Link;
    c-link -> pn-c-link;
    C-User -> PN-C-User;
    c-user -> pn-c-user;
    Set-Of-C-Users -> Set-Of-PN-C-Users;
    set-of-c-users -> set-of-pn-c-users;
End PN-C-Component;
```

Specification 6: Cooperative Petri Net Components.

```
Abstract Class PN-C-H-Component;
Inherit PN-C-Component;
 Rename pn-c-component -> pn-c-h-component;
Interface
 Use Coord, Picture, PN-C-Component;
 Subtype pn-c-h-component < pn-c-component;
 Methods
  add-child _ ,
  del-child _ : pn-c-component;
  get-child _ at _ : pn-c-component coord;
  get-parent _ : pn-c-component;
Body
 Use Depth, Set-Of-PN-C-Components;
 Places
  parent   _ : pn-c-component;
  children _ : set-of-pn-c-components;
  disp-depth _ : depth;
 Axioms
  add-child c ::
   children compnts -> children compnts+c;
  del-child c ::
   children compnts+c -> children compnts;
  get-child c at p
   with c.(get-pos p) ::
   children compnts+c -> children compnts+c;
  get-parent c : parent c -> parent c;
  where
   p : coord;    c : pn-c-component;
   compnts : set-of-pn-c-components;
End PN-C-H-Components;
```

Specification 7: Hierarchical Petri Net Components.

The four effective Petri net components, i.e. the transitions, places, arcs, and tokens, are constructed from both `PN-C-H-Component` and `PN-C-A-Component` classes. We only have provided, in Specification 8, the `PN-C-Trans` class which describes the class of the hierarchical transitions involved in the hierarchical Petri net editor. The sub-typing relationship, explicitly declared within the **`Subtype`** field, follows the hierarchy of Figure 4 in Section 5.4.

The **`Initial`** field in Specification 8 defines the static initialization of the instance variables. For instance, one may see the dimension of the transitions and the position of their four anchors. The dynamic creation and initialization is given by the `create-trans` method within the **`Creation`** field and by its behavioral axiom which, in this case, just takes charge of the initialization of the `position` instance variable.

The `component-pic` method, abstractly modeled here, is used to return the graphic aspects of the transition (just a rectangle) which will be transmitted to the appropriate viewports.

Server

The server is a crucial element which must :

- manage the identification of the users,

```
Class PN-C-Trans;
Inherit PN-C-H-Component;
 Rename pn-c-h-component -> pn-c-trans;
 Redefine component-pic _ : picture;
Interface
 Use PN-C-H-Component;
 Subtype pn-c-trans < pn-c-h-component;
 Creation create-trans _ : coord;
Body
 Initial
  dimension <50 100>;
  rotation 0;
  anchors []+<25 0>+<0 50>+<49 50>+<25 99>;
  links [];
  disp-depth 0;
 Axioms
  component-pic pic :: -> ;
  create-trans p :: -> position p;
  where
   p   : coord;
   pic : picture;
End PN-C-Trans;
```

Specification 8: Hierarchical Petri Net Transitions.

- save and restore a document,
- provide all the services required by the users,
- cope with the accesses to the shared document in accordance with the coordination protocol,
- allow the users to simultaneously access to the document,
- send to the viewports the relevant information to be displayed.

A partial textual form of the `C-Server` class which allows several users to edit simultaneously the same document is given in Specification 9.

In a centralized architecture only one server is present. Thus, we define in the `C-Server` interface within the **`Object`** field a static instance of this class `the-server`. Thus at system initialization this instance will be created and each class using the class server will be able to use this server instance.

In order to give an insight of the behavior of such a server, we explain in detail the `_move_wth_` method which may also be observed in all the classes presented in this paper. Moreover, in Figure 6, we have provided a graphic outline of the `C-Server` class stressing the `move` method behavior. The dashed arrows represent the synchronization expressions of the behavioral axioms.

The set of `triple-uvc` contained in the instance variable `assoc` associates each logged-in user with the viewport he is connected to, and the current component he is currently acting upon.

The behavioral axiom of the `move` method indicates that the user `u` wants to move the selected com-

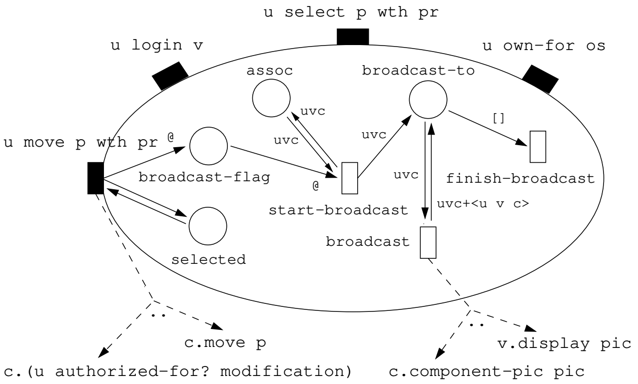

Figure 6: Outline of the Cooperative Server.

ponent according to its privilege pr to location p. To accomplish this task the server has to determine the already selected component and be synchronized, sequentially, with it. An additional necessary verification is to be sure that the user possesses the edit privilege. The synchronization expression first ensures that the user u has the authorization for the modification of the component c and then asks the component to move itself to the position p. The production of a black token @ in the place broadcast-flag triggers off the transition start-broadcast to be fired.

As one can see in Figure 6, the start-broadcast internal transition copies the assoc place into the broadcast-to which will be able to send the information to be displayed to each viewport by means of two synchronization requests. The first synchronization request gets the graphic aspect of the expanded current component, while the second synchronization sends the picture to the appropriate viewport v. This process is ended when all triples of the broadcast-to place have been removed and only the empty set resource is left. Finally, the finish-broadcast transition consumes the empty set.

Here, it is necessary to stress that the internal transitions are invisible to the outside world and are considered as being spontaneous events. Thus, for the outside world the events which compose the move method seem to be performed simultaneously, whereas they are actually serialized.

Finally, from the abstract C-Server class we derive the PN-C-Server which describes the effective server involved in the hierarchical Petri net editor. The PN-C-Server class is presented in Specification 10. This class introduces the missing services related to the creation of the effective Petri net components.

```
Abstract Class C-Server;
Interface
 Use Coord, Size, Angle, Privilege,
     Ownership, C-User, C-Viewport;
 Type c-server;
 Object the-server : c-server;
 Methods
  _ login _ : c-user c-viewport;
  _ select _ wth _ ,
  _ move   _ wth _ : c-user coord
                     privilege;
  _ own-for _ : c-user ownership;
  save;   restore;   new-document;
Body
 Use Unique, C-Component,
  Set-Of-C-Components, C-Triple-uvc,
  Set-Of-C-Triple-uvc, Picture;
 Places
  document _ : c-component;
  broadcast-flag _ : unique;
  broadcast-to   _ : set-of-c-triple-uvc;
  assoc _ : set-of-c-triple-uvc;
  _ selected _ : c-user set-of-c-components;
 Initial
  broadcast-to [];
  assoc [];
 Transitions
  start-broadcast;
  finish-broadcast;
  broadcast;
 Axioms
  u login v ::
   assoc uvc, document doc ->
   assoc (uvc + <u v doc>),
   u selected [], document doc;
  u select p wth pr
   with c.get-child c' at p ::
   (pr = edit) or (pr = view) =>
   assoc (uvc + <u v c>),
   u selected compnts
   ->
   assoc (uvc + <u v c>),
   u selected (compnts + c'),
   broadcast-flag @;
  u move p wth pr
   with c'.(u authorized-for? modification)
        .. c'.move p ::
   pr = edit =>
   u selected ([] + c')
   ->
   u selected [], broadcast-flag @;
  u own-for os with c'.(u own-for os) ::
   u selected ([] + c') -> u selected [];
  start-broadcast ::
   broadcast-flag @, assoc uvc
   ->
   assoc uvc, broadcast-to uvc;
  broadcast
   with c.component-pic pic
        .. v.display pic ::
   broadcast-to (uvc + <u v c>)
   ->
   broadcast-to uvc;
  finish-broadcast :: broadcast-to [] -> ;
  where
   p   : coord;
   pr  : privilege;   os  : ownerskip;
   pic : picture;
   uvc : set-of-c-triple-uvc;
   c, c', doc : c-component;
   compnts : set-of-c-components;
   u : c-user;    v : c-viewport;
End C-Server;
```

Specification 9: Cooperative Server Class.

