Formal Development of Java Programs[1]

Giovanna Di Marzo Serugendo[1,2], Nicolas Guelfi[1],
[1] LGL-DI, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland
[2] CUI, University of Geneva, CH-1211 Genève 4, Switzerland
tel. +41 22 7057643 email: dimarzo@di.epfl.ch
tel. +41 21 6936768 email: guelfi@di.epfl.ch

[1]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 "Formal Methods for Concurrent Systems".

Abstract

The Java object-oriented programming language has been the subject of an important involvement from programmers and industry. Especially for applications related to the Web. The problem of such rapid penetration of Java programs into commercial product is that software engineers does not have any methodology and have to develop complex parallel applications. In this paper, we present, using a real (http://lglsun.epfl.ch/Team/GDM/DSGamma.html) but simple Web parallel application, a possible formal development methodology based on the stepwise refinement of CO-OPN/2 formal specifications. The application chosen is based on the Gamma paradigm, in which additions are realized across distributed multisets of integers maintained by local Java applets. CO-OPN/2 is an object-oriented specification formalism allowing the formal specification of abstract data types based on algebraic specification and of concurrent objects using a Petri net like approach. An interesting feature of CO-OPN/2 is that it gives a coherent solution for the expression of complex properties combining inheritance, concurrency and synchronizations. The refinement of formal descriptions for building progressively the Java program is presented in several refinement steps. This refinement process focuses on the distribution of data and treatments, and validation and verification are studied through the refinement of system properties.

Keywords: Software Engineering, Formal Methods, Petri Nets, Algebraic Specifications, Refinement, Concurrent and Distributed Systems, Object-Oriented Systems, Java, Web.

Contents

Chapter 1

Introduction

Java is a recent programming language [2, 10] which is widely used to develop distributed applications over the Web. Thus, software engineering techniques must be introduced in order to support the software life cycle for this application domain. The high complexity of such applications is due to the concurrent components and their coordination algorithm.

In this paper we provide a top-down engineering methodology for the development of Java based web parallel applications (written JPA for short) based on formal specifications. The advantage of a formal specification is that it allows a precise system description necessary when complex applications are developed and is useful for verification and validation purposes. We have chosen to use the CO-OPN/2 (Concurrent Object Oriented Petri Nets) specification formalism [6, 5] which is developed by our team and which provides many features adapted to the problem addressed in this paper[1]. More precisely, CO-OPN/2 integrates, in an object-oriented approach, Petri nets for the description of concurrent behaviors, and, algebraic specification for the specification of the structured data evolving in the Petri nets.

The formal development methodology proposed in this paper consists in: (1) starting with a set of informal application's requirements including validation objectives in terms of a set of desired properties; (2) building an initial CO-OPN/2 specification of the JPA based on the informal requirements and abstract enough to be as independent as possible of implementation constraints, this first abstraction must validate the desired properties; (3) do a series of refinement steps concerning both the formal specification and the properties. These refinement steps must reach a JPA together with property descriptions such that there is an evidence on their satisfaction by the JPA. Each of these refinement steps is guided by: the hardware and software environment and by the properties. Evidence (by formal or informal proof) must be given for the properties preservation between two consecutive refinement levels.

In order to control the properties evolution between two formal specifications, we must state clearly the consequences of each refinement step on each property. General property evolution rules can be given with respect to standard refinement rules, which depend on the refinement step itself as well as on the development strategy used. For example, the refinement of a transition by a sub Petri net will have known consequences on atomicity and data distribution will have consequences on data integrity. Some basic concepts of this methodology are inspired from [9, 15, 1]. Other approaches, more focused on Petri nets, can be found in [12, 17, 14].

We propose, in the case of Java based web parallel applications, to have at least, four distinct refinement steps based on an initial specification which provides an abstract view of the application, where the problem is neither distributed nor decentralized: (1) a first refinement step provides a decentralized view of the application, where the data is distributed but not yet the behavior; (2) a second refinement provides the application with a client/server architecture where both the data and the behavior are distributed; (3) a third refinement step introduces the communication layer, i.e. the TCP/IP based sockets; this step leads to a view *close to the code*, where the problem has been deployed on the Web but not yet implemented with Java; (4) a last refinement step produces the Java program, i.e. the applets, the server, the sockets, and all necessary threads handling the sockets.

In order to present this formal development methodology for Java based web parallel applications, we refer to a concrete example (whose implementation is running at the following address

[1] other object- oriented specification formalisms could have been chosen with an adaptation of our methodology. One can refer to [4] to find a comparison of object-oriented specification formalisms including CO-OPN/2.

http://lglsun.epfl.ch/Team/GDM/DSGamma.html).

The chosen application is based on the Gamma paradigm [3] (a programming paradigm integrating chemical reaction concepts). It is called *DSGamma* for "Distributed Gamma". The implementation is using Java applets. A Gamma-like addition is realized on distributed multisets of integers maintained by local Java applets. Each applet locally maintains a multiset of integers and a user may enter new integers in the system with the help of a graphical user interface provided by the applet. The global multiset is given by the union of all the multisets maintained by the applets; chemical reactions are realized by several cooperative Java threads which collect pairs of integers, add them and put them in the multiset maintained by one of the applets.

Concerning verification and validation, the following property is chosen and studied during each refinement step: *if there is only one integer in the multiset, then it must be the result of a parallel computation of the sum of all the integers entered by all the users since the application launch.* Of course, verification and validation activities will address a more complex set of properties, but it is not the scope of this paper.

The plan of the paper is the following. Firstly, we introduce the basic concepts of the specification formalism CO-OPN/2; secondly we present in more details the refinement of CO-OPN/2 specifications and summarize our formal development methodology; thirdly we present in details the methodology by applying it in four refinement steps to the chosen distributed application; finally we explain how some features of the Java programming language are specified with CO-OPN/2.

Chapter 2

The CO-OPN/2 Specification Formalism

2.1 Concepts

CO-OPN/2 is an hybrid specification formalism based on algebraic specifications [16] and Petri nets which are combined in a way that is similar to algebraic nets [13]. The former is used to describe the data structures and the functional aspects of a system, while the latter allows to model the system's concurrent features. However, both these formalisms are not suitable to specify "in the large". To compensate for Petri nets' lack of structuring capabilities, 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.

In order to overcome some limitations of the first version of CO-OPN [7], CO-OPN/2 [6] introduces some notions peculiar to object-orientation such as the notions of class, inheritance, and subtyping. For the sake of homogeneity regarding the notion of subtyping, order-sorted algebraic specifications have been adopted for the description of data structures.

The interest of a formal specification language is its mathematical foundation which allows a precise and rigorous approach. Especially it provides a formal syntax and semantics. Concerning CO-OPN/2 the formal semantics is given in terms of concurrent transition systems expressing all the possible evolutions of objects' states. As we have adopted a step semantics approach, state changes are associated to a multiset of events which are simultaneously executable. Thus, the full concurrency of the specification is expressed in the semantics including intra-concurrency (between services of an object) and inter-concurrency (between services of several objects). The complete semantics of CO-OPN/2 can be found in [5].

Object and Class. An object is considered as an independent entity composed of an internal state and which provides some services to the exterior. The only way to interact with an object is to ask for its services; the internal state is then protected against uncontrolled accesses. Our point of view is that encapsulation is an essential feature of object-orientation and there should be no way of violating it.

CO-OPN/2 defines an object as being an encapsulated algebraic net in which the places compose the internal state and the transitions model the concurrent events of the object. A place consists of a multi-set of 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 to the outside, while the latter compose the internal behaviors of an object. Contrary to 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 their potential dynamic evolution in terms of the number of objects they may include. Thus, the dynamic creation of objects is a major objective. A class describes all the components of a set of objects and is considered as an object template. Thus, all the objects of one class have the same structure.

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 joined by 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 service request to a third object.

Concurrency. Intuitively, 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 of true concurrency which is not the case of interleaving semantics. A set of method calls can be concurrently performed on the same object.

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. An order-sorted algebra of object identifiers is constructed in order to reflect the subtyping relation which is established between the classes types, i.e. two carrier sets of object identifiers are related by inclusion if, and only if, the two corresponding types are related by subtyping. Since object identifiers are algebraic values, it is possible to define data structures which are built upon object identifiers, e.g. a stack or a queue of object identifiers. Obviously, the places of algebraic nets may contain object identifiers.

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

Our subtyping relationship is based upon the strong version of the substitutability principle. This principle implies that, in any context, any class instance of a type may be substituted for a class instance of its super-type while the behavior of the whole system remains unchanged.

Both inheritance and subtyping relationships must be explicitly given but the respective hierarchies generated by these relationships do not necessarily coincide.

2.2 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. Both 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, of the behavior and of the state of the objects. In the following we do not describe the algebraic abstract data type modules as they follow in an intuitive way the classical approaches.

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 will not be 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 interface definition. The **`Type`** field declares the name of the instances type which is used whenever an object identity has to be defined. This field has been introduced in order to avoid any confusion between the name of a module or a class and the name of its type, especially in cases where inheritance and subtyping are required. Both names are often very similar but address two different concepts: module names are used for inheritance relationship, while type names are used for subtyping relationship. 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 mix-fix 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 creation methods which create and initialize the objects; these methods may be used only once for a given object. A predefined creation method **`create`** is provided when the **`Creation`** field is empty or absent. The **`Body`** section includes a **`Use`** section and some internal or spontaneous transitions declared under

[1] Textual and graphical descriptions of the CO-OPN/2 specifications, given in the next section, may help the reader to understand the meaning of the various keywords introduced in this subsection.

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

A behavioral axiom is established as follows

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

where *Cond* is an optional condition imposed on 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 in the different places composing the net. Finally, all the variables used within the body section are grouped together in the **where** field.

Chapter 3

Formal Refinement

The formal development of Java based web parallel applications (JPA) should follow a specific engineering life cycle based on formal specifications. We propose to choose CO-OPN/2 as the specification formalism since, as it is shown in this paper, its features are well adapted to this end. Nevertheless, another specification formalism could have been chosen and the reader can refer to the comparison of object oriented specification formalisms presented in [4] (which includes CO-OPN/2). Our approach could partly be translated in some other specification formalism.

3.1 Formal Refinement applied to CO-OPN/2

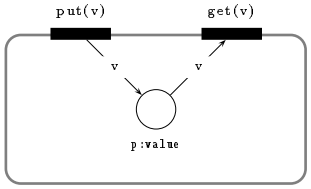
The refinement of CO-OPN/2 specifications concerns the data structure part as well as the behavior part. The main problem is to control the influence on the system properties made by these refinements. In order to present clearly the different types of refinements, we propose to follow a step of refinement between two CO-OPN/2 specifications of a simple buffer modeling a site storing products. Table 3.1 lists the initial specification and two possible refinements. Each specification is described by (1) the ADT used in the specification, by (2) the class names, and types names appearing in the external behavior part of the specification, and by (3) the class names, and types names appearing in the internal behavior part of the specification.

The initial specification describes a buffer in which it must be possible to put and get three kinds of products (`A, B, C` specified in the algebraic specification given in Figure 3.1). The system must be fully parallel. For example, depending on the resources available, it must be possible to execute simultaneously `put(A), put(A), put(B), get(C), get(A)`. This parallelism is expressed using the semantics of CO-OPN/2 which follows the basic Petri net semantics. The initial CO-OPN/2 specification of such a buffer is given by the class `Site` in Figure 3.2, in the graphical and textual forms. It defines a static object `S` of type `site`.

```
Adt Value;
Interface
  Sort value;
  Generators
    A , B , C : -> value;
End Value;
```

Figure 3.1: The Algebraic Specification of the Type `value`

```
Class Site;
Interface
  Use Value;
  Type site;
  Object S : site;
  Methods
    put _ ,  get _ : value;
Body
  Place p : value;
  Axioms
    put v :: -> p v;
    get v :: p v -> ;
    where  v : value;
End Site;
```

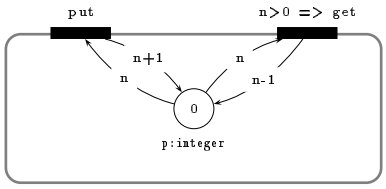


Class Site

Figure 3.2: The Initial CO-OPN/2 Site Specification

The refinement of this specification is done according to the following implementation constraint: each product kind must be managed in different distributed sites. Now, let us see a CO-OPN/2 refinement of the system. Each site managing a product kind is modeled by an object instance of the class `RefSite`, described in Figure 3.3. A site only keeps the number of products available in a site. We have then three objects, `S_A`, `S_B`, `S_C`, distinguished by their name and associated to each product kind.

```
Class RefSite;
Interface
  Use Integer;
  Type site;
  Object S_A , S_B , S_C : site;
  Methods
    put , get;
Body
  Place p : integer;
  Initial p 0
  Axioms
    put :: p n -> p n+1;
    get ::
      n > 0 => p n -> p n-1;
    where  n : integer;
End RefSite;
```



Class RefSite

Figure 3.3: The refined CO-OPN/2 Site Specification (Version 1)

We must explicitly complete the `RefSite` specification with the necessary information in order to let it become a refinement of the initial `Site` specification. Thus, according to the initial specification, we must at least provide the services `put(v), get(v)` with `v` of type `value`. They are no more available in the refined `RefSite` specification. Thus, we introduce an object of class `RefStep`, given in Figure 3.4, which has only two methods `put(v), get(v)`. The method `put(v)` has only one external behavior and the internal reaction depends on the value of `v`. A call of `put(v)` on `RefStep` is immediately, and in an atomic way, followed by a call of `put` in one of the `RefSite` objects depending on the value of `v`. Indeed, the CO-OPN/2 `int` transitions actually call `put` of `S_A`, `S_B`, or `S_C` objects if `v` has value `A`, `B` or `C` respectively. `get(v)` has three exclusive external behaviors distinguished by the value of `v`.

```
Class RefStep;
Interface
  Use Value , RefSite ;
  Type refstep;
  Objects step1 : refstep;
  Methods put _ , get _ : value ;
Body
  Transitions int;
  Place p : value;
Axioms
    put(v) :: -> p v ;

    int with S_A.put :: p A -> ;
    int with S_B.put :: p B -> ;
    int with S_C.put :: p C -> ;
    get(v) with S_A.get ::
      v = A => -> ;
    get(v) with S_B.get ::
      v = B => -> ;
    get(v) with S_C.get ::
      v = C => -> ;
End RefStep;
```

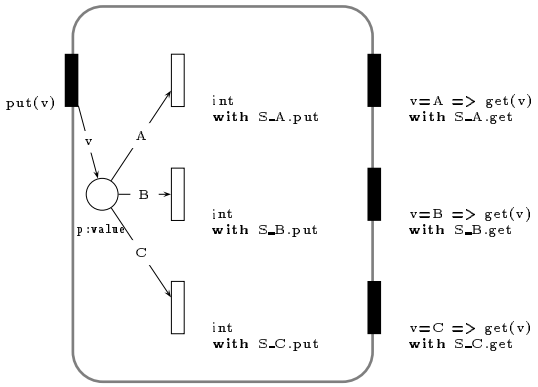


Figure 3.4: CO-OPN/2 Refinement (Version 1)

The refinement (version 1) of the initial specification is given by CO-OPN/2 classes `RefStep` and `RefSite`. Now, if we try to prove that the `RefStep` specification is actually a refinement of the initial specification, then we will see that some behaviors are no more available. Indeed, in the initial specification, it was possible to have several `get(A)`, for instance, occurring simultaneously. In the refinement (version 1), it is not possible to have several `get` of a `RefSite` object occurring simultaneously. The resource `n` is unique, and is requested entirely by each `get`, thus only one `get` can be executed at once. Thus, this behavior is propagated to the `RefStep` object, it is not possible to have several simultaneous `get(A)`. For what is concerning the `put(A)` method: the problem raised by the unique `n` has been postponed to the `int` internal transitions, which cannot actually occur several times simultaneously for the same value, and thus, simultaneous `put(A)` services are allowed in both the initial and the refined (version 1) specification.

If the properties, which must be verified until implementation, are concerned by this restriction of behaviors, we *must* redefine this refinement step. Several solutions are then possible. The one given in Figure 3.5 and 3.6, keeps in each site the independence of each product item (class `RefSite` becomes class `NewRefSite`). The unique integer `n` of class `RefSite` has been changed by `n` independent integers (of value `1`), thus each product item can be accessed separately and concurrently. The `NewRefSite` class keeps the original behavior modeling made in the first version of the specification.

```
Class NewRefSite;
Interface
  Use Integer;
  Type site;
  Object S_A, S_B, S_C : site;
  Methods
    put ,  get;
Body
  Place p : integer;
  Axioms
    put v :: -> p 1;
    get v :: p 1 -> ;
End NewRefSite;
```

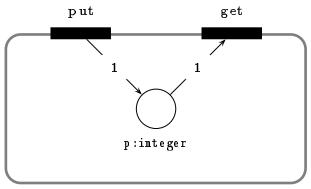


Figure 3.5: The refined CO-OPN/2 Site Specification (Version 2)

```
Class NewRefStep;
Interface
  Use Value , NewRefSite ;
  Type refstep;
  Objects step1 : refstep;
  Methods put _ , get _ : value ;
Body
  Transitions int;
  Place p : value;
Axioms
    put(v) :: -> p v ;

    int with S_A.put :: p A -> ;
    int with S_B.put :: p B -> ;
    int with S_C.put :: p C -> ;
    get(v) with S_A.get ::
      v = A => -> ;
    get(v) with S_B.get ::
      v = B => -> ;
    get(v) with S_C.get ::
      v = C => -> ;
End NewRefStep;
```

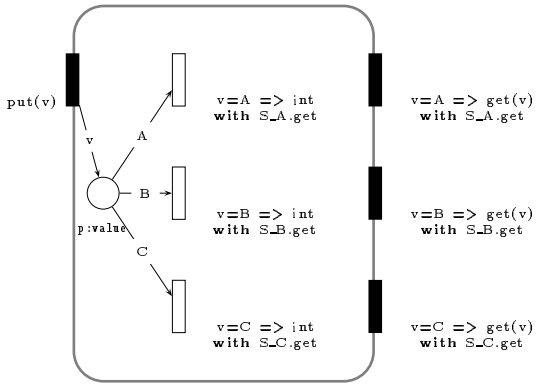


Figure 3.6: CO-OPN/2 Refinement (Version 2)

The refinement (version 2) of the initial specification is given by CO-OPN/2 classes `NewRefStep` and `NewRefSite`. The `NewRefStep` class is identical to the `RefStep` class, except that it uses the `NewRefSite` class instead of the `RefSite` class.

Table 3.1 summarizes the initial specification and the two refinements (version 1 and version 2) described above. The initial specification is given by the CO-OPN/2 class `Site` which defines the `site` type, and uses the `Value` algebraic specification. The refinement (version 1) defines the `RefStep` class, which uses the `RefSite` class, and `Value` ADT. The type provided by `RefStep` is called `refstep`, and the type provided by `RefSite` is `site` (as in the initial specification). The refinement (version 2) defines the `NewRefStep` class, which uses the `NewRefSite` class, and `Value` ADT. The type provided by `NewRefStep` is called `refstep`, and the type provided by `RefSite` is `site`.

	Initial Specification	Refinement (Version 1)	Refinement (Version 2)	
ADT	Value	Value	Value	
Class	Site	RefStep	NewRefStep	External
Type	site	refstep	refstep	Behavior
Class		RefSite	NewRefSite	Internal
Type		site	site	Behavior

Table 3.1: Initial Specification and two possible Refinements

3.2 Methodology

We now sketch the methodology followed in order to develop Java parallel web applications using formal refinement steps based on the CO-OPN/2 specification formalism.

The first step, starts with a set of informal application's requirements which must include the definition of validation objectives in terms of a set of desired properties.

The second step is to build a first formal specification of the JPA based on the informal requirements. In the case of JPA, the first abstraction must be as independent as possible of implementation constraints, which ones will be gradually integrated. This first abstraction must validate the desired properties.

The third step is in fact a series of similar refinement steps concerning both the formal specification and the properties. These refinement steps must reach a JPA together with properties descriptions such that there is an evidence on their satisfaction by the JPA. Each of these refinement steps is guided by: (1) the desired final implementation, and (2) the properties. Evidence (by formal or informal proof) must be given for the correspondence of properties between two consecutive refinement levels. For this reason, a refinement step must be associated with a refinement of the properties description. The proposed refinement have known consequences on the properties, like atomicity, parallelism, value domains, ... In order to control the properties life cycle we must establish the possible refinements allowed by the formalism itself (like refining a Petri net transition by a sub Petri net, or refining arrays by algebraic sets or Petri net places).

Chapter 4

Refinement Steps: From CO-OPN/2 to Java

This section presents a real Java based Web application, and how a formal development has been applied to it. The formal development provides stages (1) to (6) below: (1) the informal requirements of the application together with a set of expected properties; (2) a first step which provides an abstract view of the application by the means of an initial formal CO-OPN/2 specification, called **I**, where the problem is neither distributed nor decentralized; (3) a first refinement step, **R1**, which provides a decentralized view of the application, where the data is distributed but not yet the behavior; (4) a second refinement step, **R2**, which provides the application with a client/server architecture, at this step both the data and the behavior are distributed; (5) a third refinement step, **R3**, which introduces the communication layer, i.e. the TCP/IP based sockets, this step leads to a view *close to the code*, where the problem has been deployed on the Web but not yet implemented with Java; (6) a fourth, and last, refinement step, which produces the Java program, i.e. the applets, the server, the sockets, and all necessary threads handling the sockets.

We formalize our problem at each of the refinement steps using both (1) a CO-OPN/2 specification describing the behavior of the considered application, and (2) one informally expressed property the application has to fulfill. We intend to show how the refinement process is conducted on both the behavior and on this property. More precisely, a refinement step is conducted in two stages: firstly we refine the CO-OPN/2 specification, and secondly we refine the property. The newly obtained property depends on both the property obtained at the previous step and on the newly obtained CO-OPN/2 specification.

4.1 Informal Requirements and Properties

The Gamma paradigm [3] advocates a way of programming which is close to the chemical reactions. The Gamma paradigm consists in applying one or more chemical reactions on a multiset. A chemical reaction usually removes some values from the multiset, computes some results and inserts them into the multiset. In order to illustrate the Gamma paradigm and to introduce our application, we consider the following example: computing the sum of the integers present in a multiset. A multiset of integers is a bag of integers, with possibly multiple occurrences of the same integer in the bag. For convenience, we note in the same manner a set and a multiset (i.e. we use { and } to delimit the unordered list of integers present in the multiset).

Illustration 4.1.1 *Consider the following multiset:* $\{1, 2, 2, 3\}$. *We intend to compute the sum of all these integers according to the Gamma paradigm. A chemical reaction removes two integers from the multiset, adds them, and inserts their sum in the multiset. A chemical reaction can occur alone or concurrently with other chemical reactions, possibly of the same type. Consider two chemical reactions working concurrently on the multiset* $\{1, 2, 2, 3\}$, *as shown in figure 4.1. Firstly one of them removes* $3, 2$ *while the other one removes* $1, 2$; *they compute the sum and insert* 5 *and* 3 *respectively in the multiset. The multiset has now the following value:* $\{3, 5\}$. *Secondly only one of them can work, because there are only two integers remaining in the multiset, it removes* $3, 5$ *and inserts* 8, *which is the final result, in the multiset and no more chemical reactions can occur.*

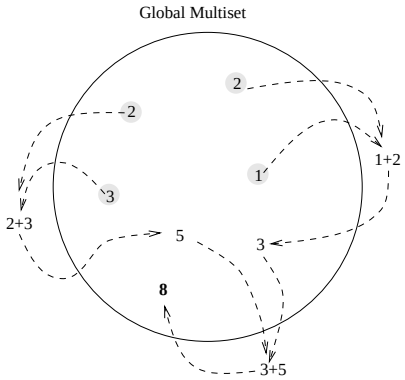


Figure 4.1: Addition according to the Gamma paradigm

4.1.1 Informal Requirements

We intend to develop an application allowing several users to insert integers into a possibly distributed multiset. According to the Gamma paradigm, chemical reactions are applied on the multiset, they have to perform the sum of all the integers entered by all the users. The system made of the users, the multiset and the chemical reaction is called the DSGamma (Distributed Gamma) system.

We present the informal requirements in two parts. The first one presents the system operations which must be provided to the users, and the second one, the details about the data and internal computations.

System Operations

[1] A new user can be added to the system at any moment; **[2]** A user may add new integers in the system, at any moment, between his entering time and his exit time; **[3]** At any moment, the application can give a partial view of the state of the multiset; **[4]** A user may exit the system provided he has entered the system.

State and Internal Behavior

[5] The integers entered by the users are stored in a multiset; **[6]** The application realizes the sum of all the integers entered by all the users; **[7]** The sum is performed by chemical reactions according to the Gamma paradigm; **[8]** A chemical reaction removes two integers from the multiset, adds them, and inserts the sum into the multiset; **[9]** There is only one type of chemical reactions, but several of them can occur simultaneously and concurrently on the multiset; **[10]** A chemical reaction may occur as soon as the state of the multiset is such that the chemical reaction can occur, i.e. as soon as there are at least two integers in the multiset.

4.1.2 Properties

In order to observe the property preservation during refinement, we will establish one fundamental property and follow it during each refinement step.

Property: *if there is only one integer in the multiset, then it must be the result of a parallel computation of the sum of all the integers entered by all the users since the application launch.*

4.2 Initial Specification: Centralized View

The external behavior of the application is modeled with CO-OPN/2 methods, while its internal behavior is modeled with CO-OPN/2 transitions and places.

4.2.1 CO-OPN/2 Specification

The initial CO-OPN/2 specification, **I**, gives a centralized view of our application. It is given by `DSGammaSystem` class depicted by figure 4.2. It is the most abstract view of the application: there is a global multiset with several chemical reactions occurring concurrently on it. We have

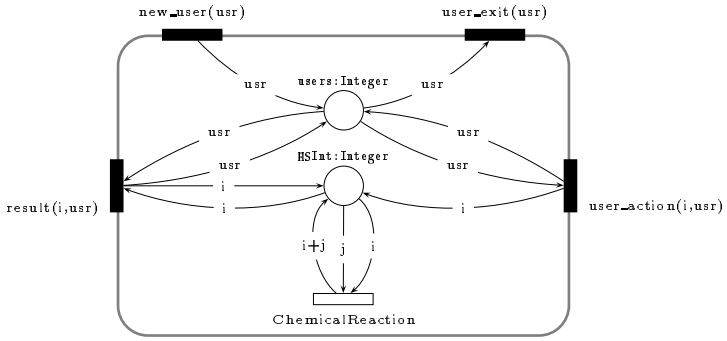


Figure 4.2: The Initial CO-OPN/2 specification, **I**

a non distributed data (the multiset), several processes (the chemical reactions), and each process, considered separately, is not distributed. The CO-OPN/2 textual specification of the initial specification **I** is given by appendix A.

System Operations:

The four CO-OPN/2 methods, `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`, and `user_exit(usr)` specify the four services, system operations **[1]** to **[4]**, that the system furnishes to the outside.

The `new_user(usr)` method inserts the users' identity, `usr`, into the `users` place. It is worth noting, that a user may add itself several times into the system under the same identity, because no check of a possibly previous entry of the same user is performed.

The `user_action(i,usr)` method checks if `usr` has already entered the system (i.e. if `usr` is in the place `users`), and inserts the `i` value, in the multiset `MSInt`. If the user `usr` has not yet entered the system, the method cannot be fired, thus the `i` value is not inserted in the multiset[1].

The `result(i,usr)` method checks if `usr` has already entered the system, and reads one integer `i` in the place `MSInt`.

The `user_exit(usr)` method removes `usr` (one time) from the `users` place if it is in. If `usr` was the last reference of the user, then, it becomes impossible for this user to ask for other system operation than the `new_user` one.

State:

A multiset of integers stores the integers entered in the system by all the users. The CO-OPN/2 `MSInt` place, of type `Integer`, models this multiset (the type `Integer` is specified using algebraic specifications as equivalent to natural numbers). Due to the CO-OPN/2 Petri net semantics of places, the content of a place is always given by a multiset.

The requirements impose that a user may enter integers, get a result, and exit the system provided he has previously entered the system. For this reason, it is necessary to store the identity of the users. The CO-OPN/2 place `users` of type `Integer` stores the identity of the users as integers.

Internal Behavior:

The CO-OPN/2 `ChemicalReaction` transition models the chemical reaction. It takes two integers `i,j` from the `MSInt` place, and inserts their sum `i+j` in `MSInt`. Due to the CO-OPN/2 semantics an object's transition is fired as soon as its pre-condition is fulfilled, and as longer as the pre-condition is fulfilled, in the meanwhile, no method of this object can be fired.

Illustration 4.2.1 *Example of figure 4.1 is modeled by the initial specification* ***I*** *of figure 4.2 in the following way. Three users,* `usr1`, `usr2`, `usr3` *enter the four integers* $\{1, 2, 2, 3\}$. *For instance, by the means of the methods* `user_action(1,usr1)`, `user_action(2,usr1)`, `user_action(2,usr2)`,

[1] recall that if one element needed by a method or transition event is not available than its execution is impossible.

and ***user_action(3,usr3)*** *occurring simultaneously.* ***usr1*** *enters two integers,*$\{1,2\}$, ***usr2*** *and* ***usr3*** *enter one integer each,* 2 *and* 3 *respectively. Thus,* ***users*** *contains* ***usr1,usr2,usr3*** *and* ***MSInt*** *contains integers* $\{1,2,2,3\}$. *The* ***ChemicalReaction*** *transition can be fired immediately, it removes randomly two pairs of integers (it is the maximum of available pairs in* ***MSInt****). For instance, pair* $\{2,3\}$ *and pair* $\{1,2\}$, *and inserts their sum,* 5 *and* 3 *respectively in* ***MSInt****. The firing of* ***ChemicalReaction*** *is not finished because two integers are available in* ***MSInt****. Thus,* ***ChemicalReaction*** *removes the pair* $\{5,3\}$ *from* ***MSInt*** *and inserts their sum* 8.

Since ***ChemicalReaction*** *is firable, it is not possible to have one of the methods be firable. As soon as* ***ChemicalReaction*** *is no longer firable it becomes possible for a user to call the* ***result(i,usr)*** *method for instance. As illustrated above, only one integer lies in* ***MSInt*** *as soon as* ***ChemicalReaction*** *has been fired, thus* ***result(i,usr)*** *returns the current sum. The* ***user_exit(usr)*** *method does not interfere with the computation of the sum, thus a user leaving the system before other users have finished entering integers does not affect the final result.*

4.2.2 Properties

We consider now the property stated on section 4.1.2, and we check how it has evolved during this step. We will informally prove, how and why, the property is fulfilled.

The computation of the result is realized by the transition `ChemicalReaction`. Due to the CO-OPN/2 semantics, the `ChemicalReaction` transition can be fired simultaneously several times, provided the pre-condition is fulfilled for each occurrence, and the firing is repeated until the pre-condition is no longer fulfilled, i.e. if only one integer remains in the multiset. At the beginning of the system, no integer is present in `MSInt`. We assume that n integers enter the system simultaneously (several simultaneous `user_action(i,usr)`). Thus, $[n/2]$ pairs[2] of integers are present in `MSInt` and `ChemicalReaction` will be fired $[n/2]$ times simultaneously, after these firings, there will remain in `MSInt` $m = n - [n/2]$ integers. The firings of `ChemicalReaction` proceeds similarly on these m integers, and stops when $m = 1$. The specification provides the following: (1) after a firing of `user_action(i,usr)` only one integer remains in `MSInt`; (2) the computation is realized fully in parallel over all available pairs of integers present in `MSInt`; (3) the remaining integer is the sum of the integers present in `MSInt` before the firing of `ChemicalReaction`.

4.3 First Refinement: Data Distribution

The initial specification, **I**, provides a centralized view of the application. As we intend to obtain an implemented application distributed over the Web, it is now necessary to introduce distributivity in the specification. This refinement step, called **R1**, is concerned with data distributivity. The CO-OPN/2 textual specification of refinement **R1** is given by appendix B.

4.3.1 Refinement Process

The informal view of the DSGamma system, is given by figure 4.3. The multiset of integers is physically distributed over several different locations. We call *local multiset*, the portion MS_i of the multiset present in a given location, and we call the *global multiset*, the multiset obtained by the union of all the local multisets. As before, the purpose of the system is to compute the sum. We give in Figure 4.3 an illustration of these parallel computations over the distributed multisets MS_i which compute the result 8.

The refinement process has to preserve the system operations. Thus, the overall specification of the DSGamma system must provide the four methods `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`, and `user_exit(usr)`[3]. As the global multiset is splitted over several local multisets we must redefine the system operations and internal behaviors.

The global multiset is logically given by the union of several local multisets. Thus, instead of one global multiset, we have now several local multisets, as many as the number of users. We chose to map each user with a local multiset. Two places, `MSInt` and `MSIntToEmpty`, store pairs `<usr,bag>`, where `usr` is a user's identity, and `bag` is the user's corresponding local multiset. `MSInt` place is used when the users are participating in the system, while `MSIntToEmpty` is used when a user leaves the system in order to dispatch his integers over the remaining user's multisets.

[2] $[n/2]$ stands for the integer part of the real number $n/2$.

[3] this will be the case for all the next refinements.

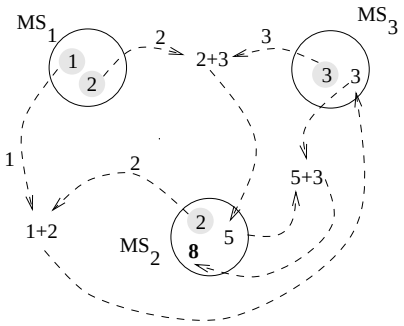


Figure 4.3: Distributed Gamma-like Addition

A chemical reaction removes two integers from one or two local multisets, adds them, and inserts the result into a local multiset. This kind of chemical reaction can be realized in 8 different ways: each one describes an atomic way of removing two integers from one or two `<usr,bag>` pairs and inserting one integer into a `<usr,bag>` pair.

Class DSGammaSystem

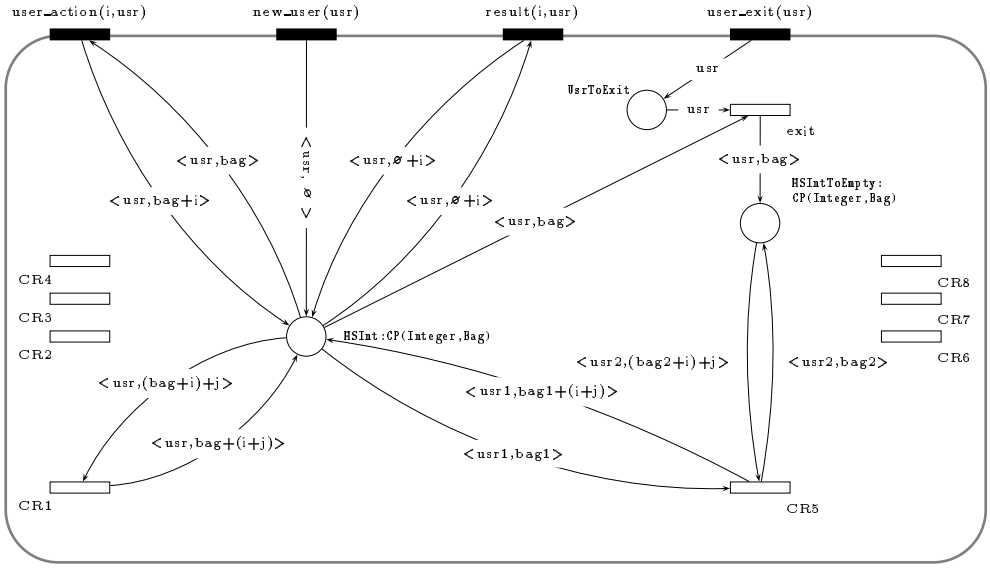


Figure 4.4: First Refinement **R1**: Data Distribution

4.3.2 CO-OPN/2 Specification

The CO-OPN/2 specification of our application with distributed multisets is given by the `DSGammaSystem` class depicted by figure 4.4. We have a distributed data (the local multisets) and several concurrent processes (the chemical reactions).

System Operations:

Four CO-OPN/2 methods, also provided by the initial specification **I**, `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`, and `user_exit(usr)` specify the four services that the system furnishes to the outside.

The `new_user(usr)` method inserts `<usr,∅>` into the `MSInt` place. A new user joins the system with an empty bag, representing a local multiset. As for the initial CO-OPN/2 specification **I**, a user may add itself several times in `MSInt`.

The `user_action(i,usr)` method checks if `usr` has already entered the system, i.e. removes the pair `<usr,bag>` from the place `MSInt`, and inserts the `i` value into `bag`, i.e. inserts the pair

<usr,bag+i> into MSInt. bag+i stands for a new bag made of the union of bag[4] and the set {i}. This method cannot be fired if usr has not already joined the system.

The result(i,usr) can be fired iff the bag of user usr contains exactly one element i (i.e. ∅ + i). It is worth noting that due to the CO-OPN/2 semantics, after each firing of the chemical reactions, only one integer remains in the global multiset.

The user_exit(usr) method inserts usr into the UsrToExit place. The system has the knowledge that usr wants to leave the system, the actual processing of leaving the system will happen later.

State:

The MSInt place stores the local multiset of users currently in the system, while the MSIntToEmpty place stores the local multiset of users wishing to leave the system. More precisely, they are of type CP(Integer,Bag), an algebraic specification for Cartesian products of Integers and Bags; they store pairs <usr,bag>. Thus we handle a multiset as a whole, and not through its elements as it is the case in the initial specification **I**.

Internal Behavior:

The exit transition actually realizes the exit of a user. It removes the pair <usr,bag> from the MSInt place and inserts it into the MSIntToEmpty place. The integer of bag will be involved in chemical reactions, and the sum will be put to another user. As the user is tightly coupled with a local multiset, it is necessary to introduce here a treatment for dispatching his values. After having exited the system, a user may no longer enter new integer, nor get the result, nor exit the system, unless it reenters the system, and the system itself cannot add integers in the user's local multiset.

Four ways of realizing the chemical reaction have been defined on MSInt only: (CR1) two integers i,j are removed from the same local multiset given by <usr,(bag+i)+j>, and their sum is inserted in this local multiset, the new state of the local multiset is given by <usr,bag+(i+j)>. (bag+i)+j stands for the bag which is given by the union of bag bag+i with the set {j}. bag+(i+j) is given by the union of bag bag, with the set {i+j}; (CR2) the i,j integers are removed from two different local multisets, and their sum is added to one of them; (CR3) the i,j integers are removed from the same local multiset and their sum is inserted into another local multiset; (CR4) the i,j integers are removed from two different local multisets, and their sum is added to a third local multiset.

Four ways of realizing the chemical reaction have been defined on both MSInt and MSIntToEmpty. They are basically the same as those described above, except the fact that they have to remove integers from local multisets stored in the MSIntToEmpty place, and they have to insert integers into local multisets stored in the MSInt place. These four ways specify the fact that once a user has decided to leave the system, then his local multiset has to be emptied, no new integers may be inserted in his local multiset: (CR5) two integers i,j are removed from the same local multiset in MSIntToEMpty, and their sum is added to another local multiset in MSInt; (CR6) the i,j integers are removed from two different local multisets, one in MSInt, and another one in MSIntToEMpty, their sum is added to the multiset of MSInt; (CR7) the i,j integers are removed from two different local multisets, one in MSInt, and another one in MSIntToEMpty, their sum is added to a third local multiset of MSInt; (CR8) the i,j integers are removed from two different local multisets of MSIntToEMpty, and their sum is added into a third local multiset of MSInt.

For simplicity purpose, figure 4.4 depicts only the behavior of CR1 and CR5 described above.

Illustration 4.3.1 *Example of figure 4.3 is modeled by refinement **R1** of figure 4.4 in the following way. Three users, usr1, usr2, usr3, enter the four integers* $\{1, 2, 2, 3\}$. *For instance, by the means of the methods user_action(1,usr1), user_action(2,usr2), user_action(3,usr3), occurring simultaneously, and user_action(2,usr1) occurring later. It is worth noting that due to the model of the user_action(i,usr) method, a single user cannot enter two or more integers simultaneously. Indeed, the user_action(i,usr) requires an access to the whole usr's bag in order to insert only one integer i. For this reason, the three above users cannot enter simultaneously four integers (as it was the case in the initial specification **I**), they can enter at most three integers simultaneously (one per user), and only after that they can enter other integers.*

[4] the specification of the type bag is made using an algebraic specification which defines an empty bag and usual operations.

Firstly three integers enter simultaneously the system, thus **MSInt** *has the following values:* <**usr1**,{1}>, <**usr2**,{2}>, <**usr3**,{3}>. *As soon as two or more integers are available in the union of* **bags** *of* **MSInt**, *the* **CRi** *can be fired. For instance* **CR4** *can occur, it removes* 1 *from the* **usr1**'*s bag, and* 2 *from* **usr2**'*s bag, and inserts their sum,* 3 *into* **usr3**'*s bag. For this, it removes* <**usr1**,{1}>, <**usr2**,{2}> *and* <**usr3**,{3}> *from* **MSInt** *and inserts* <**usr1**,∅>, <**usr2**,∅> *and* <**usr3**,{3,3}> *into* **MSInt**. *As two integers at least are still present in the global multiset, the chemical reaction can occur. For instance* **CR3** *can occur, it removes the two integers from the* **usr3**'*s bag and inserts their sum into the* **usr2**'*s bag. For this, it removes* <**usr2**, ∅> *and* <**usr3**,{3,3}> *from* **MSInt** *and inserts* <**usr2**,{6}> *and* <**usr3**,∅>. *At this point only one integer remains in the global multiset, thus the chemical reaction cannot occur. Note that the remaining integer is the sum of the integers present in the global multiset just before the firing of the chemical reactions.* **MSInt** *has the following values:* <**usr1**,∅>, <**usr2**,{6}>, <**usr3**,∅>.

The methods of a given CO-OPN/2 object cannot be fired as long as one of its internal transitions can be fired. As soon as no internal transition can be fired, a method can be fired. Thus, **user_action(2,usr1)** *can be fired, and* **MSInt** *has the following values:* <**usr1**,{2}>, <**usr2**,{6}>, <**usr3**,∅>. *The chemical reaction occurs, for instance,* **CR2** *removes* 2 *from* **usr1**'*s bag, and* 6 *from* **usr2**'*s bag and inserts* 8 *into* **usr2**'*s bag.* **MSInt** *has the following values:* <**usr1**,∅>, <**usr2**,{8}>, <**usr3**,∅>.

4.3.3 Properties

The computation of the result is realized by transitions `CR1` to `CR8`. Due to the CO-OPN/2 semantics, these `CRi` transitions can be fired simultaneously and each of them can be fired simultaneously several times, provided the pre-condition is fulfilled. The firing is repeated until the pre-condition is no longer fulfilled for none of the `CRi`, i.e. if only one integer remains in the global multiset. At the beginning of the system, only empty bags are present in `MSInt`. We assume that n integers enter the system simultaneously (several simultaneous `user_action(i,usr)`). These n integers are distributed over exactly n bags, because each occurrence of `user_action(i,usr)` has to access a whole bag in order to insert one integer. Note that there may be some more empty bags, for the users who enter no integers. In the initial specification **I**, $[n/2]$ pairs of integers were immediately involved in chemical reactions. This is always the case as the n bags are accessed concurrently by the `CRi` internal transitions. The firing of the `CRi` transitions proceeds until none of them can be fired, thus only one integer remains in the union of all the bags.

Refinement **R1** provides the following: (1) after a firing of `CRi` transitions only one integer remains in a bag of `MSInt`; (2) the computation is realized fully in parallel over all available pairs present in the bags of `MSInt` and `MSIntToEmpty`; (3) the remaining integer is the sum of the integers present in all the bags of `MSInt` and `MSIntToEmpty` before the firing of `CRi`.

4.4 Second Refinement: Behavior Distribution

Refinement **R1** provides a distributed view of the application at the level of the data. As we intend to obtain a Java application distributed over the Web, it is necessary to think about applets. These applets will store the local multiset related to the user who launches the applet. Several of these applets need to communicate with each other in order to realize the DSGamma system. The Java programming language constrains an applet to connect exclusively to the host where it comes from. For this reason, it is necessary to have in this refinement step, called **R2**, a server running on the host where the applets come from. The CO-OPN/2 textual specification of refinement **R2** is given by appendix C.

4.4.1 Refinement Process

The informal view of the DSGamma system, is given by figure 4.5. We chose to insert a server, `GlobalRelay`, which acts as a buffer between all the `Applet`s, and which is running on the host where the applets come from. This server is only able to receive integers from a set of applets, $\texttt{Applet}_1$ to $\texttt{Applet}_3$, for instance, and to forward these integers to this same set of applets. Thus an integer goes from one applet to another one via the server.

The global multiset is logically given by the union of (1) several local multisets, each one located into an applet, (2) the FIFO buffer maintained by the `GlobalRelay` object.

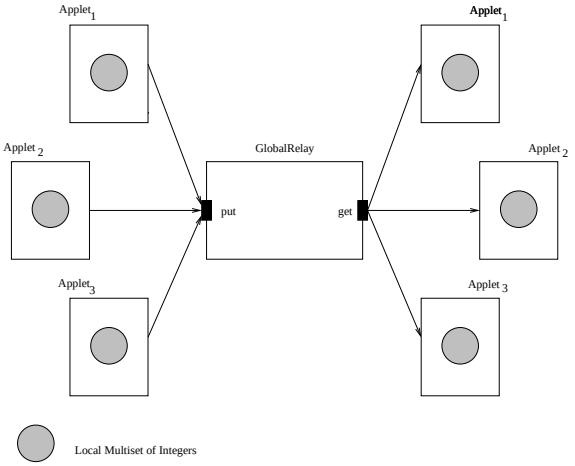


Figure 4.5: DSGamma with a Server

In refinement **R1**, the chemical reaction can happen in 8 different ways, but this way is atomically specified by one CO-OPN/2 transition (one `CRi`). In refinement **R2**, one chemical reaction is specified by four transitions, and a whole chemical reaction's behavior emerges from the firing of these four transitions possibly distributed across several different applets.

4.4.2 CO-OPN/2 Specification

Refinement **R2** provides a client/server architecture of our application. It is given by figures 4.6 and 4.7. It is the most abstract view of the application when it is considered as a client/server application. We have a distributed data (the local multisets), and processes (the chemical reactions), whose behavior is distributed across the system.

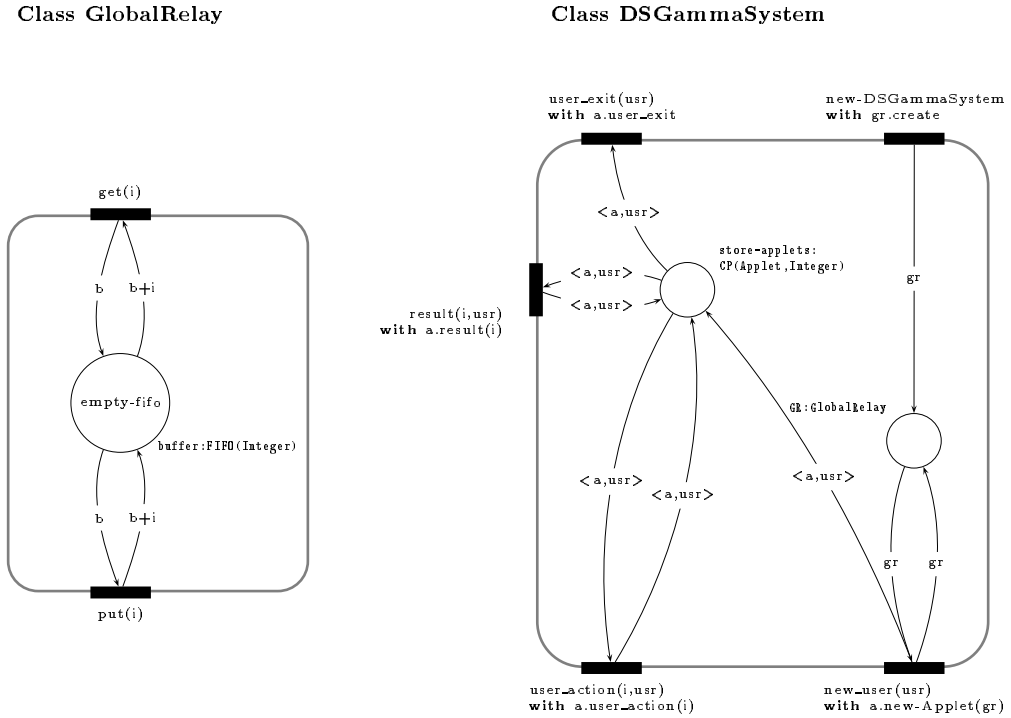


Figure 4.6: Refinement **R2**: Server side

System Operations:

The overall DSGamma system is specified by the `DSGammaSystem` class, it keeps the same CO-OPN/2 methods than refinement **R1**: `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`,

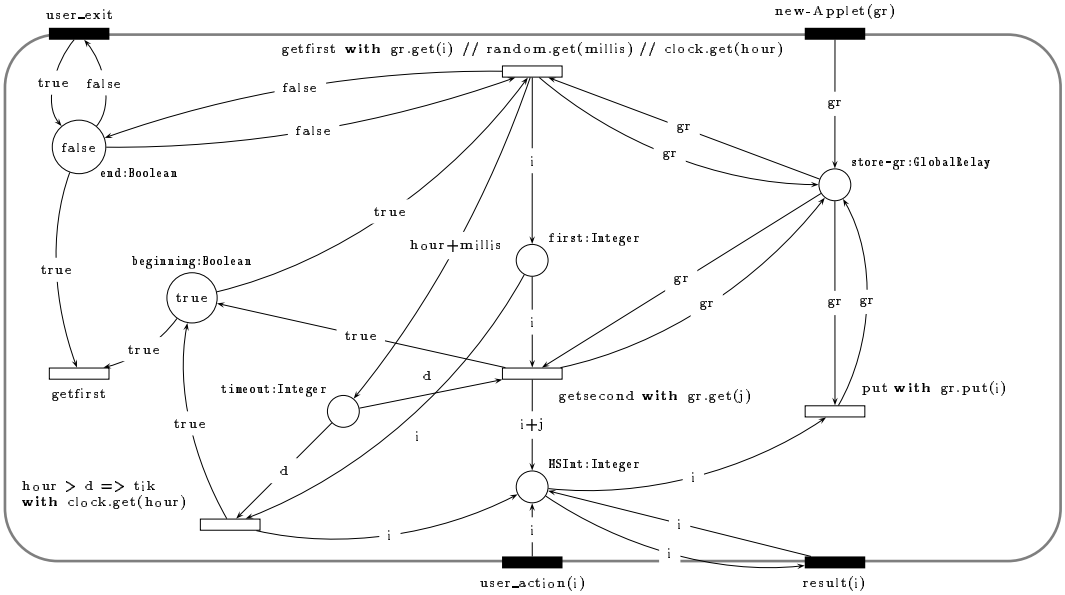


Figure 4.7: Refinement **R2**: Client Side

and `user_exit(usr)`. A CO-OPN/2 constructor (method creation) `new-DSGammaSystem` has been added because a default constructor is no longer sufficient.

The CO-OPN/2 constructor `new-DSGammaSystem` requires that, as soon as a DSGamma system exists, a `GlobalRelay` buffer `gr` is created (calling `gr.create`), where `gr` is a CO-OPN/2 object of class `GlobalRelay`, and `create` is the default constructor. `gr` is then stored in the `GR` place.

The `new_user(usr)` implies the dynamic creation of a new applet `a` by the synchronization expression `with a.new-Applet(gr)`. It stores the pair <`a,usr`> in the `store-applets` place. In refinement **R1**, the DSGamma system stores pairs of users and bags. In refinement **R2**, the DSGamma system stores pairs of users and applets. Indeed, logically it stores always the same information, but as the handling of the local bag has become more complex, it is dedicated to an applet.

The `user_action(i,usr)` method checks if the pair <`a,usr`> already exists, and if so forwards the action to the dedicated applet, `a` by the synchronization expression `with a.user_action(i)`.

The `result(i,usr)` method checks if `usr` already exists and requires the result from the `usr`'s dedicated applet, `a`, by the synchronization expression `with a.result(i)`.

The `user_exit(usr)` method removes the pair <`a,usr`> from the `store-applets` place, if it exists; and forwards this information to `usr`'s dedicated applet, `a`, by the synchronization expression `with a.user_exit`.

State:

A local multiset is given by the `MSInt` place of type `Integer` of the `Applet` class. It stores integers. The global multiset is given by these places, but also by several other places: `first` of type `Integer` in `Applet` class, and by `buffer` of `FIFO(Integer)` type in class `GlobalRelay`. Formally the FIFO of integers is specified with an algebraic specification. Integers transit from `MSInt` places directly to `buffer`, and from then they go to a `first` place or they are immediately added to an integer in a a `first` place and end in oa `MSInt` place. In refinement **R1**, the local multisets are specified separately, but stored in a same place. In refinement **R2**, the local multisets are specified separately and stored in different places located in distinct CO-OPN/2 objects of class `Applet`, and `GlobalRelay`.

Several other places are used for: (1) storing the pairs of applets and users: `store-applets` of type `CP(Applet,Integer)` in the `DSGammaSystem` class; (2) storing the `GlobalRelay` buffer: `GR` and `store-gr` places in classes `DSGammaSystem` and `Applet` respectively; (3) storing a timeout, an exit indication and a first integer to check, `timeout`, `end` and `beginning` respectively.

Internal Behavior:

The internal behavior is specifed by classes `GlobalRelay` and `Applet`. The `GlobalRelay` class specifies a FIFO buffer of integers, it is initially empty. An integer `i` is inserted into this FIFO by the means of the `put(i)` method, and is removed from the FIFO by the means of the `get(i)` method.

The `Applet` class specifies three CO-OPN/2 methods: `user_action(i)`, `user_exit`, `result(i)`, and one non default constructor `new-Applet(gr)`. As soon as a new user enters the DSGamma system, a new applet is created by the means of the `new-Applet(gr)` constructor.

The constructor creates a CO-OPN/2 object of the class `Applet`, stores the `gr` object identity of the `GlobalRelay` in the place `store-gr`, initializes the `end` place with `false`, and the `beginning` place with `true`. The `end` place stores the value `false` if the user is currently in the system and stores the value `true` if the user exits. The `beginning` place stores the value `true` if a first integer has to be requested, and stores nothing if a first integer has already been obtained. This place is used to ensure that a new first integer is requested only after the previous sum has been realized.

The `user_action(i)` inserts the integer `i` into the local multiset specified with the `MSInt` place.

The `user_exit` method replaces the token `false` by the token `true` in place `end`.

The chemical reactions are specified by the means of the four CO-OPN/2 transitions: `getfirst`, `getsecond`, `tik`, `put`. The `getfirst` transition is responsible for obtaining the first integer being involved in a sum. It has two possible firings. Firstly, it removes one integer from FIFO `gr` and enables a timeout. `getfirst with (gr.get(i) // random(millis) // clock(hour))` has the following behavior: (1) it checks if a first integer has to be taken, i.e. removes the `true` token in place `beginning`, (2) it checks if the user has not exited, i.e. it checks in the `end` place for the `false` token, simultaneously (3a) removes integer `i` from the FIFO `gr` and stores it in the `first` place; (3b) requires a random amount of milliseconds; (3c) requires the current `hour` from the `clock`; finally (4) stores the deadline `hour+millis` in the `timeout` place[5].

Secondly, it stops any new entering of integers in the applet, because the user wants to exit. `getfirst` has the following behavior: (1) it removes the `true` token from the `end` place, (2) it removes the `true` token from the `beginning` place. If the user has exited, and if a first integer has to be removed from `gr`, the `getfirst` transition, by removing these two tokens, stops the entering of new integers in the applet.

The `getsecond` transition is responsible to remove a second integer from the FIFO `gr`, and to disable the timeout. The `getsecond with gr.get(i)` has the following behavior: (1) it removes the first integer `i` from place `first`; (2) removes the deadline `d` from the `timeout` place; (3) removes integer `j` from the FIFO `gr`; (4) inserts the sum `i+j` in the `MSInt` place; (5) put the `true` token in the `beginning` place in order to mention that a new first integer can be requested.

The `tik` transition handles a timeout event occurring before a second integer can be obtained by the `getsecond` transition. It is responsible to disable the timeout and to insert the first integer (instead of a sum) into the local multiset. It has the following behavior: (1) it removes the first integer `i` from place `first`; (2) removes the deadline `d` from the `timeout` place; (3) inserts the sum `i` in the `MSInt` place; (4) put the `true` token in the `beginning` place in order to mention that a new first integer can be requested.

This timeout is necessary, because a deadlock occurs as soon as the number of integers present in the global multiset (the union of the local multisets) is lower than or equal to the number of users. Indeed, consider a system with only two integers in the global multiset and two or more applets in the system. If these two integers are taken by two different applets, then without timeout, each of these two applets would be blocked infinitely waiting for a second integer. Consequently, the whole system would be in a deadlock state.

The `put` transition randomly removes integers of the local multiset, and sends them to the FIFO buffer. `put with gr.put(i)` has the following behavior: (1) it removes `i` from `MSInt` place, (2) inserts `i` at the end of the FIFO buffer `gr`.

Illustration 4.4.1 *Example of figure 4.3 is modelled by refinement **R2** of figures 4.6 and 4.7 in the following way. Three users, `usr1`, `usr2`, `usr3`, enter the four integers $\{1, 2, 2, 3\}$. For instance, by the means of the methods `user_action(1,usr1)`, `user_action(2,usr2)`, `user_action(3,usr3)`, occurring simultaneously, and `user_action(2,usr1)` occurring later. It is worth noting that due to the modelisation of the `user_action(i,usr)` method, a single user cannot enter two or more integers simultaneously. Indeed, the `user_action(i,usr)` requires an access to the pair <`a,usr`>,*

[5] the CO-OPN/2 specification of the part concerning the clock is not detailed in this paper.

in order to insert only one integer ***i****. For this reason, the three above users cannot enter simultaneously four integers, they can enter at most three integers simultaneously (one per user), and only after that they can enter other integers.*

Let us call ***Appleti*** *for* $i \in \{1, 2, 3\}$ *the applet related to* ***usri****, and* ***MSInti*** *the corresponding local multiset. Firstly three integers enter simultaneously the system, thus* ***MSInt1*** *has the value* $\{1\}$*,* ***MSInt2*** *has the value* $\{2\}$*,* ***MSInt3*** *has the value* $\{3\}$*. See (a) on figure 4.8.*

As soon as one integer is present in a ***MSInt****, the* ***put*** *transition of the corresponding* ***Applet*** *can be fired. An execution scenario can be the following,* ***put*** *of* ***Applet1*** *removes* 1 *from* ***MSInt1*** *and sends it to* ***GlobalRelay****,* ***put*** *of* ***Applet2*** *removes* 2 *from* ***MSInt2*** *and sends it to* ***GlobalRelay****,* ***put*** *of* ***Applet3*** *removes* 3 *from* ***MSInt3*** *and sends it to* ***GlobalRelay****. The* ***buffer*** *of* ***GlobalRelay*** *contains values* $1, 2, 3$ *in this order, and* ***MSInti*** *are empty. See (b) on figure 4.8.*

The ***getfirst*** *of* ***Applet1*** *can get the first available integer in the* ***GlobalRelay*** *buffer, i.e.* 1*, the* ***getfirst*** *of* ***Applet2*** *can get the next available integer in the* ***GlobalRelay*** *buffer, i.e.* 2*, and the* ***getsecond*** *of* ***Applet2*** *can get the next available integer in the* ***GlobalRelay*** *buffer, i.e.* 3*, it makes the sum with the* 2 *stored in* ***first*** *of* ***Applet2****, and inserts the sum* 5 *into* ***MSInt2****. See (c) on figure 4.8.*

As no transitions of ***Applet1*** *can be fired,* ***usr1*** *can enter a new integer,* 2 *in the system, with the* ***user_action(2,usr1)****. Thus* ***MSInt1*** *has the value* $\{2\}$*, while* ***MSInt2*** *has value* $\{5\}$*. See (d) on figure 4.8.*

The ***put*** *of* ***Applet2*** *then removes* 5 *from* ***MSInt2*** *and sends it to* ***GlobalRelay****. The* ***put*** *of* ***Applet1*** *removes* 2 *from* ***MSInt1*** *and sends it to* ***GlobalRelay****. The* ***buffer*** *of* ***GlobalRelay*** *contains values* $5, 2$ *in this order,* ***MSInti*** *are empty, and* ***first*** *of* ***Applet1*** *contains* 1*.*

The timeout of ***Applet1*** *elapses, thus the* ***tik*** *transition removes* 1 *from* ***first*** *and inserts it into* ***MSInt1****, the* ***put*** *then immediately removes* 1 *from* ***MSInt1*** *and sends it to* ***GlobalRelay****. The* ***buffer*** *of* ***GlobalRelay*** *contains values* $5, 2, 1$ *in this order, and* ***MSInti*** *are empty. See (e) on figure 4.8.*

getfirst *of* ***Applet2*** *can get the first available integer in the* ***GlobalRelay*** *buffer,* 5*,* ***getfirst*** *of* ***Applet3*** *can get the next available integer in the* ***GlobalRelay*** *buffer,* 2*, and* ***getsecond*** *of* ***Applet3*** *can get the next available integer in the* ***GlobalRelay*** *buffer,* 1*, makes the sum with the* 2 *stored in* ***first*** *of* ***Applet3****, and inserts the sum* 3 *into* ***MSInt3****. The* ***buffer*** *of* ***GlobalRelay*** *is empty,* ***MSInt1****,* ***MSInt2*** *are empty,* ***MSInt3*** *has value* $\{3\}$*, and* ***first*** *of* ***Applet2*** *contains* 5*. See (f) on figure 4.8.*

put *of* ***Applet3*** *immediately removes* 3 *from* ***MSInt3*** *and sends it to* ***GlobalRelay****. The* ***buffer*** *of* ***GlobalRelay*** *contains only value* 3*,* ***MSInti*** *are empty,* ***first*** *of* ***Applet2*** *contains* 5*.*

getsecond *of* ***Applet2*** *can get the first available integer in the* ***GlobalRelay*** *buffer,* 3*, makes the sum with the* 5 *stored in* ***first*** *of* ***Applet2****, and inserts the sum* 8 *into* ***MSInt2****. See (g) on figure 4.8.*

If no more integers are entered by the users, the 8 *value will transit from an* ***MSInt*** *to another, via the* ***buffer*** *of* ***GlobalRelay****, due to the* ***tik*** *transitions.*

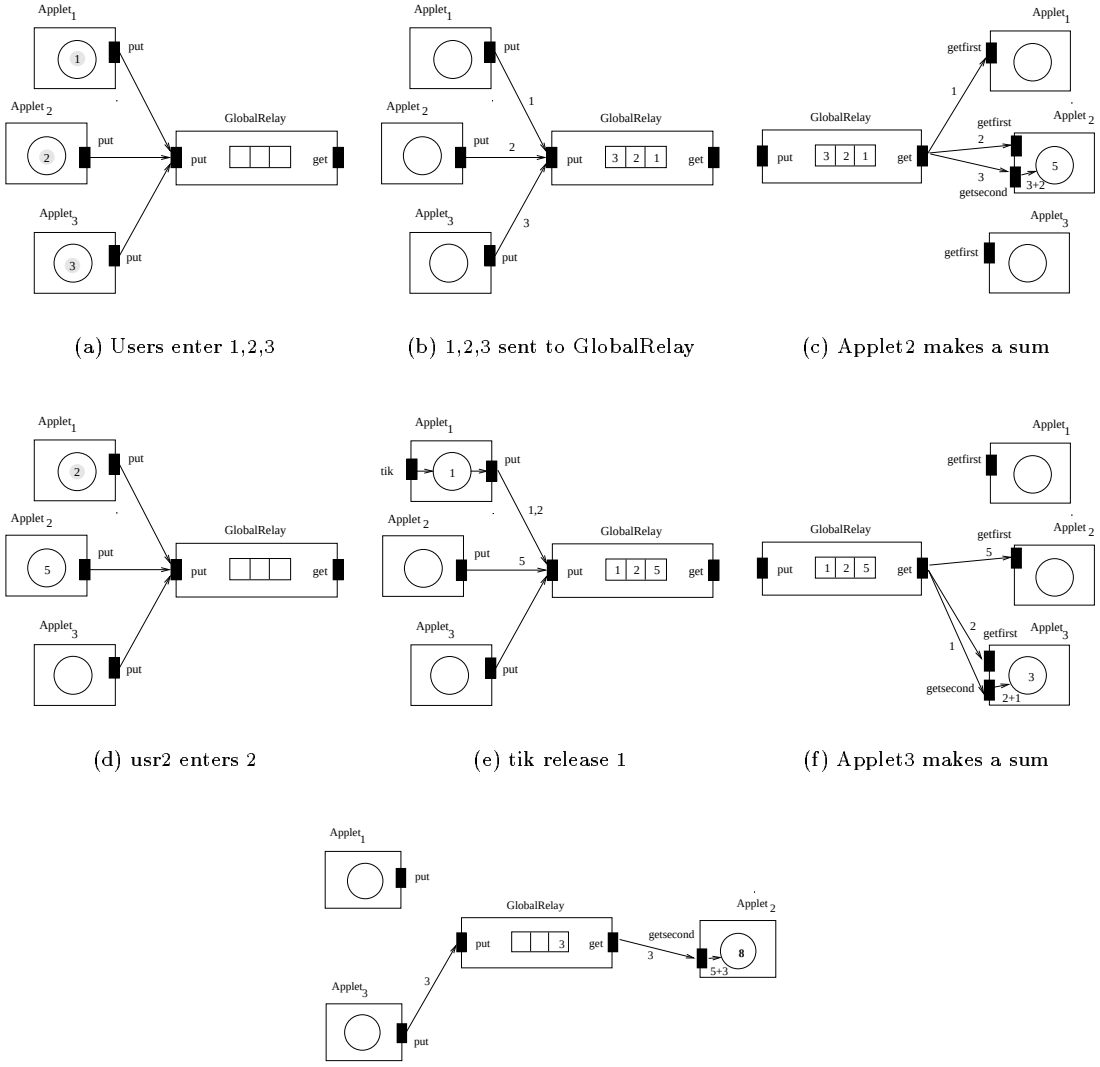
Note that the way how ***GlobalRelay*** *has been modeled, implies an access to the whole buffer in order to either remove one integer or insert one integer from/into the buffer. Thus, the* ***getfirst****,* ***getsecond****,* ***tik*** *and* ***put*** *transitions occur in an interleaved way.*

4.4.3 Properties

The computation of the result is realized by the four transitions `getfirst`, `getsecond`, `tik`, and `put`. All these transitions are fired concurrently, simultaneously, and each one several times simultaneously, as soon as it is possible and as longer as their pre-conditions are fulfilled.

At the beginning of the system, no integers are present in the whole system. We assume that n integers enter the system simultaneously (several simultaneous `user_action(i,usr)`). These n integers are distributed over n applets, because each occurrence of `user_action(i,usr)` has to access a whole applet. Note that there may be more applets, for the users who enter no integer. From now on and for simplicity purpose, we suppose that the users will not add new integers in the system. `put`, `getfirst`, `getsecond`, and `tik` are now firable.

`getfirst`, `getsecond`, and `tik` collaborate to compute sums of integers. Due to the specification of the FIFO buffer, the `get(i)`, `put(i)` methods of `GlobalRelay` must access to the whole FIFO in order to insert one or remove one integer. Thus, one chemical reaction occurs in parallel with another chemical reaction, however, the smallest steps (transitions) that realize one chemical



(a) Users enter 1,2,3

(b) 1,2,3 sent to GlobalRelay

(c) Applet2 makes a sum

(d) usr2 enters 2

(e) tik release 1

(f) Applet3 makes a sum

(g) Applet2 makes a sum

Figure 4.8: Example of chemical reactions in Client/Server Architecture

reaction occur in an interleaved way with those of another chemical reaction. We can see two chemical reactions as two sequences of several transitions, each transition requires an exclusive access to the FIFO buffer, thus the transitions occur in an interleaved way. The chemical reactions stop either when only one integer remains in the global multiset, note that due to the `tik` transitions this integer will go from one applet to the other one, or when the number of integers present in the global multiset is lower than the number of applets, a short period of deadlock occurs, before the `tik` transitions are firable. After a possibly long time there will remain only one integer in the system, because pairs of integers will succeed to meet in the same applet.

As soon as a user exits, the `getfirst` transition stops receiving integers. If all the users leave the system simultaneously, then the applets will send all their integers, stored in `MSInt`, and stop receiving integers, thus `GlobalRelay` will store all the integers in its FIFO.

Refinement **R2** provides the following: (1) after a firing of the chemical reactions and of the `tik` transitions, it remains only one integer in the system; (2) the computation is realized in an interleaved way by non atomic chemical reactions; (3a) a remaining integer is obtained provided that at least one user remains in the system otherwise, `GlobalRelay` stores several integers; (3b) the remaining integer is the sum of the integers present in `MSInt` before the firing of the chemical reactions.

4.5 Third Refinement : Communication Layer

Refinement **R2** provides a client/server view of our application, with applets communicating with each other through a relay server. The applets communicate directly with the server. As our targeted application has to run across several physically distributed hosts, it is now time to introduce the sockets, i.e. the communication layer between the applets and the server. Refinement **R3**, provided at this stage, is also intended to be the last one before the Java program. For this reason, this specification takes into account some features of the Java programming language, and specifies all the Java components that will be part of the final program. The CO-OPN/2 textual specification of refinement **R3** is given by appendix D.

4.5.1 Refinement Process

The informal view of both refinement **R3** and the implementation of the DSGamma system is given by figure 4.9.

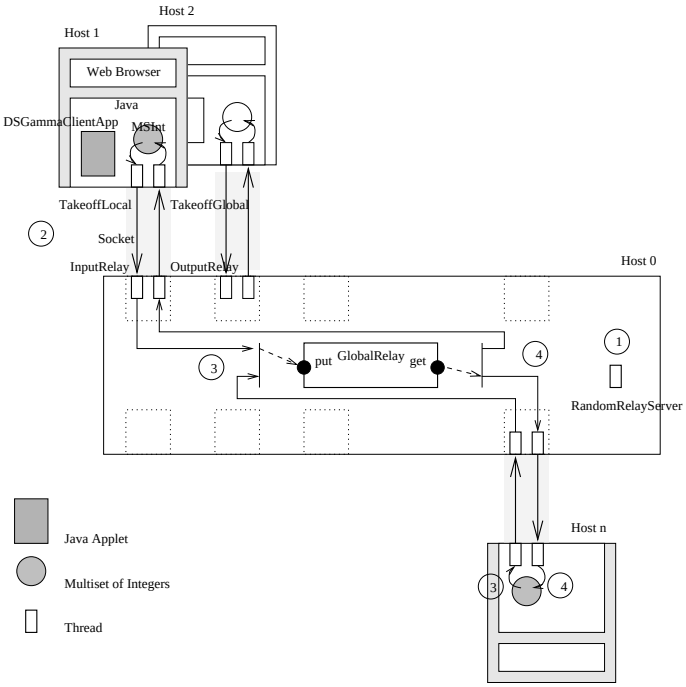


Figure 4.9: The implemented architecture of the DSGamma system

The relay server is bigger than it was in refinement **R2**, class `RandomRelayServer` defines now the server, see position 1 on figure 4.9. It handles the following elements: a FIFO buffer of integers of class `GlobalRelay`; and for each applet a pair of threads of classes `OutputRelay`, `InputRelay`, they are dedicated to the handling of the communication with an applet. See position 2 on figure 4.9.

The global multiset is logically given by the union of (1) several local multisets, each one located into an applet, (2) the FIFO buffer maintained by the `GlobalRelay` object, and (3) the sockets buffers.

The communication layer is given by the sockets. A socket has been specified by four classes: `Socket` class, `FIFO(Bytes)` class, and `DataInputStream`, `DataOutputStream` classes for two FIFO buffers of integers. As soon as an applet connects to the `RandomRelayServer`, a `Socket` is created together with two streams requiring on buffer each. The first stream goes from the server to the applet, it is made of one `DataInputStream` at the client side and one `DataOutputStream` at the server side working upon a `FIFO(Bytes)`. The second stream goes from the applet to the server, it is made of one `DataInputStream` at the server side and one `DataOutputStream` at the client side working upon another `FIFO(Bytes)` .

The applets are more complex than what they were in refinement **R2**. As soon as the applet connects to the server, two threads are created of classes `TakeoffLocal`, `TakeoffGlobal`. These

threads are responsible to handle the chemical reactions, the timeout and the quitting protocol. See position 2 on figure 4.9. The applet also handles the local multiset `MSInt`.

4.5.2 CO-OPN/2 Specification

Refinement **R3** views our application as a Java applet based application. It is given by several CO-OPN/2 classes. Firstly, a set of basic Java classes have been specified into CO-OPN/2 classes, respecting the same inheritance tree than the Java programming language. A Java scheduler has been also specified. The Java `Object` class has been specified by the `JavaObject` CO-OPN/2 class, idem for the Java `Thread`, `Applet`, `Socket`, `ServerSocket`, `DataInputStream`, `DataOutputStream` classes. Upon this layer of CO-OPN/2 classes, we have built the CO-OPN/2 specification of our future program.

System Operations:

The overall DSGamma system is specified with the `DSGammaSystem` class, it also keeps the same CO-OPN/2 methods than refinement **R2**: `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`, and `user_exit(usr)`. It is depicted by figure 4.10.

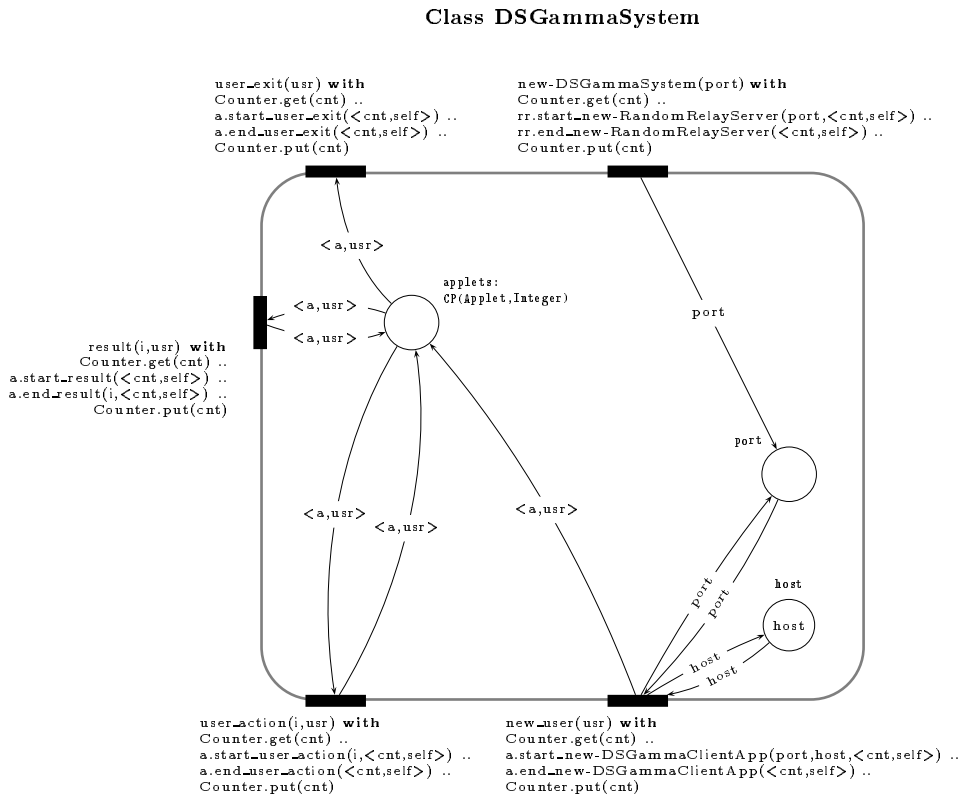


Figure 4.10: Refinement **R3**: External Behavior

The CO-OPN/2 constructor `new-DSGammaSystem(port)` has been changed, wrt refinement **R2**, in order to take into account the fact that the server waits on a given `port` for applets connections. The constructor `new-DSGammaSystem(port)` creates an CO-OPN/2 object, `rr`, of the class `RandomRelayServer` waiting on `port`.

The `new_user(usr)` method: (1) creates and inits a new applet `a` of the CO-OPN/2 class `DSGammaClientApp` class; (2) stores the pair <`a,usr`> in the `store-applets` place.

The `user_action(i)` method checks for the pair <`a,usr`> in the `store-applets` place, and forwards the action to applet `a`.

The `result(i,usr)` method checks for the pair <`a,usr`> in the `store-applets` place, and requires the result from the user's dedicated applet.

The `user_exit(usr)` method checks for the pair <`a,usr`>, forwards this exit to the user's dedicated applet, and removes the pair from the `applets` place.

State:

The local multisets are given by the `MSInt` places of each applet. The global multiset is given by the union of these local multisets and by the buffers of all the sockets streams and by the FIFO buffer of `GlobalRelay`.

The state is given by the same places than the previous specification, with in addition the buffers of the socket streams.

Internal Behavior:

The internal behavior is specifed by classes `RandomRelayServer`, `InputRelay`, `OutputRelay`, `GlobalRelay` at the server side and by, `DSGammaClientApp`, `TakeoffLocal`, `TakeoffGlobal` classes at the client side. The communication layer is specified by the `Socket`, `DataInputStream`, `DataOuputStream` and `ServerSocket` classes. These last four classes specify the omonymous classes of the Java programming language.