```
Class PN-C-Server;
Inherit C-Server;
 Rename
 c-server -> pn-c-server;
 C-User   -> PN-C-User;
 c-user   -> pn-c-user;
 Set-Of-C-Triple-uvc->Set-Of-PN-C-Triple-uvc;
 set-of-c-triple-uvc->set-of-pn-c-triple-uvc;
Interface
 Methods
  _ new-trans-wth _ : pn-c-user privilege;
  _ new-plac-wth  _ : pn-c-user privilege;
  _ new-token-wth _ : pn-c-user privilege;
  _ new-arc-wth   _ : pn-c-user privilege;
Body
 Axioms
  u new-trans-wth pr
   with c.(u authorized-for? modification)
        .. c'.create-trans <0 0>
        .. c.add-child c' ::
   pr = edit =>
   assoc (uvc + <u v c>)
   ->
   assoc (uvc + <u v c>), broadcast-flag @;
  where
   uvc : set-of-triple-pn-c-uvc;
   u : pn-c-user;   v : pn-c-viewport;
   c : pn-c-component;
 ;; ... and more axioms ...
End PN-C-Server;
```

Specification 10: Petri Net Server Class.

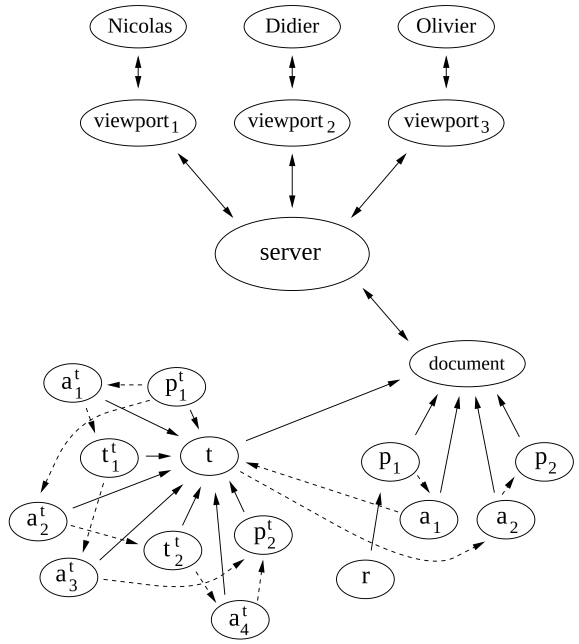

Figure 7: Class Instances of the Example.

Within the **Axioms** field, one may see how a new Petri net transition is created at a position `p` for an user `u` with the privilege `pr`. First the ownership for modification of the current component is verified, then (sequentially) the creation of the transition in itself is requested followed the addition of the new transition to the current component.

6.3 Behavioral Properties of the Classes

In Figure 7 we have provided a snapshot of a system of objects taking part in the development of a hierarchical Petri net, in accordance with the example given in Section 4.3 and illustrated in Figure 1. Two kinds of relationships are represented, the solid arrows correspond to the synchronizations that can be performed, and the dashed arrows represent the link between the connected components.

The server object is a key element of the whole system. All the modifications concerning any component are transmitted to the server which dispatches the result of the modifications to the appropriate viewports. In CO-OPN/2, the concurrency is naturally managed by the places which are multi-sets of algebraic values and build states concurrently accessible by the object methods. Despite this modeling power, it is sometimes difficult to model concurrent accesses when operations on global multi-set states have to be considered. It is, for instance, the case for the `children` place of the `PN-C-H-Component` class which is a set of references modeled by means of an algebraic abstract data type and not a multi-set. This modeling choice prevents from accessing simultaneously some of the sub-components of a node.

The server component allows intra-concurrency. Indeed, most of the services (not `login` and `logout`) of the server may be simultaneously activated by two different users for two different components, for instance the `_move_wth_` method. Moreover, some services may be requested by two different users for the same component, for instance the rotation and the shifting of a component. We did not have optimized the broadcast performed after a modification, for example a simultaneous modification of two components will produce two simultaneous request for the `display` service of the viewports (synchronously performed after the component modification).

7 Modeling of Other Editors

In this section we give one direction for the evolution of our modeling of the hierarchical Petri net editor in order to obtain other kinds of cooperative editors.

The class structure presented in this paper is flexible enough to take into account other cooperative editors such as an editor of SADT diagrams as well as an editor of CO-OPN/2 diagrams. The mandatory modifications only concern the classes derived from second level of Figure 3. Thus, the lowest level of Figure 3 as well as the component hierarchy must be redefined according to the type of the diagram. Here follows two suggestions concerning the sub-typing relationship between the components of SADT diagrams and CO-OPN/2 diagrams.

SADT

SADT diagrams are quite simple, they consist of two kinds of components the boxes which may be arbitrarily nested, and the arrows. Thus, Figure 8 depicts the component hierarchy which could be adopted for the design of a cooperative editor of SADT diagrams.

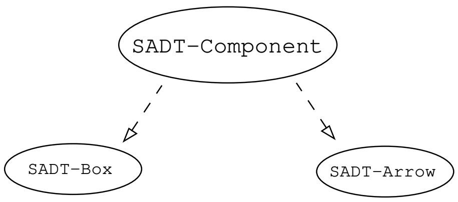

Figure 8: Sub-typing Hierarchy for SADT Components.

CO-OPN/2

A CO-OPN/2 specification is a set of interconnected classes which consist of an encapsulated Petri net equipped with some methods or parameterized transitions. Thus, CO-OPN/2 formalism defines only two levels of hierarchy with regard to the component nesting: the net level and the class level. At the net level, the CO-OPN/2 components are the places, the internal transitions, the methods and the arrows that represent the control flow. At the class level, the components are the classes and the arrows which express the synchronization requests. In Figure 9 we establish a possible component hierarchy for the a CO-OPN/2 diagram editor.

8 Conclusion

In this paper we modeled a case study on groupware or, more specifically, on a cooperative editor of hierarchical diagrams. The aim of this case study was to provide a general example in order to compare various approaches which combined a formal model of concurrency and the object-oriented paradigm. First, we presented the CO-OPN/2 specification language

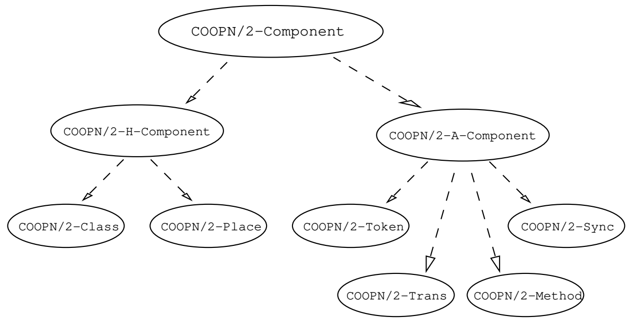

Figure 9: Sub-typing Hierarchy for CO-OPN/2.

by giving some explanations of the specific notions of object-orientation. As for the proposed case study, we showed how the CO-OPN/2 formalism is adapted for the modeling of cooperative applications in which concurrency is a major objective.

The distinction established between inheritance and sub-typing has allowed us to provide a flexible structure of classes and, at the same time, enabled us to treat the components which compose the diagrams in a consistent and regular way. However, it has appeared that a notion of sub-typing in which some semantic constraints could be relaxed, would increase the regularity of the treatments. The formalization of such a sub-typing relationship is one of our present research activities.

In order to provide a complete development system based on CO-OPN/2, we are working on the development of tools adapted to our formalism, and also studying the other phases of the life cycle such as the analysis and the test of object oriented software as well as the techniques of incremental implementation.

9 Acknowledgments

We would like to thank Alain Karsenty and Jan Vitek for their helpful suggestions and for the valuable discussions which all engaged in with regard to groupware and object orientation.

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'87: European conference on object-oriented programming: proceedings*, volume 276 of *LNCS*, pages 234–242, Paris, France, June 1987. Springer-Verlag.

[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.

[Bas95] Rémy Bastide. Approaches in unifying Petri nets and the object-orientation approach. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[BB95] Olivier Biberstein and Didier Buchs. Structured Algebraic Nets with Object-Orientation. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[BBG96a] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. CO-OPN/2 applied to the modeling of cooperative structured editors. Technical Report 96/184, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1996.

[BBG96b] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Using the CO-OPN/2 formal method for groupware applications engineering. July 1996. To appear in the IMACS-IEEE-SMC conference on Computational Engineering in Systems Application (CESA'96), Lille, France.

[BCC95] E. Battiston, A. Chizzoni, and F. De Cindio. Inheritance and concurrency in CLOWN. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[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.

[DQV92] D. Decouchant, V. Quint, and I. Vatton. L'édition coopérative de documents avec griffon. In *Actes des quatrièmes journées sur l'ingéniérie des interfaces homme-machine (IHM'92)*, pages 137–142, 1992.

[EW94] Clarence (Skip) Ellis and Jacques Wainer. A conceptual model of groupware. In *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, Models of Cooperative Work, pages 79–88, 1994.

[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.

[GRWB92] Saul Greenberg, Mark Roseman, Dave Webster, and Ralph Bohnet. Human and technical factors of distributed group drawing tools. *Interacting with Computers: Special Issue on CSCW: Part 1*, 4:3:364–392, 1992.

[Kar94] Alain Karsenty. *GroupDesign : un collectitiel synchrone pour l'édition partagée de documents.* PhD thesis, Université Paris XI Orsay, 1994. Also in Computing System: "GroupDesign: Shared Editing in a Heterogeneous Environment", vol. 6, no. 2, pp. 167–192, 1993.

[Kie89] Astrid Kiehn. Petri net systems and their closure properties. In G. Rozenberg, editor, *Advance in Petri Nets*, number 424 in LNCS, pages 306–328. Springer-Verlag, 1989.

[Lak95] Charles Lakos. From coloured Petri nets to object Petri nets. In *Proceedings of the "Application and Theory of Petri Nets 1995"*, volume 935 of *LNCS*, pages 278–297, Torino, Italy, June 1995. Springer.

[LW93] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

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

[RS77] D. T. Ross and K. E. Shoman. Structured analysis for requirements specification. *IEEE Transactions on Software Engineering*, 3(1):6–15, January 1977.

[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.

A Full CO-OPN/2 Specification