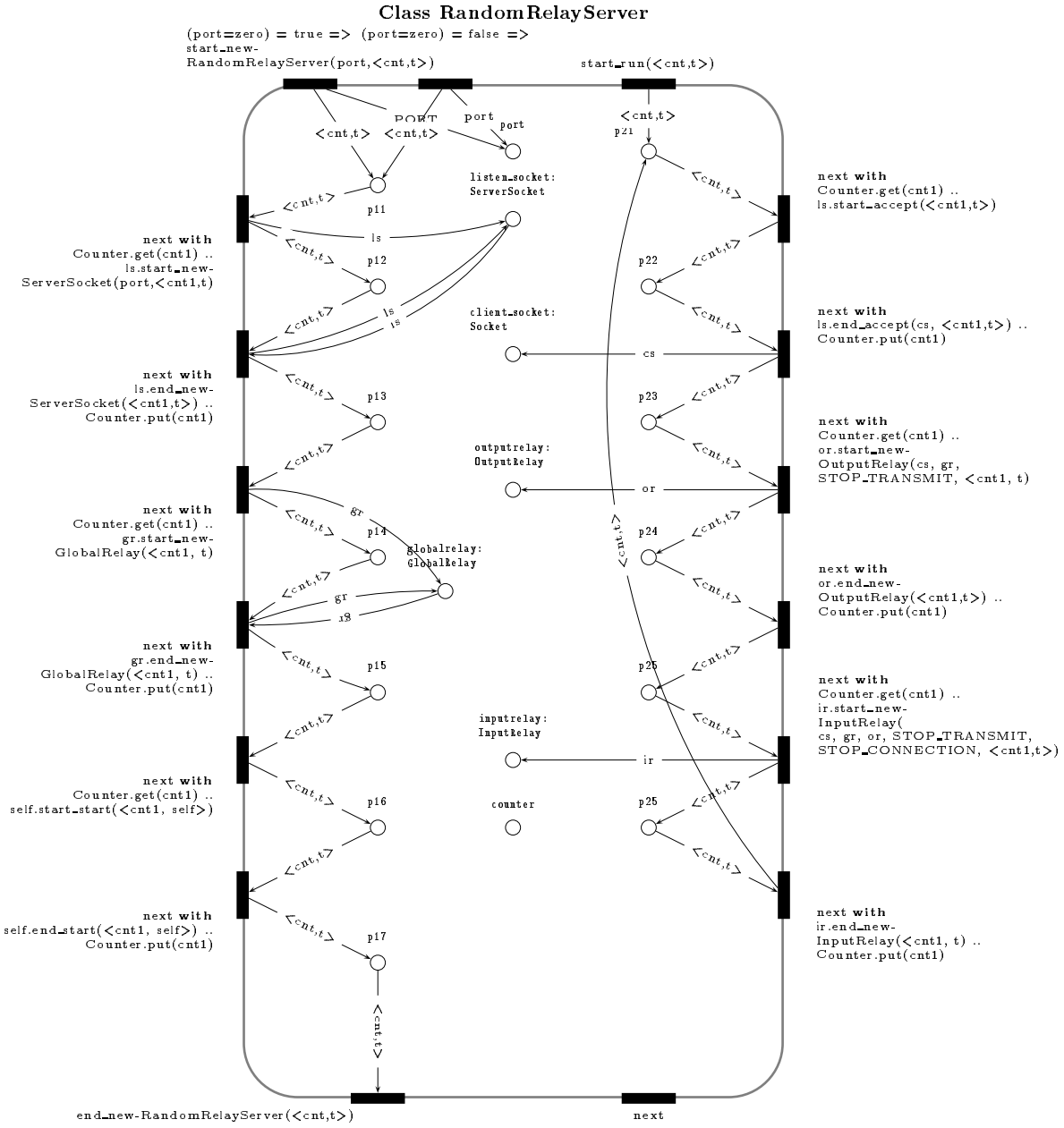


Figure 4.11: Refinement **R3**: Server Side (1)

Refinement **R3** has to stick to the Java language programming semantics. For this reason, and because of the CO-OPN/2 semantics, we have modelled a body of Java method, as a sequence of CO-OPN/2 `next` methods. Chapter 5 explains the details of this specification. In order to better understand this part, the reader should refer to this chapter.

Server Side: The CO-OPN/2 constructor of the `RandomRelayServer` class is given by methods `start_new-RandomRelayServer(port,<cnt,t>)` and `end_new-RandomRelayServer(<cnt,t>)`. Depending on the value of `port` a default `PORT` is used. The constructor creates the `GlobalRelay` FIFO buffer, and a `ServerSocket` on port `port`. A `RandomRelayServer` is a thread. The `run` method infinitely waits for connections on the `ServerSocket`, and as soon as an applet connects, it creates two threads `or`, and `ir`, of class `OutputRelay`, `InputRelay` respectively connected to the applet's socket.

The `RandomRelayServer` class is given by figure 4.11. For simplicity purpose, the reading arcs are no drawn on this figure. For instance, all input parameters used in a method call must be taken from a place and inserted in the same place in order to be able to use the parameter. Also for simplicity purpose, arcs related to place `counter` are not drawn.

The `InputRelay` class is given by figure 4.12.

The creation of an `InputRelay` thread implies the creation of an `DataInputStream`. The main task of this thread is to read integers from the `DataInputStream`, and to forward them to the `GlobalRelay` FIFO buffer, see positions 3 on figure 4.9. It is also responsible to handle for end signals incoming from the applet. There are two types of signals incoming from the applet: (1) the `stop_transmit` signal which indicates that the applet does not want any more integers to come from the server, thus the `InputRelay` thread notifies that to the `OutputRelay`; (2) the `stop_connection` signal which indicates that the applet has received from `OutputRelay` the confirmation that the `stop_transmit` signal has been received and that its local multiset is empty, thus the `InputRelay` thread stops itself.

Figure 4.12: Refinement **R3**: Server Side (2)

The `OutputRelay` class is given by figure 4.13.

The creation of an `OutputRelay` thread implies the creation of an `DataOutputStream`. The main task of this thread is to remove integers from the `GlobalRelay` FIFO buffer, to write them to `DataOutputStream`, see positions 4 on figure 4.9. It is also responsible to check for the end signal incoming from the `InputRelay` thread: it then writes `stop_transmit` on the `DataOutputStream`.

Figure 4.13: Refinement **R3**: Server Side (3)

The `GlobalRelay` FIFO buffer is specified in refinement **R2**.

Client Side: The `DSGammaClientApp` class is given by figure 4.14.

The constructor of the `DSGammaClientApp` class stores the remote host and the port of the server. An `init` method is used, it creates: (1) the `Socket`, (2) the `DataInputStream` and `DataOutputStream` at the client side, (3) the local multiset: `MSInt` vector (specifying a Java vector); (4) two threads, `TakeoffLocal`, `TakeoffGlobal` which realize the chemical reaction, the timeout, and the quitting protocol.

An applet keeps the same methods as in refinement **R2**: `user_action(i)`, `result(i)` and `user_exit` .

Figure 4.14: Refinement **R3**: Client Side (1)

The applets' `user_action(i)` method only stores `i` in the `MSInt` vector. The `result(i)` method returns an integer of the local multiset maintained by the applet. The `user_exit` method sends an end signal to the server. It writes the `stop_transmit` signal on the `DataOutputStream` at the applet's side.

The `TakeoffLocal` class is given by figure 4.15.

The `TakeoffLocal` thread permanently checks for integers in `MSInt`, removes randomly one and writes it to `DataOutputStream` at the client side. If the server has already sent back the `stop_transmit` signal, then `TakeoffLocal` empties the `MSInt` vector a last time, then sends the `stop_connection` signal to the server, i.e. writes `stop_connection` on the `DataOutputStream`, finally `TakeoffLocal` stops.

Figure 4.15: Refinement **R3**: Client Side (2)

The `TakeoffGlobal` class is given by figure 4.16.

The `TakeoffGlobal` thread reads a first integer from the `DataInputStream` at the client side. As soon as it has obtained it, `TakeoffGlobal` enables a timeout, and reads a second integer. If the second integer arrives before the timeout deadline, then it is added with the first one, and inserted into `MSInt`. Otherwise, a `tik` transition prevents a deadlock, by inserting the first integer into `MSInt`. `TakeoffGlobal` handles also the `stop_transmit` signal that it can receive from the server. In that case, either the `stop_transmit` arrives instead of a first integer, then `TakeoffLocal` is notified of that and `TakeoffGlobal` stops, or the `stop_transmit` arrives instead of a second integer, then the first one is inserted into `MSInt`, `TakeoffLocal` is notified and `TakeoffGlobal` stops.

Figure 4.16: Refinement **R3**: Client Side (3)

In refinement **R2**, the timeout had been already specified, it has been used exactly in the same manner in this specification. The quitting protocol of refinement **R2** is more simple, because there are no intermediate buffers storing integers. It has been enhanced in refinement **R3**, in order to firstly notify the server that the user wants to exit, to secondly, receive from the server eventual integers present in the `DataOutputStream` at the server's side, and finally, to empty the local multiset `MSInt` a last time before stopping.

Communication Layer: The `DataOuputStream` and `DataInputStream` are used to insert or removes integers into a FIFO buffer of bytes, realizing the converstion. The `Socket` class creates two `FIFO(Bytes)`, so that the sockets realize the TCP/IP protocol (they neither lose nor disorder the packets). The `Socket` class actually specifies the connection with a `ServerSocket` given a remote host and a port. Chapter 5 provides the CO-OPN/2 specifications for the Java class related to the sockets.

Illustration 4.5.1 *Example of figure 4.3 is modelled, in the following way, by refinement **R3** of figures 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, and socket related specifications. Three users, `usr1`, `usr2`, `usr3`, enter the four integers* $\{1, 2, 2, 3\}$. *For instance, by the means of the methods `user_action(1,usr1)`, `user_action(2,usr2)`,*
`user_action(3,usr3)`, occurring simultaneously, and `user_action(2,usr1)` occurring later. It is worth noting that due to the modelisation of the `user_action(i,usr)` method, a single user cannot call this method two or more times simultaneously. Indeed, the `user_action(i,usr)` requires an access to the pair <a,usr>. However, the insertion of the value i, in the `MSInt` of the `usr`'s applet, is not actually realized by the `user_action(i,usr)` method, but by a `next` in the body of the `user_action(i)` method of the `Applet` class. Thus, before the arrival of integer i in `MSInt`, `usr` can call the `user_action(j,usr)` method, in order to insert another integer j. Depending on the execution order of the CO-OPN/2 method, several scenario may happen: (1) the four integers $\{1, 2, 2, 3\}$ *reach the local multisets before the chemical reactions begin; or (2) as soon as the first one has reached a local multiset, a chemical reaction removes it and sends it the server, before a second one arrives in another local multiset; etc.*

*Due to the fact that refinement **R3** has to respect the Java semantics, there are no CO-OPN/2 transitions in refinement **R3**. Thus, it is worth keeping in mind: (1) each "action" (chemical reaction, new integer arriving) on a local multiset is splitted over several CO-OPN/2 methods (no transitions are used); (2) all these methods can be executed at any moment, provided their pre-conditions are fulfilled; (3) an "action" occur in an interleaved way with another action; (4) one CO-OPN/2 method can occur several times simultaneously and several of them can occur simultaneously; (5) only the methods accessing directly one integer in `MSInt` or `buffer` of `GlobalRelay`*

cannot occur simultaneously or several times simultaneously because **MSInt** *has to be accessed as a whole.*

Due to the lack of CO-OPN/2 transitions, the chemical reactions and the insertion of new integers by the users occur fully in parallel and in an interleaved way, thus the four inserted integers may be moved and summed to each other in a plenty of different manners.

Let us call **Appleti** *for* $i \in \{1, 2, 3\}$ *the applet related to* **usri**, **MSInti** *the corresponding local multiset,* **TakeoffLocali**, **TakeoffGlobali**, **InputRelayi**, **OutputRelayi** *the four threads,* **Client-DataOutputStreami**, *and* **Client-DataInputStreami**, **Server-DataOutputStreami**, *and* **Server-DataInputStreami** *the two streams at the client side, and the two streams at the server side respectively.*

We describe now one possible scenario.

Firstly four integers enter the system, thus **MSInt1** *has the value* $\{1, 2\}$, **MSInt2** *has the value* $\{2\}$, **MSInt3** *has the value* $\{3\}$. *See (a) on figure 4.17.*

On the client side, **TakeoffLocal1** *removes firstly* 2, *and then* 1 *from* **MSInt1** *and writes them in this order on* **Client-DataOutputStream1**. *Simultaneously,* **TakeoffLocal2** *removes* 2 *from* **MSInt2** *and writes it on* **Client-DataOutputStream2**, *while* **TakeoffLocal3** *removes* 3 *from* **MSInt3** *and writes it on* **Client-DataOutputStream3**. *On the server side,* **InputRelay1** *reads firstly* 2, *and then* 1 *from* **Server-DataInputStream1**, *and sends them in this order to* **GlobalRelay**, *while* **InputRelay2** *reads* 2 *from* **Server-DataInputStream2**, **InputRelay3** *reads* 3 *from* **Server-DataInputStream3**, *and sends respectively* 2 *and* 3 *to* **GlobalRelay**. *The four integers reach* **GlobalRelay**, *we suppose that they arrive in the following order* $2, 3, 1, 2$. *Note that they could arrive in any order provided that* $2, 1$ *sent by* **InputRelay1** *arrive in this order. See (b) on figure 4.17. Note that a chemical reaction can begin to happen before the four integers reach* **GlobalRelay**.

On the server side, **OutputRelay3** *gets* 2 *from* **GlobalRelay**, **OutputRelay2** *gets* 3 *from* **GlobalRelay**, **OutputRelay3** *gets* 1 *from* **GlobalRelay**, **OutputRelay2** *gets* 2 *from* **GlobalRelay**. *They write these integers on* **Server-DataOuputStream3** *and* **Server-DataOuputStream2** *respectively. On the client side,* **TakeoffGlobal3** *reads* $2, 1$ *from* **Client-DataInputStream3**, *and* **TakeoffGlobal2** *reads* $3, 2$ *from* **Client-DataInputStream2**. *They make the sum, and insert* 3 *and* 5 *in* **MSInt3**, *and* **MSInt2** *respectively. See (c) on figure 4.17.*

TakeoffLocal3 *removes* 3 *from* **MSInt3** *and writes it on* **Client-DataOutputStream3**. **TakeoffLocal** *removes* 5 *from* **MSInt2** *and writes it on* **Client-DataOutputStream2**. *On the server side,* **InputRelay3**, *and* **InputRelay2** *read* 3 *and* 5 *from* **Server-DataInputStream3**, *and* **Server-DataInputStream2** *respectively, and send them to* **GlobalRelay**. *We can suppose that they reach* **GlobalRelay** *in the following order* $5, 3$. *See (d) on figure 4.17.*

OutputRelay1 *gets* 5 *from* **GlobalRelay**, **OutputRelay2** *gets* 3 *from* **GlobalRelay**. *They write these integers to* **Server-DataOuputStream1** **Server-DataOuputStream2** *respectively. On the client side,* **TakeoffGlobal1** *reads* 5 *from* **Client-DataInputStream1**, *while* **TakeoffGlobal2** *reads* 3 *from* **Client-DataInputStream2**. *See (e) on figure 4.17.*

A period of deadlock occurs, until **tik** *of* **TakeoffGlobal1** *releases* 5 *into* **MSInt1**. *Through,* **TakeoffLocal1**, **Client-DataOutputStream1**, *and then* **Server-DataInputStream1** *and* **InputRelay1** *it reaches* **GlobalRelay**. **OutputRelay2** *gets* 5 *from* **GlobalRelay**, *and through* **Server-DataOutputStream2**, **Client-DataInputStream2**, *it reaches* **TakeoffGlobal2**, *where is it summed with* 3, *and* 8 *is inserted in* **MSInt2**. *See (f) on figure 4.17.*

4.5.3 Properties

In refinement **R2**, the computation of the sum is distributed over several applets' transitions. In refinement **R3** this computation is distributed over the four threads handling an applet's socket. Actually, the `getfirst`, and `getsecond` transitions of refinement **R2** have been splitted into two transitions each: one in `TakeoffGlobal` and one in `OutputRelay`, the `put` transition has been splitted over `TakeoffGlobal` and `InputRelay`, and the `tik` has been delocated to the `TakeoffGlobal` thread. Provided this extension of the chemical reaction to the communication layer, the same conclusion than refinement **R2** applies. Refinement **R3** provides the following: (1) after a firing of the chemical reaction only one integer remains in the global multiset; (2) the computation is realized in an interleaved way by non atomic chemical reactions; (3a) a remaining integer is obtained provided that at least one user remains in the system otherwise, `GlobalRelay` stores several integers; (3b) the remaining integer is the sum of the integers present in `MSInt` before the firing of

chemical reactions.

4.6 Fourth Refinement: The Java Program

The Java program has exactly the same classes than refinement **R3**. Their behavior is exactly the same as those specified by refinement **R3**. The Java program is given by appendix E.

4.6.1 Refinement Process

The only difference with refinement **R3** are the following. Firstly, in refinement **R3** we assumed that the streams worked on `FIFO(Bytes)`, in the concrete program, we need to be aware of the fact that intermediary `InputStream` and `OutputStream` classes are necessary. Secondly, due to the Java semantics a CO-OPN/2 transition is firable as soon as its precondition is fulfilled, in the Java program, the four involved thread classes: `TakeoffGlobal`, `TakeoffLocal`, `InputRelay`, `OutputRelay` use `wait`, `notify` methods in order to avoid polling. Thirdly, the applet provides a graphical user interface enabling a user to enter some integers and to see the state of the local multiset.

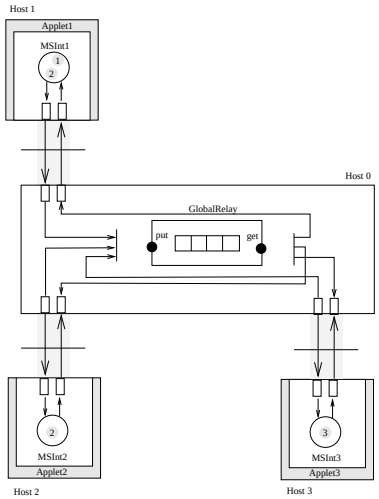
4.6.2 Properties

The Java program and refinement **R3** show very few differences. Thus, we reach exactly the same conclusion for the property.

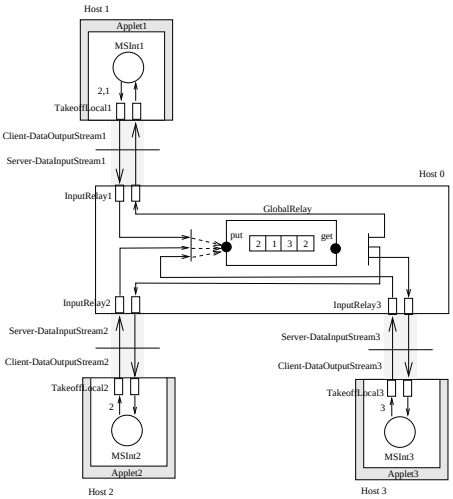
The Java program provides the following: (1) after a firing of the chemical reaction only one integer remains in the global multiset; (2) the computation is realized in an interleaved way by non atomic chemical reactions; (3a) a remaining integer is obtained provided that a least one user remains in the system otherwise, `GlobalRelay` stores several integers; (3b) the remaining integer is the sum of the integers present in `MSInt` before the firing of the chemical reactions.

4.7 Summary

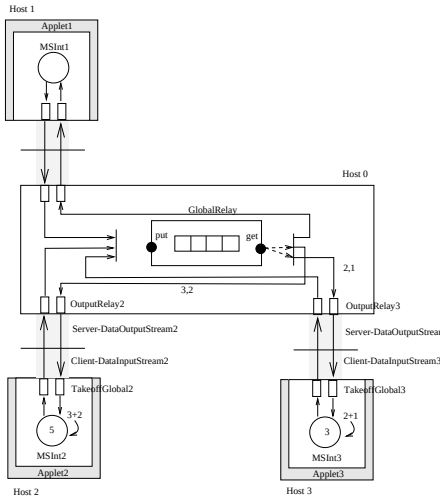
Table 4.1 summarizes the initial specification, **I**, and the three refinement steps described above, **R1**, **R2** and **R3**. The initial specification, **I**, is given by class `DSGammaSystem`, it uses the `Integer` ADT. The initial specification **I** and all its subsequent refinements **R1**, **R2** and **R3**, offer the same interface with the external world, `DSGammaSystem`. These four different `DSGammaSystem` classes offer the same set of methods to the outside. The initial specification **I** is concerned with a centralized view of the application, the `DSGammaSystem` class defines both the internal and external behavior. Refinement **R1** is a refinement of **I**, it is concerned with data distribution, it does not add any specialized class for the internal behavior. Refinement **R2** is a refinement of **R1**, it is concerned with the introduction of a client/server architecture. For this reason, the internal behavior is described by two additional classes `GlobalRelay` for the server side, and `Applet` for the client side. Refinement **R3** is a refinement of **R2**, it is concerned with the introduction of a communication layer between the server and the clients. The internal behavior is given by three sets of classes: (1) classes `RandomRelayServer`, `GlobalRelay`, `InputRelay`, `OutputRelay` at the server side; these four classes are the refinement of class `GlobalRelay` of refinement **R2**; (2) classes `Applet`, `TakeoffLocal`, `TakeoffGlobal` at the client side; these three classes are the refinement of class `Applet` of refinement **R2**; (3) classes `ServerSocket`, `Socket`, `DataInputStream`, `DataOutputStream` define the communication layer between the server and the client.



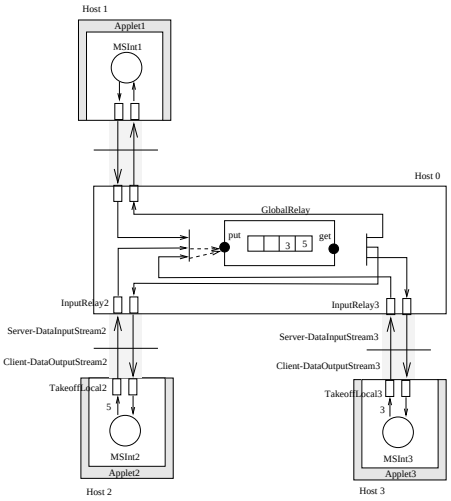
(a) Users enter 1,2,2,3



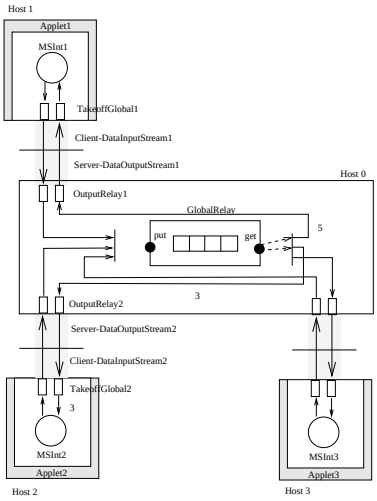
(b) 2,3,1,2 sent to Server



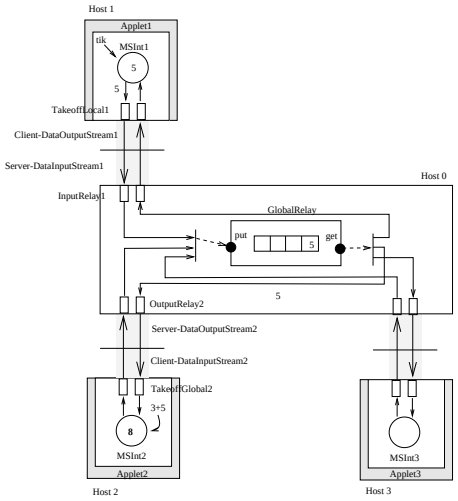
(c) TakeoffGlobal2, TakeoffGlobal3 make a sum



(d) 5,3 sent to Server



(e) Deadlock



(f) tik release 5

Figure 4.17: Example of chemical reactions in the Concrete Architecture

	I	R1	R2	R3	
ADT	Integer	Integer, Bag	Integer	Integer	
Class	DSGammaSystem	DSGammaSystem	DSGammaSystem	DSGammaSystem	External Behavior
Class			GlobalRelay Applet	RandomRelayServer GlobalRelay InputRelay OutputRelay DSGammaClientApp TakeoffLocal TakeoffGlobal ServerSocket Socket DataInputStream DataOutputStream	Internal Behavior

Table 4.1: Initial Specification and Three Refinement Steps

Chapter 5

CO-OPN/2 Interpretation of Java Underlying Concepts

This chapter provides a CO-OPN/2 specification of the Java `Object` class, focus is given on the `wait`, `notify`, `notifyall` methods and on the locks. The `Thread`, `Applet`, `Socket` classes are also considered. The CO-OPN/2 textual specification of these Java classes is given by appendix F.

Several Java concepts are introduced. They are either part of the Java Programming Language [2, 10] or part of the Java Virtual Machine [11]. For each of them, we give our design decisions for their specification in the CO-OPN/2 language.

5.1 Java Types

There are 3 kinds of Java types: (1) *primitive* types, (2) *reference* type, and (3) the `null` type. (pages 7-8-9 of [11])

- *Primitive* types are: the (a) `boolean` type with two values `true`, `false`, and (b) the *numeric* types.

  Numeric types are:
  (a) *integral*, i.e. `byte` (8-bits), `short` (16-bits), `int` (32-bits), `long` (64-bits), signed two's complement integers, `char` (16-bits) unsigned integers.
  (b) *floating-point* types, i.e. `float` (32-bits) and `double` (64-bits).

- *Reference* types are:
  (a) the *class* types, (b) the *interface* types, and (c) the *array* types.

  Reference values are *pointers* to *objects*. An object is a dynamically created class instance or an array. Reference types form a hierarchy. Each class type is a subclass of another class type, except for the class `Object`, which is the superclass of all their class types.

- `null` type:
  This type can always be converted to any reference type, it has only one possible value, the null value.

CO-OPN/2 Specification

Primitive Types

The primitive types are not part of the Java class hierarchy. However, the `Boolean`, `Character`, `Double`, `Float`, `Integer` and `Long` classes are Java classes which enclose the corresponding primitive type.

Primitive types allow to pass parameters by value, while reference types only allow to pass parameters by reference. In Java, in order to pass also primitive types by reference, each primitive type has a corresponding reference type.

We decided to specify, in CO-OPN/2 only the Java reference types, and not the primitive types. If, for instance, in a Java program the `int` type is used, then we will specify it as if it was the

`Integer` class. The specifier has to be careful to the fact that if he is specifying a method using a parameter of this type, changes to such a parameter will not be propagated to the method's caller.

Our purpose is to study the behavior of our application, and not all the details of the Java Programming language. We assume that the Java distinction between `int` and `Integer`, for instance does not affect the behavior of our application, provided that the rule above is applied.

Note that if we wanted to stick to the Java programming language, it would be possible to define primitive types as `ADT` (algebraic data types). The corresponding CO-OPN/2 classes, e.g. `Integer`, then would use the ADT, e.g. `int`, and will be able to convert the ADT variables into objects, and vice-versa.

Reference Types

As explained above, we do not specify the Java primitive types, but only the Java reference types.

For each Java *class*, we propose to specify a dedicated CO-OPN/2 class. The inheritance tree of our CO-OPN/2 classes is exactly the same as the inheritance tree of the Java classes. The `Object` Java class is the superclass of all Java classes. In CO-OPN/2 we called this class the `JavaObject` class, thus the `JavaObject` CO-OPN/2 class is the superclass of all our CO-OPN/2 class related to Java. The `JavaObject` class is explained in section 5.4.

We propose to not specify Java *interfaces*. Indeed, the Java programming language do not support multiple inheritance, i.e. each class has exactly one parent class, except the `Object` class, which is the root. Java interfaces are used to define the interface of methods together with their parameters. A class which implements an interface has to implement the body of the methods listed in the interface. For this reason, we think that we need only to specify the classes. If a Java class implements a Java interface, then the corresponding CO-OPN/2 class has in its interface both the methods of the class and the methods of the Java interface.

Java *array* are manipulated by reference, but are not defined with Java classes. Thus, we propose to define a CO-OPN/2 `JavaArray` class for the Java array type. It is defined as an array whose elements are of `JavaObject` class. Java arrays do not inherit from the Java `Object` class, it goes the same for the CO-OPN/2 `JavaArray` class. There is no inheritance relationship between the CO-OPN/2 `JavaArray` class and the `JavaObject` class. An instance of the CO-OPN/2 `JavaArray` class has a reference, given by the CO-OPN/2 semantics, that can be used as a parameter by other CO-OPN/2 classes. We propose that the CO-OPN/2 `JavaObject` class `uses` the CO-OPN/2 `JavaArray` class, thus any Java class in CO-OPN/2 is able to use arrays. Vice-versa the `JavaArray` class `uses` the `JavaObject` class, because it specifies arrays of Java objects.

Java references are *pointers* to class instances. CO-OPN/2 provides the similar notion of *object's identity*. Thus handling a CO-OPN/2 object's identity in a CO-OPN/2 specification, corresponds to handling a Java reference in a Java program. In Java, a class instance handles its own reference using the keyword `this`, in CO-OPN/2, an object handles its own object's identity using the keyword `self`.

Null Type

The Java `null` type can be used instead of any other Java type. The CO-OPN/2 semantics does not provide such an object. It is necessary to define for each CO-OPN/2 type a null object. For this reason, we will not specify the Java `null` type. When necessary, we will formalize the use of the `null` type with an explicit (perhaps complex) specification.

5.2 Java Methods

Methods Sequentiality

A Java method is a *sequential* code operating on data. It is through the method invocations that data is modified or checked. Interfaces of methods, i.e. their name and parameters, are visible for a programmer, but their implementation is not visible for the programmer. The method's caller is blocked until the method returns.

Parameters and Global, Local Method's Variables

A method may access a *global* variable, i.e. a variable that other methods of the same object can also access. A method may also use *local* variables, and input/output *parameters*, these variables

and parameters are not visible for other methods. In Java, two methods accessing a global variable, access the *same* copy of the variable. On the contrary, parameters and local variables are *duplicated* when two concurrent calls to the same method occur.

Swap and Out-of-Order Writes

A thread always works on a copy of a global variable. This gives rise to subtle execution schemes.

Swap

Consider the Java class `Sample` of figure 5.1. This class defines two methods: `hither()` and `yon()`.

```
1   class Swap{
2     int a=1, b=2;
3     void hither(){
4       a=b;
5     }
6     void yon(){
7       b=a;
8     }
9   }
```

Figure 5.1: Swapping of Global Variables

We assume that two different threads `t1`, `t2` call simultaneously these two methods of a same instance `o` of this class. For instance, thread `t1` calls `o.hither()` while thread `t2` calls `o.yon()`. Three possible executions may happen giving rise to three different results:

- Thread `t1` reads the value of `b`, it reads `2`, and modifies `a` accordingly. Afterwards, thread `t2` reads the value of `a`, it sees `2` because `t1` already modified it, `t2` then modifies `b`.
  Final result: `a=2`, `b=2`
- Thread `t2` reads the value of `a`, it reads `1`, and modifies `b` accordingly. Afterwards, thread `t1` reads the value of `b`, it sees `1` because `t2` already modified it, `t1` then modifies `a`.
  Final result: `a=1`, `b=1`
- Thread `t1` reads the value of `b`, it stores `2`. Simultaneously, thread `t2` reads the value of `a`, it stores `1`. Thread `t1`, then modifies `a`, thus `a` has value `2`. Finally, thread `t2`, then modifies `b`, thus `b` has value `1`.
  Final result: `a=2`, `b=1`.

These three possible executions are due to the fact that Java threads work on copies of global variables that they store in their working memory and that an instruction of the form `a=b` consists of several actions: (1) the value of `b` is `load`ed, (2) the value of `a` is `assign`ed. Between these actions other actions may happen.

Out-of-Order Writes

Two consecutive instructions happen necessarily in sequence but in the meanwhile some other instruction may also happen.

We consider the Java class `OutOfOrder` of figure 5.2. It defines two methods: `to()` which is made of two instructions, and `fro()` which prints some output.

```
1   class OutOfOrder{
2     int a=1, b=2;
3     void to(){
4       a=3;
5       b=4;
6     }
7     void fro(){
8       System.out.println("a= " + a + ", b= " + b);
9     }
10  }
```

Figure 5.2: Out-of-Order Writes of Global Variables

If we assume that two threads simultaneously call these two methods of a same instance of a class. Several different output may happen, and several different values may reside in main memory.

- The `fro()` method is executed after the `to()` method has returned.
  The output is: `a=3, b=4`.
  The main memory has values: `a=3, b=4`.
- The `to()` method is executed after the `fro()` method has returned.
  The output is: `a=1, b=2`.
  The main memory has values: `a=3, b=4`.
- The `fro()` and `to()` methods occur in an interleaved way. The main memory has values `a=3, b=4`, but several different outputs may happen:
  `a=3, b=2` or `a=1, b=4`.

Propagation of Thread's Reference

Except `Runnable` objects, there are no active objects in Java. In our model, we restrict the `Runnable`s to Threads only. Thus, from now on, we will only consider threads, but the same discussion applies for `Runnable` objects in general. Every method call is embedded in a body of another method, which is currently being called, and so on till the most enclosing method which is part of the body of a `run` method of a thread. This thread has generated, by the means of its `run` method, all this cascade of method's call. Thus, this thread is actually the caller of all these methods. This point is particularly important when we consider the notions of object's locks and of locker of an object. The locker of an object is the thread which is behind the cascade of method calls that conducted to the action of performing a lock on an object. See 5.4.1.

The Java reference of the thread which initiates the cascade of method's call is propagated to each method called in the cascade. The Java program of figure 5.3 contains four classes: (1) the `T` class, which is a thread class; (2) the `O1` class which is a Java `Object` class with an extra `m1` method; (3) the `O2` class which is a Java `Object` class with an extra `m2` method; (4) the `Z` class which is simply a Java `Object` class.

A thread `t` of class `T` creates an object `z` of class `Z` and performs a `synchronized` statement on `z`. The `synchronized` statement requires a call to method `m1` of a newly created object `o1` of class `O1`. Method `m1` requires a call to method `m2` of a newly created object `o2` of class `O2`. The reference of `z` is passed as parameters to both `m1` and `m2`, thus method `m2` is able to require a `synchronized` statement on `z`.

The cascade of methods is listed below from the most enclosing one to the most embedded one:

1. `run()` (line 9);
2. `o1.m1(z)` (line 14);
3. `o2.m2(z)` (line 28);
4. `System.out.println("Synchronized on z from within o2")` (line 35).

A method caller is blocked until the method returns. Thus, thread `t` is delayed until `o1.m1(z)` returns, which is itself delayed until `o2.m2(z)` returns, this method is, in turn, delayed until `System.out.println( ... )` returns. Remember that the original call to `o1.m1(z)` is part of a `synchronized` statement on `z`. Due to the Java semantics of the `synchronized` keyword, only one thread can lock an object at a time. In our example, thread `t` is currently locking object `z` when method `m2` requires the lock on `z`, because the `synchronized` statement on `z` from the `run` method is not yet finished.

The output of this program is the following:

```
Synchronized on z from within t
Synchronized on z from within o2
```

Then the program stops. This program shows that the Java reference of `t` is propagated to method `m1` and then to method `m2`. Thus, when method `m2` performs the `synchronized` statement on `z` the program is not blocked waiting for `t` to release its lock on `z`.

```
1   package ThreadPropagation;
2
3   import java.io.*;
4   import java.net.*;
5   import java.util.*;
6
7
8    class T extends Thread {
9     public void run(){
10        Z z = new Z();
11        synchronized(z){
12        System.out.println("Synchronized on z from within t");
13        O1 o1 = new O1();
14        o1.m1(z);
15      }
16    }
17
18    public static void main(String argv[]){
19      System.out.println("0");
20      T t= new T();
21      t.start();
22    }
23  }
24
25   class O1{
26    public void m1(Z z){
27      O2 o2 = new O2();
28      o2.m2(z);
29    }
30  }
31
32   class O2{
33    public void m2(Z z){
34      synchronized(z){
35        System.out.println("Synchronized on z from within O2");
36      }
37    }
38  }
39
40   class Z{
41   }
```

Figure 5.3: Propagation of Thread's Reference in a cascade of methods calls

The propagation of the thread's reference ends when a new thread is created, i.e. when a method `start()` is reached in the cascade of method's calls. In this case the reference of the caller is no longer propagated, instead the reference of the newly created thread is propagated, firstly from its `start()` method to its `run()` method, and subsequently to all the methods that are called from within its `run` method.

Inheritance and Overriding

Subclasses *inherit* the methods of their superclasses. A subclass may keep unchanged the method, thus it inherits of the superclass implementation. A subclass may change the method's implementation, thus it *overrides* the superclass method. The implementation provided by the superclass is no longer available for the subclass, unless it invokes explicitly the superclass implementation using the `super` keyword in calls of the form `super.m()` where `m` is the father's implementation of the method `m`. The `super` keyword can be used from within a direct subclass, i.e. constructions of the form `super.super.m()` calling method `m` of the grandfather class are not allowed. A subclass may add new methods, they are available only for the subclass and its children, but not for its superclass.

CO-OPN/2 Specification

Methods Sequentiality

Consider a Java thread `t` performing instruction `y1=o1.m1(x1)`. This instruction calls method `m1(x1)` of object `o1`, method `m1(x1)` requires a parameter `x1` and returns a value `y1`. Due to the Java semantics, both `x1` and `y1` are references of two Java objects.

The Java objects `t` and `o1` are specified by CO-OPN/2 objects. For convenience, we call `t` and `o1` their respective CO-OPN/2 object's identity. The Java method `m1(x1)` begins with a { and ends with a }. In between, several sequential Java instructions actually build the method's body.

Java method `m1(x1)` is specified by several CO-OPN/2 methods:

1. A CO-OPN/2 `start_m1(x1,id1)` method
   This CO-OPN/2 method is called by the CO-OPN/2 object `t` (modeling the Java thread `t`). The `start_m1(x1,id1)` method stores: (1) any input parameter `x1` as a pair `<x1,id1>` into a dedicated place (one for each input parameter), (2) the *caller's identity* `id1` into a dedicated place, and (3) any local variable `local1` needed by the method as a pair `<local1,id1>` into a dedicated place (one for each local variable). The `start_m1(x1,id1)` corresponds to the { of the Java method. We will explain in the sequel what we mean by caller's identity, and why we need it.

2. Several CO-OPN/2 `next` methods
   The sequential code of the Java method is specified by several CO-OPN/2 methods. Each instruction of the method's body is specified by one or more CO-OPN/2 methods, called `next`. Such a `next` method can be fired only if the previous `next` has finished, and as soon as itself finishes it allows the consecutive `next` to be fired. The first `next` is firable only if `start_m1(x1,id1)` method has been fired, and the firing of the last `next` enables the `end_m1(y1,id1)` method to be fired. The sequence of `next` methods respects the sequence of instructions of the Java method's body. A `next` always needs a caller's identity, i.e. an `id1` token, in a place, removes it from this place and puts it in another place, where it is waited by the consecutive `next`.

   All these CO-OPN/2 `next` methods are only called by the CO-OPN/2 object specifying the Java scheduler, they are not called by the CO-OPN/2 `t` object, which is the caller of the `start_m1` method. They are all called with the name `next` (for every method of every object). The scheduler is another CO-OPN/2 object formalizing the scheduler of a Java Virtual Machine. The scheduler permanently loops: it calls one firable `next` method, waits for its complete execution, and then calls another firable `next` method (possibly of another object), etc.

3. A CO-OPN/2 `end_m1(y1,id1)` method
   This CO-OPN/2 method is called by the CO-OPN/2 object `t`. It removes the caller's identity from a dedicated place, as well as all the local variables and input/output parameters from their own places. In addition it returns all the output parameters `y1`. The action of removing the caller's identity, and the local variables and parameters corresponds to the } of the Java method.

   It is worth noting, that an input parameter `x1` is passed to a CO-OPN/2 method as an object's identity, thus the method may have modified its internal state. The method's caller also has the knowledge of the input parameter's identity, thus, at the end of the method, the caller handles the object `x1` with a possibly modified state. It goes the same in Java, the input parameter is the reference of an object.

   We decided to model the end of a method with a dedicated CO-OPN/2 method, `end_m1`. It has to be explicitly called by the caller of the method, in order to: (1) obtain a result (eventually a void one) and (2) wait for the end of the method's computation.

Consider the Java thread `t` performing instruction `y1=o1.m1(x1)`. Figure 5.4 depicts the CO-OPN/2 specification of the Java method `m1` of object `o1`.

The `start_m1(x1,id1)` method is the only method of this figure which can be called by an external object, for instance `t`. As soon as it is called, it stores the input parameter `x1` as the pair `<x1,id1>` in the place called `x1`, and a local variable `local1` (needed later in the method's body), as the pair `<local1,id1>` in the place `local1`. Finally, it stores the caller's (`t`) identity `id1` in the place `p1`.

Once `start_m1(x1,id1)` has finished, a `next` method (in the upper right corner of the figure) can be called by the scheduler. As soon as the scheduler calls it, the `next` method removes `id1` from place `p1` to place `p2`, and possibly calls some other method of some other object. Even though it is not depicted in the figure, every `next` method, may need to access to the input parameter `x1`, or the local variable `local1` or a global variable, when it is fired.

The sequence of `next` proceeds. The last `next` of this method, is depicted in the lower right corner of the figure. As soon as it is called by the scheduler it removes `id1` from place `pn-1` to place `pn`, and possibly calls some other method of some other object. In the example chosen here,

this method also provides the output parameter `y1` of the Java `m1` method, and stores it as the pair `<y1,id1>` in the place `y1`. It is worth noting that the output parameter could have been provided by any of the `next` methods, or could already exist in the object `o` as a global variable. Finally, the `end_m1(y1,id1)` method can be called by the method's caller, `t`. The `end_m1(y1,id1)` method removes the local variable `local` from the `local` place, the input and output parameters `x1` and `y1` from their respective places, and the caller's identity from the `pn` place.

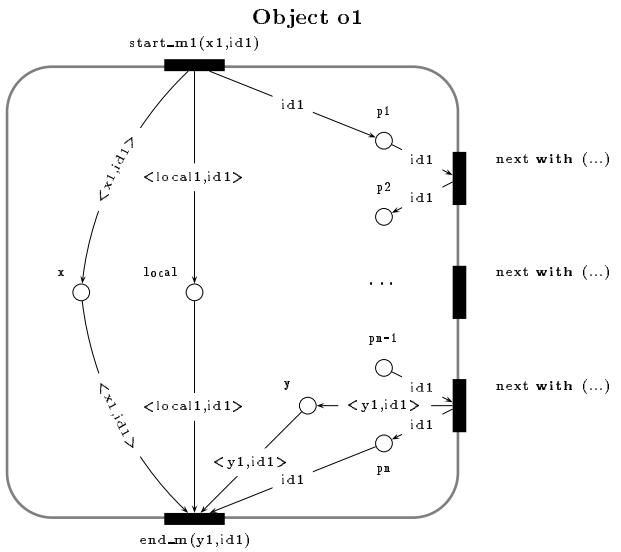


Figure 5.4: CO-OPN/2 Specification of a Java Method

Remark 5.2.1 *There were three possible ways of realizing the sequentiality of Java methods within CO-OPN/2: (1) a sequence in a synchronization expression (i.e. without extra transition or method), (2) a sequence of CO-OPN/2 transitions, (3) a sequence of CO-OPN/2* `next` *methods. We decided to use the third possibility,* `next` *methods, and not a sequence in a synchronization expression, or CO-OPN/2 transitions. Due to the CO-OPN/2 semantics, if we used a sequence of methods calls in a synchronization expression, all the method's body would be realized in an atomic way (all or nothing), this would be in contradiction with the Java semantics. Several instructions of a Java method's body can be executed, before the system could enter in a state, where no more instructions can be executed for instance.*

The CO-OPN/2 semantics forces a stabilization of all the transitions between two firings of CO-OPN/2 methods. If we used a sequence of CO-OPN/2 transitions, as soon as a CO-OPN/2 method (modeling a Java method) is fired, the whole body of the Java method would be immediately executed. This would prevent concurrency and interleaving of the Java instructions of several methods.

Parameters and Global, Local Method's Variables

A Java method may be called simultaneously by several different objects or several times by a same object, with several different parameters. A method handles *global variables*, *parameters* and *local variables*. In Java, as soon as a method is invoked, the parameters and the local variables of the method are duplicated, so that every method invocation induces a method execution with a separate memory space for parameters and local variables. On the contrary, global variables are not duplicated and every method invocation accesses the same instance of the global variables.

In CO-OPN/2, in order to identify each method invocation and execution, together with their private memory space for local variables and parameters we introduce the notion of *caller's identity*. The caller's identity `id` is a pair `id=<cnt,t>`, where `cnt` is an integer and `t` is the CO-OPN/2 object's identity of the threads which has initiated the cascade of methods calls leading to the current method call. The `cnt` is used to distinguish concurrent calls to a same method, and the `t` part is the reference of the caller thread, it stands for the Java reference of this thread. The `t` part is used to propagate the reference of the thread which is the initiator of the method calls. A special `Counter` object provides unique counters, `cnt`, to caller objects. Before calling a method, the caller object must require this unique counter.

As soon as a method is invoked, each input parameter `x` is stored in a dedicated place under the pair `<x,id>`, where `id=<cnt,t>` is the caller's identity. Local variables, `local`, and output parameters, `y`, are created when necessary and stored, similarly to input parameters, under pairs `<local,id>` and `<y,id>` respectively, into dedicated places. They are stored as pairs in order to be retrieved when concurrent calls to the same method occur. Any global variable `global` needed by the method is (already) stored in a separate place as the `global` token.

In a Java program, global as well as local variables and parameters may be used only once or several times, i.e. by several different instructions. In CO-OPN/2 this implies that, all the variables (global or local) and parameters have to be stored in a *separate* places (one for each parameter of variable), in order to be retrieved later by CO-OPN/2 methods or transitions.

Parameters are always passed by reference, thus input/output parameters, `x`,`y`, local and global variables `local` and `global` are CO-OPN/2 object's identity.

We consider now two concurrent calls to the same method of the same object. Two cases may arise: (1) two different Java threads `t1` and `t2` perform simultaneously a call to `y1=o1.m1(x1)` for `t1` and a call to `y2=o1.m1(x2)` for `t2`; (2) a Java thread `t` performs instruction `y1=o1.m1(x1)`, and due to the behavior of method `m1`, it happens that the same thread `t` performs instruction `y2=o1.m1(x2)`, *before* the previous call to `m1` has returned. For instance, if method `m1` calls recursively itself, due to the propagation of the thread's identity, thread `t` is simultaneously calling two times the same method `m1` of the same object `o1`.

In both cases the CO-OPN/2 method `start_m1(x,id)` is called twice simultaneously. What distinguishes the two calls is the `id` part.

Figure 5.5 depicts the case of two simultaneous calls to method `start_m1(x,id)` of object `o1`.

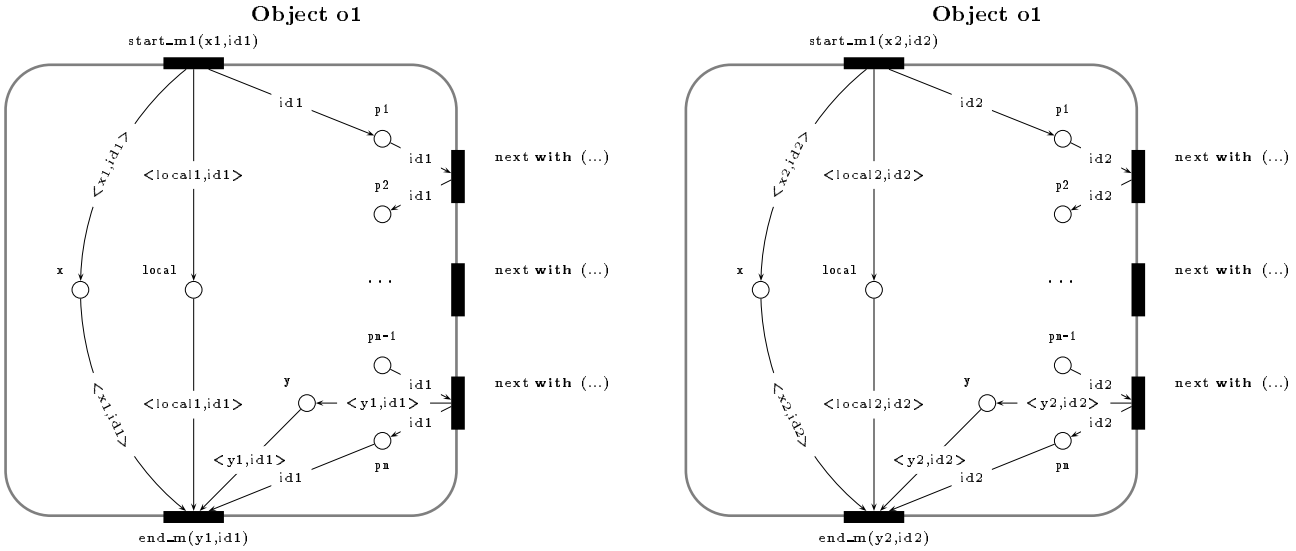


Figure 5.5: CO-OPN/2 Specification of two Concurrent Method Calls

The left part of the figure depicts what happens when the call `o1.start_m1(x1,id1)` is performed, and the right part what happens when the call `o1.start_m1(x2,id2)` is performed. It is worth noting that even if we have depicted separately what happens inside object `o1`, the two calls occur simultaneously inside the *same* object. Thus, place `x`, for instance, is the *same* place in both pictures.

Once `o1.start_m1(x1,id1)` is called variable `x1` is stored as the pair `<x1,id1>` in place `x`. Once `o1.start_m1(x2,id2)` is called variable `x2` is stored as the pair `<x2,id2>` in the same place `x`. The execution of the method proceeds separately for each call. Indeed, the `next` methods are guided by the `id1` caller's identity for the first call and by the `id2` caller's identity for the second call. No confusion may arise in the use of variables: variable `x2` may not be used by the execution needing variable `x1`. No disorder of `next` executions may arise: a `next` may not be performed if the previous `next` of the `same` execution has not been performed.

The caller's identity have the following values:

- If two different threads `t1`, `t2` call respectively
  `o1.start_m1(x1,id1)` and `o1.start_m1(x2,id2)`.
  Caller's identity `id1=<cnt1,t1>`

Caller's identity `id1=<cnt2,t2>`
with `cnt1` different from `cnt2`.

- If a given thread `t` calls simultaneously `o1.start_m1(x1,id1)` and `o1.start_m1(x2,id2)`.
  Caller's identity `id1=<cnt1,t>`
  Caller's identity `id1=<cnt2,t>`
  with `cnt1` different from `cnt2`, but the same thread's identity.

Inheritance and Overriding

The CO-OPN/2 language is an object-oriented language, thus it provides naturally the inheritance and overriding of methods for subclasses.

The Java `super` keyword allows a subclass to call an overridden method of its direct superclass. In CO-OPN/2 is this possible ?

Swap and Out-of-Order Writes

The use of `next` methods and the CO-OPN/2 semantics naturally provide the *Out-Of-Order* writes of global variables described above. However, the problem of *swap* described above is not naturally provided, we have to specify it.

Swap

In order to enable a CO-OPN/2 specification model all the possible behaviors of Java class `Swap` given by figure 5.1, we propose to model Java method `hither()` as if it was defined in the way depicted by figure 5.6. Local variables `tmpb` and `tmpa` are introduced in method `hither()` and

```
1   class Swap{
2     int a=1, b=2;
3     void hither(){
4       int tmpb;
5       tmpb=b;
6       a=tmpb;
7     }
8     void yon(){
9       int tmpa;
10      tmpa=a;
11      b=tmpa;
12    }
13  }
```

Figure 5.6: Swapping of Global Variables

`yon()` respectively. They act as buffers, capturing the value of the global variables `b` respectively `a`. In this manner, the act of reading the value of the global variable and that of assigning a value to another global value are splitted over two instructions.

Figure 5.7 depicts the CO-OPN/2 specification of the Java `Swap` class version with local variables. Java Method `hither()` is specified top-down on the left part. Java method `yon()` is specified bottom-up on the right part of the figure.

The specification of Java method `hither()` needs two `next` methods, one `next` is used to read the value of `b` and to store it in the local variable `tmpb`, and the consecutive `next` actually assigns the value of `tmpb` to global variable `a`. It goes the same for Java method `yon()`.

The adjunction of a local variable used as a buffer enables the CO-OPN/2 specification to model all the three possible executions discussed before. Indeed, the three following transitions may occur in the transition system:

- Method `hither()` before method `yon()`.
  The two `next` of method `hither()` occur before the two `next` of method `yon()`.
  The final result is: `tmpb=2`, `tmpa=2`, `a=2`, `b=2`.

- Method `yon()` before method `hither()`.
  The two `next` of method `yon()` occur before the two `next` of method `hither()`.
  The final result is: `tmpa=1`, `tmpb=1`,`a=1`, `b=1`.

- Interleaving of `next`. Four possible interleaving occur:

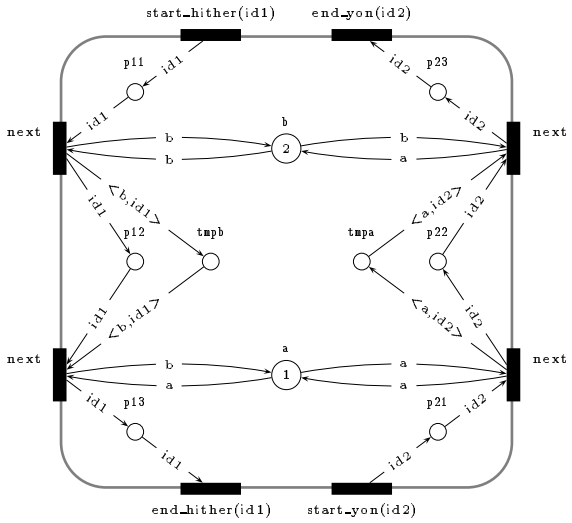


Figure 5.7: CO-OPN/2 Specification of Java Swap of Global Variables

1. First `next` of `hither()` followed by first `next` of `yon()` followed by second `next` of `hither()` followed by second `next` of `yon()`.
   Final result: `tmpb=2`, `tmpa=1`, `a=2`, `b=1`
2. First `next` of `hither()` followed by first `next` of `yon()` followed by second `next` of `yon()` followed by second `next` of `hither()`.
   Final result: `tmpb=2`, `tmpa=1`, `a=2`, `b=1`
3. First `next` of `yon()` followed by first `next` of `hither()` followed by second `next` of `yon()` followed by second `next` of `hither()`.
   Final result: `tmpa=1`, `tmpb=2`, `a=2`, `b=1`
4. First `next` of `yon()` followed by first `next` of `hither()` followed by second `next` of `hither()` followed by second `next` of `yon()`.
   Final result: `tmpa=1`, `tmpb=2`, `a=2`, `b=1`

- Parallel execution of `next`.
  The sequence of `next` for method `hither()` can happen in parallel with the sequence of `next` for method `yon()`. The only restriction is given by the unique token in place `a` and the unique token in place `b`. If two methods wants to access the same token in a place, one method is delayed.
  Final result: `tmpa=1`, `tmpb=2`, `a=2`, `b=1`

If we do not use these extra local variables, the CO-OPN/2 specification has only one `next` for Java method `hither()` and only one `next` for Java method `yon()`. These `next` atomically read the value of `b` (respectively `a`), and replace the value of `a` by that of `b` (respectively the value of `b` by that of `a`).

Due to the CO-OPN/2 semantics, only two transitions occur in the transition system:

- Method `hither()` before method `yon()`.
  The `next` of method `hither()` occurs before the `next` of method `yon()`. The final result is: `a=2`, `b=2`

- Method `yon()` before method `hither()`.
  The `next` of method `yon()` occurs before the `next` of method `hither()`. The final result is: `a=1`, `b=1`

The simultaneity of the two `next`, that of method `hither()` and that of method `yon()` cannot happen, because only one token is available in place `b`, and only one token is available in place `a`. Thus, there are not sufficiently tokens available to allow both `next` simultaneously, and the case leading to the final result `a=2`, `b=1` cannot happen.

Out-Of-Order

The CO-OPN/2 specification of a Java method requires the use of `next` methods. The CO-OPN/2 semantics provides several cases: (1) the sequence of these `next` methods in all possible ways, (2) the simultaneity of these `next` methods if it is possible to fire several of them simultaneously.

The CO-OPN/2 specification of the `OutOfOrder` class of figure 5.2 is given by figure 5.8.

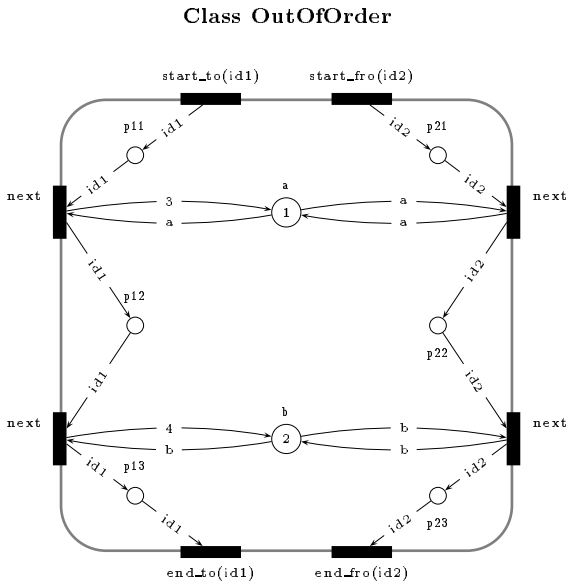


Figure 5.8: CO-OPN/2 Specification of Java Out-Of-Order Writes of Global Variable

Java method `to()` is specified in the left part of the figure. Java method `fro()` is specified in the right part of the figure. Both of them use two `next`. For Java method `to()`, the first `next` specifies the `a=3;` instruction, while the second `next` specifies the `b=4;` instruction. For Java method `fro()`, the first `next` specifies the `"a= " + a` part of the instruction, while the second `next` specifies the `"b= " + b` part of the instruction.

The CO-OPN/2 semantics naturally provides the interleaving of `next` methods. In the Java `OutOfOrder` class, variables `a` and `b` are directly assigned. They are not read for a further use as in the Java `Swap` class. The specification of Java methods with `next` methods is sufficient to model the Out-Of-Order writes behavior. Indeed, given the CO-OPN/2 specification of Java class `OutOfOrder`, CO-OPN/2 semantics provides the following behaviors:

- Method `to()` before method `fro()`.
  The output of `fro()` is: `a=3,b=4`.
- Method `fro()` before method `to()`.
  The output of `fro()` is: `a=1,b=2`.
- Interleaving of methods `fro()` and `to()`.
  The four possible outputs are: (1) `a=1,b=2`; (2) `a=1, b=4`; (3) `a=3,b=2`; (4) `a=3,b=4`.

Method Call

How to actually model the call to a Java method `m1`? As before, we suppose that the Java thread `t` performs the following instruction: `y1=o1.m1(x1)`. It is worth noting that this instruction has to be in the method `run` of the `t` thread.

Figure 5.9 depicts the CO-OPN/2 modeling of the call of Java method `m1` from the `run` method of a thread of class `T`.

The Java `run` method of a thread behaves like any other method wrt the caller's identity. The `run` method receives the caller's identity of the thread which is behind the cascade of methods that reaches this `run`.

Usually, a thread is started with its Java `start` method, and this method calls the Java `run` method. In this case, the caller's identity is a pair `<cnt,t>` where `t=self` is the own identity of the thread.

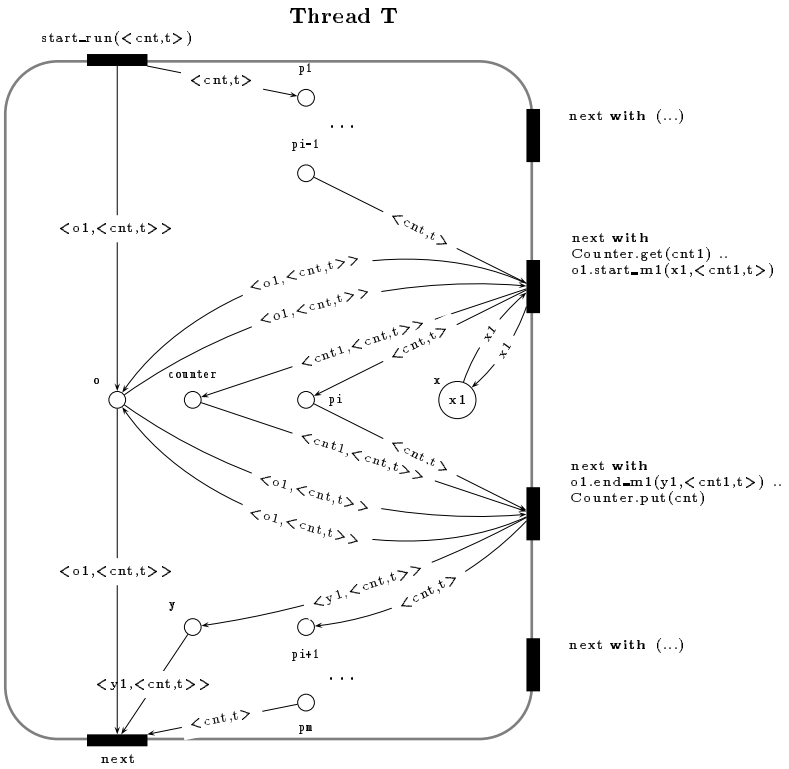


Figure 5.9: CO-OPN/2 Specification of a Method Call

It is also possible for a thread to call directly the `run` method of another thread. In this case, the caller's identity is a pair `<cnt,t>` where `t` is the identity of the thread which called the `run` method.

In the example depicted in figure 5.9, we assume a general case, where the method `run` has been started by some thread `t`, where `t` may be or not `self`.

This `run` is modeled with a `start_run(id)` method, and several `next` methods. No `end_run(id)` method is necessary because the Java `run()` method is the unique Java method that does not block the caller until it returns. A `next` method is used instead of the `end_run(id)` method.

In the body of this `run` is modeled the Java instruction `y1=o1.m1(x1)`. The CO-OPN/2 call to `o1.m1()` and the waiting upon the returned value are splitted into two phases. The call of the method is given by the two following axioms:

```
next with (Counter.get(cnt1) ..  o1.start_m1(x1,<cnt1,t))
next with (o.end_m1(y1,<cnt1,t) ..  Counter.put(cnt1)).
```

These two axioms are part of the method `run`. They have to happen in this order, and with no other axiom of the method `run` they belong to occurring in between. The specification has to ensure that. Remember that a call to a CO-OPN/2 method is realized using the CO-OPN/2 keyword `with()`.

Firstly a counter `cnt1` is acquired, by invocation of method `Counter.get(cnt1)`. The `Counter` object is a global CO-OPN/2 object which provides counters for every other object.

Figure 5.10 depicts the global object providing a counter to every CO-OPN/2 objects specifying a Java object. It has only two methods, `get(cnt)` and `put(cnt)`, which respectively provides a unique counter to an object, and re-acquires a counter from an object. The range of counters goes from 0 to 1000.

After having obtained the counter, the thread composes caller's identity, `<cnt1,t>`. The caller's identity is always the pair `cnt1` (the just obtained counter), and the CO-OPN/2 object's identity of the thread obtained by propagation `t`. Method `start_m1(x1,<cnt,self>)` is then sequentially called using the CO-OPN/2 special character `..` .

The `cnt1` value is stored in the place `Counter`. It will be used later it in order to wait for the end of the called method.

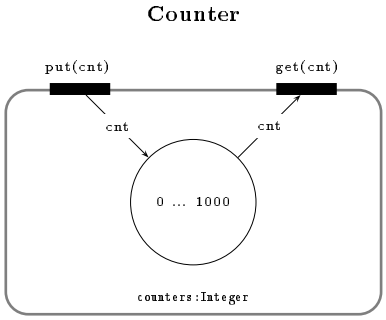


Figure 5.10: CO-OPN/2 specification of `Counter` object

The body of the method then begins to be executed and the caller waits for the end of the method `end_m1(y1,<cnt1,t>` and for a returned value `y`. The same caller's identity is used for both the `start_m1` and the `end_m1` method. After the method has finished, the counter is released `Counter.put(cnt1)` and removed from the `Counter` place. Indeed, a counter is valid only for one method call.

As a summary, we adopt the following convention: a Java method `m` is specified with a CO-OPN/2 method called `start_m`, all the instructions of the method's body are under the control of `next` CO-OPN/2 methods, the end of the Java method is specified with a CO-OPN/2 method called `end_m`. `next` methods are called by the scheduler, while `start_m` and `end_m` are called by method's caller. The first action of `start_m` is to put the thread's identity in a place enabling the first `next` to be fired. The last action of `end_m` is to remove the thread's identity.

If a Java object calls one of its own methods `this.m()`, then the corresponding CO-OPN/2 object uses the recursion on an object provided by the CO-OPN/2 axioms:

```
next with Counter.get(cnt) ..  self.start_m(<cnt,t>)
next with self.end_m(<cnt,t>) ..  Counter.put(cnt)
```

where `t` is the thread's identity obtained by propagation.

Propagation of Thread's Reference

The propagation of thread's reference is realized by the means of the *caller's identity*. Remember that we modeled the caller's identity as a pair: `id=<cnt,t>`, where `cnt` is an integer identifying the method call, and `t` is the CO-OPN/2 object's identity of the thread which initiated the cascade of method's calls. The propagation of the thread's reference is actually performed in the method call. Indeed, each Java method `m(x)` is modeled with the `start_m(x,id)` and the `end_m(y,id)` methods. An extra parameter `id` is needed, this parameter is the caller's identity, it is used to (1) distinguish two or more concurrent calls to a given method, and to (2) propagate the thread's identity.

Figure 5.9 depicts the CO-OPN/2 specification of the thread `t` performing instruction `y1=o1.m1(x1)`. Figure 5.11 depicts the CO-OPN/2 specification of instruction `y2=o2.m2(x2)` performed from within the body of Java method `m1`.

According to figure 5.9, method `start_m1(x,id)` is called by the thread with the caller's identity `id=<cnt1,t>`. In the body of the method is specified the call to the Java method `o2.m2(x2)`.

Similarly to Java method `m1`, Java method `m2` is specified with a `start_m2(x2,id2)` and a `end_m2(y2,id2)` method.

The CO-OPN/2 call and the waiting upon the returned value are splitted into two phases. The call of the method is given by the two following axioms:

```
next with (Counter.get(cnt2) ..  o2.start_m2(x2,<cnt2,t>))
next with (o2.end_m2(y2,<cnt2,t>) ..  Counter.put(cnt2)).
```

These two axioms are part of the CO-OPN/2 specification of method `m1` of object `o1`. These two axioms have to happen in this order, and with no other axiom of the method `m1` occurring in between. The specifier has to ensure that.

The caller, `o1`, firstly acquires a counter `cnt2`, by invocation of method `Counter.get(cnt2)`.

After having obtained the counter, the object composes the caller's identity, `<cnt2,t>`, where `t` is the thread's identity received, by propagation, by method `start_m1(x1,<cnt1,t>)`. It is worth noting that object `o1` calls a method of object `o2` and propagates to this method the thread's identity `t` of the thread which is the initiator of the cascade of methods. However, it does not propagate the counter `cnt1` used by `t` when it called `start_m1()`, because a counter is valid only for one method call. `o1` stores the `cnt2` value in the place `Counter`, in order to use it later.

Method `start_m2(x2,<cnt2,t>)` is then started sequentially.

The body of the method `m2` begins to be executed. The end of the method is reached when `end_m2(y2,<cnt2,t>)` returns, with and for the returned value `y2`. The same caller's identity is used the `start_m2` method and the `end_m2` method. After the method has finished, the counter `Counter.put(cnt2)` released and `cnt2` value is removed from the `Counter` place.

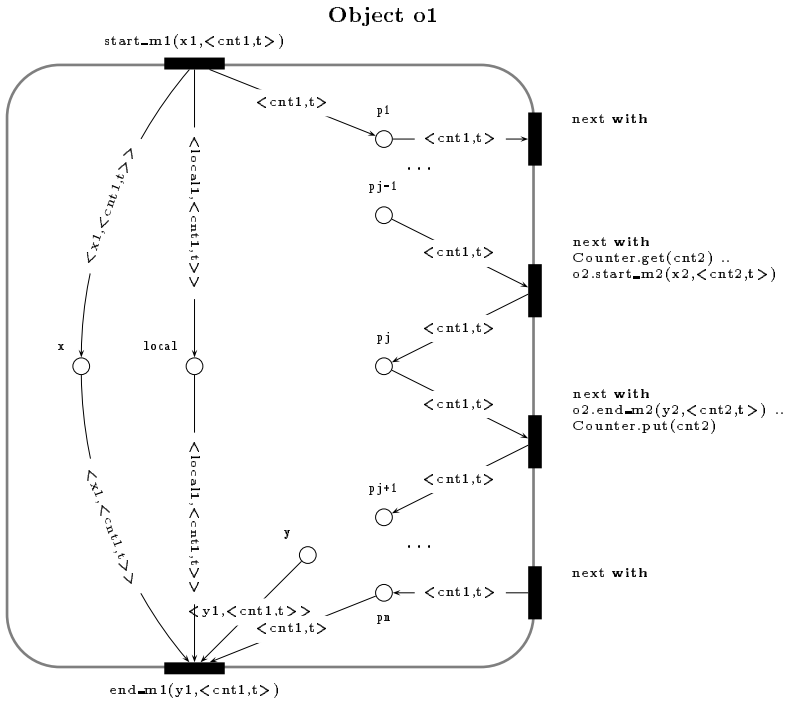


Figure 5.11: CO-OPN/2 Propagation of Thread's Reference

Remark 5.2.2 *The caller's identity* `id=<cnt,t>` *is used: (1) to force the sequence of* `next` *methods; (2) to discriminate local variables and parameters when concurrent calls to the same method occur (*`cnt` *part); (3) to propagate the thread's identity of the thread which initiated the cascade of method calls (*`t` *part); (4) the identity of this thread is used to which perform locks on objects.*

Remark 5.2.3 *Note that, the first action of every Java method specified in CO-OPN/2 is to put the caller's identity in a dedicated place (to mark the beginning), thus even if the first instruction of a method is to call another method there is no CO-OPN/2 call of the form*

```
start_m(x,<cnt,t>) with Counter.get(cnt2) ..  start_m2(x2,<cnt2,t>)
```

There are only CO-OPN/2 calls of the form

```
next with Counter.get(cnt2) ..  start_m2(x2,<cnt2,t>)
```

This way of specifying methods' calls has the following advantages: (1) it respects the sequentiality of the method's call, i.e. the method is called first, and its body is executed afterwards; (2) it avoids circularity of `with` *calls, i.e.* `next` *methods should not be called by other objects than the scheduler, and only* `next` *methods can be combined with other methods' calls.*

5.3 Constructors

In Java, constructors are not inherited, therefore they are not subject to hiding or overriding. If a constructor body does not begin with an explicit constructor invocation and the constructor being

declared is not part of the primordial class `Object`, then the constructor body is implicitly assumed by the compiler to begin with a superclass constructor invocation `super()`. A call to `super()` can only occur from within a method of the direct subclass. A call of the form `super.super()` which would invoke the default constructor of the grandfather class is not allowed.

The CO-OPN/2 Specification

In CO-OPN/2 a field called `Creation` contains all the methods that have to be invoked once an instance of a class is created. This field is never inherited. The method `create` exists by default for every class, and cannot be overridden by the specifier. If a non-default constructor is required, the specifier must add in the `Creation` field the non-default constructor. In the CO-OPN/2 semantics, if, for example, the method `new-constructor` belongs to the `Creation` field of a class `A`, then the call `o.new-constructor`, where `o` is an instance of class `A`, is actually treated as a call to `o.create .. o.new-constructor`.

Multiple constructors can coexist in the `Creation` field of a CO-OPN/2 specification.

The Java `Object()` constructor is empty, we do not specify a CO-OPN/2 constructor in our CO-OPN/2 `JavaObject` class. The CO-OPN/2 default `create` is sufficient.

If a constructor body does not begin with an explicit constructor and the constructor being declared is not part of the primordial class `Object`, CO-OPN/2 assumes that the `create` provided by default is used. Thus, unlike Java, there is no implicit call to the superclass constructor. Therefore, we propose the following: if a Java class has no explicit constructor, then in CO-OPN/2 this class has an explicit constructor which is called `super()` and is the exact copy of the default constructor of the direct superclass.

An explicit Java constructor `B()` for instance for a class `B` is specified in the same way than Java methods. Indeed, two CO-OPN/2 methods `start_new-B` and `end_new-B` specify the begin and end of the constructor, the body is specified with several `next` methods. The `start_new-B` and `end_new-B` appear in the `Creation` field of the CO-OPN/2 specification. As for the specification of Java methods, the specification of Java constructors uses the notion of caller's identity, local variables and parameters are stored in separate places as pairs of value and caller's identity.

A Java constructor may support an overloading of parameters, i.e. the same constructor name can be used with parameters that can vary in quantity and type. Such a constructor is modeled in CO-OPN/2 using several different methods names, one for each possible Java constructor.

Example

In the Java code below, we have three classes `A,B,C`. Class `A` uses both classes `B` and `C`. Class `A` has a default constructor and two methods `mt1` and `mt2`, whose behavior is to create an instance of class `B` and an instance of class `C` respectively. Class `B` has a constructor `public B(Integer i)`, this constructor is not inherited by class `C`. Class `C` is a subclass of class `B`. The `C` constructor has one more parameter, `y`, than the `B` constructor. It uses the `B` constructor by the means of the `super` keyword.

The corresponding CO-OPN/2 specification for classes `B` and `C` is given by figure 5.13.

Usually, a constructor is implemented with several instructions, for this reason, we have treated the specification of a constructor in the same way as the specification of a method. Indeed, the Java constructor `B` is specified with the two methods `start_new-B(x,id)` and `end_new-B(id)`. The caller firstly calls `start_new-B(x,id)` and waits for the end of the constructor calling `end_new-B(id)`. The body of the constructor is modeled with only one `next`. It specifies the Java instruction `j=x;` of line 14 in figure 5.12. It replaces the current value of place `j` by the value of place `x`.

It goes the same for the Java constructor `C()`. The call to the `super()` constructor of line 22 in figure 5.12 has been replaced by the specification of the whole constructor, i.e. in the CO-OPN/2 specification there is no call to a `super()` constructor, thus all the constructor of `B` has been copied in the specification at the place corresponding to the `super()` call. The `next` in the upper right part of the graphical representation of CO-OPN/2 `C` class is exactly the copy of the `next` of CO-OPN/2 `B` class. Java class `C` has one more global variable `k` and one more parameter in the constructor, thus two new places are needed in the CO-OPN/2 specification of class `C`. One for the global variable `k` and the other one for the input parameter `y`. As class `C` is a subclass of class `B`, places `j` and `x` are inherited from `B`. The `next` in the below part of the graphical representation of CO-OPN/2 `C` class specifies instruction `k=y;` at line 23 of figure 5.12.

```
1   public class A{
2     ;; no constructor
3     public void m1(Integer x){
4       B b = new B(x);
5     }
6     public void m2(Integer x, Integer y){
7       C c = new C(x,y);
8     }
9   }
10
11  public class B{
12    Integer j;
13    public B(Integer x){
14      j = x;
15    }
16  }
17
18
19  public class C extends B{
20    Integer k;
21    public C(Integer x, Integer y){
22      super(x);
23      k = y;
24    }
25  }
```

Figure 5.12: Java Constructor

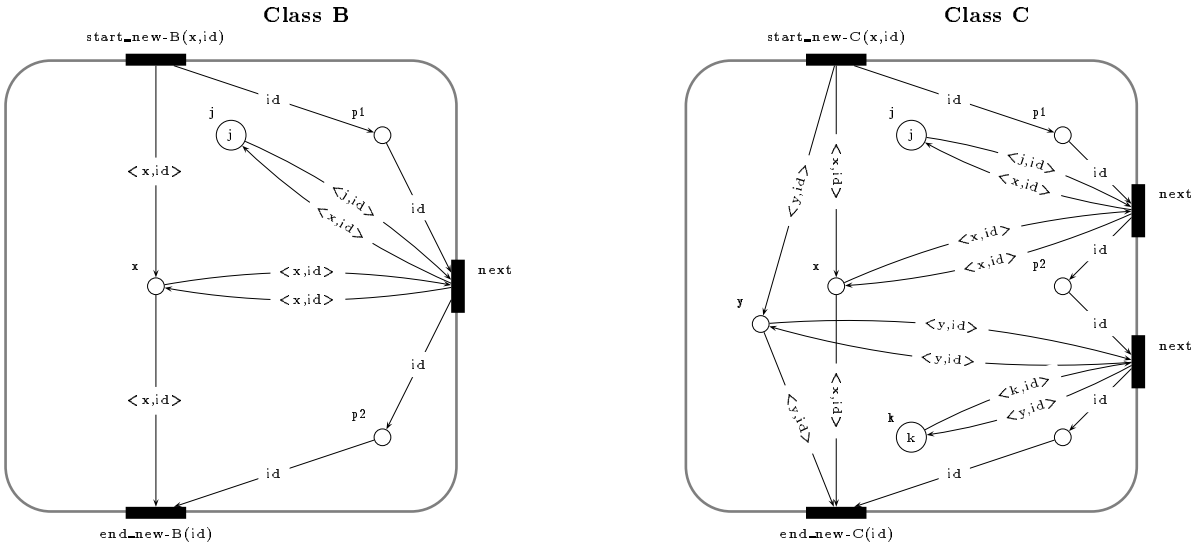


Figure 5.13: CO-OPN/2 Constructors

Constructor Call

Class `A` uses a default constructor, thus there is no CO-OPN/2 specification of a constructor for this class. The default CO-OPN/2 `create` constructor has to be used for creating instances of class `A`. An instance of class `A` is created in CO-OPN/2 by an axiom of the form:

```
next with a.create
```

where `a: A`, the type of `a` is `A`.

Method `m1` of class `A` creates an instance of class `B`, while method `m2` creates an instance of class `C`. As explained above, calls to the constructor are handled as call to methods. Thus, instruction `B b = new B(x);`, of line 4 in figure 5.12, is specified with the two axioms:

```
next with Counter.get(cnt1) ..  b.start_new-B(x,<cnt1,t>)
next with b.end_new-B(<cnt1,t>) ..  Counter.put(cnt1)
```

where `t` has been obtained by propagation from method `start_m1(x,<cnt,t>)` specifying Java method `m1`.

It goes the same for instruction `C c = new C(x,y);`.

```
next with Counter.get(cnt2) ..  b.start_new-C(x,y,<cnt2,t>)
next with b.end_new-C(<cnt2,t>) ..  Counter.put(cnt2)
```

where `t` has been obtained by propagation from method `start_m2(x,y,<cnt,t>)` specifying Java method `m2`.

5.4 The Java Object Class

The CO-OPN/2 `JavaObject` class models the Java `Object` class. We model three Java methods of the Java `Object` class: the `wait`, the `notify` and the `notifyall` methods. Besides these three methods, we model the mechanism for acquiring and releasing a lock on an object. This section firstly presents the Java notion of locks, and how in CO-OPN/2 we have modeled it, and then presents the three methods `wait`, `notify` and `notifyall`, and their model with CO-OPN/2.

5.4.1 Java Locks

"Synchronization is implemented by exclusively accessing the underlying and otherwise inaccessible internal *lock* (sometimes called *mutex*) that is associated with each Java `Object` (...). Each lock acts as a counter. If the count value is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is delayed (*blocked*) until the count is zero. On entry, the count value is incremented. The count is decremented on exit from each `synchronized` method or block, even if it is terminated via an exception." ([10], p.25)

The CO-OPN/2 Specification

The CO-OPN/2 `JavaObject` class maintains a special `locker` place. Two methods interact with this place: `lock(t)` and `unlock(t)`. The type of the `locker` place is given by the pair `<Thread, Integer>`.

As the CO-OPN/2 `JavaObject` class is the superclass of all the CO-OPN/2 classes related to Java, every subclass is provided with the same mechanism of lock described above.

Each class instance `o` is provided with its own `locker` place and the `lock(t)` and `unlock(t)` methods. Roughly speaking, the `lock(t)` methods acquires the lock of object `o` on behalf of the thread `t`. After the firing of the `lock(t)` method, the `t` thread is the locker of the `o` object. Similarly, after the firing of the `unlock(t)` method, thread `t` releases one lock of object `o`.

An extra `locked` place is used to specify that the object is currently locked by no thread.