```
;; COOPN/2 specification of a Cooperative Editor of Hierarchical Diagrams
;; --------------------------------------------------------------------
;; Contains all the classes and ADT modules involved in the case study except the very
;; basic abstract data types such as the booleans, uniques, naturals, and strings;
;; === Component class
;; This class provides the basic elements of the graphic components involved in the editor.
;; It will be used so as to build more complex components.  The services given by this class
;; are classified into three categories:
;; 1- the services in relation with the graphical aspects of the component (move, resize,
;;    rotate, get-pos, put-label, ret-pic);
;; 2- the services which allow the user to attach a new component (add-link, del-link,
;;    get-links);
;; 3- the services which allow the user to add/remove a new position where components can be
;;    attached (add-anchor, del-anchor, get-anchors).
Abstract Class Component;
Interface
  Use  Coord, Size, Angle, String, Link, Set-Of-Links, Anchor, Set-Of-Anchors;
  Type component;
  Methods
    move      _ : coord;                        ;; modify the component aspects
    resize    _ : size;
    rotate    _ : angle;
    get-pos   _ : coord;                        ;; provide the position
    put-label _ : string;                       ;; modifie the component's name
    add-link  _ ,                               ;; add and remove a given link
    del-link  _ : link;
    get-links _ : set-of-links;                 ;; provide all component links
    add-anchor  _ ,                             ;; add and remove a given anchor
    del-anchor  _ : anchor;
    get-anchors _ : set-of-anchors;             ;; provide all the anchors
Body
  Places
    position  _ : coord;                        ;; where the component is located
    dimension _ : size;                         ;; how big is the component
    rotation  _ : angle;                        ;; how the component is oriented
    label     _ : string;                       ;; component's name
    links   _ : set-of-links;                   ;; which components are linked with
    anchors _ : set-of-anchors;                 ;; where the components may be linked
  Axioms
    move p        :: position p' -> position p;
    resize s      :: size s'     -> size s;
    rotate r      :: rotation r' -> rotation r;
    get-pos p     :: position p  -> position p;
    put-label lb  :: label lb'   -> label lb;
    add-link l    :: links lks -> links (lks + l);
    del-link l    :: links lks -> links (lks - l);
    get-links lks :: links lks -> links lks;
    add-anchor a      :: anchors anchs -> anchors (anchs + a);
    del-anchor a      :: anchors anchs -> anchors (anchs - a);
    get-anchors anchs :: anchors anchs -> anchors anchs;
    where
      p, p' : coord;
      s, s' : size;
      r, r' : angle;
      lb, lb' : string;
      l : link;
      a : anchor;
      lks     : set-of-links;
      anchs   : set-of-anchors;
End Component;
```

```
;; === Cooperative components

;; This class enriches the Component class with the cooperative aspects of the editor.

;; Four ownerships are taken into account
;;   deletion      : requires that no-one else has any ownership
;;   inspection    : only the owner can view the attributes
;;   modification  : only the owner can modify the attributes
;;   encapsulation : only the owner can view the internal details,
;;                   can persist between editing sessions,

;; The method authorized-for? plays the role of a predicate.  The method can be fired
;; iff the given user owns the component for the given ownership.  This method uses
;; the place authorised-for which associates a set of users for each ownership level.

;; The method own-for is called whenever a user wants to own the component for a given
;; ownership.  This method verifies the compatibility between the ownership and the
;; privileges of the user.

Abstract Class C-Component;
Inherit Component;
  Rename component -> c-component;
         Link -> C-Link;
         link -> c-link;
         Set-Of-Links -> Set-Of-C-Links;
         set-of-links -> set-of-c-links;
Interface
  Use Attributes, Ownership, Picture, C-User;
  Methods
    get-attrib _ ,                              ;; provide and mofify the attributes
    put-attrib _ : attributes;

    _ own-for _ ,                               ;; own the component for an ownership
    _ authorized-for? _ : c-user ownership;     ;; is the user authorized for an ownership ?

    surrender;                                  ;; surrender all ownerships except encapsulation
    component-pic _ : picture;                  ;; provide the picture of the component
Body
  Use Set-Of-C-Users;
  Places
    attrib _ : attributes;                      ;; t-uple of attributes
    _ authorized-for _ : set-of-c-users ownership;
  Initial
    [] authorized-for deletion;                 ;; nobody owns the component for any ownership
    [] authorized-for inspection;               ;; [] denotes the empty set
    [] authorized-for modification;
    [] authorized-for encapsulation;
  Axioms
    get-attrib t :: attrib t  -> attrib t;
    put-attrib t :: attrib t' -> attrib t;

    u own-for deletion ::
      (((users1 = []) or (users1 = ([] + u))) and  ;; guaranty that nobody (expect u itself)
       ((users2 = []) or (users2 = ([] + u))) and  ;; owns the component for any ownership
       ((users3 = []) or (users3 = ([] + u))) and
       ((users4 = []) or (users4 = ([] + u)))
      ) =>
        users1 authorized-for deletion,
        users2 authorized-for inspection,
        users3 authorized-for modification,
        users4 authorized-for encapsulation
        ->
        ([] + u) authorized-for deletion,       ;; u is the only one who owns the component
        users2   authorized-for inspection,     ;; for deletion and may be for inspection
        users3   authorized-for modification,   ;; modification and encapsulation
        users4   authorized-for encapsulation;

    u own-for os ::                             ;; verify that nobody owns the component for
      not (os = deletion) =>                    ;; deletion and add u as owner for os
        [] authorized-for deletion, users authorized-for os
        ->
        [] authorized-for deletion, (users + u) authorized-for os;

    u authorized-for? os ::                     ;; verify that the user u owns for ownership os
        (users + u) authorized-for os -> (users + u) authorized-for os;
```

```
    surrender ::                               ;; remove all the owners for each ownership
      users1 authorized-for deletion,          ;; except for encapsulation
      users2 authorized-for modification,
      users3 authorized-for inspection
      ->
      [] authorized-for deletion,
      [] authorized-for modification,
      [] authorized-for inspection;
    where
      t  : attributes;
      os : ownership;
      u  : c-user;
      users, users1, users2, users3, users4 : set-of-c-users;
End C-component;


;; === Cooperative users
;; A user may join of leave an editing session, and be connected to a viewport.
;; A user performs some actions sending the appropriate message to a the viewport
;; he is connected to.
Abstract Class C-User;
Interface
  Use  Coord, Size, Angle, Privilege, Ownership;
  Type c-user;
  Methods                                      ;; the user's actions
    join _ : privilege;                        ;; a user wants to join or leave an editing
    leave;                                     ;; session with a given privilege.
    select _ : coord;                          ;; select a component located in x,y
    unselect;                                  ;; unselect all the selected components
    go-down;                                   ;; visit the selected component
    go-back;                                   ;; go back up in the hierarchy
    delete;                                    ;; delete the selected component
    move   _ : coord;                          ;; modify the selected component aspects
    resize _ : size;
    rotate _ : angle;
    own-for _ : ownership;                     ;; modify the ownership of a sel. component
Body
  Use Unique, C-Viewport;
  Place
    idle   _ ,                                 ;; the user is either idle xor active
    active _ : unique;
    cur-vp _ : c-viewport;                     ;; current viewport he is connected to
  Initial
    idle @;                                    ;; default user's state
  Axioms
    join pr with v.(self join pr) :: idle @ -> active @, cur-vp v;
    leave with v.leave :: active @, cur-vp v -> idle @;
    select p with v.select p :: active @, cur-vp v -> active @, cur-vp v;
    unselect with v.unselect :: active @, cur-vp v -> active @, cur-vp v;
    go-down with v.go-down :: active @, cur-vp v -> active @, cur-vp v;
    go-back with v.go-back :: active @, cur-vp v -> active @, cur-vp v;
    delete with v.delete :: active @, cur-vp v -> active @, cur-vp v;
    move p with v.move p :: active @, cur-vp v -> active @, cur-vp v;
    resize s with v.resize s :: active @, cur-vp v -> active @, cur-vp v;
    rotate r with v.rotate r :: active @, cur-vp v -> active @, cur-vp v;
    own-for os with v.own-for os :: active @, cur-vp v -> active @, cur-vp v;
    where
      p  : coord;
      s  : size;
```

```
      r  : angle;
      v  : c-viewport;
      pr : privilege;
      os : ownership;
End C-User;


;; === Cooperative viewport
;; A viewport transmits each action of the user connected  to the server and collects
;; the results.  For each user's action, the server sends back to the viewport the picture
;; to be displayed by the viewport.
Abstract Class C-Viewport;
Interface
  Use  Coord, Size, Angle, Picture, Privilege, Ownership, C-User;
  Type c-viewport;
  Methods
    _ join _ : c-user privilege;              ;; a user asks to join/leave an editing
    leave;                                    ;; session with a given privilege
    select _ : coord;                         ;; select a component located in x,y
    unselect;                                 ;; unselect all the selected components
    go-down;                                  ;; visit the selected component
    go-back;                                  ;; go back up in the hierarchy
    delete;                                   ;; delete the selected component
    move   _ : coord;                         ;; modify the selected component aspects
    resize _ : size;
    rotate _ : angle;
    own-for _ : ownership;                    ;; modify the ownership of the sel. component
    display _ : picture;                      ;; display the picture returned by the server
    clear-screen;                             ;; used whenever a user join or leave
Body
  Use C-Server, C-User;
  Places
    priv _ : privilege;                       ;; privileges of the current user
    cur-user _ : c-user;                      ;; who is currently logged in
  Axioms
    u join pr with the-server.(u login self) :: -> priv pr, cur-user u;
    leave with the-server.(u logout) :: cur-user u, priv pr -> ;
    select p with the-server.(u select p wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    unselect with the-server.(u unselect) :: cur-user u -> cur-user u;
    go-down with the-server.(u go-down-wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    go-back with the-server.(u go-back) :: cur-user u -> cur-user u;
    delete with the-server.(u delete-wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    move p with the-server.(u move p wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    resize s with the-server.(u resize s wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    rotate r with the-server.(u rotate r wth pr) :: cur-user u, priv pr -> cur-user u, priv pr;
    own-for os with the-server.(u own-for os) :: cur-user u -> cur-user u;
    display pic  :: -> ;                      ;; abstractly defined here
    clear-screen :: -> ;
    where
      p  : coord;
      s  : size;
      r  : angle;
      pr : privilege;
      os : ownership;
      u  : c-user;
      pic : picture;
End C-Viewport;


;; === Cooperative server
```

```
;; The server collects the users' actions sent by the viewports and performs them.
;; It mainly solves the accesses to the various components by means of a simple
;; protocol and broadcast to each viewport currently connected the results of the
;; users actions, i.e. send a different picture to each viewport.

;; The place assoc makes an association between each loggeg-on user, the viewport he is
;; connected to, and the component he is currently working on. This place consists of
;; a set which will be sequentially consulted during the broadcast which will be
;; triggered off when a token ill be present in the broadcast-flag place.
;; The selected place associates each logged-on user with thier set of selected components.