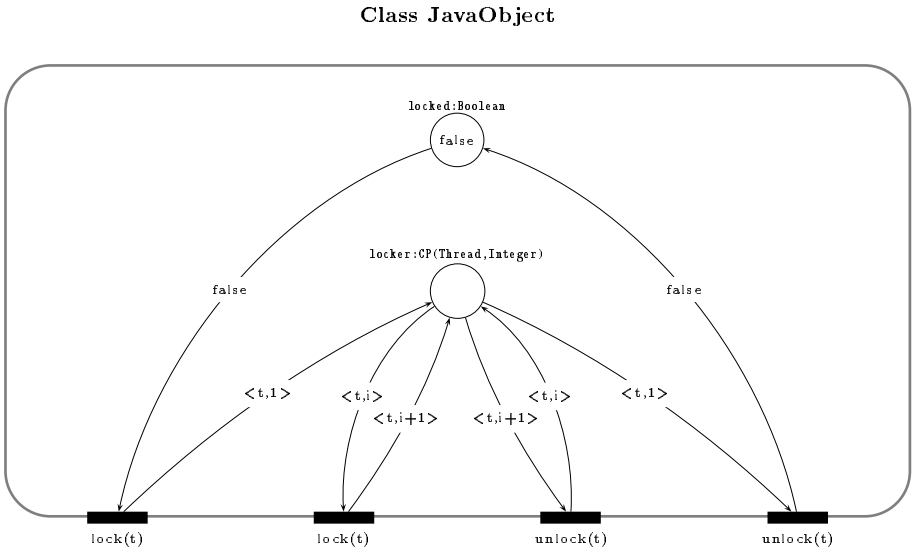


Figure 5.14: CO-OPN/2 specification of Java Locks

Figure 5.14 depicts a part of the `JavaObject` class: the `locker` and the `locked` places, and the two CO-OPN/2 methods `lock(t)` and `unlock(t)`. The `locker` place stores pairs `<t,i>`, where `t` is a thread's identity and `i` is the number of lock that thread `t` has acquired on object `o`.

The `lock(t)` method is given by two axioms. The first axiom (given by the CO-OPN/2 `lock(t)` method on the left of the figure) specifies that if there is no current locker object, then `t` becomes the current locker with one lock on the current object: value `false` in place `locked` is removed and value `<t,1>` is inserted in place `locker`. The second axiom (given by the CO-OPN/2 `lock(t)` method on the middle of the figure) specifies that if the current locker is already `t`, then its number of locks is increased by one: token `<t,i>` is replaced by token `<t,i+1>`. It is worth noting that if `t` is not the current locker, then neither the first axiom nor the second axiom for `lock(t)` can be fired, thus, `t` is blocked until one of these two axioms is firable.

The `unlock(t)` method is given by two axioms. The first axiom (given by the CO-OPN/2 `unlock(t)` method on the middle of the figure) specifies that if the current locker is `t` and if it possesses more than one lock on the current object, then `t` releases one lock: token `<t,i+1>` is replaced by token `<t,i>`. The second axiom (given by the CO-OPN/2 `unlock(t)` method on the right of the figure) specifies that if the current locker is `t` and if it possesses exactly one lock on the current object, then, `t` releases its last lock on the current object, and is no longer the current locker: value `<t,1>` is removed from place `locker` and value `false` is inserted in place `locked`.

This specification allows a thread, different from `t` to release the locks of `t`. A correct specification should avoid that.

5.4.2 Java Synchronized Methods

In order to allow exclusive access to an object, Java offers only one primitive which is the `synchronized` keyword. A Java `synchronized` method `m2` is declared in the following way:

```
synchronized public m2(Integer x)
```

Exclusive access is guaranteed only if *every* method is declared as `synchronized`. Otherwise, the exclusive access is not guaranteed.

How does `synchronized` methods work? In order to execute a `synchronized` method, a thread has to compete for the lock on the object which is the method's owner. In the sequel, this thread is called the locker thread.

- A `synchronized` method ensures that only one thread, at a time can be executing this method. It is the locker thread.
- The locker thread can be executing concurrently several `synchronized` or not methods of a given object.
- Several `synchronized` methods of the same object ensure that only the locker thread can execute them at the same time. Note that this thread can execute some of them simultaneously.
- Consider a given object with some of its methods declared as `synchronized` and some of them not. In this case, exclusive access to the object is not ensured, because *any* thread (locker or not) can execute at any time a non `synchronized` method, even if the locker thread is already executing a `synchronized` method.

"A `synchronized` method automatically performs a *lock* operation when it is invoked; its body is not executed until the *lock* operation has successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked (...). If the method is `static`, it locks the lock associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an *unlock* operation is automatically performed on that same lock."([11] (p.387))

Remark 5.4.1 *Consider two objects making simultaneously an invocation to the same not* `synchronized` *method of a given instance of a class. If the method uses only local variables (local to the method), each invocation will have its own private stack of variables, if the method uses not only local variables, but shared variables, the two invocations will access to the same instance of these variables, thus the result is not predictable.*

CO-OPN/2 Specification

Section 5.2 describes the CO-OPN/2 specification of Java methods, `synchronized` or not. A `synchronized` Java method is specified in the same way than an non `synchronized` method. The CO-OPN/2 method's caller calls the `synchronized` method as if it was a non `synchronized` method. The acquisition of the lock is performed internally by the method's body.

Figure 5.15 depicts the CO-OPN/2 specification of a `synchronized` method. If we assume that in a Java program object `o1` performs instruction `y2=o2.m2(x2)`, where the Java method `m2` is declared in the following way:

```
synchronized public m2(Integer x)
```

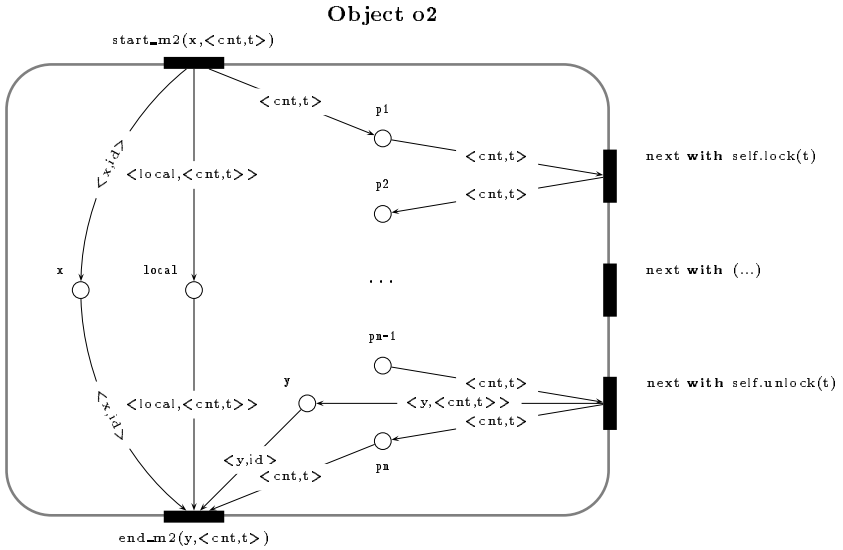


Figure 5.15: CO-OPN/2 specification of Java Synchronized Methods

The CO-OPN/2 specification of the Java `m2` method is realized with: (1) the `start_m2(x,id)` method, (2) a `next with self.lock(t)`,(3) several `next` methods specifying the Java `mt` method body, (4) a `next with self.unlock(t)`, and (5) the `end_m2(y,id)` method. It is the same structure than for a non synchronized method. The difference with a non synchronized method is that two extra `next` methods are needed: one which is fired just after the `start_mt(x,id)` method, and another one which is fired just before the `end_mt(y,id)`. The first `next` is responsible to acquire the lock of object `o2` on behalf of thread `t`. The identity of this thread is obtained by propagation, `id=<cnt,t>`. This thread is actually the thread being the initiator of the cascade of method calls leading to method `start_m2(x,id)`. The last `next` is responsible to release the lock of object `o` which is in possession of the caller, i.e. `t`. The specification of the Java `m2` method is embraced between this pair of `next`: the method's body can be executed only if the lock has been acquired by the caller's thread, and as soon as the method's body is finished the lock is released.

Remark 5.4.2 *Object `o1` calls method `start_m2(x,id)` of object `o2` with the caller's identity `id=<cnt,t>`. The `t` part of the caller's identity is used to realize the synchronized mechanism. Indeed, the `t` part of the caller's identity is used to acquire the lock of `o2`, i.e. the sequence of `next` begins to be executed only when `t` has succeeded to be the locker of object `o2`.*

Remark 5.4.3 *Note that we need both* `cnt` *and* `t` *to discriminate threads. Indeed, if we used only* `cnt` *it would not be possible to know if a given thread is holding a lock on an object, because the* `cnt` *counter is acquired at the beginning of a method and released at the end. The counter is dependent of the method's call, because it is used to discriminate local variables and parameter related to a method call. A same thread has to use two different counters if it is simultaneously in two different method's calls. If we used only* `t` *it would be possible to manage the lock problem, but it would be impossible to discriminate two concurrent calls of the same method by the same thread (recursion), even if the method is a* `synchronized` *method.*

5.4.3 Java Synchronized Statements

A Java `synchronized` statement is a more basic construct than `synchronized` methods. It is of the form

```
synchronized(z) { I }
```

where `z` is an object, and `I` is a block of instructions. A `synchronized` statement always is part of the body of a method.

In order to execute a `synchronized` statement, a thread has to compete for the lock on the object `z` . In the sequel, this thread is called the locker thread.

- A `synchronized` statement ensures that only one thread at a time can be executing the block of instructions `I`. This thread is the locker thread of object `z`.
- The locker thread may be executing simultaneously several `synchronized` or not methods and `synchronized` statements of a given object.
- `synchronized` methods and `synchronized` statements ensure that only the locker thread at a time is executing them (possibly simultaneously).
- Consider a given object with non `synchronized` methods. Any thread can be executing at any time any non `synchronized` statement inside the non `synchronized` methods, even if the locker thread is executing the `synchronized` methods or statements.
- A given thread may request simultaneous locks on several different objects.

"The `synchronized` statement computes a reference to an object; it then attempts to perform a *lock* operation on that object and does not proceed further until the *lock* operations on that object has successfully completed. (...) After the lock operation has been performed, the body of the `synchronized` statement is executed. If execution of the body is ever completed, either normally or abruptly, an *unlock* operation is automatically performed on that same lock." ([11] p.386)

CO-OPN/2 Specification

A (`synchronized` or not) Java method with, in its body a `synchronized` statement is specified in the same way than an non `synchronized` method. The CO-OPN/2 method's caller calls the method as if it there were no `synchronized` statement in its body. The acquisition of the lock is performed internally by the method's body.

Figure 5.16 depicts the CO-OPN/2 specification of a (`synchronized` or not) Java `m2` method, having a `synchronized` statement in its body. We assume that object `o1` performs Java instruction `y2=o2.m2(x2)`, and that in the body of Java method `mt` there is the following synchronized statement: `synchronized(z) { I }`.

The CO-OPN/2 specification of the Java `m2` method is realized with: (1) the `start_m2(x,id)` method, (2) several `next` methods (zero or more),(3) a `next with z.lock(t)`, (4) several `next` instructions specifying the block of instructions `I`, (5) a `next with z.unlock(t)`, (6) several `next` methods (zero or more), and (7) the `end_m2(y,id)` method.

Two extra `next` methods are needed: one which is fired just before the block of instructions `I`, and another one which is fired just after `I`. The `next` before `I` is responsible to acquire the lock of object `z` on behalf of thread `t`, which is the thread that has initiated the cascade of method call leading to the call to `start_m2(x,id)`. The thread's identity is given by the caller's identity, because `id=<cnt,t>`. The `next` after `I` is responsible to release the lock of object `z` which is in possession of thread `t`. The specification of the Java `I` block of instructions is embraced between this pair of `next`: the `I` block can be executed only if the lock has been acquired by thread `t`, and as soon as the `I` block is executed, the lock is released.

5.4.4 Wait, Notify, NotifyAll

"Every object, in addition to having an associated lock, has an associated wait set, which is a set of threads. When an object is first created, its wait set is empty. Wait sets are used by the methods `wait, notify, notifyall` of class `Object`. These methods also interacts with the scheduling

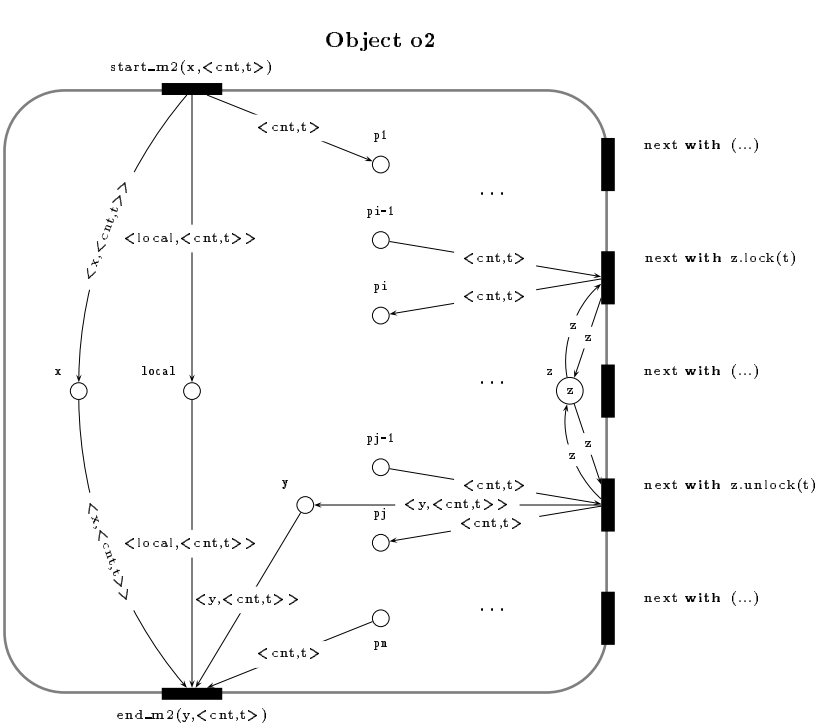


Figure 5.16: CO-OPN/2 specification of Java Synchronized Statements

mechanism for threads. The method **wait** should be invoked for an object only when the current thread T has already locked the object's lock. Suppose that thread T has in fact performed N *lock* operations that have not been matched by *unlock* operations. The **wait** method then adds the current thread to the wait set for the object, disables the current thread for thread scheduling purposes, and performs as many unlock operations as the numbers of locks performed by T on the object to relinquish the lock. The thread T then lies dormant until one of three things happens: (1) some other thread invokes the **notify** method for that object, and thread T happens to be the one arbitrarily chosen as the one to notify; (2) Some other thread invokes the **notifyall** method for that object; (3) If the call by thread T to the **wait** method specified a time-out interval, then the specified amount of real time has elapsed. The thread T is then removed from the wait set and re-enabled for thread scheduling. It then locks the object again (which may involve competing in the usual manner with other threads); once it has gained control of the lock, it performs N - 1 additional *lock* operations and then returns from the invocation of the **wait()** method. Thus on return from the **wait** method, the state of the object's lock is exactly as it was when the **wait** method was invoked." ([11], p.388).

"The **notify, notifyall** methods should be invoked for an object only when the current thread has already locked the object's lock. In the case of the **notify** method, one thread is arbitrarily chosen in the wait set, removed from the wait set and re-enabled; in the case of the **notifyall** method, all the threads in the wait set are removed from the wait set and re-enable. (([11] p.388)

The **wait()** method causes the thread to release only the lock of the object on which it is performing a **wait()**. If the thread is currently locking other object's locks, then these locks will not be released. This can cause deadlocks.

wait() and **notify()** requires a lock on the object.

If a thread T calls its own **wait** method (**this.wait()**), a reference to this thread is put in its own wait-set, and the thread is no longer scheduled by the scheduler until another threads performs a call to **T.notify()** and the scheduler removes T from the wait-set. A thread which performs a **wait()** is blocked in its **run()** method, the other methods can always be called by other threads. Therefore even if a thread is waiting (on himself), other threads can call its methods.

The CO-OPN/2 `JavaObject` class maintains a special place named `wait-set`, of type `Thread`. The Java methods `wait()`, `notify()`, and `notifyall()` are specified as other Java methods. For instance, the Java `wait()` method is specified with: CO-OPN/2 `start_wait(id)`, `end_wait(id)` and several `next` methods, where `id` is the caller's identity, `id=<cnt,t>` with `cnt` a counter number and `t` the thread's identity of the thread which is the initiator of the cascade of methods leading to a call to the Java `wait()` method.

As the CO-OPN/2 `JavaObject` class is the superclass of all the CO-OPN/2 classes related to Java, every subclass is provided with wait sets and the CO-OPN/2 methods specifying Java `wait()` and `notify()` methods. It goes the same for each class instance.

Figure 5.17 depicts a part of the `JavaObject` class: the `wait_set` place and the specification of the Java methods `wait()` and `notify()`. The body of Java method `wait` is depicted on the right part of the figure, while the body of Java method `notify()` is depicted on the left part of the figure.

The `start_wait(<cnt,t>)` method firstly checks if the thread which wants to perform a wait is currently locking the object, i.e. if there is some token `<t,i>` in the place `locker`. Place `locker` has been introduced in section 5.4.1. Token `<t,i>` in place `locker` means that thread `t` locks the object with `i` locks. If `t` is not currently locking the object, the `start_wait(<cnt,t>)` method cannot be fired, and `t` is delayed until it locks the object. As soon as, `t` obtains a lock on the object, the `start_wait(<cnt,t>)` inserts the caller's identity `<cnt,t>` in place `p11`. The first `next` specifying the body of the Java method `wait()` is in the upper right corner of the figure. As soon as the `start_wait(<cnt,t>)` method has been fired, this `next`: (1) removes token `<t,i>` from place `locker` and inserts token `false` in place `locked`, i.e. it releases all locks that `t` maintains on the object; (3) stores this number of locks in place `wait_set`; (3) moves the caller's identity, `<cnt,t>` from place `p11` to place `p12`.

At this point, no methods concerning the execution of Java method `wait()`, with caller's identity `<cnt,t>` is firable, unless a `start_notify(<cnt1,t1>)` is performed.

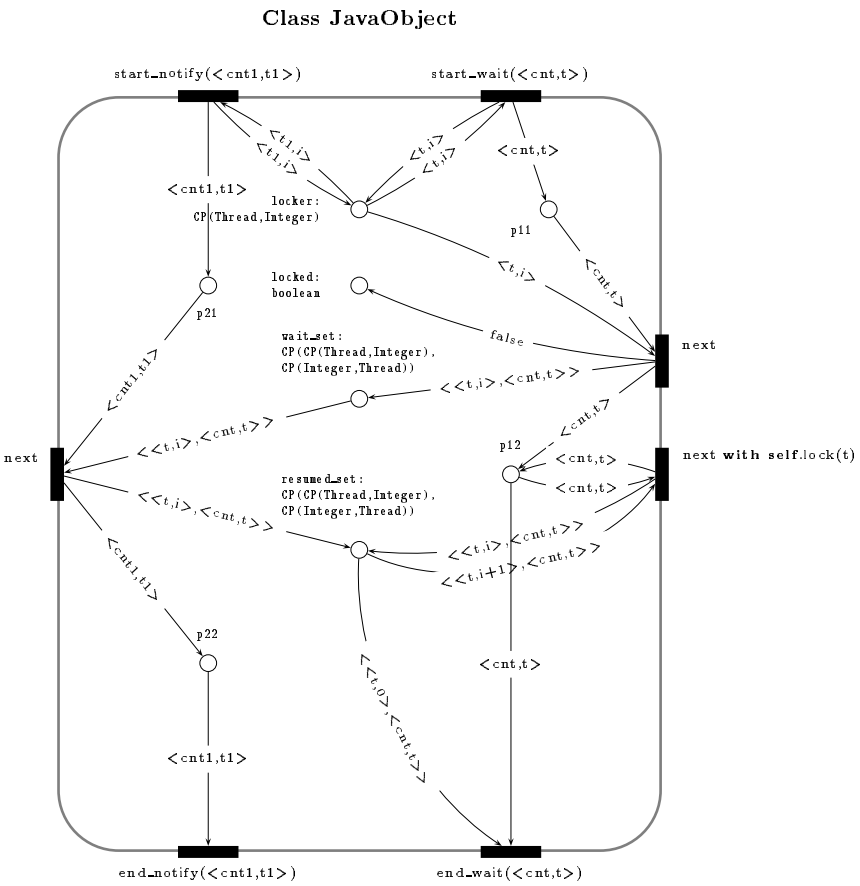


Figure 5.17: CO-OPN/2 specification of the Java Object Class

We consider now that thread `t1` calls `start_notify(<cnt1,t1>`. The `start_notify(<cnt1,t1>`

method checks if t1 owns the lock of the object. If it is not the case, then the start_notify(<cnt1,t1> is not firable until t1 acquires at least one lock. If we assume that t1 owns at least one lock on the object, then start_notify(<cnt1,t1> inserts the caller's identity <cnt1,t1> into place p21. The next on the left part of the figure can then be fired. It moves a token, randomly chosen, from the wait_set place to the resumed_set place. It also moves the caller's identity <cnt1,t1> of the thread which performed the start_notify(<cnt1,t1>) from the p21 place to the p22 place. If the wait_set was empty, then thread t1 is blocked until one thread performs a start_wait(<cnt,t>. Finally, the end_notify(<cnt1,t1>) removes the <cnt1,t1> from place p22 and returns. The CO-OPN/2 specification of the Java notify() method essentially moves one thread from the wait set to the resumed set.

We come back now to the wait() method. As soon as the thread, which performed the start_wait(<cnt,t>) method, arrives in the resumed_set, the next on the below right corner of the figure can be fired. It re-acquires all the locks that have been released by t. When all the locks have been re-acquired, the end_wait(<cnt,t>) method can be fired, and returns.

5.5 Java Keywords

The Java static keyword is specified by the means of the CO-OPN/2 Object field. The Java public keyword has no direct CO-OPN/2 keyword associated, however, the CO-OPN/2 Use field enables a specification to access another (used) specification. The Java private keyword has no direct CO-OPN/2 keyword associated, however, the use of methods in the body or in the interface of a CO-OPN/2 specification let the method begin private or not. The Java extends keyword is specified by the means of the CO-OPN/2 inherit keyword. The Java implements keyword has no CO-OPN/2 field or keyword associated. The Java synchronized keyword has to be specified with several CO-OPN/2 methods (see 5.4.2). void and non void
Abstract Java classes, are classes that are not implemented, in CO-OPN/2 Abstract classes are classes not completely implemented, i.e. instances of such classes make no sense. We can conclude that Abstract Java classes can be mapped into Abstract CO-OPN/2 classes. The Java Generic keyword is specified with the CO-OPN/2 Generic keyword.

5.6 Exceptions

"When a Java program violates the semantic constraints of the Java language, a Java Virtual Machine signals this error to the program as an *exception*. (...) Java specifies that an exception will be thrown when semantic constraints are violated and will cause a nonlocal transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to caught at the point to which control is transferred." ([11], p.34)

"Every exception is represented by an instance of the class Throwable or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by catch clauses of try statements. During the process of throwing an exception, a Java Virtual Machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, static initializers, and field initialization expressions that have begun but not completed execution in the current thread. (...) until a handler is found that indicates that it handles the thrown exception (...). Locks are properly releases as synchronized statements and invocations of synchronized methods complete abruptly." ([11], p.35)

"When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically enclosing catch clause of a try statement that handles the exception. A statement or expression is *dynamically enclosed* by a catch clause if it appears within the try block of the try statement of which the catch class is a part, or if the caller of the statement or expression is dynamically enclosed by the catch clause. ([11], p.36)

If an exception occurs inside a try statement made of several instructions, control is transferred from the point in the code where the exception has occurred to the handler. Thus, any instruction contained in the try statement and following the instruction that caused the exception is not executed.

5.7 Java Threads

Each Java Virtual Machine can support many threads of execution at once. These threads independently execute Java code that operates on Java values and objects residing in a shared main memory. Threads may be supported by having many hardware processor, or by time-slicing many hardware processors. ([11], p.53)

The Java Virtual Machine initially starts up with a single non-daemon thread which typically calls the method `main` of some class. ([11], p.53)

To synchronize threads, Java uses *monitors*, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of *locks*. A thread can suspend itself with a `wait` until such time as another thread awakens it using `notify` or `notifyall`. ([11], p.54)

If two or more concurrent threads act on a shared variable, there is a possibility that the actions on the variable will produce time-dependent results. Every thread has a *working memory* in which it keeps its own *working copy* of variables that it must use or assign. As the thread executes a Java program, it operates on these working copies. The main memory contains the *master copy* of every variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. ([11], p. 371).

"A thread can *use, assign, load, store, lock, unlock* any object (possibly a thread), while the main memory subsystem can *read, write, lock, unlock* an object. Each of these operations is atomic (indivisible)." ([11], p.371).

The *use, assign* operations are interactions concerning the working memory of the thread, while the *read, load* and *store, write* are operations concerning both the working memory and the main memory. "There may be some transit time between main memory and a working memory, and the transit time may be different for each transaction; thus, operations initiated by a thread on different variables may be viewed by another thread as occurring in a different order. For each variable, however, the operations in main memory on behalf of any one thread are performed in the same order as the corresponding operation by that thread." ([11], p.372).

The rules for passing data from the working memory to the main memory and vice-versa are such that each time a thread wants to *use* or *assign* a value it is required to first load it from the main memory, then *use* or *assign* it only once, and in the case of an assign then it is required to store it in the main memory, before a subsequent *use* or *assign*.

Consider two methods `a, b` accessing the same variables of a given object, and two threads are created, and that one calls `a` and the other one calls `b`. If neither `a` nor `b` are `synchronized`, the actions seen by the thread calling `a` and those seen by the thread calling `b` seem to happen in a different order. It is the same if one method is synchronized and the other one not. If both methods `a` and `b` are `synchronized` then the operations are seen in the same order as they are performed. (See [11] p.383-385 for the examples)

"Threads are created and managed by the classes `Thread` and `ThreadGroup`. Creating a `Thread` object creates a thread, and that is the only way to create a thread. When the thread is created, it is not yet active, it begins to run when its `start` method is called." ([11], p. 386)

"The `run` method of the `Thread` or of the specified `Runnable` object is the "body" of the thread. It begins executing when the `start()` method of the `Thread` is called. The thread runs until the `run()` method returns or until the `stop()` method of its `Thread` object is called. The static methods of this class operate on the currently running thread. The instance methods may be called from one thread to operate on a different thread. ([8] p. 325)

For the methods we use in our applications, we can say that: `sleep()` can be invoked only by the current thread, while `start(), stop(), run()` can be invoked by other threads. `start()` is a synchronized method, thus it is necessary to acquire a lock before invoking it. `stop, run` are not declared `synchronized` by the default Java API. Nevertheless if a program overrides these methods and declares them `synchronized` then it is necessary to have a lock. Otherwise it is not necessary.

The thread's identity of a thread calling an object's method is propagated to that method and to any other method called from inside that method's body. This is used for a `synchronized` method to know the identity of the thread which is currently calling it and to compare it with the current locker's thread.

CO-OPN/2 Specification

Figure 5.18 depicts the CO-OPN/2 specification of the Java `start` and `run()` methods.

A Java `start()` method is specified with the two CO-OPN/2 methods `start_start(id)` and `end_start(id)`, with `id=<cnt,t>`. These two CO-OPN/2 methods are called by a thread `t`, directly or from a cascade of method calls.

The Java `start()` method is `synchronized`. The body of the Java `start()` method is specified with several `next` methods, in the same manner than other Java `synchronized` methods.

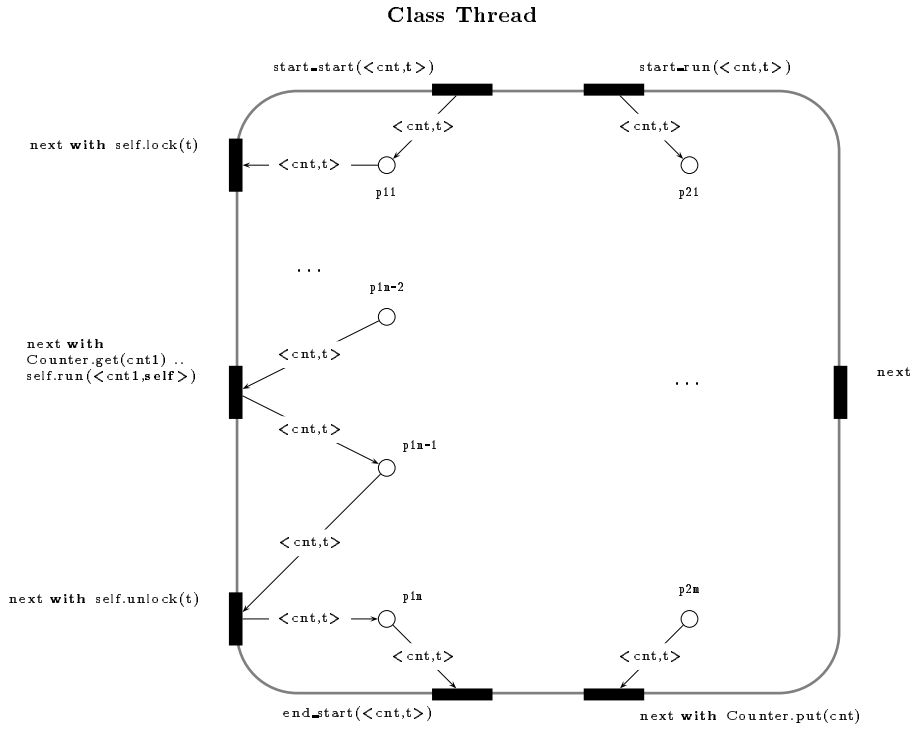


Figure 5.18: CO-OPN/2 Specification of a Java Thread

The important thing to note here is that, just before returning the Java `start()` method calls the `run()` method of the thread which is started. Just before the CO-OPN/2 `end_start(id)` method, the

```
next with Counter.get(cnt1) ..  self.run(<cnt1,self>) ..  Counter.put(cnt1)
```

is fired. This `next` acquires a counter `cnt1`, calls the `run()` method of the current thread `self`, and releases the counter without waiting for the return of the `run()` method. The caller's identity which is sent to the `run()` method is `<cnt1,self>`. It is made of the just obtained counter, and of the own identity of the current thread. The last `next` of the specification of the Java `start()` method *does not propagate* the thread's identity of the thread `t` which called `start_start(<cnt,t>)`. This point is the actual point where a new execution flow is started, which will control its own cascade of method calls. As the Java `start()` method is `synchronized`, the body is embedded into a call to `self.lock(t)` and a call to `self.unlock(t)`.

The Java `run()` method is specified like any other Java method. The `start_run(id1)` method is called by a thread which wants to make the current thread run its execution. The particularity of the threads is that they are separate flow of control. This means, that if a thread `t1` starts a thread `t2`, the first thread `t1` continues its execution just after the `t2.start()` method finishes its execution. At this point both `t1` and `t2` are in their respective `run()` method. Simply said, `t1` is not blocked waiting for `t2` to end its execution. For this reason, there is no `end_run(id1)` method because, the Java `run()` method is the unique Java method that does not block the caller until it returns. For this reason, the CO-OPN/2 specification of the Java `run()` method ends with a `next`, called by the Java scheduler.

Our CO-OPN/2 specification abstracts the notion of working memory, the threads work all on the same reference of an object. This object is responsible to provide its own semantic for allowing

several concurrent accesses. We assume that the specifier has to specify objects shared between several threads so that concurrent accesses are not allowed or properly handled.

The Java `stop()` and `sleep()` methods have not yet been specified, as well as the notion of priorities. Inserts tokens in some places, that the scheduler will check.

Threads constructors enable to pass a `Runnable` target, `r`. The `start()` method of the thread does not calls the `self.run()` method but the `run()` method of the `Runnable` target `r`. The CO-OPN/2 specification may be changed in order to let the `Thread(r)` constructor insert the target `r` in a place, so that later the appropriate `next` specifying the body of the Java `start()` methods performs a call to `r.run(<cnt,self>)` instead of a `self.run(<cnt,self>)`.

5.8 Java Applets

Java applets are piece of code that are moved from one machine to another one, and that are launched by an appletviewer or a Web browser.

The Java `init()` method is used to perform any one-time initialization that is necessary when the applet is first created.

The Java `start()` method is called by the system. It is like the `init()` method, but it may be called multiple times throughout the applet's life.

The Java `stop()` method stops the applet from executing.

The Java `destroy()` method free up any resources that the applet is holding.

The `Applet()` constructor provided by the Java `Applet` class is a default constructor.

Several other methods.

All these methods are called by an applet viewer or a Web browser, they are never called by another object.

In addition, an applet subclass may define methods handling user events Methods used to interact with a user (input,output,graphical user interface). `action`.

These methods are called by the system when the events occur.

CO-OPN/2 Specification

In our CO-OPN/2 specification of the Java `Applet` class 5.19, we only model, the `init()`, `start()`, and `stop()` methods. Java does not provide any body for these methods, i.e. their body is empty. For this reason, the corresponding CO-OPN/2 Specification, depicted in figure 5.19, provides only the pairs of CO-OPN/2 methods: (1) `start_init(id)`, `end_init(id)`; (2) `start_start(id)`, `end_start(id)`; (3) `start_stop(id)`, `end_stop(id)`.

We do not provide a constructor, because the Java constructor of the `Applet` class is a default one, thus we provide also a default constructor.

We skip all the other Java methods being part of the Java `Applet` class.

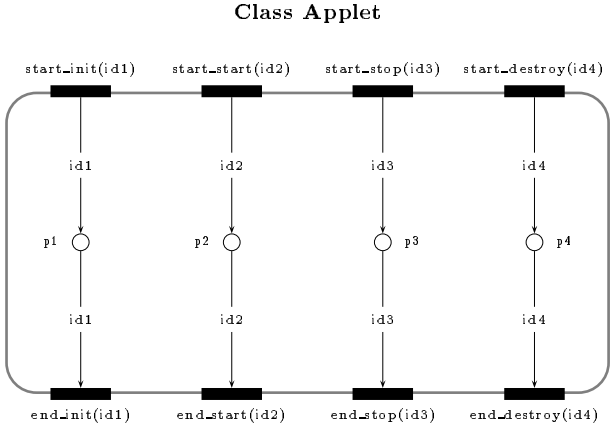


Figure 5.19: CO-OPN/2 Specification of a Java Applet

For what is concerning graphical user interfaces and all the method intended to handle user events, a specifier of a particular applet has to define them.

5.9 Java Sockets

The Java Programming language defines several classes to work with sockets. Two types of communication through a socket are available: (1) a communication based on an underlying reliable connection-based stream protocol; (2) a communication based on an underlying unreliable datagram protocol. A stream protocol is the default.

Our DSGamma application uses the stream protocol, thus the CO-OPN/2 specification only provides a specification for the communication through a socket based on a reliable connection-based stream protocol. Such a protocol implies the following: (1) a connection is established between the partners of the communication before any exchange of messages is performed; (2) messages between partners are received in the same order than the order in what they are sent; (3) no message is lost during the communication.

We focus more precisely on some classes related to the sockets, and on some methods of these classes. We list below the Java classes and their related methods we are interested in:

- `ServerSocket` class:
  The `public ServerSocket(int port)` constructor creates an instance of the class which will wait on `port`.
  The `public Socket accept()` method accepts the requested connection and returns a `Socket` for future communication between client and server.
  The `void close()` method closes the `ServerSocket`.
  Usually, a server creates an object of class `ServerSocket`, this object waits for clients connections on the port specified by the constructor method.

- `Socket` class:
  The `public Socket(String host, int port)` constructor creates the socket and connect it to the specified host and port. The creation of the socket implies the creation of two additional objects: an object of class `InputStream`, and an object of class `OutputStream` for the two communication streams.
  The `public synchronized void close()` closes the `Socket`.
  Usually, a client creates an object of class `Socket` which will connect to a server waiting on a well known port of a well known host.

- `InputStream` class:
  It is an `abstract` class thus its methods are intended to be overridden by subclasses.
  The `public InputStream()` method .. The `public int read(byte[] b)` method .. The `public void close()` method ..

- `OutputStream` class:
  It is an `abstract` class thus its methods are intended to be overridden by subclasses.
  The `public OutputStream()` method .. The `public int write(byte[] b)` method .. The `public void close()` method ..

- `FilterInputStream` class is an actual implementation of the `InputStream` class

- `FilterOutputStream` class is an actual implementation of the `OutputStream` class

- `DataInputStream` class is a subclass of the `FilterInputStream` class, which is able amongst other to communicate integers and not only bytes.

- `DataOutputStream` class is a subclass of the `FilterOutputStream` class, which is able, amongst other, to communicate integers and not only bytes.

CO-OPN/2 Specification

A socket is specified as two queues of bytes, more precisely as two `FIFO(Bytes)` objects: one of these queues is used by the client to write information and by the server to read information, while the other one is used by the server to write information and by the client to read information. The `FIFO(Bytes)` class enables to read information from the queue, i.e. to read and remove the first item at the head of the queue; and to write information to the queue, i.e. to insert an item at the end of the queue.

We skipped the Java `InputStream` and `OutputStream` classes. They are actually replaced with the `FIFO(Bytes)`.

We skipped the Java `FilterInputStream` and `FilterOutputStream` classes.

The CO-OPN/2 `DataInputStream` class requests an underlying `FIFO(Bytes)` to actually read a byte, and convert this byte into an integer. The `DataOutputStream` class converts an integer into a byte and requests the underlying `FIFO(Bytes)` to actually write this byte.

The CO-OPN/2 `Socket` class creates two `FIFO(Bytes)` and connects to the server by the means of the overall system.

The CO-OPN/2 `ServerSocket` class checks the overall system for socket connections. It receives two `FIFO(Bytes)`.

Figures 5.20, 5.21, 5.22, 5.23, 5.24 depict the CO-OPN/2 specifications of respectively the `FIFO(Bytes)`, `ServerSocket`, `Socket`, `DataInputStream`, `DataOutputStream` classes.

- `FIFO(Bytes)` class.

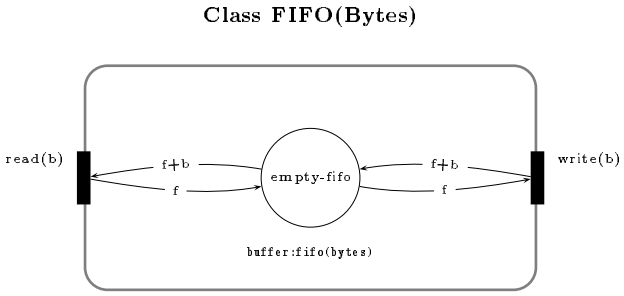


Figure 5.20: CO-OPN/2 Specification of a FIFO of bytes

- `ServerSocket` class.
  The constructor is given by `start_new-ServerSocket(port,id)` and `end_new-ServerSocket(port,id)`. It stores the `port` as a global variable in a dedicated place.
  The Java `accept()` method is given by `start_accept(id)` and `end_accept(id)`. It checks the system for a socket which has requested a connection and returns two `FIFO(Bytes)`. One of them is used to read data from the client, and the other one is used to write data to the client. This is different from the Java `accept()` method which returns a socket.