Abstract Class C-Server;
Interface
  Use  Coord, Size, Angle, Privilege, Ownership, C-User, C-Viewport;
  Type c-server;
  Object the-server : c-server;               ;; static class instance
  Methods
    _ login _ : c-user c-viewport;            ;; same services as the c-user and c-viewport
    _ logout  : c-user;                       ;; except that the user who ask the service
                                              ;; is sent by the viewport as well as ist
    _ select _ wth _ : c-user coord privilege;  ;; privilege when necessary
    _ unselect : c-user;

    _ go-down-wth _ : c-user privilege;
    _ go-back : c-user;

    _ delete-wth _ : c-user privilege;

    _ move   _ wth _ : c-user coord privilege;
    _ resize _ wth _ : c-user size  privilege;
    _ rotate _ wth _ : c-user angle privilege;

    _ own-for _ : c-user ownership;

    save;                                     ;; save the document iff nobody use it
    restore;                                  ;; load the document previously saved
    new-document;                             ;; create a new document, abstractly defined
Body
  Use Unique, C-Component, Set-Of-C-Components, C-Triple-uvc, Set-Of-C-Triple-uvc, Picture;
  Places
    document _ : c-component;                 ;; the document currently edited
    broadcast-flag _ : unique;                ;; trigger the broadcast
    broadcast-to   _ : set-of-c-triple-uvc;   ;; auxiliary set of connected users/viewport
    assoc _ : set-of-c-triple-uvc;
    _ selected _ : c-user set-of-c-components;  ;; the set of selected components of each user
  Initial
    broadcast-to [];                          ;; nobody is connected yet
    assoc [];                                 ;; nobody is connected yet
  Transitions
    start-broadcast;                          ;; internal transitions so as to broadcast
    finish-broadcast;                         ;; the information to each connected viewport
    broadcast;
  Axioms
    u login v ::
        assoc uvc, document doc
        ->
        assoc (uvc + <u v doc>), u selected [], document doc;

    u logout :: assoc (uvc + <u v c>), u selected compnts -> assoc uvc;

    u select p wth pr with c.get-child c' at p ::
      (pr = edit) or (pr = view) =>
        assoc (uvc + <u v c>), u selected compnts
        ->
        assoc (uvc + <u v c>), u selected (compnts + c'), broadcast-flag @;

    u unselect :: u selected compnts -> u selected [], broadcast-flag @;

    u go-down-wth pr with c'.(u authorized-for? encapsulation) ::
      ((pr = edit) or (pr = view)) =>
        assoc (uvc + <u v c>), u selected ([] + c')
        ->
        assoc (uvc + <u v c'>), u selected [], broadcast-flag @;
```

```
    u go-back with c.get-parent c' ::
        assoc (uvc + <u v c>), u selected compnts
        ->
        assoc (uvc + <u v c'>), u selected [], broadcast-flag @;
    u delete-wth pr with c.(u authorized-for? deletion) .. c.del-child c' ::
      pr = edit =>
        assoc (uvc + <u v c>), u selected []+c'
        ->
        assoc (uvc + <u v c>), u selected [], broadcast-flag @;
    u move p wth pr with c.(u authorized-for? modification) .. c.move p ::
      pr = edit =>
        u selected ([] + c) -> u selected [], broadcast-flag @;
    u resize s wth pr with c.(u authorized-for? modification) .. c.resize s ::
      pr = edit =>
        u selected ([] + c) -> u selected [], broadcast-flag @;
    u rotate r wth pr with c.(u authorized-for? modification) .. c.rotate r ::
      pr = edit =>
        u selected ([] + c) -> u selected [], broadcast-flag @;
    u own-for os with c.(u own-for os) :: u selected ([] + c) -> u selected [];
    start-broadcast ::
        broadcast-flag @, assoc uvc -> assoc uvc, broadcast-to uvc;
    broadcast with c.component-pic pic .. v.display pic ::
        broadcast-to (uvc + <u v c>) -> broadcast-to set-ucv;
    finish-broadcast :: broadcast-to [] -> ;
    save with doc.surrender :: document doc, assoc [] -> assoc [];
    restore :: -> document doc;
    new-document :: -> document doc;
    where
      p   : coord;
      s   : size;
      r   : angle;
      pr  : privilege;
      os  : ownerskip;
      pic : picture;
      uvc : set-of-c-triple-uvc;
      c, c', doc : c-component;
      compnts : set-of-c-components;
      u : c-user;
      v : c-viewport;
End C-Server;

;; === Petri nets cooperative components

;; This class forms the root of the sub-typing hierarchy of the concrete components
;; involved in the editor.

Abstract Class PN-C-Component;
Inherit C-Component;
  Rename
    c-component -> pn-c-component;
    C-Link -> PN-C-Link;
    c-link -> pn-c-link;
    C-User -> PN-C-User;
    c-user -> pn-c-user;
    Set-Of-C-Users -> Set-Of-PN-C-Users;
    set-of-c-users -> set-of-pn-c-users;
End PN-C-Component;

;; === Hierarchical Petri nets cooperative components

;; This class introduces all the elements necessary for the hierarchical components
;; such as the instance variable parent and children and the associated methods.
;; Moreover the instance variable disp-depth is introduced, it is used by the component-pic
;; so as to make some components not opaque such as the place in which the token are displayed.
;; Opaqueness correspond to the value 0. (e.g. 0 for the transitions, 1 for the places).
;; The concrete classes in relation with the hierarchical components of the editor
```

```
;; will be derived from this class (transition, place).
Abstract Class PN-C-H-Component;
Inherit PN-C-Component;
  Rename pn-c-component -> pn-c-h-component;
Interface
  Use Coord, Picture, PN-C-Component;
  Subtype pn-c-h-component < pn-c-component;
  Methods
    add-child _ ,                            ;; add and remove a child (sub-component)
    del-child _ : pn-c-component;
    get-child _ at _ : pn-c-component coord;    ;; return the child located in position x,y
    get-parent _ : pn-c-component;           ;; return the parent of the component
Body
  Use Depth, Set-Of-PN-C-Components;
  Places
    parent   _ : pn-c-component;             ;; who is the ancestor
    children _ : set-of-pn-c-components;     ;; who are the children (sub-components)
    disp-depth _ : depth;                    ;; how many levels have to be displayed
  Axioms
    add-child c :: children compnts -> children (compnts + c);
    del-child c :: children (compnts + c) -> children compnts;
    get-child c at p with c.(get-pos p) :: children (compnts + c) -> children (compnts + c);
    get-parent c : parent c -> parent c;
    where
      p   : coord;
      pic : picture;
      c   : pn-c-component;
      compnts : set-of-pn-c-components;
End PN-C-H-Components;


;; === Atomic Petri nets cooperative components
;; The concrete classes in relation with the atomic components of the editor
;; will be derived from this class (arc, token).
Abstract Class PN-C-A-Component;
Inherit PN-C-Component;
  Rename pn-c-component -> pn-c-a-component;
End PN-C-A-Component;


;; === Petri net transitions
;; A transition is a hierarchical component graphically represented by a rectangle
;; equiped with four anchors.
Class PN-C-Trans;
Inherit PN-C-H-Component;
  Rename pn-c-h-component -> pn-c-trans;
  Redefine component-pic _ : picture;
Interface
  Use PN-C-H-Component;
  Subtype pn-c-trans < pn-c-h-component;        ;; a transition is a hierarchical component
  Creation
    create-trans _ : coord;                  ;; creation of a transition located in x,y
Body
  Initial
    dimension <50 100>;                      ;; horizontal and vertical sizes
    rotation 0;                              ;; no rotation
    anchors []+<25 0>+<0 50>+<49 50>+<25 99>;   ;; 4 anchors (midle of each side of the rect.)
    links [];                                ;; no link
    disp-depth 0;                            ;; a transition is opaque
  Axioms
    component-pic pic :: -> ;                ;; abstractly defined here;
    create-trans p :: -> position p;         ;; initialize the last instance variable
    where
```

```
      p   : coord;
      pic : picture;
End PN-C-Trans;


;; === Petri net places

;; A place is a hierarchical component graphically represented by a circle
;; equiped with four anchors.

Class PN-C-Plac;
Inherit PN-C-H-Component;
  Rename pn-c-h-component -> pn-c-plac;
  Redefine component-pic _ : picture;
Interface
  Use PN-C-H-Component;
  Subtype pn-c-plac < pn-c-h-component;         ;; a place is a hierarchical component
  Creation
    create-plac _ : coord;                      ;; creation of a place at a given coordinate
Body
  Initial
    dimension <100 100>;                        ;; horizontal and vertical sizes
    rotation 0;                                 ;; no rotation
    anchors []+<50 0>+<0 50>+<99 50>+<50 99>;   ;; four anchors (east,west,north,south)
    links [];                                   ;; no link
    disp-depth 1;                               ;; sub-components (tokens) must be visible
  Axioms
    component-pic pic :: -> ;                   ;; abstractly defined here;

    create-plac p :: -> position p;             ;; initialize the last instance variable
    where
      p   : coord;
      pic : picture;
End PN-C-Plac;


;; === Petri net tokens

;; A token is an atomic component graphically represented by a disc.

Class PN-C-Token;
Inherit PN-C-A-Component;
  Rename pn-c-a-component -> pn-c-token;
  Redefine component-pic _ : picture;
Interface
  Use PN-C-A-Component;
  Subtype pn-c-token < pn-c-a-component;        ;; a token is an atomic component
  Creation
    create-token-at _ : coord;                  ;; creation of a token at a given coordinate
Body
  Initial
    dimension <5 5>;                            ;; horizontal and vertical sizes
    rotation 0;                                 ;; no rotation
    anchors [];                                 ;; no anchor
    links [];                                   ;; no link
  Axioms
    component-pic pic :: -> ;                   ;; abstractly defined here;

    create-token p :: -> position p;            ;; initialize the last instance variable
    where
      p   : coord;
      pic : picture;
End PN-C-Token;


;; === Petri net arc

;; An arc is an atomic component graphically represented by an arrow.

Class PN-C-Arc;
Inherit PN-C-A-Component;
  Rename pn-c-a-component -> pn-c-arc;
```

```
  Redefine component-pic _ : picture;
Interface
  Use PN-C-A-Component;
  Subtype pn-c-arc < pn-c-a-component;          ;; an arc is an atomic component
  Creation
    create-arc _ : coord;                       ;; creation of an arc at a given coordinate
Body                                            ;; rotation and length
  Initial
    dimension <200 20>;                         ;; horizontal and vertical sizes
    rotation 0;                                 ;; no rotation
    anchors []+<0 10>+<199 10>;                 ;; no anchor
    links [];                                   ;; no link
  Axioms
    component-pic pic :: -> ;                   ;; abstractly defined here
    create-arc p :: -> position p;              ;; initialize the last instance variable
    where
      p    : coord;
      pic  : picture;
End PN-C-Arc;


;; === Petri nets cooperative users
;;
;; From the cooperative user class, this class introduces the necessary services
;; for the creation of the four Petri nets components.
Class PN-C-User;
Inherit C-User;
  Rename
    c-user -> pn-c-user;
    C-Viewport -> PN-C-Viewport;
    c-viewport -> pn-c-viewport;
Interface
  Objects Didier, Nicolas, Olivier : c-user;    ;; here three users statically created
  Methods
    new-trans;                                  ;; creation of the components
    new-plac;
    new-token;
    new-arc;
Body
  Axioms
    new-trans with v.new-trans :: cur-vp v -> cur-vp v;
    new-plac  with v.new-plac  :: cur-vp v -> cur-vp v;
    new-token with v.new-token :: cur-vp v -> cur-vp v;
    new-arc   with v.new-trans :: cur-vp v -> cur-vp v;
    where
      v : pn-c-viewport;
End PN-C-User;


;; === Petri nets cooperative viewport
;;
;; From the cooperative viewport class, this class introduces the necessary services
;; for the creation of the four Petri nets components.
;; Each service calls the-server, unselect all selected components, and selects the
;; new component.
Class PN-C-Viewport;
Inherit C-Viewport;
  Rename
    c-viewport -> pn-c-viewport;
    C-User   -> PN-C-User;
    c-user   -> pn-c-user;
    C-Server -> PN-C-Server;
    c-server -> c-server;
Interface
  Methods
    new-trans;                                  ;; creation of the components
    new-plac;
```

```
    new-token;
    new-arc;
Body
  Axioms
    new-trans with the-server.(u new-trans-wth pr) ::
      cur-user u, priv pr -> cur-user u, priv pr;
    new-plac  with the-server.(u new-plac-wth pr) ::
      cur-user u, priv pr -> cur-user u, priv pr;
    new-token with the-server.(u new-token-wth pr) ::
      cur-user u, priv pr -> cur-user u, priv pr;
    new-arc   with the-server.(u new-arc-wth pr) ::
      cur-user u, priv pr -> cur-user u, priv pr;
    where
      pr : privilege;
      u  : pn-c-user;
End PN-C-Viewport;


;; === Petri net cooperative server
;; From the cooperative server class, this class introduces the necessary services
;; for the creation of the four Petri nets components.
;; Each service verify that the current component is own for modification, create the
;; new component and adds it to the children of the current component.  Finally the
;; broadcast is triggered off.
Class PN-C-Server;
Inherit C-Server;
  Rename
    c-server -> pn-c-server;
    Set-Of-C-Component -> Set-Of-C-Component;
    set-of-c-component -> set-of-c-component;
    Set-Of-C-Triple-uvc -> Set-Of-PN-C-Triple-uvc;
    set-of-c-triple-uvc -> set-of-pn-c-triple-uvc;
Interface
  Methods
    _ new-trans-wth _ : pn-c-user privilege;
    _ new-plac-wth  _ : pn-c-user privilege;
    _ new-token-wth _ : pn-c-user privilege;
    _ new-arc-wth   _ : pn-c-user privilege;
Body
  Axioms
    u new-trans-wth pr
      with c.(u authorized-for? modification) .. c'.create-trans <0 0> .. c.add-child c'
      pr = edit =>
        assoc (uvc + <u v c>) -> assoc (uvc + <u v c>), broadcast-flag @;
    u new-plac-wth pr
      with c.(u authorized-for? modification) .. c'.create-plac <0 0> .. c.add-child c'
      pr = edit =>
        assoc (uvc + <u v c>) -> assoc (uvc + <u v c>), broadcast-flag @;
    u new-token-wth pr
      with c.(u authorized-for? modification) .. c'.create-token <0 0> .. c.add-child c'
      pr = edit =>
        assoc (uvc + <u v c>) -> assoc (uvc + <u v c>), broadcast-flag @;
    u new-arc-wth pr
      with c.(u authorized-for? modification) .. c'.create-arc <0 0> .. c.add-child c'
      pr = edit =>
        assoc (uvc + <u v c>) -> assoc (uvc + <u v c>), broadcast-flag @;
    where
      uvc : set-of-triple-pn-c-uvc;
      u : pn-c-user;
      v : pn-c-viewport;
      c : pn-c-component;
End PN-C-Server;


;; *** Cooperative Editor of Hierarchical Diagrams
```

```
;; abstract data types part

;; A coordinate is a pair of naturals
Adt Coord as CartesianProduct2(Natural, Natural);
  Morphism
    elem1 -> natural;
    elem2 -> natural;
    _ = _ in ComparableElem1 -> _ = _ in Natural;
    _ = _ in ComparableElem2 -> _ = _ in Natural;
  Rename
    cp2   -> coord;
    fst _ -> xcoord _ ;
    snd _ -> ycoord _ ;
End Coord;