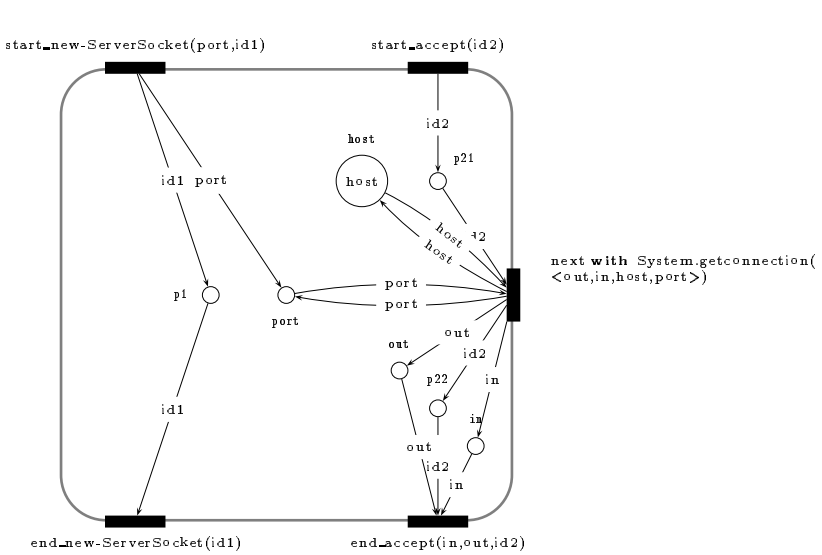


Figure 5.21: CO-OPN/2 Specification of the Java `ServerSocket` class

- `Socket` class:
  The constructor is given by `start_new-Socket(host,port,id)` and `end_new-Socket(id)`. It creates a `Socket` object, and two additional `FIFO(Bytes)`: `in`, `out`. It sends to the system the tuple `<out,in,host,port>`, where `in`, `out` are the two newly created `FIFO(Bytes)`, and `host`, `port` are the host and the port where the server is waiting for connections. The `out` stream is used for bytes send from the client to the server, while the `in` stream is used for bytes sent from the server to the client.
  Java methods `getInputStream`, `getOutputStream` are specified with `start_getInputStream(id)`, `end_getInputStream(in,id)` and `start_getOutputStream(id)`, `end_getOutputStream(out,id)`. They check the place storing respectively the `FIFO(Bytes)`, `in`, and `FIFO(Bytes)`, `out`, related to the socket, and return their reference.

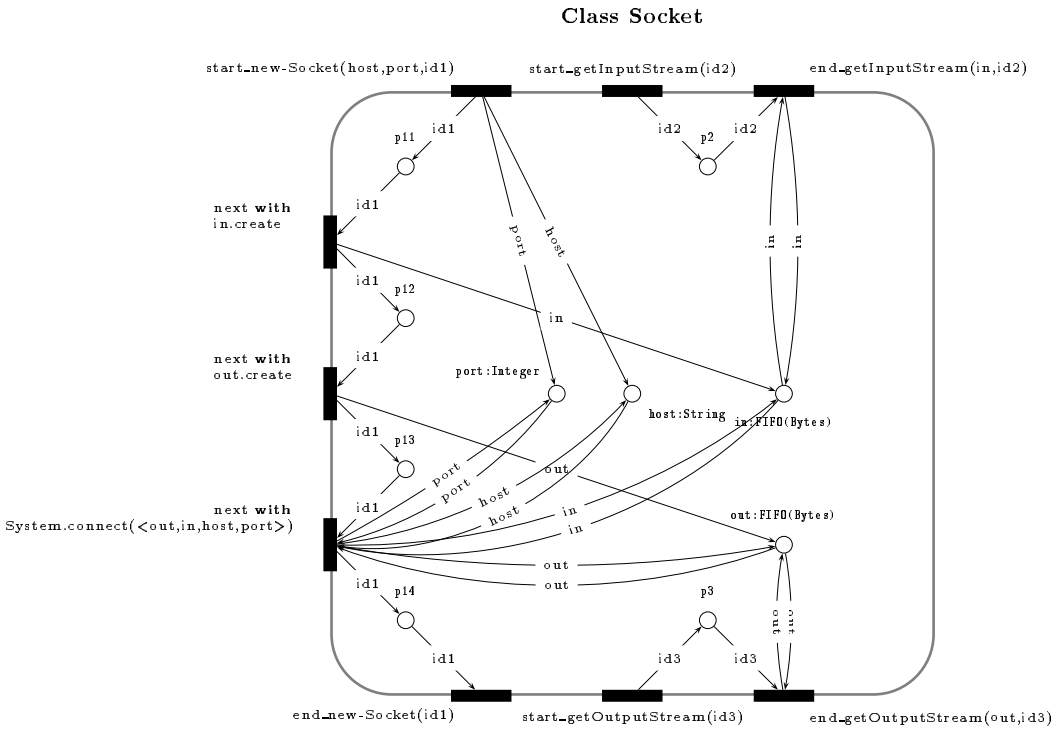


Figure 5.22: CO-OPN/2 Specification of a Java `Socket` class

- `InputStream` class, `OutputStream` class are not specified. The `FIFO(Bytes)` are used instead.

- `FilterInputStream` class and the `FilterOutputStream` class are not specified.

- `DataInputStream` class:
  The constructor is given by `start_new-DataInputStream(in,id)` and `end_new-DataInputStream(id)`. It creates the `DataInputStream` and stores the `FIFO(Bytes)`, `in`, on which the data has to be read from.
  It allows only to read integers from the underlying `in` stream. Java method `readInt()` is specified with `start_readInt(id)` and `end_readInt(i,id)`. It calls the `read` method of the `in` stream and converts the returned byte into an integer `i`. The `in` stream actually reads a byte from the stream.

- `DataOutputStream` class:
  The constructor is given by `start_new-DataOutputStream(out,id)` and `end_new-DataOutputStream(id)`. It creates the `DataOutputStream` and stores the `FIFO(Bytes)`, `out`, on which the data has to be actually written to.
  It allows only to write integers to the underlying `out` stream. Java method `writeInt()` is specified with `start_writeInt(i,id)` and `end_writeInt(id)`. It converts the integer `i` into a byte and calls the `write` method of the `out` stream, which will actually write the byte in the stream.

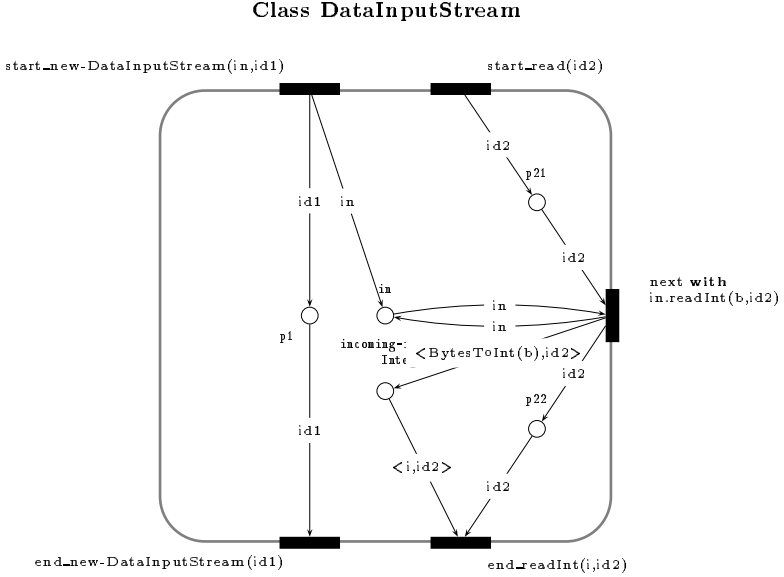


Figure 5.23: CO-OPN/2 Specification of the Java `DataInputStream` class

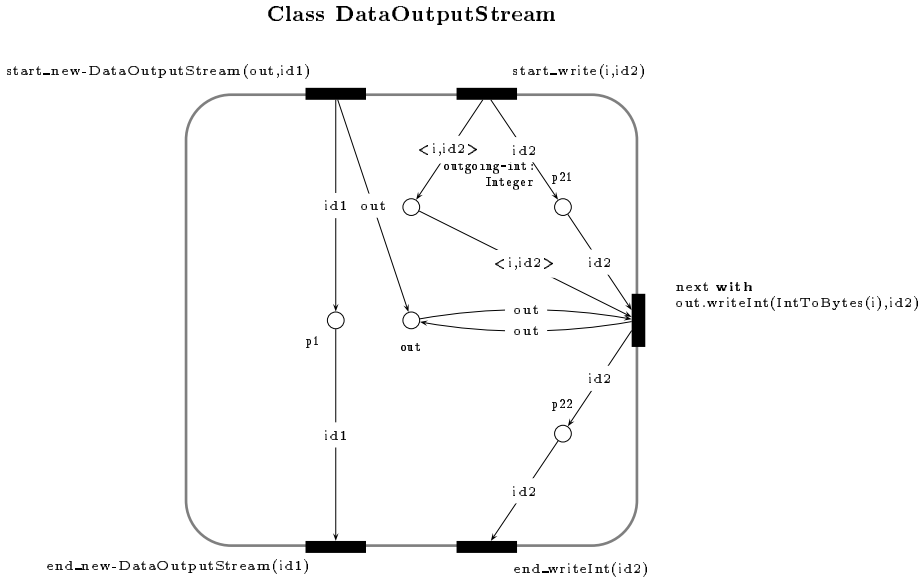


Figure 5.24: CO-OPN/2 Specification of the Java `DataOutputStream` class

The overall system responsible to actually realize the connection between the client and the server is specified with the CO-OPN/2 `System` class. This class has nothing to do with the Java `System` class.

Figure 5.25 shows the place `connections` of type a 4-tuple of two `FIFO(Bytes)`, a host and a port. The `connect` method stores a tuple into the place, and the `getconnection` method removes such a tuple from the place. A `Socket` which connects to a `host` on a port `port` and which uses the `out` and `in` streams sends a tuple `<in,out,host,port>` to the `System`. A `ServerSocket` waiting on `port` of `host` requests connections of the form `System.getconnection(out,in,host,port)`. It is worth noting the interversion of the streams. Indeed, the client uses `in` stream to read from the server, the server uses the `same` stream but to write informations, thus it is an output stream for the server, and input stream for the client. It goes in the similarly for the `out` stream.

5.10 Java Virtual Machine Scheduler

A Java thread is executed sequentially. Several of them are being executed concurrently and simultaneously. A thread suspends its execution when it performs a `wait()` and resumes its execution when the corresponding `notify()` is performed. A method call in the `run()` method of

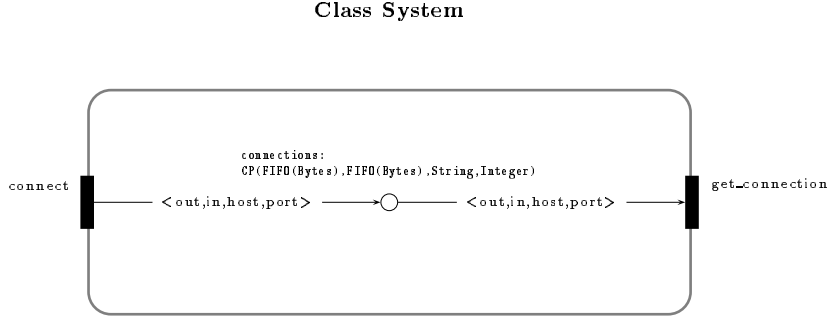


Figure 5.25: CO-OPN/2 Specification of the overall System

a thread returns only after all the nested methods calls have returned. The Java virtual machine is responsible amongst others to correctly halt and resume an execution thread, as well as to correctly perform nested method calls. The Java virtual machine also handles several threads executions.

CO-OPN/2 Specification

The CO-OPN/2 specification of Java methods and Java threads treats the following cases: (1) the `wait()` and `notify()` calls as wells as the halt and resume of the calling thread; (2) the nesting of method calls and the propagation of the thread identity of the calling thread; (3) a sequential specification of the instructions of method body using CO-OPN/2 `next` methods.

The CO-OPN/2 specification of the Java virtual machine scheduler is only responsible to call the `next` methods of every objects present at the beginning of the system's life or dynamically created during the system's life. The CO-OPN/2 specification of the scheduler calls randomly one or more firable `next`. In that way we model the concurrent and simultaneous execution of sequential threads.

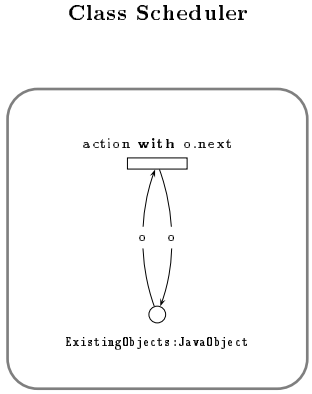


Figure 5.26: CO-OPN/2 Specification of the Java Virtual Machine Scheduler

The `Scheduler` maintains the list of all objects of type `JavaObject` present in the specified system. This list of objects may evolved during the time, because new objects may be dynamically created, or destroyed. A CO-OPN/2 transition, `action` actually calls the `next` methods of the objects. As it is a CO-OPN/2 transition is fired without being called by another object, it is spontaneously fired as soon as the pre-conditions are satisfied and the requested `next` method is itself firable. In the case of the scheduler the pre-condition requires at least one object in the specified system. Due to the CO-OPN/2 semantics the `action` transition may be fired several times simultaneously, thus if several `JavaObjects` have a `next` method firable, it may happen that they are fired all simultaneously.

The `Scheduler is` the actual `engine` that let the whole system specified with CO-OPN/2 evolve. Without this object, the other CO-OPN/2 objects related to a Java application are not able to change their state.

5.11 Appletviewer or Web Browser

A Java appletviewer or a web browser running on a host `host` is responsible to call the methods of all the Java applets running on the same host `host`.

CO-OPN/2 Specification

The CO-OPN/2 specification of an appletviewer of a web browser calls every firable method of one or more applets currently running. For every host, an object of CO-OPN/2 class `Appletviewer` is created. It will handle all the CO-OPN/2 objects of class `Applet` that are specified to be running on the same host. It is worth noting that the CO-OPN/2 `next` methods of an applet's method are called by the CO-OPN/2 `Scheduler`.

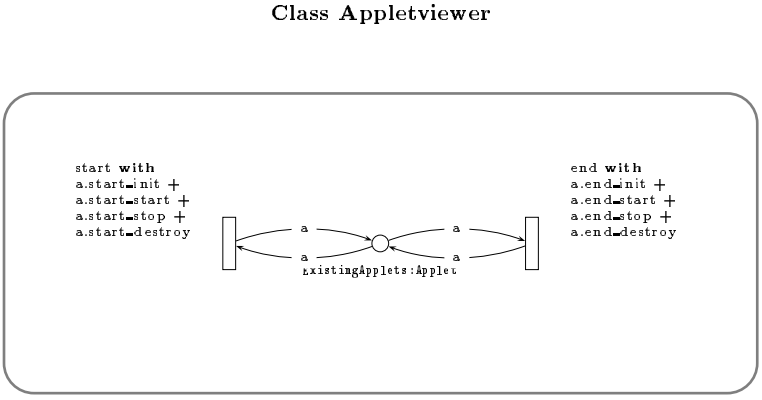


Figure 5.27: CO-OPN/2 Specification of a Java Appletviewer

The `Appletviewer` object maintains the list of `Applet` currently running on its host. The `Appletviewer` object has two CO-OPN/2 transitions `start` and `end` that are spontaneously fired, as soon as an applet `a` has one of the following methods is firable: `a.init`, `a.start`, `a.stop`, `a.destroy`.

5.12 Summary

propagation of thread's identity except with start
each method is called with start and end except run
scheduler calls next and end-run
scheduler handles wait, notify.

Chapter 6

Conclusion

We have proposed in this paper a formal development methodology for Java Web based parallel applications (JPA). Our methodology is based on several refinement steps of both a formal specification and a set of properties the application has to fulfill. Each refinement step is guided by the targeted implementation and by the necessity to fulfill the properties. We applied this methodology on a real Java application, using the CO-OPN/2 formal specification language. Thus we have provided four refinement steps adapted for JPA and we have explained how these refinement steps were influencing one particular system property. We have, using a very simple program, shown that only a precise description of the system and a progressive development based on formal refinement could help in verifying and validating JPA. An automatic generator could be envisaged, it could take in input the most concrete CO-OPN/2 specification and could produce in output a Java program.

Currently, we express only informally the properties, but we envisage to use temporal logic for expressing, verifying and validating the properties. The idea is to give a formal semantics to CO-OPN/2 specifications in terms of temporal logic formulae. Then the properties of the most abstract specification will be proved using a temporal logic deduction system, and each refinement step will have proof obligation over the temporal logic formulae.

This top-down approach for the development of an application may also be considered upon a bottom-up point of view. The idea is to apply a reverse engineering process on a Java program in order to verify and validate its expected properties. If necessary, a re-engineering process may lead to a new safe implementation.

Appendix A

Initial CO-OPN/2 Specification I

```
;; DSGammaSystem class: Centralized View
;; ------------------------------------
;; The global system made of one global MSInt
Class DSGammaSystem;
Interface
  Use Integer;
  Methods
    new_user      _ : Integer;
    user_action _ _ : Integer Integer;
    result      _ _ : Integer Integer;
    user_exit     _ : Integer;
Body
  Places
    MSInt : Integer;
    users : Integer;
  Transition
    ChemicalReaction;
  Axioms
    new_user(usr)
        :: -> users usr;
    user_action(i,usr)
        :: users usr -> users usr, MSInt i;
    result(i,usr)
        :: users usr, MSInt i -> users usr, MSInt i;
    user_exit(usr)
        :: users usr -> ;
        ;; All the possible Chemical Reactions
    ChemicalReaction
        :: MSInt i, MSInt j -> MSInt i+j;
    Where
      i, j : Integer;
      usr  : Integer;
End DSGammaSystem;
```

Appendix B

First Refinement R1: CO-OPN/2 Specification

```
;; DSGammaSystem class: Data Distribution
;; -------------------------------------
;; The global system made of the distributed MSInts(s)
Class DSGammaSystem;
Interface
  Use Integer;
  Methods
    new_user      _ : Integer;
    user_action _ _ : Integer Integer;
    result      _ _ : Integer Integer;
    user_exit     _ : Integer;
Body
  Use Bag, CP(Integer,Bag);
  Places
    MSInt, MSIntToEmpty : CP(Integer, Bag);
  Transition
    CR1, CR2, CR3, CR4, CR5, CR6, CR7, CR8;
  Axioms
    new_user(usr)
        :: -> MSInt <usr,empty-bag>;
    user_action(i,usr)
        :: MSInt <usr,bag> -> MSInt <usr,bag+i>;
    result(i,usr)
        :: MSInt <usr,empty-bag+i> -> MSInt <usr,empty-bag+i>;
    user_exit(usr)
        :: MSInt <usr,bag> -> MSIntToEmpty <usr,bag>;
        ;; All possible Chemical Reactions
    CR1         :: MSInt <usr, (bag + i) + j> -> MSInt <usr, bag + (i+j)>;
    CR2         :: MSInt <usr1, bag1 + i>, MSInt <usr2, bag2 + j>
        -> MSInt <usr1, bag1 + (i+j)>, MSInt <usr2, bag2>;
    CR3         :: MSInt <usr1, (bag1 + i) + j>, MSInt <usr2, bag2>
        -> MSInt <usr1, bag1>, MSInt <usr2, bag2 + (i+j)>;
    CR4         :: MSInt <usr1, bag1 + i>, MSInt <usr2, bag2 + j>,
           MSInt <usr3, bag3>
        -> MSInt <usr1, bag1>, MSInt <usr2, bag2>,
           MSInt <usr3, bag3 + (i+j)>;
        ;; do not add integers in MSIntToEmpty
    CR5         :: MSInt <usr1, bag1>, MSIntToEmpty <usr2, (bag2 + i) + j>
        -> MSInt <usr1, bag1 + (i+j)>, MSIntToEmpty <usr2, bag2>;
    CR6         :: MSInt <usr1, bag1 + i>, MSIntToEmpty <usr2, bag2 + j>
        -> MSInt <usr1, bag1 + (i+j)>, MSIntToEmpty <usr2, bag2>;
    CR7         :: MSInt <usr1, bag1 + i>, MSInt <usr2, bag2>,
           MSIntToEmpty <usr3, bag3 + j>
        -> MSInt <usr1, bag1>, MSInt <usr2, (bag2 + i) + j>,
           MSIntToEmpty <usr3, bag3>
    CR8         :: MSInt <usr1, bag1>, MSIntToEmpty <usr2, bag2 + i>,
           MSIntToEmpty <usr3, bag3 + j>
        -> MSInt <usr1, bag1 + (i+j)>, MSIntToEmpty <usr2, bag2>
           MSIntToEmpty <usr3, bag3>;
    Where
        bag, bag1, bag2, bag3 : Bag;
        usr, usr1, usr2, usr3 : Integer;
        i, j : Integer;
End DSGammaSystem;

Generic ADT Bag(Item);
Interface
  Use Item;
  Generators
    empty-bag : -> Bag(Item);
  Operations
```

```
61      _ + _      : Bag(Item) Item -> Bag(Item);
62  Body
63    Axioms ?
64  End Bag;
```

Appendix C

Second Refinement R2: CO-OPN/2 Specification

```
1  ;; DSGammaSystem class: Behavior Distribution
2  ;; ----------------------------------------
3  ;; The global system made of the applets and the
4  ;; GlobalRelay
5  Class DSGammaSystem;
6  Interface
7    Use Integer;
8    Methods
9      new_user      _ : Integer;
10     user_action _ _ : Integer Integer;
11     result      _ _ : Integer Integer;
12     user_exit     _ : Integer;
13   Creation
14     new-DSGammaSystem;
15 Body
16   Use Applet, GlobalRelay, Integer, CP(Applet, Integer);
17   Places
18     store-applets        : CP(Applet, Integer);
19     GR                   : GlobalRelay;
20   Axioms
21     new-DSGammaSystem with gr.create
22         :: -> GR gr;
23     new_user(usr) with a.new-Applet(gr)
24         :: GR gr
25         -> GR gr,
26            store-applets <a, usr>;
27     user_action(i,usr) with a.user_action(i)
28         :: store-applets <a, usr>
29         -> store-applets <a, usr>;
30     result(i,usr) with a.result(i)
31         :: store-applets <a, usr>
32         -> store-applets <a, usr>;
33
34     user_exit(usr) with a.user_exit
35         :: store-applets <a, usr>
36         -> ;
37     Where
38       gr        : GlobalRelay;
39       usr       : Integer;
40       a         : Applet;
41       i         : Integer;
42 End DSGammaSystem;
43 ;; Class GlobalRelay;
44 ;; -----------------
45 Class GlobalRelay;
46 Interface
47   Use Integer;
48   Methods
49     put _ : Integer;
50     get _ : Integer;
51 Body
52   Use FIFO(Integer);
53   Places
54     buffer : FIFO(Integer);
55   Initial
56     buffer empty-fifo;
57   Axioms
58     put(i) :: buffer b -> buffer b+i;
59     get(i) :: buffer b+i -> buffer b;
60   Where
```

```
    i : Integer;
End GlobalRelay;

;; Class Applet;
;; -------------------------
Class Applet;
Interface
  Use Integer, GlobalRelay;
  Methods
    user_action _ : Integer;
    result      _ : Integer;
    user_exit;
  Creation
    new-Applet  _ : GlobalRelay;
Body
  Use Random, Clock;
  Places
    store-gr      : GlobalRelay;
    MSInt, first  : Integer;
    end           : Boolean;
    beginning     : Boolean;
    timeout       : Integer;
  Transitions
    getfirst, getsecond, tik, put;
  Initial
    end             false;
    beginning       true;
  Axioms
        ;; store gr
    new-Applet(gr)
        :: -> store-gr gr;
        ;; add new integer to MSInt
    user_action(i)
        :: -> MSInt i;
        ;; change flag
    user_exit
        :: end false
        -> end true;
        ;; receive a first integer from system
        ;; provided the user has not exit
    getfirst with (gr.get(i) // random.get(millis) // clock.get(hour))
        :: end false, beginning true, store-gr gr
        -> end false, store-gr gr,
           first i, timeout (hour + millis);
    getfirst
        :: end true, beginning true
        -> ;

        ;; receive a second integer, adds it to first and inserts into
        ;; MSInt
    getsecond with gr.get(j)
        :: first i, timeout d, store-gr gr
        -> beginning true, MSInt i+j, store-gr gr;
        ;; to prevent deadlock when no sufficient integers in the
        ;; system, add only first integer to MSInt.
    tik with clock.get(hour)
        :: (hour > d)
        => timeout d, first i
        -> beginning true, MSInt i;
        ;; removes integer from MSInt until no more integer
    put with gr.put(i)
        :: store-gr gr, MSInt i
        -> store-gr gr;
    Where
      gr              : GlobalRelay;
      i, j            : Integer;
      hour, millis, d : Integer;
End Applet;
```

Appendix D

Third Refinement R3: CO-OPN/2 Specification

D.1 System Operations

```
1  ;; DSGammaSystem class: Communication Layer
2  ;; ------------------------------------
3  ;; The global system made of the RandomRelayServer,
4  ;; the Applets and the ommunication layer.
5  Class DSGammaSystem;
6  Inherit Thread;
7          ;; used only to send its reference as a Thread
8  Interface
9    Use Integer;
10   Type dsgammasystem;
11   Methods
12     new_user      _ : Integer;
13     user_action _ _ : Integer Integer;
14     result      _ _ : Integer Integer;
15     user_exit     _ : Integer;
16   Creation
17     new-DSGammaSystem _ : Integer;
18 Body
19   Use DSGammaClientApp, RandomRelayServer,
20       CounterProvider, CP(Integer,Thread);
21   Places
22     applets        : CP(Applet, Integer);
23     port           : Integer;
24     host           : String;
25   Initial
26     host host;
27     port port;
28   Axioms
29     new-DSGammaSystem(port) with
30         Counter.get(cnt) ..
31         rr.start_new-RandomRelayServer(port,<cnt,self>) ..
32         rr.end_new-RandomRelayServer(<cnt,self>) ..
33         Counter.put(cnt)
34         :: -> port port;
35     new_user(usr) with
36         (Counter.get(cnt) ..
37         a.start_new-DSGammaClientApp(port,host,<cnt,self>) ..
38         a.end_new-DSGammaClientApp(<cnt,self>) ..
39         Counter.put(cnt))
40         :: port port, host host
41         -> port port, host host, applets <a, usr>;
42         ;; a user enters a new integer
43         ;; then propagate this to the user's applet
44     user_action(i,usr) with
45         (Counter.get(cnt) .. a.start_user_action(i,<cnt,self>) ..
46         a.end_user_action(<cnt,self>) .. Counter.put(cnt))
47         :: applets <a, usr>
48         -> applets <a, usr>;
49         ;; a user wants a result
50         ;; then propagate this to the user's applet
51     result(i,usr) with
52         (Counter.get(cnt) .. a.start_result(<cnt, self>) ..
53         a.end_result(i,<cnt, self>) .. Counter.put(cnt))
54         :: applets <a, usr>,
55         -> applets <a, usr>;
56         ;; a user leaves
57         ;; then propagate this to the user's applet
```

```
    user_exit(usr) with
        (Counter.get(cnt) .. a.start_user_exit(<cnt, self>) ..
         a.end_user_exit(<cnt, self>) .. Counter.put(cnt))
        :: applets <a, usr>,
        -> ;

    Where
      port         : Integer;
      host         : String;
      rr           : RandomRelayServer;
      a            : DSGammaClientApp;
      usr          : Integer;
      i            : Integer;
      cnt          : Integer;
End DSGammaSystem;

;; Counter Provider Class
;; ----------------------
Class CounterProvider;
Interface
  Use Integer;
  Type counterprovider;
  Methods
     put _ : Integer;
     get _ : Integer;
Body
  Places
    counters : Integer;
        ;; a multiset of integers;
  Objects
    Counter;
  Initial
    counters  1 2 3 4 5 6 7 8 ... 1000;
  Axioms
    put(cnt) :: -> counters cnt;
    get(cnt) :: counters cnt ->;
    Where
      cnt : Integer;
End CounterProvider;

;; Defaults
;; ------------------
ADT DEFAULTS;
Interface
  Generators
    PORT, REMOTE-HOST, STOP_TRANSMIT, STOP_CONNECTION
        -> DEFAULTS;
Body
End DEFAULTS;
```

D.2 Server Side

```
;; RandomRelayServer class
;; ----------------------------
Class RandomRelayServer;
        ;;public class RandomRelayServer extends Thread
Inherit Thread;
Interface
  Use Integer, CP(Integer, Thread);
  Methods
    start_run     _ : CP(Integer, Thread);
        ;; public void run();
  Creation
    start_new-RandomRelayServer _ _ : Integer, CP(Integer, Thread);
    end_new-RandomRelayServer     _ : CP(Integer, Thread);
Body
  Use ServerSocket, GlobalRelay, Socket,
      InputRelay, OutputRelay,
      CP(FIFO(Bytes), CP(Integer, Thread)),
      CP(OutputRelay, CP(Integer, Thread)),
      CP(InputRelay, CP(Integer, Thread));
  Places
        ;; Global Variables
    port          : Integer;
    listen_socket : ServerSocket;
    globalrelay   : GlobalRelay;
        ;; Local Variables
    in,out        : CP(FIFO(Bytes), CP(Integer, Thread));
    outputrelay   : CP(OutputRelay, CP(Integer, Thread));
    inputrelay    : CP(InputRelay, CP(Integer, Thread));
    p11, .., p17,
    p21, .., p25 : CP(Integer, Thread);
  Axioms
    start_new-RandomRelayServer(port, <cnt,t>)
        :: (port = zero) = true
        =>
```

```
        -> p11 <cnt,t>, port PORT;
    start_new-RandomRelayServer(port, <cnt,t>)
        :: (port = zero) = false
         =>
        -> p11 <cnt,t>, port port;
    next with (Counter.get(cnt1) ..
               ls.start_new-ServerSocket(port,<cnt1,t>))
        :: p11 <cnt,t>, port port
        -> p12 <cnt,t>, port port,
           listen_socket ls,
           counter <cnt1, <cnt,t>>;
    next with (ls.end_new-ServerSocket(<cnt1,t>) ..
                      Counter.put(cnt1))
        :: p12 <cnt,t>,
           listen_socket ls,
           counter <cnt1, <cnt,t>>
        -> p13 <cnt,t>,
           listen_socket ls;
    next with (Counter.get(cnt1) ..
               gr.start_new-GlobalRelay(<cnt1, t>))
        :: p13 <cnt,t>
        -> p14 <cnt,t>,
           globalrelay gr,
           counter <cnt1, <cnt,t>>;
    next with (gr.end_new-GlobalRelay(<cnt1, t) ..
               Counter.put(cnt1))
        :: p14 <cnt,t>,
           globalrelay gr,
           counter <cnt1, <cnt,t>>
        -> p15 <cnt,t>,
           globalrelay gr;
    next with (Counter.get(cnt1) .. self.start_start(<cnt1, t>))
        :: p15 <cnt,t>
        -> p16 <cnt,t>,
           counter <cnt1, <cnt,t>>;
    next with (self.end_start(<cnt1, t) .. Counter.put(cnt1))
        :: p16 <cnt,t>,
           counter <cnt1, <cnt,t>>
        -> p17 <cnt,t>;
    end_new-RandomRelayServer(<cnt,t>)
        :: p17 <cnt,t> -> ;

       ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
        :: alive ok -> alive ok, p21 <cnt,t>;
    next with (Counter.get(cnt1) .. ls.start_accept(<cnt1,t>))
        :: p21 <cnt,t>, listen_socket ls,
        -> p22 <cnt,t>, listen_socket ls,
           counter <cnt1, <cnt,t>>;
    next with (ls.end_accept(in, out, <cnt1,t>) ..
               Counter.put(cnt1))
        :: p22 <cnt,t>, listen_socket ls,
           counter <cnt1, <cnt,t>>
        -> p23 <cnt,t>, listen_socket ls,
           in <in, <cnt,t>>, out <out, <cnt,t>>;
    next with (Counter.get(cnt1) ..
               or.start_new-OutputRelay(out, gr,
                         STOP_TRANSMIT, <cnt1, t))
        :: p23 <cnt,t>, out <out, <cnt,t>>,
           globalrelay gr
        -> p24 <cnt,t>,
           globalrelay gr,
           outputrelay <or, <cnt,t>>,
           counter <cnt1, <cnt,t>>;
    next with (or.end_new-OutputRelay(<cnt1,t>) ..
               Counter.put(cnt1))
        :: p24 <cnt,t>, outputrelay <or, <cnt,t>>,
           counter <cnt1, <cnt,t>>
        -> p25 <cnt,t>, outputrelay <or, <cnt,t>>;

   next with (Counter.get(cnt1) ..
               ir.start_new-InputRelay(in, gr, or,
                        STOP_TRANSMIT, STOP_CONNECTION, <cnt1,t>))
        :: p25 <cnt,t>, in <in, <cnt,t>>,
           globalrelay gr,
           outputrelay <or, <cnt,t>>,
        -> p26 <cnt,t>,
           globalrelay gr,
           outputrelay <or, <cnt,t>>,
           inputrelay <ir, <cnt,t>>,
           counter <cnt1, <cnt,t>>;
    next with (ir.end_new-InputRelay(<cnt1, t) ..
               Counter.put(cnt1))
        :: p26 <cnt,t>, inputrelay <ir, <cnt,t>>,
           counter <cnt1, <cnt,t>>
        -> p21 <cnt,t>, inputrelay <ir, <cnt,t>>;

```

```
        ;; this thread loops infinitely !
    next(<cnt,t>)
        :: stopped ok,  p21 <cnt,t> -> ;
    Where
      port      : Integer;
      in,out    : FIFO(Bytes);
      ls        : ServerSocket;
      gr        : GlobalRelay;
      ir        : InputRelay;
      or        : OutputRelay;
      t               : Thread;
      cnt, cnt1 : Integer;
End RandomRelayServer;
```

```
;; InputRelay class
;; ----------------------------
Class InputRelay;        ;;class InputRelay extends Thread;
Inherit  Thread;
Interface
  Use   Integer, CP(Integer, Thread),
        FIFO(Bytes), GlobalRelay, OutputRelay, ;
  Methods
    start_run     _ : CP(Integer, Thread);
        ;; public void run();

  Creation
    Creation
      start_new-InputRelay _ _ _ _ _ _ : FIFO(Bytes) GlobalRelay OutputRelay
                        Integer Integer CP(Integer, Thread);
      end_new-InputRelay _ : CP(Integer, Thread);
        ;; public InputRelay(Socket client_socket,
        ;;        GlobalRelay gr,
        ;;        OutputRelay or,
        ;;        int stop_transmit, int stop_connection);
Body
  Use DataInputStream, Boolean, CP(Integer, CP(Integer, Thread));
  Places
        ;; Global Variables
    globalrelay     : GlobalRelay;
    outputrelay     : OutputRelay;
    stop_transmit   : Integer;
    stop_connection : Integer;
    datainputstream : DataInputStream;
    inputstream     : FIFO(Bytes);
        ;; Local Variables
    elem : CP(Integer, CP(Integer, Thread));
    p11, .., p15,
    p21, .., p27: CP(Integer, Thread);
  Axioms
    start_new-InputRelay(in, gr, or,
                  stop_transmit, stop_connection, <cnt, t>)
        :: -> inputstream in,
              globalrelay gr,
              outputrelay or,
              stop_transmit stop_transmit,
              stop_connection stop_connection,
              p11 <cnt, t>;
        ;; create DataInputStream
     next with (Counter.get(cnt1)] ..
               datain.start_new-DataInputStream(in, <cnt1,t>))
        :: p11 <cnt, t>, inputstream in
        -> p12 <cnt, t>, inputstream in,
           datainputstream datain,
           counter <cnt1,<cnt,t>>;
    next with (datain.end_new-DataInputStream(<cnt1,t>) ..
               Counter.put(cnt1))
        :: p12 <cnt, t>,
           datainputstream datain,
           counter <cnt1,<cnt,t>>
        -> p13 <cnt, t>,
           datainputstream datain ;
        ;; starts itself
    next with (Counter.get(cnt1)] .. self.start_start(<cnt1,t))
        :: p13 <cnt, t>
        -> p14 <cnt, t>,
           counter <cnt1,<cnt,t>>;
    next with (self.end_start(<cnt1,t) .. Counter.put(cnt1))
        :: p14 <cnt, t>,
           counter <cnt1,<cnt,t>>
        -> p15 <cnt, t>;
    end_new-InputRelay(<cnt,t>)
        :: p15 <cnt, t> -> ;

        ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
```

```
        :: alive ok -> alive ok, p21 <cnt,t>;

        ;; wait for an integer from DataInputStream.
    next with (Counter.get(cnt1)] .. datain.start_readInt(<cnt1,t>))
        :: p21 <cnt,t>, datainputstream datain,
        -> p22 <cnt,t>, datainputstream datain,
           counter <cnt1,<cnt,t>>;
    next with (datain.end_readInt(elem, <cnt1,t>) .. Counter.put(cnt1))
        :: p22 <cnt,t>, datainputstream datain,
           counter <cnt1,<cnt,t>>
        -> p23 <cnt,t>, datainputstream datain,
           elem <elem,<cnt,t>>;

        ;; if the received integer is the stop_connection signal then stops
     next with (Counter.get(cnt1) .. self.start_stop(<cnt1,t>))
        :: (elem = stop_connection) = true
        => p23 <cnt,t>, elem <elem,<cnt,t>>,
           stop_connection stop_connection
        -> p24 <cnt,t>, elem <elem,<cnt,t>>,
           counter <cnt1,<cnt,t>>,
           stop_connection stop_connection;
     next with (self.end_stop(<cnt1,t>) .. Counter.put(cnt1))
        :: p24 <cnt,t>, counter <cnt1,<cnt,t>>
        -> p25 <cnt,t>;

        ;; it the received integer is the stop_transmit signal then forwards
        ;; the signal to outputrelay
     next with (Counter.get(cnt1)] ..
                or.start_setnotify_end_sending(true, <cnt1,t>))
        :: (elem = stop_transmit) = true
        => p23 <cnt,t>, elem <elem,<cnt,t>>,
           stop_transmit stop_transmit, outputrelay or
        -> p26 <cnt,t>, elem <elem,<cnt,t>>,
           counter <cnt1,<cnt,t>>
           stop_transmit stop_transmit, outputrelay or;
     next with (or.end_setnotify_end_sending(<cnt1,t>) ..
                Counter.put(cnt1))
        :: p26 <cnt,t>, counter <cnt1,<cnt,t>>,
           outputrelay or
        -> p21 <cnt,t>,
           outputrelay or;

        ;; the received integer is not a stop signal, then forward it to
        ;; globalrelay
     next with (Counter.get(cnt1)] ..
                gr.start_put(elem, <cnt1,t>))
        :: ((elem = stop_transmit) = false ) AND
           ((elem = stop_connection) = false ) AND
        => p23 <cnt,t>, elem <elem,<cnt,t>>,
           stop_transmit stop_transmit,
           stop_connection stop_connection,
           globalrelay gr
        -> p27 <cnt,t>, elem <elem,<cnt,t>>,
           stop_transmit stop_transmit,
           stop_connection stop_connection,
           globalrelay gr,
           counter <cnt1,<cnt,t>>;
     next with (gr.end_put(<cnt1,t>) ..
                Counter.put(cnt1))
        :: p27 <cnt,t>, counter <cnt1,<cnt,t>>,
           globalrelay gr
        -> p21 <cnt,t>,
           globalrelay gr;

    next
        :: stopped ok -> ;

    Where
      gr       : GlobalRelay;
      or       : OutputRelay;
      stop_transmit, stop_connection : Integer;
      datain           : DataInputStream;
      in               : FIFO(Bytes);
      elem             : Integer;
      t                : Thread;
      cnt1, cnt        : Integer;
End InputRelay;


;; OutputRelay class
;; ----------------------
Class OutputRelay;        ;; class OutputRelay extends Thread
Inherit  Thread;
Interface
  Use   Integer, CP(Integer, Thread),
        Boolean, FIFO(Bytes), GlobalRelay;
  Methods
    start_run     _ : CP(Integer, Thread);
        ;; public void run();
    start_setnotify_end_sending _ _ : Boolean CP(Integer, Thread);
    end_setnotify_end_sending     _ : CP(Integer, Thread);
        ;; public void setnotify_end_sending(boolean value);
```

```
  Creation
    start_new-OutputRelay _ _ _ _ : FIFO(Bytes) GlobalRelay Integer
                                    CP(Integer, Thread);
    end_new-OutputRelay _ : CP(Integer, Thread);
        ;; public  OutputRelay(Socket cs,
        ;;         GlobalRelay gr, int stop_transmit)
Body
  Use DataOutputStream, OutputStream, CP(Integer, CP(Integer, Thread));
  Places
            ;; Global Variables
    globalrelay      : GlobalRelay;
    stop_transmit    : Integer;
    end_sending      : Boolean;
    dataoutputstream : DataOutputStream;
    outputstream     : FIFO(Bytes);
        ;; Local Variables
    elem : CP(Integer, CP(Integer, Thread));
    p11, .., p15,
    p21, .., p25,
    p31 : CP(Integer, Thread);
  Initial
    end_sending false;
  Axioms
    start_new-OutputRelay(out, gr,
                          stop_transmit, <cnt, t>)
        :: -> outputstream  out,
              globalrelay   gr,
              stop_transmit stop_transmit,
              p11 <cnt, t>;
        ;; create DataOutputStream
    next with (Counter.get(cnt1) ..
               dataout.start_new-DataOutputStream(out, <cnt1,t>))
        :: p11 <cnt, t>, outputstream out
        -> p12 <cnt, t>, outputstream out,
           dataoutputstream dataout,
           counter <cnt1,<cnt,t>>;
    next with (dataout.end_new-DataOutputStream(<cnt1,t>) ..
               Counter.put(cnt1))
        :: p12 <cnt, t>, dataoutputstream dataout,
           counter <cnt1,<cnt,t>>
        -> p13 <cnt, t>, dataoutputstream dataout;
        ;; starts itself
    next with (Counter.get(cnt1) .. self.start_start(<cnt1,t))
        :: p13 <cnt, t>
        -> p14 <cnt, t>,
           counter <cnt1,<cnt,t>>;
    next with (self.end_start(<cnt1,t) .. Counter.put(cnt1))
        :: p14 <cnt, t>,
           counter <cnt1,<cnt,t>>
        -> p15 <cnt, t>;
    end_new-OutputRelay(<cnt,t>)
        :: p15 <cnt, t> -> ;
        ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
        :: alive ok -> alive ok, p21 <cnt,t>;
        ;; if stop_transmit then stop
    next with (Counter.get(cnt1) ..
               dataout.start_writeInt(stop_transmit, <cnt1,t>))
        :: p21 <cnt,t>, end_sending true, dataoutputstream out,
           stop_transmit stop_transmit
        -> p22 <cnt,t>, end_sending true, counter <cnt1, <cnt,t>>,
           stop_transmit stop_transmit, dataoutputstream dataout;
    next with (dataout.end_writeInt(<cnt1,t>) .. Counter.put(cnt1))
        :: p22 <cnt,t>, counter <cnt1, <cnt,t>>,
           dataoutputstream dataout
        -> p23 <cnt,t>, dataoutputstream dataout;
     next with (Counter.get(cnt1) .. self.start_stop(<cnt1,t>))
        :: p23 <cnt,t> -> p24 <cnt,t>, counter <cnt1, <cnt,t>>;
    next with (self.end_stop(<cnt1,t>) .. Counter.put(cnt1))
        :: p24  <cnt,t>, counter <cnt1, <cnt,t>> -> ;

        ;; if not stop_transmit, then take from globalrelay and go to
        ;; p21
        ;; !!! in the code it is not so safe
    next with (Counter.get(cnt1) .. gr.start_get(<cnt1,t>))
        :: p21 <cnt,t>, end_sending false,
           globalrelay gr
        -> p25 <cnt,t>, end_sending false,
           globalrelay gr, counter <cnt1, <cnt,t>>;
    next with (gr.end_get(elem, <cnt1,t>) .. Counter.put(cnt1))
        :: p25 <cnt,t>, end_sending false,
           globalrelay gr, counter <cnt1, <cnt,t>>
        -> p21 <cnt,t>, end_sending false,
           globalrelay gr, elem <elem, <cnt,t>>;