;; The depth is a natural number
Adt Depth as Natural;
End Depth;

;; An anchor is a coordinate which represent where a component can be attached.
Adt Anchor as CartesianProduct2(Natural, Natural);
  Morphism
    elem1 -> natural;
    elem2 -> natural;
    _ = _ in ComparableElem1 -> _ = _ in Natural;
    _ = _ in ComparableElem2 -> _ = _ in Natural;
  Rename
    cp2 -> anchor;
End Anchor;

;; A link attaches a current component with another component by means two anchors.
;; The first anchor is the anchor of the current component.
;; The component is the component with which the current component is attached
;; The second anchor is the anchor of the component with which the current component is attached.
Adt Link as CartesianProduct3(Anchor, Component, Anchor);
  Morphism
    elem1 -> anchor;
    elem2 -> component;
    elem3 -> anchor;
    _ = _ in ComparableElem1 -> _ = _ in Anchor;
    _ = _ in ComparableElem2 -> _ = _ in Component;
    _ = _ in ComparableElem3 -> _ = _ in Anchor;
  Rename
    cp2 -> link;
End Link;

;; A triple uvc makes an association between a user and the viewport he is connected with
;; and the current component he is working with.
Adt Triple-uvc as CartesianProduct3(User, Viewport, Component);
  Morphism
    elem1 -> user;
    elem2 -> viewport;
    elem3 -> component;
    _ = _ in ComparableElem1 -> _ = _ in User;
    _ = _ in ComparableElem2 -> _ = _ in Viewport;
    _ = _ in ComparableElem3 -> _ = _ in Component;
  Rename
    cp2 -> triple-uvc;
End Triple-uvc;

;; The four required owerships levels are defined.
Adt Ownership;
```

```
Interface
  Use  Boolean;
  Sort ownership;
  Generators
    deletion,
    inspection,
    modification,
    encapsulation : -> ownership;
  Operation
    _ = _ : ownership ownership -> boolean;
Body
  Axioms
    (deletion = deletion)           = true;
    (deletion = encapsulation)      = false;
    (deletion = modification)       = false;
    (deletion = inspection)         = false;

    (inspection = deletion)         = false;
    (inspection = encapsulation)    = false;
    (inspection = modification)     = false;
    (inspection = inspection)       = true;
    (modification = deletion)       = false;
    (modification = encapsulation)  = false;
    (modification = modification)   = true;
    (modification = inspection)     = false;

    (encapsulation = deletion)      = false;
    (encapsulation = encapsulation) = true;
    (encapsulation = modification)  = false;
    (encapsulation = inspection)    = false;
End Ownership;

;; Both the required privileges of a user are defined.
;; A user may merely view a diagram or be able to edit (modify) it
Adt Privilege;
Interface
  Use  Boolean;
  Sort privilege;
  Generators
    view, edit : -> privilege;
  Operation
    _ = _ : privilege privilege -> boolean;
Body
  Axioms
    (edit = edit) = true;
    (edit = view) = false;
    (view = edit) = false;
    (view = view) = true;
End Privilege;

;; Actually the attributes consist of a t-uple
;; This ADT must be more specific
Adt Attributes;
Interface
  Sort attributes;
End Attributes;

;; Actually a pictures consists of a matrix of pixels
;; This ADT must be more specific
Adt Picture;
Interface
  Sort picture;
End Picture;

;; Abstract data type which are in relation with the cooperative aspects
```

```
Adt C-Link as CartesianProduct3(Anchor, C-Component, Anchor);
  Morphism
    elem1 -> anchor;
    elem2 -> c-component;
    elem3 -> anchor;
    _ = _ in ComparableElem1 -> _ = _ in Anchor;
    _ = _ in ComparableElem2 -> _ = _ in C-Component;
    _ = _ in ComparableElem3 -> _ = _ in Anchor;
  Rename
    cp2 -> c-link;
End C-Link;

;; A triple uvc makes an association between a user and the viewport he is connected with
;; and the current component he is working with.
Adt C-Triple-uvc as CartesianProduct3(C-User, C-Viewport, C-Component);
  Morphism
    elem1 -> c-user;
    elem2 -> c-viewport;
    elem3 -> c-component;
    _ = _ in ComparableElem1 -> _ = _ in C-User;
    _ = _ in ComparableElem2 -> _ = _ in C-Viewport;
    _ = _ in ComparableElem3 -> _ = _ in C-Component;
  Rename
    cp2 -> c-triple-uvc;
End C-Triple-uvc;


Adt PN-C-Link as CartesianProduct3(Anchor, PN-C-Component, Anchor);
  Morphism
    elem1 -> anchor;
    elem2 -> pn-c-component;
    elem3 -> anchor;
    _ = _ in ComparableElem1 -> _ = _ in Anchor;
    _ = _ in ComparableElem2 -> _ = _ in PN-C-Component;
    _ = _ in ComparableElem3 -> _ = _ in Anchor;
  Rename
    cp2 -> pn-c-link;
End PN-C-Link;

;; A triple uvc makes an association between a user and the viewport he is connected with
;; and the current component he is working with.
Adt PN-C-Triple-uvc as CartesianProduct3(PN-C-User, PN-C-Viewport, PN-C-Component);
  Morphism
    elem1 -> pn-c-user;
    elem2 -> pn-c-viewport;
    elem3 -> pn-c-component;
    _ = _ in ComparableElem1 -> _ = _ in PN-C-User;
    _ = _ in ComparableElem2 -> _ = _ in PN-C-Viewport;
    _ = _ in ComparableElem3 -> _ = _ in PN-C-Component;
  Rename
    cp2 -> pn-c-triple-uvc;
End PN-C-Triple-uvc;

;; Various set used in the specification
Adt Set-Of-Anchors as Set(Anchor);
  Morphism
    elem  -> anchor;
    _ = _ in ComparableElem -> _ = _ in Anchor;
  Rename
    set -> set-of-anchors;
End Set-Of-Anchors;

Adt Set-Of-Components as Set(Component);
  Morphism
    elem -> component;
    _ = _ in ComparableElem -> _ = _ in Component;
  Rename
    set -> set-of-components;
End Set-Of-Components;
```

```
Adt Set-Of-C-Components as Set(C-Component);
  Morphism
    elem -> c-component;
    _ = _ in ComparableElem -> _ = _ in C-Component;
  Rename
    set -> set-of-c-components;
End Set-Of-C-Components;

Adt Set-Of-PN-C-Components as Set(PN-C-Component);
  Morphism
    elem -> pn-c-component;
    _ = _ in ComparableElem -> _ = _ in PN-C-Component;
  Rename
    set -> set-of-pn-c-components;
End Set-Of-PN-C-Components;

Adt Set-Of-Links as Set(Link);
  Morphism
    elem -> link;
    _ = _ in ComparableElem -> _ = _ in Link;
  Rename
    set -> set-of-links;
End Set-Of-Links;

Adt Set-Of-C-Links as Set(C-Link);
  Morphism
    elem -> c-link;
    _ = _ in ComparableElem -> _ = _ in C-Link;
  Rename
    set -> set-of-c-links;
End Set-Of-C-Links;

Adt Set-Of-PN-C-Links as Set(PN-C-Link);
  Morphism
    elem -> pn-c-link;
    _ = _ in ComparableElem -> _ = _ in PN-C-Link;
  Rename
    set -> set-of-pn-c-links;
End Set-Of-PN-C-Links;