```

```
    next
        :: stopped ok -> ;
    start_setnotify_end_sending(value, <cnt,t>)
        :: end_sending old_value
        -> end_sending value, p31 <cnt,t>;
    end_setnotify_end_sending(<cnt,t>)
        :: p31 <cnt,t> -> ;
    Where
      gr               : GlobalRelay;
      stop_transmit    : Integer;
      out              : FIFO(Bytes);
      dataout          : DataOutputStream;
      value, old_value : Boolean;
      t                : Thread;
      cnt1, cnt        : Integer;

End OutputRelay;
```

```
;; GlobalRelay class
;; ---------------------------
;; A GlobalRelay is a thread which realizes a FIFO buffer.
Class GlobalRelay;
        ;; class GlobalRelay extends Thread
Inherit JavaObject;
Interface
  Use  Integer, CP(Integer, Thread);
  Type globalrelay;
  Methods
    start_put _ : Integer CP(Integer, Thread);
    end_put   _ : CP(Integer, Thread);
        ;; public synchronized void put(int input_elem);
    start_get _ : CP(Integer, Thread);
    end_get   _ : Integer CP(Integer, Thread);
        ;; public synchronized int get();
  Creation
    start_new-GlobalRelay : CP(Integer, Thread);
    end_new-GlobalRelay : CP(Integer, Thread);
        ;; public GlobalRelay();
Body
  Use Vector, CP(Integer,CP(Integer, Thread))
  Places
        ;; Global Variables
    buffer        : Vector;
        ;; Local Variables
    input_elem    : CP(Integer,CP(Integer, Thread));
    elem_to_relay : CP(Integer,CP(Integer, Thread));
    p11, p12,
    p21, .., p25,
    p31, .., p35 : CP(Integer, Thread);
  Axioms
    start_new-GlobalRelay(<cnt,t>) ::
        -> p11 <cnt,t>;
    next with b.create
        :: p11 <cnt,t>
        -> p12 <cnt,t>, buffer b;
    end_new-GlobalRelay(<cnt,t>)
        :: p12 <cnt,t> -> ;
        ;; put and get are synchronized !!!
    start_put(input_elem, <cnt,t>)
        :: -> p21 <cnt,t>, input_elem <input_elem, <cnt,t>>;
    next with self.lock(t)
        :: p21 <cnt,t>
        -> p22 <cnt,t>;
    next with (Counter.get(cnt1) ..
               b.start_addElement(input_elem, <cnt1,t>))
        :: p22 <cnt,t>, buffer b,
           input_elem <input_elem, <cnt,t>>
        -> p23 <cnt,t>, buffer b, counter <cnt1,<cnt,t>>;
    next with (b.end_addElement(<cnt1,t>) .. Counter.put(cnt1))
        :: p23 <cnt,t>, counter <cnt1,<cnt,t>> -> p24  <cnt,t>;
    next with self.unlock(t)
        :: p24 <cnt,t> -> p25 <cnt,t>;
    end_put(<cnt,t>)
        :: p25 <cnt,t> -> ;
         ;; put and get are synchronized !!!
    start_get(<cnt,t>)
        :: -> p31 <cnt,t>;
    next with self.lock(t)
        :: p31 <cnt,t> -> p32 <cnt,t>;
    next with (Counter.get(cnt1) .. b.start_elementAt(zero, <cnt1,t>))
        :: p32 <cnt,t>, buffer b, counter <cnt1,<cnt,t>>,
        -> p33 <cnt,t>, buffer b;
    next with (b.end_elementAt(elem_to_relay, <cnt1,t>))..
               Counter.put(cnt1))
        :: counter <cnt1,<cnt,t>>, p33 <cnt,t>
        -> elem_to_relay <elem_to_relay, <cnt,t>>, p34  <cnt,t>;
```

```
    next with self.unlock(t)
        :: p34 <cnt,t> -> p35 <cnt,t>;
    end_get(elem_to_relay, <cnt,t>)
        :: p35 <cnt,t>,
           elem_to_relay <elem_to_relay, <cnt,t>> -> ;
    Where
      b             : Vector;
      input_elem    : Integer;
      elem_to_relay : Integer;
      t             : Thread;
      cnt, cnt1     : Integer;
End GlobalRelay;

```

D.3 Client Side

```
;; DSGammaClientApp Class
;; ------------------------
;; Client Applet maintains:
;; (1) a graphical interface with the user (TextField, TextArea, Button)
;;     (the gui is not specified)
;; (2) a local multiset of integers and
;; (3) a socket to communicate with a server.
Class DSGammaClientApp;
        ;; public class DSGammaClientApp extends Applet;
Inherit Applet;
Interface
  Use Integer,CP(Integer,Thread);
  Methods
    start_user_action _ _ : Integer CP(Integer,Thread);
    end_user_action     _ : CP(Integer,Thread);
    start_result        _ : CP(Integer,Thread);
    end_result        _ _ : Integer CP(Integer,Thread);
    start_user_exit     _ : CP(Integer,Thread);
    end_user_exit       _ : CP(Integer,Thread);
  Creation
    start_new-DSGammaClientApp _ _ _ : Integer, String, CP(Integer,Thread);
    end_new-DSGammaClientApp       _ : CP(Integer,Thread);


Body
  Use TakeoffGlobal, TakeoffLocal,
      Socket, FIFO(Bytes),DataInputStream, DataOutputStream,
      Vector, CP(Integer,CP(Integer,Thread));
  Places
        ;; Global Variables
    socket           : Socket;
    datainputstream  : DataInputStream;
    dataoutputstream : DataOutputStream;
    inputstream      : FIFO(Bytes);
    outputstream     : FIFO(Bytes);
    MSInt            : Vector;
    takeofflocal     : TakeoffLocal;
    takeoffglobal    : TakeoffGlobal;

    port             : Integer;
    host             : Integer;
    stop_transmit    : Integer;
    stop_connection  : Integer;
        ;; Local Variables
    entering-int     : CP(Integer,CP(Integer,Thread));
    result           : CP(Integer,CP(Integer,Thread));
    p11,
    p21, .., p216,
    p31, .., p35,
    p41, .., p43,
    p51, .., p54,
    p61, .., p63     : CP(Integer, Thread);
  Initial
    stop_transmit   STOP_TRANSMIT;
    stop_connection STOP_CONNECTION;
  Axioms
    start_new-DSGammaClientApp(port,host,<cnt,t>)
        :: -> port port, host host, p11 <cnt,t>;
    end_new-DSGammaClientApp(<cnt,t>)
        :: p11 <cnt,t> -> ;
        ;; respecify Applet.start_init
    start_init(<cnt,t>)
        :: -> p21 <cnt,t>;
        ;; create a socket
    next with (Counter.get(cnt1) ..
               s.start_new-Socket(host, port, <cnt1,t>))
        :: p21 <cnt,t>,
           host host, port port
        -> p22 <cnt,t>,
```

```
            host host, port port,
            socket s, counter <cnt1, <cnt, t>>;
      next with (s.end_new-Socket(<cnt1,t>) .. Counter.put(cnt1))
          :: p22 <cnt,t>, socket s, counter <cnt1, <cnt, t>>
          -> p22 <cnt,t>, socket s;
          ;; get InputStream associated to the socket
      next with (Counter.get(cnt1) .. s.start_getInputStream(<cnt1,t>))
          :: p23 <cnt,t>,
             socket s
          -> p24 <cnt,t>,
             socket s, counter <cnt1, <cnt,t>>;
      next with (s.end_getInputStream(in, <cnt1,t>) .. Counter.put(cnt1))
          :: p24 <cnt,t>,
             socket s, counter <cnt1, <cnt,t>>
          -> p25 <cnt,t>,
             socket s, inputstream in ;
          ;; create DataInputStream
      next with (Counter.get(cnt1) ..
                 datain.start_new-DataInputStream(in, <cnt1,t>))
          :: p25 <cnt,t>, inputstream in
          -> p26 <cnt,t>, inputstream in,
             datainputstream datain,
             counter <cnt1, <cnt,t>>
      next with (datain.end_new-DataInputStream(<cnt1,t>) ..
                 Counter.put(cnt1))
          :: p26 <cnt,t>,
             datainputstream datain,
             counter <cnt1, <cnt,t>>
          -> p27 <cnt,t>,
             datainputstream datain;
           ;; get OutputStream associated to the socket
      next with (Counter.get(cnt1) .. s.start_getOutputStream(<cnt1,t>))
          :: p27 <cnt,t>,
             socket s
          -> p28 <cnt,t>,
             socket s, counter <cnt1, <cnt,t>>;
      next with (s.end_getOutputStream(out, <cnt1,t>) .. Counter.put(cnt1))
          :: p28 <cnt,t>,
             socket s, counter <cnt1, <cnt,t>>
          -> p29 <cnt,t>,
             socket s, outputstream out ;
          ;; create DataOutputStream
      next with (Counter.get(cnt1) ..
                 dataout.start_new-DataOutputStream(out, <cnt1,t>))
          :: p29 <cnt,t>, outputstream out
          -> p210 <cnt,t>, outputstream out,
             dataoutputstream dataout,
             counter <cnt1, <cnt,t>>
      next with (dataout.end_new-DataOutputStream(<cnt1,t>) ..
                 Counter.put(cnt1))
          :: p210 <cnt,t>,
             dataoutputstream dataout,
             counter <cnt1, <cnt,t>>
          -> p211 <cnt,t>,
             dataoutputstream dataout;

               ;; Create MSInt
      next with  MSInt.create((cnt1,t>)
          :: p211 <cnt,t>
          -> p212 <cnt,t>, MSInt MSInt;
          ;; Create TakeoffLocal
      next with (Counter.get(cnt1) ..
                 takeofflocal.start_new-TakeoffLocal(dataout,
                                MSInt, stop_connection, <cnt1,t>))
          :: p212 <cnt,t>, dataoutputstream dataout,
             MSInt MSInt, stop_connection stop_connection
          -> p213 <cnt,t>, dataoutputstream dataout,
             MSInt MSInt, stop_connection stop_connection,
             takeofflocal takeofflocal, counter <cnt1, <cnt,t>>;
      next with (takeofflocal.end_new-TakeoffLocal(<cnt1,t>) ..
                 Counter.put(cnt1))
          :: p213 <cnt,t>,
             takeofflocal takeofflocal,
             counter <cnt1, <cnt,t>>
          -> p214 <cnt,t>,
             takeofflocal takeofflocal;
          ;; Create TakeoffGlobal
      next with (Counter.get(cnt1) ..
                 takeoffglobal.start_new-TakeoffGlobal(datain,
                     MSInt, takeofflocal, stop_transmit, <cnt1,t>))
          :: p214 <cnt,t>, datainputstream datain,
             MSInt MSInt, takeofflocal takeofflocal,
             stop_transmit stop_transmit
          -> p215 <cnt,t>, datainputstream datain,
             MSInt MSInt,takeofflocal takeofflocal,
             stop_transmit stop_transmit,
```

```
            takeoffglobal takeoffglobal, counter <cnt1, <cnt,t>>;
    next with (takeoffglobal.end_new-TakeoffGlobal(<cnt1,t>) ..
                Counter.put(cnt1))
        :: p215 <cnt,t>,
           takeoffglobal takeoffglobal,
           counter <cnt1, <cnt,t>>
        -> p216 <cnt,t>,
           takeoffglobal takeoffglobal;
        ;; respecify Applet.end_init
    end_init(<cnt,t>)
        :: p216 <cnt,t> -> ;
        ;; respecify Applet.start_stop
    start_stop(<cnt,t>)
        :: -> p31 <cnt,t>;
        ;; close datainputstream
    next with (Counter.get(cnt1) .. datain.start_close(<cnt1, t>))
        :: p31 <cnt,t>,
           datainputstream datain
        -> p32 <cnt,t>, counter (<cnt1, <cnt,t>>)
           datainputstream datain;
    next with (datain.end_close(<cnt1, t>) .. Counter.put(cnt1))
        :: p32 <cnt,t>, counter (<cnt1, <cnt,t>>)
           datainputstream datain
        -> p33 <cnt,t>;
        ;; close dataoutputstream
    next with (Counter.get(cnt1) .. dataout.start_close(<cnt1, t>))
        :: p33 <cnt,t>,
           dataoutputstream dataout
        -> p34 <cnt,t>, counter (<cnt1, <cnt,t>>)
           dataoutputstream dataout;
    next with (dataout.end_close(<cnt1, t>) .. Counter.put(cnt1))
        :: p34 <cnt,t>, counter (<cnt1, <cnt,t>>)
           dataoutputstream dataout
        -> p35 <cnt,t>;
        ;; close socket
    next with (Counter.get(cnt1) .. s.start_close(<cnt1, t>))
        :: p33 <cnt,t>,
           socket s
        -> p34 <cnt,t>, counter <cnt1, <cnt,t>>
           socket s;
    next with (s.end_close(<cnt1, t>) .. Counter.put(cnt1))
        :: p34 <cnt,t>, counter <cnt1, <cnt,t>>
           socket s
        -> p35 <cnt,t>;

        ;; respecify Applet.end_stop
    end_stop(<cnt,t>)
        :: p35 <cnt,t> -> ;
    start_user_action(i, <cnt,t>)
        :: -> p41 <cnt,t>, entering-int <i,<cnt,t>>;
        ;; add new integer to MSInt
    next with (Counter.get(cnt1) .. MSInt.start_addElement(i, <cnt1, t>))
        :: p41 <cnt,t>, entering-int <i,<cnt,t>,
           MSInt MSInt
        -> p42 <cnt,t>, MSInt MSInt,
           counter <cnt1, <cnt,t>>;
    next with (MSInt.end_addElement(<cnt1, t>) .. Counter.put(cnt1))
        :: p42 <cnt,t>, MSInt MSInt,
           counter <cnt1, <cnt,t>>
        -> p43 <cnt,t>, MSInt MSInt;
    end_user_action(<cnt,t>)
        :: p43 <cnt,t> -> ;
    start_result(<cnt,t>)
        :: -> p51 <cnt,t>;
    next with (Counter.get(cnt1) .. MSInt.start_elementAt(zero, <cnt1, t>))
        :: p52 <cnt,t>, MSInt
        -> p53 <cnt,t>, MSInt;
    next with (MSInt.end_elementAt(i, <cnt1, t>) .. Counter.put(cnt1))
        :: p53 <cnt,t>, MSInt
        -> p54 <cnt,t>, MSInt, result <i,<cnt,t>>;
    end_result
        :: p54 <cnt,t>,result <i,<cnt,t>> ->;
    start_user_exit(<cnt,t>)
        :: -> p61 <cnt,t>;
        ;; send stop_transmit signal to server
    next with (Counter.get(cnt1) ..
               dataout.start_writeInt(stop_transmit,<cnt1,t>))
        :: p61 <cnt,t>, stop_transmit stop_transmit,
           dataoutputstream dataout
        -> p62 <cnt,t>, stop_transmit stop_transmit,
           dataoutputstream dataout, counter <cnt1,<cnt,t>>;
    next with (dataout.end_writeInt(<cnt1,t>) .. Counter.put(cnt1))
        :: p62 <cnt,t>, dataoutputstream dataout,
           counter <cnt1,<cnt,t>>
```

```
        -> p63 <cnt,t>, dataoutputstream dataout;
    end_user_exit(<cnt,t>)
        :: p63 <cnt,t>  ->;
    Where
      t            : Thread;
      s            : Socket;
      in, out      : FIFO(Bytes);
      datain       : DataInputstream;
      dataout      : DataOutputstream;
      takeofflocal : TakeoffLocal;
      takeoffglobal: TakeoffGlobal;
      MSInt        : Vector;
      cnt, cnt1    : Integer;
      i            : Integer;
      host         : String;
      port         : Integer;
End DSGammaClientApp;
```

```
;; TakeoffLocal class
;; ------------------------
Class TakeoffLocal;
        ;; class TakeoffLocal extends Thread;
Inherit Thread;
Interface
  Use   Integer, CP(Integer, Thread),
        Boolean, DataOutputStream, Vector;
  Methods
    start_run     _ : CP(Integer, Thread);
        ;; public void run();
    start_set_end_reception _ _ : Boolean CP(Integer, Thread);
    end_set_end_reception     _ : CP(Integer, Thread);
       ;; public void set_end_reception()
  Creation
    start_new-TakeoffLocal _ _ _ _ : DataOutputStream Vector Integer
                                     CP(Integer, Thread);
    end_new-TakeoffLocal _ : CP(Integer, Thread);
        ;; public TakeoffLocal(DataOutputStream out,
        ;; Vector MSInt, TextArea textarea, int stop_connection)
Body
  Use Random, CP(Integer, CP(Integer, Thread));
  Places
          ;; Global Variables
    end_reception    : Boolean;
    dataoutputstream : DataOutputSteam;
    MSInt            : Vector;
    stop_connection  : Integer;
        ;; Local Variables
    random           : CP(Integer, CP(Integer, Thread));
    p11, .., p14,
    p21, .., p214,
    p31              : CP(Integer, Thread);
  Initial
    end_reception  false;
  Axioms
    start_new-TakeoffLocal(dataout, MSInt, stop_connection, <cnt,t>)
        :: -> dataoutputstream dataout, MSInt MSInt,
              stop_connection stop_connection,
              p11  <cnt,t>;
    next with (Counter.get(cnt1) .. self.start_start(<cnt1,t))
        :: p11 <cnt, t>
        -> p12 <cnt, t>,
           counter <cnt1,<cnt,t>>;
    next with (self.end_start(<cnt1,t) .. Counter.put(cnt1))
        :: p13 <cnt, t>,
           counter <cnt1,<cnt,t>>
        -> p14 <cnt, t>;
    end_new-TakeoffLocal(<cnt,t>)
        :: p14 <cnt, t> -> ;
           ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
        :: alive ok -> alive ok, p21 <cnt,t>, p29 <cnt,t>;
        ;; the stop signal has been received,
        ;; then check if MSInt is empty
    next with (Counter.get(cnt1) .. MSInt.start_isEmpty(<cnt1,t>))
        :: p21 <cnt,t>, counter <cnt1,<cnt,t>>,
           end_reception true, MSInt MSInt
        -> p22 <cnt,t>, MSInt MSInt;
    next with (MSInt.end_isEmpty(true, <cnt1,t>) .. Counter.put(cnt1))
        :: p22 <cnt,t>, MSInt MSInt, counter <cnt1,<cnt,t>>
        -> p23 <cnt,t>, MSInt MSInt;
        ;; loop until MSInt is empty
    next with (MSInt.end_isEmpty(false, <cnt1,t>) .. Counter.put(cnt1))
        :: p22 <cnt,t>, MSInt MSInt, counter <cnt1,<cnt,t>>
        -> p21 <cnt,t>, MSInt MSInt;
```

```
68
69          ;; stop signal has been received and MSInt is empty
70          ;; then send the stop signal to server ...
71      next with (Counter.get(cnt1) ..
72                 dataout.start_writeInt(stop_connection,<cnt1,t>))
73          :: p23 <cnt,t>, stop_connection stop_connection,
74             dataoutputstream dataout
75          -> p24 <cnt,t>, stop_connection stop_connection,
76             dataoutputstream dataout, counter <cnt1,<cnt,t>>;
77      next with (dataout.end_writeInt(<cnt1,t>) .. Counter.put(cnt1))
78          :: p24 <cnt,t>, dataoutputstream dataout,
79             counter <cnt1,<cnt,t>>
80          -> p25 <cnt,t>, dataoutputstream dataout;
81       next with (Counter.get(cnt1) ..
82                 dataout.start_flush(<cnt1,t>))
83          :: p25 <cnt,t>,
84             dataoutputstream dataout
85          -> p26 <cnt,t>,
86             dataoutputstream dataout, counter <cnt1,<cnt,t>>;
87      next with (dataout.end_flush(<cnt1,t>) .. Counter.put(cnt1))
88          :: p26 <cnt,t>, dataoutputstream dataout,
89             counter <cnt1,<cnt,t>>
90          -> p27 <cnt,t>, dataoutputstream dataout;
91
92          ;; ... and stop itself
93      next with (Counter.get(cnt1) .. self.start_stop(<cnt1,t>))
94          :: p27 <cnt,t> -> p28 <cnt,t>, counter <cnt1, <cnt,t>>;
95      next with (self.end_stop(<cnt1,t>) .. Counter.put(cnt1))
96          :: p28  <cnt,t>, counter <cnt1, <cnt,t>> -> ;
97
98          ;; MSInt has to be emptied.
99      next with (Random.get(random) .. Counter.get(cnt1) ..
100                MSInt.start_elementAt(random, <cnt1,t>))
101         :: p29 <cnt,t>, MSInt MSInt
102         -> p210 <cnt,t>, counter <cnt1,<cnt,t>>,
103            MSInt MSInt, random <random,<cnt,t>>;
104     next with (MSInt.end_elementAt(i, <cnt1,t>) .. Counter.put(cnt1))
105         :: p210 <cnt,t>, MSInt MSInt, counter <cnt1,<cnt,t>>
106         -> p211 <cnt,t>, MSInt MSInt,
107            elem-to-send <i,<cnt,t>>;
108     next with (Counter.get(cnt1) ..
109                MSInt.start_removeElementAt(random, <cnt1,t>))
110         :: p211 <cnt,t>, MSInt MSInt, random <random,<cnt,t>>
111         -> p212 <cnt,t>, MSInt MSInt, counter <cnt1,<cnt,t>>;
112     next with (MSInt.end_removeElementAt(<cnt1,t>) .. Counter.put(cnt1))
113         :: p212 <cnt,t>, MSInt MSInt, counter <cnt1,<cnt,t>>
114         -> p213 <cnt,t>, MSInt MSInt;
115         ;; send integer to server and loop until MSInt is empty
116     next with (Counter.get(cnt1) .. dataout.start_writeInt(i, <cnt1,t>))
117         :: p213 <cnt,t>, elem-to-send <i,<cnt,t>>,
118            dataoutputstream dataout
119         -> p214 <cnt,t>, counter <cnt1,<cnt,t>>,
120            dataoutputstream dataout;
121     next with (dataout.end_writeInt(<cnt1,t>) .. Counter.put(cnt1))
122         :: p214 <cnt,t>, counter <cnt1,<cnt,t>>,
123            dataoutputstream dataout
124         -> p29 <cnt,t>, dataoutputstream dataout;
125         ;; return from run
126     next
127         :: stopped ok -> ;
128
129     start_set_end_reception(value, <cnt,t>)
130         :: end_reception old_value
131         -> end_reception value, p31 <cnt,t>;
132     end_set_end_reception(<cnt,t>)
133         :: p31 <cnt,t> -> ;

134     Where
135       value, old_value  : Boolean;
136       stop_connection   : Integer;
137       dataout           : DataOutputStream;
138       MSInt             : Vector;
139       t                 : Thread;
140       cnt, cnt1         : Integer;
141       random            : Integer;
142 End TakeoffLocal;
143
144


1   ;; TakeoffGlobal class
2   ;;----------------------
3   Class TakeoffGlobal
4           ;; class TakeoffGlobal extends Thread;
5   Inherit Thread;
6   Interface
7     Use   Integer, DataInputStream, Vector,
8           CP(Integer, Thread), TakeoffLocal;
9     Methods
```

```
    start_run     _ : CP(Integer, Thread);
        ;; public void run();
  Creation
    start_new-TakeoffGlobal  _ _ _ _: DataInputStream Vector
                         TakeoffLocal Integer CP(Integer, Thread);
    end_new-TakeoffGlobal _ : CP(Integer, Thread);
        ;; public TakeoffGlobal(DataInputStream in,
        ;; Vector MSInt, TextArea textarea,
        ;; TakeoffLocal tl, int stop_transmit)
Body
  Use Boolean, Random, Clock, CP(Integer, CP(Integer, Thread));
  Transitions
    tik;
  Places
        ;; Global Variables
    datainputstream   : DataInputStream;
    MSInt             : Vector;
    takeofflocal      : TakeoffLocal;
    stop_transmit     : Integer;
    timeout           : Integer;
        ;; Local Variables
    first, second,
    result : CP(Integer, CP(Integer, Thread));

    p11, .., p14,
    p21, .., p215     : CP(Integer, Thread);
  Axioms
    start_new-TakeoffGlobal(datain, MSInt, tl,
                stop_transmit, <cnt,t>)
        :: -> datainputstream datain
              MSInt           MSInt
              takeofflocal    tl
              stop_transmit   stop_transmit,
              p11 <cnt,t>;
    next with (Counter.get(cnt1) .. self.start_start(<cnt1,t))
        :: p11 <cnt, t>
        -> p12 <cnt, t>,
           counter <cnt1,<cnt,t>>;
    next with (self.end_start(<cnt1,t) .. Counter.put(cnt1))
        :: p13 <cnt, t>,
           counter <cnt1,<cnt,t>>
        -> p14 <cnt, t>;
    end_new-TakeoffGlobal(<cnt,t>)
        :: p14 <cnt, t> -> ;

        ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
        :: alive ok -> alive ok, p21 <cnt,t>;

        ;; get the first integer
    next with (Counter.get(cnt1) .. datain.start_readInt(<cnt1,t>))
        :: p21 <cnt,t>, datainputstream datain
        -> p22 <cnt,t>, datainputstream datain,
           counter <cnt1,<cnt,t>>;
        ;; first integer is not a stop signal
    next with ((datain.end_readInt(first, <cnt1,t>)) ..
               Counter.put(cnt1)) //  random.get(millis) // clock.get(hour))
        :: (first = stop_transmit) = false
        => p22 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           counter <cnt1,<cnt,t>>
        -> p23 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           first <first, <cnt,t>>, timeout (hour + millis);
        ;; first integer is a stop signal
    next with (datain.end_readInt(first, <cnt1,t>)) .. Counter.put(cnt1))
        :: (first = stop_transmit) = true
        => p22 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           counter <cnt1,<cnt,t>>
        -> p210 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           first <first, <cnt,t>>;
        ;; get the second integer
    next with (Counter.get(cnt1) .. datain.start_readInt(<cnt1,t>))
        :: p23 <cnt,t>, datainputstream datain, timeout d
        -> p24 <cnt,t>, datainputstream datain,
           counter <cnt1,<cnt,t>>;
        ;; second integer is not a stop signal
    next with (datain.end_readInt(second, <cnt1,t>)) .. Counter.put(cnt1))
        :: (second = stop_transmit) = false
        => p24 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           MSInt MSInt, counter <cnt1,<cnt,t>>
        -> p25 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
```

```
            second <second, <cnt,t>>;
        ;; add first+second to MSInt
    next with (Counter.get(cnt1) ..
                MSInt.start_addElement(first + second, <cnt1,t>))
         :: p25 <cnt,t>, MSInt MSInt,
           first <first, <cnt,t>>,
           second <second, <cnt,t>>
        -> p26  <cnt,t>,  MSInt MSInt, counter <cnt1, <cnt,t>>;
    next with (MSInt.end_addElement(<cnt1,t)  .. Counter.put(cnt1))
        :: p26  <cnt,t>,  counter <cnt1, <cnt,t>>,
           MSInt MSInt
        -> p27  <cnt,t>, MSInt MSInt;
        ;; second integer is a stop signal
    next with (datain.end_readInt(second, <cnt1,t>)) .. Counter.put(cnt1))
        :: (second = stop_transmit) = true
        => p24 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit,
           counter <cnt1,<cnt,t>>
        -> p28 <cnt,t>, datainputstream datain,
           stop_transmit stop_transmit;
        ;; add only first to MSInt
    next with (Counter.get(cnt1) ..
                MSInt.start_addElement(first, <cnt1,t>))
         :: p28 <cnt,t>, MSInt MSInt,
           first <first, <cnt,t>>
        -> p29  <cnt,t>, MSInt MSInt, counter <cnt1, <cnt,t>>;
    next with (MSInt.end_addElement(<cnt1,t)  .. Counter.put(cnt1))
        :: p29  <cnt,t>,  counter <cnt1, <cnt,t>>,
           MSInt MSInt
        -> p210  <cnt,t>, MSInt MSInt;
        ;; prevent deadlock when no sufficient integers.
        ;; tik adds only first to MSInt and loop for new integers.
    tik with clock.get(hour)
        :: (d > hour)
        => p23 <cnt,t>, timeout d -> p214 <cnt,t>;
        ;; add only first to MSInt
    next with (Counter.get(cnt1) ..
                MSInt.start_addElement(first, <cnt1,t>))
         :: p214 <cnt,t>, MSInt MSInt,
           first <first, <cnt,t>>
        -> p215  <cnt,t>, MSInt MSInt, counter <cnt1, <cnt,t>>;
    next with (MSInt.end_addElement(<cnt1,t)  .. Counter.put(cnt1))
        :: p215  <cnt,t>,  counter <cnt1, <cnt,t>>,
           MSInt MSInt
        -> p21  <cnt,t>, MSInt MSInt;


        ;; a stop signal has been received
    next with (Counter.get(cnt1) ..
                tl.start_set_end_reception(true, <cnt1,t>))
        :: p210 <cnt,t>, takeofflocal tl
        -> p211 <cnt,t>, takeofflocal tl,
           counter <cnt1, <cnt,t>>;
    next with (tl.start_set_end_reception(<cnt1,t>) ..
                Counter.put(cnt1))
        :: p211  <cnt,t>, takeofflocal tl,
           counter <cnt1, <cnt,t>>
        -> p212 <cnt,t>, takeofflocal tl, ;

    next with (Counter.get(cnt1) .. self.start_stop(<cnt1,t>))
        :: p212 <cnt,t> -> p213 <cnt,t>, counter <cnt1, <cnt,t>>;
    next with (self.end_stop(<cnt1,t>) .. Counter.put(cnt1))
        :: p213  <cnt,t>, counter <cnt1, <cnt,t>> -> ;

    next
        :: stopped ok -> ;

    Where
        datain          : DataInputStream;
        MSInt           : Vector;
        tl              : TakeoffLocal;
        t               : Thread;
        cnt1, cnt       : Integer
        stop_transmit   : Integer;
        first, second   : Integer;
        hour, millis, d : Integer;
End TakeoffGlobal;
```

D.4 Communication Layer

```
;; Socket class
;; ----------------
Class Socket;               ;;public final class Socket extends Object
Inherit JavaObject;
Interface
```

```
  Use String, Integer, FIFO(Bytes),
      CP(Integer, Thread);
  Methods
    start_close _   : CP(Integer, Thread);
    end_close   _   : CP(Integer, Thread);
        ;; public synchronized void close() throws IOException;
    start_getInputStream  _ : CP(Integer, Thread);
    end_getInputStream  _ _ : FIFO(Bytes) CP(Integer, Thread);
            ;; public InputStream getInputStream() throws IOException;
    start_getOutputStream _ : CP(Integer, Thread);
    end_getOutputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
        ;; public OutputStream getOutputStream() throws IOException;
  Creation
    start_new-Socket _ _ _ : String Integer CP(Integer, Thread);
    end_new-Socket       _ : CP(Integer, Thread);
        ;; public Socket(String host, int port) throws
        ;;        UnknownHostException, IOException;
Body
  Use System;
  Places
        ;; Global Variables
    remotehost   : String;
    port         : Integer;
    inputstream  : FIFO(Bytes);
    outputstream : FIFO(Bytes);

    p11, .., p14,
    p21, p22,
    p31, p41, : CP(Integer, Thread);
  Initial
    opened ok;
  Axioms
    start_new-Socket(host, port, <cnt,t>)
        :: -> remotehost host, port port,
              p11  <cnt,t>;
        ;; Host.connect delays to serversocket.connect(s,host)
    next with in.create
        :: p11  <cnt,t>
        -> p12  <cnt,t>, inputstream in;
    next with out.create
        :: p12  <cnt,t>
        -> p13  <cnt,t>, outputstream out;
    next with System.connect(in,out,host,port)
        :: p13  <cnt,t>, inputstream in, outputstream out,
             remotehost host, port port
        -> p14  <cnt,t>, inputstream in, outputstream out,
              remotehost host, port port;
    end_new-Socket(<cnt,t>)
        :: p14  <cnt,t> -> ;
        ;; close is a sychronized method
    start_close(<cnt,t>) :: -> p21 <cnt,t>;
    next with self.lock(t)
        :: p21 <cnt,t>,  opened ok -> p22 <cnt,t>, closed ok ;
    next :: p21 <cnt,t>, closed ok -> p23 <cnt,t>, closed ok ;
    next with self.unlock(t)
        :: p22 <cnt,t> -> p23 <cnt,t>;
    end_close(<cnt,t>) :: p23 <cnt,t> -> ;

    start_getInputStream(<cnt,t>)
        :: -> p31 <cnt,t>;
    end_getInputStream(in, <cnt,t>)
         :: p31 <cnt,t>, inputstream in
         -> inputstream in;

    start_getOutputStream(<cnt,t>)
        :: -> p41 <cnt,t>;
    end_getOutputStream(out, <cnt,t>)
        :: p41 <cnt,t>, outputstream out
        -> outputstream out;
    Where
      host : String;
      port : Integer;
      in   : FIFO(Bytes);
      out  : FIFO(Bytes);
      t    : Thread;
      cnt  : Integer;
End Socket;

Class System
Interface
  Use Socket, String, Integer;
  Methods
    connect       _ _ _ _ : FIFO(Bytes), FIFO(Bytes), String, Integer;
    getconnection _ _ _ _ : FIFO(Bytes), FIFO(Bytes), String, Integer;
  Object System;
Body
  Places
    connections : CP(FIFO(Bytes),FIFO(Bytes),String,Integer);
```

```
  Axioms
        ;; a Socket registers to the System
    connect(in,out,host,port)
        :: -> connections <in,out,host,port>;
    getconnection(in,out,host,port)
        :: connections <in,out,host,port>
        -> ;
    Where
        in, out      : FIFO(Bytes);
        host         : String;
        port         : Integer;
end System;
```

```
;; ServerSocket class
;; --------------------
Class ServerSocket;
        ;; public final class ServerSocket extends Object;
Inherit JavaObject;
Interface
  Use FIFO(Bytes), Integer, CP(Integer, Thread);
  Methods
    start_accept     _ : CP(Integer, Thread);
    end_accept   _ _ _ : FIFO(Bytes), FIFO(Bytes), CP(Integer, Thread);
        ;; public Socket accept() throws IOException;
    start_close      _ : CP(Integer, Thread);
    end_close        _ : CP(Integer, Thread);
        ;; public void close() throws IOException;
  Creation
    start_new-ServerSocket _ : Integer CP(Integer, Thread);
    end_new-ServerSocket   _ : CP(Integer, Thread);;
        ;; public ServerSocket(int port) throws IOException;
Body
  Use System;
  Places
        ;; Global Variables
    port         : Integer;
    host         : String;
    inputstream  : FIFO(Bytes);
    outputstream : FIFO(Bytes);

    p11, p21, p22,
    p31, p32,    : CP(Integer, Thread);
    opened, closed : Black_Tocken;
  Initial
    opened ok;
    host host;
  Axioms
    start_new-ServerSocket(port, <cnt,t>)
        :: -> p11 <cnt,t>, port port;
    end_new-ServerSocket(<cnt,t>)
        :: p11 <cnt,t> -> ;
        ;; accept blocks until a socket makes a connect
    start_accept(<cnt,t>)
        :: -> p21 <cnt,t>;
    next with System.getConnection(out,in,host,port)
        :: p21 <cnt,t>, host host, port port
        -> p22 <cnt,t>, host host, port port,
           inputstream in, outputstream out;
    end_accept(in, out, <cnt,t>)
        :: p22 <cnt,t>, opened ok,
           inputstream in, outputstream out
        -> inputstream in, outputstream out, opened ok;
    start_close(<cnt,t>)
        :: -> p31 <cnt,t>;
    next
        :: p31 <cnt,t>, opened ok
        -> p32 <cnt,t>, closed ok ;
    next
        :: p31 <cnt,t>, closed ok
        -> p32 <cnt,t>, closed ok ;
    end_close(<cnt,t>)
        :: p32 <cnt,t> -> ;
    Where
      port   : Integer;
      in,out : FIFO(Bytes);
      t      : Thread;
      cnt    : Integer;
End ServerSocket;
```

```
;; DataInputStream class
;; ----------------------
;; A DataInputStream is a fifo buffer of bytes, with the ability
;; to retrieve different kinds of data inside the buffer
;; (integers, chars, etc)
Class DataInputStream;
        ;; public class DataInputStream extends FilterInputStream
Inherit JavaObject; ;; (FilterInpuStream < InputStream)
```

```
Interface
  Use Integer, FIFO(Bytes), CP(Integer, Thread);
  Methods
    start_readInt   _ : CP(Integer, Thread);
    end_readInt   _ _ : Integer CP(Integer, Thread);
        ;; public final int readInt() throws IOException;
  Creation
    start_new-DataInputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
    end_new-DataInputStream     _ : CP(Integer, Thread);
Body
  Use CP(Integer,CP(Integer, Thread));
  Places
        ;; Global Variables
    inputstream : FIFO(Bytes);
        ;; Local Variables
    incoming-int      : CP(Integer,CP(Integer, Thread));
    p11,
    p21: CP(Integer, Thread);

  Axioms
    start_new-DataInputStream(in, <cnt,t>)
        :: -> inputstream in,
              p11 <cnt,t>;
    end_new-DataInputStream(<cnt,t>)
        :: p11 <cnt,t> -> ;
    start_readInt(<cnt,t>)
        :: -> p21 <cnt,t> ;
    next with (in.read(b,<cnt1,t>)
        :: p21 <cnt,t>, inputstream in
        -> p22 <cnt,t>, inputstream in,
           incoming-int (BytesToInt(b), <cnt,t>) ;
    end_readInt(i,<cnt,t>)
        :: p22 <cnt,t>, incoming-int(i,<cnt,t>)-> ;
    Where
      i, cnt, cnt1 : Integer;
      in           : FIFO(Bytes);
      t            : Thread;
      b            : Bytes;
End DataInputStream;
```

```
;; DataOutputStream class
;; ----------------------
;; A DataOutputStream is a fifo buffer of bytes, with the ability
;; insert ih the buffer different kinds of data (integers, chars, etc)
Class DataOutputStream;
        ;; public class DataOutputStream extends FilterOutputStream
Inherit JavaObject; ;; (FilterOutputStream < OutputStream)
Interface
  Use Integer, CP(Integer, Thread), FIFO(Bytes);
  Methods
    start_writeInt _ _ : Integer CP(Integer, Thread);
    end_writeInt     _ : CP(Integer, Thread);
        ;; public final void writeInt(int v) throws IOException;
  Creation
    start_new-DataOutputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
    end_new-DataOutputStream     _ : CP(Integer, Thread);
Body
  Places
        ;; Global Variables
    outputstream : FIFO(Bytes);
        ;; Local Variables
    outgoing-int : CP(Integer,CP(Integer, Thread));
    p11, p21, p22: CP(Integer, Thread);

  Axioms
    start_new-DataOutputStream(out, <cnt,t>)
        :: -> outputstream out,
              p11 <cnt,t>;
    end_new-DataOutputStream(<cnt,t>)
        :: p11 <cnt,t> -> ;
    start_writeInt(i, <cnt,t>)
        :: -> p21 <cnt,t>, outgoing-int <i,<cnt,t>> ;
    next with (out.start_write(intToBytes(i), <cnt1,self>))
        :: p21 <cnt,t>, outgoing-int <i,<cnt,t>>,
           outputstream out
        -> p22 <cnt,t>, outputstream out;
    end_writeInt(<cnt,t>)
        :: p22 <cnt,t> -> ;
    Where
      i, cnt, cnt1  : Integer;
      out           : FIFO(Bytes);
      t             : Thread;
End DataOutputStream;
```