Adt Set-Of-Triples-uvc as Set(Triple-uvc);
  Morphism
    elem -> triple-uvc;
    _ = _ in ComparableElem -> _ = _ in Triple-uvc;
  Rename
    set -> set-of-triples-uvc;
End Set-Of-Triples-uvc;

Adt Set-Of-C-Triples-uvc as Set(C-Triple-uvc);
  Morphism
    elem -> c-triple-uvc;
    _ = _ in ComparableElem -> _ = _ in C-Triple-uvc;
  Rename
    set -> set-of-c-triples-uvc;
End Set-Of-C-Triples-uvc;

Adt Set-Of-PN-C-Triples-uvc as Set(PN-C-Triple-uvc);
  Morphism
    elem -> pn-c-triple-uvc;
    _ = _ in ComparableElem -> _ = _ in PN-C-Triple-uvc;
  Rename
    set -> set-of-pn-c-triples-uvc;
End Set-Of-PN-C-Triples-uvc;

Adt Set-Of-C-Users as Set(C-User);
  Morphism
    elem -> c-user;
    _ = _ in ComparableElem -> _ = _ in C-User;
```

```
  Rename
    set -> set-of-c-users;
End Set-Of-C-Users;

Adt Set-Of-PN-C-Users as Set(PN-C-User);
  Morphism
    elem -> pn-c-user;
    _ = _ in ComparableElem -> _ = _ in PN-C-User;
  Rename
    set -> set-of-pn-c-users;
End Set-Of-PN- C-Users;



;; *** Structured basic abstract data types

;; === Comparable elements
;; An equal operation is defined as well as the usual equivalence relation properties
Parameter Adt ComparableElem;
Interface
  Use  Boolean;
  Sort elem;
  Operation
    _ = _ : elem elem -> boolean;
Body
  Theorems
     (x = x) = true;                                     ;; reflexivity
     (x = y) = true => (y = x) = true;                   ;; symmetry
     (x = y) = true & (y = z) = true => (x = z) = true;  ;; transitivity
    where
    x, y, z : elem;
End ComparableElem;

Parameter Adt ComparableElem1;
Interface
  Use  Boolean;
  Sort elem1;
  Operation
    _ = _ : elem1 elem1 -> boolean;
Body
  Theorems
     (x = x) = true;                                     ;; reflexivity
     (x = y) = true => (y = x) = true;                   ;; symmetry
     (x = y) = true & (y = z) = true => (x = z) = true;  ;; transitivity
    where
    x, y, z : elem1;
End ComparableElem1;

Parameter Adt ComparableElem2;
Interface
  Use  Boolean;
  Sort elem2;
  Operation
    _ = _ : elem2 elem2 -> boolean;
Body
  Theorems
     (x = x) = true;                                     ;; reflexivity
     (x = y) = true => (y = x) = true;                   ;; symmetry
     (x = y) = true & (y = z) = true => (x = z) = true;  ;; transitivity
    where
    x, y, z : elem2;
End ComparableElem2;

Parameter Adt ComparableElem3;
Interface
  Use  Boolean;
  Sort elem3;
  Operation _ = _ : elem3 elem3 -> boolean;
Body
```

```
  Theorems
    (x = x) = true;                                      ;; reflexivity
    (x = y) = true => (y = x) = true;                    ;; symmetry
    (x = y) = true & (y = z) = true => (x = z) = true;  ;; transitivity
    where
    x, y, z : elem3;
End ComparableElem3;

;; === Ordered elements
Parameter Adt OrderedElem;
Interface
  Use  Boolean;
  Sort elem;
  Operations
    _ < _ ,
    _ = _ : elem elem -> boolean;
Body
  Theorems
    (x < y) = true & (y < x) = true => (y = x) = true;  ;; anti-symmetry
    (x < y) = true & (y < z) = true => (x < z) = true;  ;; transitivity
    (x = x) = true;                                      ;; reflexivity
    (x = y) = true => (y = x) = true;                         ;; symmetry
    (x = y) = true & (y = z) = true => (x = z) = true;  ;; transitivity
    where
    x, y, z : elem;
End OrderedElem;

;; === Generic sets of comparable elements
Generic Adt Set(ComparableElem);
Interface
  Use  Natural, Boolean;
  Sort set;
  Generators
    []     : -> set;                        ;; empty set
    _ + _  : set elem -> set;               ;; add an element to a set
  Operations
    _ - _  : set elem -> set;               ;; remove an element to a set
    _ + _  ,
    _ - _  : set set  -> set;               ;; add/sub to sets
    _ = _  ,                                ;; equality, inclusion, strict inclusion
    _ <= _ ,
    _ < _  ,
    _ > _  ,
    _ >= _ : set set -> boolean;
    # _    : set -> natural;                ;; cardinality of a set
    _ inside? _ : elem set -> boolean;      ;; belonging predicate
Body
  Axioms
    ((s + e1) + e2) = ((s + e2) + e1);      ;; commutativity of the insertion (generator)
    ((s + e1) + e1) = (s + e1);             ;; no duplication in set (generator)
    s - []  = s;
    s - (s1 + e1)  = (s - e1) - s1;
    [] + s  = s;
    (s1 + e1) + s2 = (s1 + s2) + e1;
    (e1 = e2) = true  => ((s + e1) - e2) = s;
    (e1 = e2) = false => ((s + e1) - e2) = (s - e2) + e1;
    (s1 = s2) = (s1 <= s2) and (s2 <= s1);
    [] <= [] = false;
    (s + e1) <= [] = false;
    [] <= (s + e1) = true;
    (e1 = e2) = true  =>  (s1 + e1) <= (s2 + e2)  = s1 <= s2;
    (e1 = e2) = false => ((s1 + e1) <= (s2 + e2)) = (e1 inside? s2) and s1 <= (s2 + e2);
```

```
    s1 < s2  = s1 <= s2 and not (s2 <= s1);
    s1 >= s2 = s2 <= s1;
    s1 > s2  = s2 < s1;

    # [] = 0;
    # (s + e1) = succ (# s);

    e1 inside? [] = false;
    (e1 inside? (s + e2)) = (e1 = e2) or (e1 inside? s);
    where
      e1, e2    : elem;
      s, s1, s2 : set;
End Set;

;; === Cartesian Products

Generic Adt CartesianProduct2(ComparableElem1, ComparableElem2);
Interface
  Use  Boolean;
  Sort cp2;
  Generator
    < _ _ > : elem1 elem2 -> cp2;
  Operations
    fst _ : cp2 -> elem1;                       ;; first projection
    snd _ : cp2 -> elem2;                       ;; second projection
    _ = _ : cp2 cp2 -> boolean;
Body
  Axioms
    fst (<x y>) = x;
    snd (<x y>) = y;
    (<x y> = <u v>) = (x = u) and (y = v);
    where
      x, u : elem1;
      y, v : elem2;
End CartesianProduct2;

Generic Adt CartesianProduct3(ComparableElem1, ComparableElem2, ComparableElem3);
Interface
  Use  Boolean;
  Sort cp3;
  Generator
    < _ _ _ > : elem1 elem2 elem3 -> cp3;
  Operations
    fst _ : cp3 -> elem1;                       ;; first projection
    snd _ : cp3 -> elem2;                       ;; second projection
    thd _ : cp3 -> elem3;                       ;; third projection
    _ = _ : cp3 cp3 -> boolean;
Body
  Axioms
    fst (<x y z>) = x;
    snd (<x y z>) = y;
    thd (<x y z>) = z;
    (<x y z> = <u v w>) = (x = u) and (y = v) and (z = w);
    where
      x, u : elem1;
      y, v : elem2;
      z, w : elem3;
End CartesianProduct3;
```