Appendix E

Fourth Refinement R4: Java Program

E.1 Client Side

```
1   package Gamma;
2
3   import java.applet.*;
4   import java.awt.*;
5   import java.io.*;
6   import java.net.*;
7   import java.util.*;
8   import MyUtils.*;
9
10  /*
11    Gamma addition of integers
12    */
13
14
15  /*
16     Distributed Gamma-like addition of integers
17     */
18
19
20
21
22  /** Distributed Gamma-like addition of integers
23    DSGammaClientApp Applets allows a user to enter several integers.
24    This local multiset (Vector MSInt) of integers will be part of a global distributed
25    multiset of integers, that obtained by the union of all the other local multisets of
26    integers provided by all the other users using the same applet.
27    DSGammaClientApp is responsible for: <br>
28    a) establishing connection with a server,  <br>
29    b) entering the DSGamma system (the set of all these applets running),  <br>
30    c) managing integers entered by user and those received by the server,  <br>
31    d) properly quitting the DSGamma system (empty the local
32    MSInt of integers, stop the threads and closing socket)
33    */
34  public class DSGammaClientApp extends Applet{
35    public final static int PORT=6090;
36    public final static int STOP_TRANSMIT=-2;
37    public final static int STOP_CONNECTION=-1;
38
39    Socket s;
40    DataInputStream in;
41    DataOutputStream out;
42    TextField textfield;
43    TextArea textarea;
44    Button stop_button;
45
46    TakeoffGlobal takeoffglobal;
47    TakeoffLocal takeofflocal;
48
49    Vector MSInt;
50
51
52    /** Create a socket to communicate with a server on port 6090
53      of the host that the applet's code is on. Create streams to use
54      with the socket. Then create a TextField for user input, a TextArea
55      for output, and a Button for exitting the DSGamma system.
56      MSInt stores the integers entered by the local users, and those received by the
57      server. Finally, create two threads for interaction with
58      the server.
59      */
60    public void init(){
```

```
61
62      try{
63        s=new Socket(this.getCodeBase().getHost(),PORT);
64        in=new DataInputStream(s.getInputStream());
65        out=new DataOutputStream(s.getOutputStream());
66
67        textfield = new TextField();
68        textarea = new TextArea();
69        stop_button = new Button("Exit DSGamma System");
70        textarea.setEditable(false);
71        MSInt = new Vector();
72
73
74        setLayout(new BorderLayout());
75        add("North",textfield);
76        add("Center",textarea);
77        add("South",stop_button);
78
79        //Initializes takeofflocal and takeoffglobal threads
80        takeofflocal = new TakeoffLocal(out, MSInt, textarea, STOP_CONNECTION);
81        takeoffglobal = new TakeoffGlobal(in, MSInt, textarea,takeofflocal,
82                                          STOP_TRANSMIT);
83
84        showStatus("Connected to "
85                        + s.getInetAddress().getHostName()
86                        + ":" + s.getPort()+"\n");
87      }
88      catch (IOException e) {
89        showStatus("Exception while creating socket: "+e);
90        try{if (s!=null) {s.close();}}
91        catch (IOException e2) {
92          showStatus("Exception while closing socket: "+e2);
93        }
94        stop();
95      }
96
97    }
98
99
100
101   /** Close the socket and the input, output streams
102    */
103   public void stop(){
104
105     try{
106       if (in!=null)  {in.close(); in = null;}
107       if (out!=null) {out.close(); out = null;}
108       if (s!=null) {s.close(); s = null;}
109     }
110     catch (IOException e2) {
111      showStatus("Exception while closing socket: "+e2);
112     }
113
114     showStatus("ByeBye\n");
115   }
116
117
118   /** Capture events on the TextField or Button Components of the interface
119    */
120   public boolean action(Event event, Object what){
121
122     //User types a line in textfield, convert it to a Vector of Integer
123     if (event.target == textfield){
124       // Convert String into Vector of Integers (MSInt)
125       Convert.StringtoInteger(textfield.getText(),MSInt);
126       textfield.setText("");
127       showStatus("User entered some integers\n");
128
129       //Notifies takeofflocal, because MSInt is no more empty
130       synchronized (takeofflocal) {takeofflocal.notify();}
131
132       return true;
133     }
134
135
136     //User wants to exit the DSGamma system
137     if (event.target == stop_button){
138       //Notifies the server that the user wants to stop
139       try{
140         out.writeInt(STOP_TRANSMIT);
141         textarea.appendText("Exit DSGamma requested\n");
142       }
143       catch(IOException e) {
144         System.out.println("Client can't write on socket: "+e);
145       }
146       return true;
147     }
148     return false;
```

```
149   }
150
151
152
153 } // end of DSGammaClientApp
154
155
156 // -------------------------------------
157 /**  Randomly removes one integer from local multiset (Vector MSInt) of integers.
158  */
159 class TakeoffLocal extends Thread{
160   DataOutputStream out;
161   Vector MSInt;
162   TextArea textarea;
163   int stop_connection;
164
165   boolean end_reception = false;
166
167   public TakeoffLocal(DataOutputStream out, Vector MSInt, TextArea textarea,
168                       int stop_connection){
169     this.out = out;
170     this.MSInt = MSInt;
171     this.textarea = textarea;
172     this.stop_connection = stop_connection;
173     this.start();
174   }
175
176
177   /**Body of TakeoffLocal.
178     Wait for MSInt to be not empty, the send the content of MSInt to server.
179     If no more integers will be received from server, MSInt is emptied a last time,
180     before stopping.
181     */
182   public synchronized void run(){
183     for (;;) {
184
185       //Check if  TakeoffGlobal has finished received integers (end_reception = true).
186       //In this case no more integers will be added in MSInt, and TakeoffLocal empties
187       //MSInt a last time and stops.
188       if (end_reception) {
189         textarea.appendText("Emptying local multiset for the last time\n");
190
191         //free MSInt
192         doReactions();
193
194         //send stop_connection signal to server
195         try{
196           out.writeInt (stop_connection);
197           out.flush();
198         }
199         catch(IOException e) {
200         System.out.println("Client can't write on socket: "+e);
201         }
202         //TakeoffLocal can stop
203         finally{break;} //to stop()
204       }
205
206
207       //TakeoffGlobal is still receiving integers from server.
208       //TakeoffLocal waits for user or for TakeoffGlobal to enter integer numbers
209       //i.e. wait for MSInt to be not empty.
210       try{wait();}
211       catch(InterruptedException e) {
212         textarea.appendText("Exception while waiting: "+e);
213       }
214       finally{
215         //Free MSInt
216         doReactions();
217         }
218     }
219
220     textarea.appendText("Exit DSGamma system done\n");
221     stop();
222   }
223
224
225
226
227   /**Randomly chooses one integer in Vector MSInt, and sends it to the Server,
228     till MSInt is not empty.
229     */
230   public  void  doReactions(){
231     int i;
232
233     while (!MSInt.isEmpty()){
234
235       //Show the user the new state of Vector
236       textarea.appendText("\n");
```

```
237       for (i=0; i<MSInt.size();i++){
238         textarea.appendText(MSInt.elementAt(i).toString()+" ");
239       }
240       textarea.appendText("\n");
241
242       //Choose an index
243       i = (int) (Math.random() * MSInt.size()) % MSInt.size() ;
244
245       //Send the chosen integer to the server
246       try{
247         out.writeInt(((Integer) MSInt.elementAt(i)).intValue());
248         //Remove the integer from Vector MSInt
249         MSInt.removeElementAt(i);
250       }
251       catch (IOException e) {
252         System.out.println("Client can't write on socket: "+e);
253         stop();
254       }
255     }
256     //Ensure the sending of integers to server
257     try{
258       out.flush();
259       textarea.appendText("\nEmpty\n");
260     }
261     catch(IOException e) {
262         System.out.println("Client can't write on socket: "+e);
263         stop();}
264
265   }
266
267
268
269   //TakeoffGlobal set end_reception to true when it has finished receiving
270   //integers from server.
271   /**Set variable end_reception to value. <br>
272     It is used as an asyncronous flag to notify TakeoffLocal that nothing
273     more will be received from server.
274    */
275   public  void set_end_reception(boolean value){
276     end_reception = value;
277   }
278
279
280 }// end of TakeoffLocal
281
282
283
284
285
286 // ---------------------------------------
287 /** Wait for output (2 integers) from the server on the specified stream, adds
288   them and puts the result in its local Vector of integers.
289   */
290 class TakeoffGlobal extends Thread{
291   DataInputStream in;
292   TextArea textarea;
293   Vector MSInt;
294   TakeoffLocal takeofflocal;
295   int stop_transmit;
296
297   int result;
298
299
300   public TakeoffGlobal(DataInputStream in, Vector MSInt, TextArea textarea,
301                        TakeoffLocal takeofflocal, int stop_transmit){
302     this.in = in;
303     this.textarea = textarea;
304     this.MSInt = MSInt;
305     this.takeofflocal = takeofflocal;
306     this.stop_transmit = stop_transmit;
307     this.start();
308   }
309
310
311
312   //Body of TakeoffGlobal.
313   public synchronized void run(){
314     doReactions();
315     takeofflocal.set_end_reception(true);
316     synchronized (takeofflocal) {takeofflocal.notify();}
317     textarea.appendText("Exit DSGamma: stop receiving integers\n");
318     stop();
319   }
320
321
322   /** Read two integers from server and add their sum to MSInt
323     If the second integer does not come sufficiently soon, the first one
324     is added to MSInt. This is useful when the number of integers in the global multiset
```

```
    is less than the number of current users.
    If the second integer is the STOP_TRANSMIT signal, then TakeoffGlobal adds
    the first one to MSInt and then stops.
    If the first integer is the STOP_TRANSMIT signal, then doReactions
    returns immediately.
    */
  public void doReactions(){
    int tmp = 0;
    int i;

    //Wait for two integer, add their sum to MSInt
    //until the stop signal arrives
    while(tmp != stop_transmit) {
      try{
        result = in.readInt();
        if (result == stop_transmit) {
          //the first integer is the stop signal, it is time to return
          break; //to return
        }
        if (in.available() > 0) {
          //A second integer is available, check for the stop signal and
          //add them if necessary
          tmp = in.readInt();
          if (tmp != stop_transmit) {
            result +=tmp;
          }
        }
        // A second integer is not available immediately
        else {
          // Sleep a random amount of millis before checking a second time
          // for available data from server.
          i = (int) (Math.random() *2000);
          try{sleep(i);}
          catch(InterruptedException e) {
            textarea.appendText("Exception while sleeping: "+e);return;
          }
          finally{
            if (in.available() > 0) {
              //A second integer is available after sleeping, check of the stop signal
              //and add the first and the second if necessary
              tmp = in.readInt();
              if (tmp!= stop_transmit) {
                result +=tmp;
              }
            }
             // if no second integer is availabe, the first one is reinjected
             // in Vector, instead of the sum.
          }
        }


        // Add either the first integer received from server, or the sum of
        // two integers received from server.
        MSInt.addElement(new Integer(result));

        // Notifies TakeoffLocal that a new integer has arrived in MSInt, this is
        // necessary if MSInt was empty.
        synchronized (takeofflocal) {takeofflocal.notify();}

       }
      catch(IOException e) {
        textarea.appendText("Connection closed by server");
        break; //to return
      }
    }
    return; //to run()
  }

}// end of TakeoffGlobal
```

E.2 Server Side

```
package RelayServer;


import java.io.*;
import java.net.*;
import java.util.*;


/**Create several socket connections with several clients.
  Act as a random relay between all the clients.
  Data sent along the socket must be of type int.
  */
```

```
13  public class RandomRelayServer extends Thread {
14    /** default value for the server port is 6090 */
15    public final static int DEFAULT_PORT=6090;
16    public final static int STOP_TRANSMIT=-2;
17    public final static int STOP_CONNECTION=-1;
18
19
20    int port;
21
22    ServerSocket listen_socket;
23    GlobalRelay globalrelay;
24
25
26    /** Create a ServerSocket to listen for connections on a given port.
27      Initialize the thread GlobalRelay which will realize the random relay
28      between all clients. <br>
29      Starts itself.
30      */
31    public RandomRelayServer(int port){
32      if (port == 0) port=DEFAULT_PORT;
33      this.port = port;
34      try { listen_socket = new ServerSocket(port); }
35      catch(IOException e) {
36        System.out.println("Exception creating server socket"+e);
37      }
38      System.out.println("RandomRelayServer: listening on port "+port);
39      globalrelay = new GlobalRelay();
40      this.start();
41    }
42
43
44
45
46    /**Body of the server thread. Loop forever, listening for and
47      accepting connections from clients. For each connection, initialize two threads
48      InputRelay, and OutputRelay, handling respectively incoming and outgoing
49      communication from/to clients.
50      */
51    public void run(){
52       try{
53         while(true){
54           Socket client_socket = listen_socket.accept();
55           System.out.println("A client wants a connection\n");
56           OutputRelay outputrelay = new OutputRelay(client_socket,globalrelay,
57                                                     STOP_TRANSMIT);
58           InputRelay inputrelay = new InputRelay(client_socket, globalrelay,
59                                                  outputrelay, STOP_TRANSMIT,
60                                            STOP_CONNECTION);
61
62         }
63       }
64       catch(IOException e) {
65         System.out.println("Exception while listening for connections "+e);
66       }
67     }
68
69
70    /**Start the server up, listening on an optionally specified port. <br>
71      Default port is 6090.
72     */
73    public static void main(String[] args){
74      int port =0;
75      if (args.length == 1){
76        try {port = Integer.parseInt(args[0]);}
77        catch (NumberFormatException e) {port = 0;}
78      }
79      new RandomRelayServer(port);
80    }
81
82  } //end of RandomRelayServer class
83
84  //--------------------
85
86  /** Handle all incoming communication from a dedicated client using Socket.
87    Relay this data to GlobalRelay. Notifies OutputRelay if the stop_transmit signal
88    is received from client. Stops itself it the stop_connection signal is received
89    from client.
90    */
91  class InputRelay  extends Thread{
92    Socket client;
93    GlobalRelay globalrelay;
94    DataInputStream in;
95    OutputRelay outputrelay;
96    int stop_transmit;
97    int stop_connection;
98
99
100   /** Initialize DataInputStream and starts itself
```

```
101    */
102   public InputRelay(Socket client_socket, GlobalRelay globalrelay,
103                     OutputRelay outputrelay, int stop_transmit,
104                     int stop_connection){
105     this.client = client_socket;
106     this.globalrelay = globalrelay;
107     this.outputrelay = outputrelay;
108     this.stop_transmit = stop_transmit;
109     this.stop_connection = stop_connection;
110
111     try{in = new DataInputStream(client_socket.getInputStream());}
112     catch(IOException e){
113       try {client.close();}
114       catch(IOException e2){
115         System.out.println("Exception while getting socket streams:"+e2);};
116       System.out.println("Exception while getting socket streams:"+e);
117       return; //to RandomRelayServer
118     }
119     this.start();
120   }
121
122
123   /**Body of InputRelay.<br>
124     Read data from DataInputStream of client.
125     Put data to GlobalRelay
126     */
127   public void run(){
128     int elem;
129
130     try{
131       for(;;){
132         // Read a data from DataInputStream of client
133         try{
134           elem = in.readInt();
135           if (elem == stop_connection) {
136             //client has no more to send
137             System.out.println("InputRelay "+this.getName()+
138                                " Exit DSGamma done.\n");
139               break; // to finally
140           }
141           if (elem == stop_transmit) {
142             //client wants no more on its input (our output)
143             System.out.println("InputRelay "+this.getName()+
144                                " Exit DSGamma: stop sending received from client\n");
145             outputrelay.setnotify_end_sending(true);
146           }
147           else {
148             // Relay data to GlobalRelay thread.
149             System.out.println("InputRelay before put"+this.getName()+" "+elem);
150             globalrelay.put(elem);
151             System.out.println("InputRelay after put"+this.getName()+" "+elem);
152           }
153         }
154         catch(IOException e) {
155           System.out.println("Input Relay "+this.getName()+
156                              " not possible: "+e+"\n");
157
158           break; // to finally
159         }
160       }
161     }
162     finally {
163       try {client.close();client = null; }
164       catch(IOException e2) {
165         System.out.println("Exception closing client: "+e2+"\n");
166       }
167       finally{stop();}
168     }
169   }
170
171 } // end of InputRelay
172
173 //-------------------
174
175 /** Handle all outgoing communication to a dedicated client.
176   Relay a data from GlobalRelay thread to the dedicated client.
177   */
178 class OutputRelay  extends Thread{
179   Socket client;
180   GlobalRelay globalrelay;
181   DataOutputStream out;
182   int stop_transmit;
183
184   boolean end_sending= false;
185
186   /** Initialize DataOutputStream and starts itself
187    */
188   public OutputRelay(Socket client_socket, GlobalRelay globalrelay,
```

```
                    int stop_transmit){

    this.client = client_socket;
    this.globalrelay = globalrelay;
    this.stop_transmit = stop_transmit;

    try{out = new DataOutputStream(client_socket.getOutputStream());}
    catch(IOException e){
      try {client.close();}
      catch(IOException e2){
        System.out.println("Exception while getting socket streams:"+e2);
      }
      System.out.println("Exception while getting socket streams:"+e);
      return; //to RandomRelayServer
    }
    this.start();
  }


  /** Body of OutputRelay.<br>
    Get data from GlobalRelay <br>
    Relay data to DataOutputStream of client.
    */
  public void run(){
    int elem;

    try{
      for(;;){
        if (end_sending){
            System.out.println("OutputRelay "+this.getName()+
                               " Exit DSGamma: stop sending received\n");
            //notifies the client that the stop_transmit signal has been received
            try{
              out.writeInt(stop_transmit);
              out.flush();
            }
            catch(IOException e) {
              System.out.println("OutputRelay "+this.getName()+" not possible\n"+e);
            }
            finally{break;} //to stop
        }


        // Wait for  data from GlobalRelay
        System.out.println("OutputRelay before get"+this.getName());
        elem = globalrelay.get();
        System.out.println("OutputRelay after get"+this.getName()+" "+elem);

        // Relay data to DataOutputStream of client.
        try{
          out.writeInt(elem);
          out.flush();
          System.out.println("OutputRelay "+this.getName()+" "+elem);
        }
        catch(IOException e) {
          System.out.println("OutputRelay "+this.getName()+" not possible\n"+e);

          //Save value
          globalrelay.put(elem);
          break;  // to finally
        }
      }
    }
    finally{
      System.out.println("Output relay "+this.getName()+
                         " Exit DSGamma: stop sending done\n");
      stop();
    }

  }


  /** Set end_sending to value <br>
    It is used as an asynchronous flag to notify OutputRelay to stop sending
    data to a client.
   */
  public  void setnotify_end_sending(boolean value){
    end_sending = value;
  }



} // end of OutputRelay


//-------------------

/** Act as a FIFO buffer.
```

```
277   */
278 class GlobalRelay extends Thread{
279   Vector buffer;
280
281   /** Initializes the FIFO buffer to empty and Starts itself */
282   public GlobalRelay(){
283     buffer = new Vector();
284     this.start();
285   }
286
287
288   /**Incoming data  is stored at the end of the FIFO buffer
289    */
290   synchronized public void put(int input_elem){
291
292     //prevent two consecutive put, without intermediary get
293     System.out.println("GlobalRelay rcvd "+input_elem);
294     buffer.addElement(new Integer(input_elem));
295     notify();
296   }
297
298
299   /**First data stored in buffer is returned and removed from the FIFO buffer.
300     This method blocks  until a data to relay is available.
301     */
302   synchronized public int get(){
303     int elem_to_relay;
304
305     while (buffer.isEmpty()) {
306       try {wait();}
307       catch (InterruptedException e) {
308         System.out.println("Error while get GlobalRelay is waiting "+e);
309       }
310     }
311     elem_to_relay = ((Integer) buffer.elementAt(0)).intValue();
312     System.out.println("GlobalRelay has relayed "+elem_to_relay);
313     buffer.removeElementAt(0);
314     return elem_to_relay;
315   }
316
317 } //end of GlobalRelay
318
319
```

E.3 Utils

```
1   package MyUtils;
2
3   import java.awt.*;
4   import java.io.*;
5   import java.net.*;
6   import java.util.*;
7
8
9   /** Set of functions useful for some conversions.
10   */
11  public final class Convert{
12
13    /** Converts a String into a Vector of Integers.
14      Ex: String (12 34) becomes Vector of two Integers 12 and 34.
15      The current implementation does not consider ill-formed strings.
16      */
17    public static void StringtoInteger(String s, Vector v){
18
19        int beginIndex =0;
20        int endIndex;
21
22        // extraction of substring from a string
23     BI: while(beginIndex < s.length()){
24
25          //search for a new integer
26          while(Character.isSpace(s.charAt(beginIndex))) {
27            beginIndex++;
28            if (beginIndex == s.length()) break BI;
29          }
30          endIndex = beginIndex+1;
31          if (endIndex < s.length()) {
32            while(!Character.isSpace(s.charAt(endIndex))) {
33              endIndex++;
34              if (endIndex == s.length()) break;
35            }
36          }
37
38          // add the new integer to the Vector
39          v.addElement(Integer.valueOf(s.substring(beginIndex,endIndex)));
40          beginIndex = endIndex;
41
```

```
42      } // end of BI
43     }// end of StringtoInteger
44
45
46
47
48
49  }
```

Appendix F

Java Basics Classes: CO-OPN/2 Specification

```
1  ;; The Object class (Java Object Class)
2  ;; ----------------------------------
3  Class JavaObject;
4  Interface
5    Use Integer, Thread, CP(Integer, Thread);
6    Methods
7      start_notify  _ : CP(Integer, Thread);
8      end_notify    _ : CP(Integer, Thread);
9          ;; public final void notify();
10     start_wait    _ : CP(Integer, Thread);
11     end_wait      _ : CP(Integer, Thread);
12         ;; public final void wait();
13     lock _ , unlock _ : Thread;
14     next;
15 Body
16   Use Black_Tocken; ;; Used by most classes, even if not by JavaObject.
17   Use CounterProvider,
18       CP(CP(Thread,Integer),CP(Integer,Thread)),
19       CP(Integer, CP(Integer, Thread));
20   Initial
21     locker <null,zero>; ;; no locker object (null) and no locks (zero)
22   Places
23         ;; Global Variables
24     wait-set    : CP(CP(Thread,Integer),CP(Integer,Thread));
25         ;; set of threads waiting on the current object
26     resumed-set : CP(CP(Thread,Integer),CP(Integer,Thread));
27         ;; set of threads resumed by a notify
28     locker      : CP(Thread,Integer);
29         ;; the Thread who is currently possessing
30         ;; the object's lock, together with
31         ;; the number of current Integer locks it possesses on the object.
32     counter : CP(Integer, CP(Integer, Thread));
33     p11, p12, p13, p21, p22 : CP(Integer,Thread);
34   Axioms
35         ;; it is necessary to have a lock on the object before calling its
36         ;; wait method.
37     start_wait(<cnt, t>)
38         :: locker <t,i>
39         -> p11 <cnt,t>, locker <t,i>;
40         ;; release all the locks on the object
41     next
42         :: p11 <cnt,t>, locker <t,i>
43         -> p12 <cnt,t>, wait-set <<t,i>,<cnt,t>>, locker <null,0>;
44         ;; reacquires all the locks on the object
45     next with self.lock(t) ::
46         :: p12 <cnt,t>, resumed-set <<t,i+1>,<cnt,t>>
47         -> p12 <cnt,t>, resumed-set <<t,i>,<cnt,t>>
48     end_wait(<cnt,t>)
49         :: p12 <cnt,t>, resumed-set <<t,0>,<cnt,t>>
50         -> ;
51     start_notify(<cnt1, t1>)
52         :: locker <t,i>
53         -> p21 <cnt1,t1>, locker <t,i>;
54     next
55         :: p21 <cnt1,t1>, wait-set <<t,i>,<cnt,t>>
56         -> p22 <cnt1,t1>, resumed-set <<t,i>,<cnt,t>>;
57     end_notify(<cnt1,t1>)
58         :: p22 <cnt1,t1> -> ;
59     lock(t) :: locker <t, i> -> locker <t, i+1>;
60         ;; the current locker increments the lock
61     lock(t) :: locker <null,zero> -> locker <t, 1>;
```

```
        ;; no current locker, acquisition of the lock
    unlock(t) :: locker <t,i+1> -> locker <t,i>;
        ;; the current locker decrements the lock
    unlock(t) :: locker <t,1>   -> locker <null, zero>;
        ;; the current locker releases the lock
    Where
      t, t1     : Thread;
      cnt1, cnt : Integer;
      i         : Integer;
End JavaObject;
```

```
;; Thread class
;; ------------------------
Class Thread;  ;; public class Threads extends Object implements Runnable
Inherit JavaObject;
Interface
  Use Boolean, Integer, CP(Integer, Thread);
  Type
  Methods
    ;; Public Instance Methods
    start_isAlive _ : CP(Integer, Thread);
    end_isAlive _ _ : Boolean CP(Integer, Thread);
        ;; public final boolean isAlive();
    start_run     _ : CP(Integer, Thread);
        ;; public void run();
    start_start   _ : CP(Integer, Thread);
    end_start     _ : CP(Integer, Thread);
        ;; public synchronized void start()
        ;; throws IllegalThreadStateException;
    start_stop    _ : CP(Integer, Thread);
    end_stop      _ : CP(Integer, Thread);
        ;; public final void stop();
Body
  Use Clock;
  Places
    alive, stopped : Black_Tocken;

    p11,p21,p22,p31  : CP(Integer, Thread);
  Initial
    alive ok;
  Axioms
    start_isAlive(<cnt,t>)
        :: -> p11 <cnt,t>;
    end_isAlive(true,<cnt,t>)
        :: alive ok, p11 <cnt,t> -> alive ok;
    end_isAlive(false,<cnt,t>)
        :: stopped ok, p11 <cnt,t> -> stopped ok;
        ;; a thread returns if
        ;; 1. run ends, 2. exception interrupts, 3. there is a stop()
    start_run(<cnt,t>)
        :: alive ok -> alive ok, p21 <cnt,t>;
    next with Counter.put(cnt)
        :: p22 <cnt,t> -> ;
    next with Counter.put(cnt)
         :: stopped ok,  p22 <cnt,t> -> ;

        ;; start immediately causes run
    start_start(<cnt,t>)
        :: alive ok -> alive ok, p31 <cnt,t>;
    start_start(<cnt,t>)
        :: stopped ok -> stopped ok, p34 <cnt,t>;
        ;; start is synchronized
    next with self.lock(t)
        :: p31 <cnt,t> -> p32 <cnt,t>;
        ;; start causes run
    next with (Counter.get(cnt1) .. self.run(<cnt1,self>))
        :: p32 <cnt,t> -> p33 <cnt,t>;
        ;; it is a new execution flow, there is no need to wait for the
        ;; return, unless it is not a new execution flow
    next with self.unlock(t)
        :: p33 <cnt,t> -> p34 <cnt,t>;
    end_start(<cnt,t>)
        :: p34 <cnt,t> -> ;
    start_stop(<cnt,t>)
        :: alive ok -> stopped ok, p41 <cnt,t>;
    stop_stop(<cnt,t>)
        :: stopped ok -> stopped ok, p41 <cnt,t>;
    end_stop(<cnt,t>)
        :: p41 <cnt,t> -> ;

    Where
      t         : Thread;
      cnt, cnt1 : Integer;
End Thread;
```

```
;; Applet class
;; ----------------
```

```
Class Applet; ;; public class Applet extends Panel
Inherit JavaObject; ;; (Panel < Container < Component < Object)
Interface
  Use  String, Integer, CP(Integer,Thread);
  Type Applet;
  Methods
    ;; the methods are called by a web browser or an appletviewer
    start_init  _ : CP(Integer,Thread);
    end_init    _ : CP(Integer,Thread);
        ;; public void init();  //empty
    start_start _ : CP(Integer,Thread);
    end_start   _ : CP(Integer,Thread);
        ;; public void start(); //empty
    start_stop  _ : CP(Integer,Thread);
    end_stop    _ : CP(Integer,Thread);
        ;; public void stop();  //empty
  Creation
    start_new-Applet _  _ : String CP(Integer,Thread);
    end_new-Applet      _ : CP(Integer,Thread);
        ;; called by Web Browser or Appletviewer

Body
  Places
    p11,p21,p31,p41 : CP(Integer,Thread);
    active, stopped : Black_Tocken;
    remotehost      : String;
  Initial
    active ok;
  Axioms
    start_new-Applet(host, <cnt,t>)
        :: -> p11 <cnt,t>,
              remotehost host;
    end_new-Applet(<cnt,t>)
        :: p11 <cnt,t> -> ;
        ;; an applet is inited only once
    start_init(<cnt,t>) :: active ok  -> active ok,  p21 <cnt,t>;
    start_init(<cnt,t>) :: stopped ok -> stopped ok, p21 <cnt,t>;
    end_init(<cnt,t>)   :: p21 <cnt,t> -> ;

        ;; an applet is started many times
    start_start(<cnt,t>) :: active ok  -> active ok,  p31 <cnt,t>;
    start_start(<cnt,t>) :: stopped ok -> stopped ok, p31 <cnt,t>;
    end_start(<cnt,t>)   :: p31 <cnt,t> -> ;

    start_stop(<cnt,t>) :: active ok  -> stopped ok, p41 <cnt,t>;
    stop_stop(<cnt,t>)  :: stopped ok -> stopped ok, p41 <cnt,t>;
    end_stop(<cnt,t>)   :: p41 <cnt,t> -> ;

    Where
      t   : Thread;
      cnt : Integer;
End Applet;
```

```
;; Socket class
;; -----------------
Class Socket;               ;;public final class Socket extends Object
Inherit JavaObject;
Interface
  Use String, Integer, FIFO(Bytes),
      CP(Integer, Thread);
  Methods
    start_close _   : CP(Integer, Thread);
    end_close   _   : CP(Integer, Thread);
        ;; public synchronized void close() throws IOException;
    start_getInputStream  _ : CP(Integer, Thread);
    end_getInputStream  _ _ : FIFO(Bytes) CP(Integer, Thread);
          ;; public InputStream getInputStream() throws IOException;
    start_getOutputStream _ : CP(Integer, Thread);
    end_getOutputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
        ;; public OutputStream getOutputStream() throws IOException;
  Creation
    start_new-Socket _ _ _ : String Integer CP(Integer, Thread);
    end_new-Socket       _ : CP(Integer, Thread);
        ;; public Socket(String host, int port) throws
        ;;        UnknownHostException, IOException;
Body
  Use System;
  Places
        ;; Global Variables
    remotehost   : String;
    port         : Integer;
    inputstream  : FIFO(Bytes);
    outputstream : FIFO(Bytes);

    p11, .., p14,
    p21, p22,
    p31, p41, : CP(Integer, Thread);
  Initial
```

```
    opened ok;
  Axioms
    start_new-Socket(host, port, <cnt,t>)
        :: -> remotehost host, port port,
              p11  <cnt,t>;
        ;; Host.connect delays to serversocket.connect(s,host)
    next with in.create
        :: p11  <cnt,t>
        -> p12  <cnt,t>, inputstream in;
    next with out.create
        :: p12  <cnt,t>
        -> p13  <cnt,t>, outputstream out;
    next with System.connect(in,out,host,port)
        :: p13  <cnt,t>, inputstream in, outputstream out,
             remotehost host, port port
        -> p14  <cnt,t>, inputstream in, outputstream out,
             remotehost host, port port;
    end_new-Socket(<cnt,t>)
        :: p14  <cnt,t> -> ;
        ;; close is a sychronized method
    start_close(<cnt,t>) :: -> p21 <cnt,t>;
    next with self.lock(t)
        :: p21 <cnt,t>,  opened ok -> p22 <cnt,t>, closed ok ;
    next :: p21 <cnt,t>,  closed ok -> p23 <cnt,t>, closed ok ;
    next with self.unlock(t)
        :: p22 <cnt,t> -> p23 <cnt,t>;
    end_close(<cnt,t>) :: p23 <cnt,t> -> ;

    start_getInputStream(<cnt,t>)
        :: -> p31 <cnt,t>;
    end_getInputStream(in, <cnt,t>)
         :: p31 <cnt,t>, inputstream in
         -> inputstream in;

    start_getOutputStream(<cnt,t>)
        :: -> p41 <cnt,t>;
    end_getOutputStream(out, <cnt,t>)
        :: p41 <cnt,t>, outputstream out
        -> outputstream out;

    Where
      host : String;
      port : Integer;
      in   : FIFO(Bytes);
      out  : FIFO(Bytes);
      t    : Thread;
      cnt  : Integer;
End Socket;

Class System
Interface
  Use Socket, String, Integer;
  Methods
    connect       _ _ _ _ : FIFO(Bytes), FIFO(Bytes), String, Integer;
    getconnection _ _ _ _ : FIFO(Bytes), FIFO(Bytes), String, Integer;
  Object System;
Body
  Places
    connections : CP(FIFO(Bytes),FIFO(Bytes),String,Integer);
  Axioms
        ;; a Socket registers to the System
    connect(in,out,host,port)
        :: -> connections <in,out,host,port>;
    getconnection(in,out,host,port)
        :: connections <in,out,host,port>
        -> ;
    Where
        in, out      : FIFO(Bytes);
        host         : String;
        port         : Integer;
end System;


;; ServerSocket class
;; --------------------
Class ServerSocket;
        ;; public final class ServerSocket extends Object;
Inherit JavaObject;
Interface
  Use FIFO(Bytes), Integer, CP(Integer, Thread);
  Methods
    start_accept     _ : CP(Integer, Thread);
    end_accept   _ _ _ : FIFO(Bytes), FIFO(Bytes), CP(Integer, Thread);
        ;; public Socket accept() throws IOException;
    start_close      _ : CP(Integer, Thread);
    end_close        _ : CP(Integer, Thread);
        ;; public void close() throws IOException;
  Creation
    start_new-ServerSocket _ : Integer CP(Integer, Thread);
    end_new-ServerSocket   _ : CP(Integer, Thread);;
```

```
        ;; public ServerSocket(int port) throws IOException;
Body
  Use System;
  Places
        ;; Global Variables
    port         : Integer;
    host         : String;
    inputstream  : FIFO(Bytes);
    outputstream : FIFO(Bytes);

    p11, p21, p22,
    p31, p32,    : CP(Integer, Thread);
    opened, closed : Black_Tocken;
  Initial
    opened ok;
    host host;
  Axioms
    start_new-ServerSocket(port, <cnt,t>)
        :: -> p11 <cnt,t>, port port;
    end_new-ServerSocket(<cnt,t>)
        :: p11 <cnt,t> -> ;
        ;; accept blocks until a socket makes a connect
    start_accept(<cnt,t>)
        :: -> p21 <cnt,t>;
    next with System.getConnection(out,in,host,port)
        :: p21 <cnt,t>, host host, port port
        -> p22 <cnt,t>, host host, port port,
           inputstream in, outputstream out;
    end_accept(in, out, <cnt,t>)
        :: p22 <cnt,t>, opened ok,
           inputstream in, outputstream out
        -> inputstream in, outputstream out, opened ok;
    start_close(<cnt,t>)
        :: -> p31 <cnt,t>;
    next
        :: p31 <cnt,t>, opened ok
        -> p32 <cnt,t>, closed ok ;
    next
        :: p31 <cnt,t>, closed ok
        -> p32 <cnt,t>, closed ok ;
    end_close(<cnt,t>)
        :: p32 <cnt,t> -> ;
    Where
      port   : Integer;
      in,out : FIFO(Bytes);
      t      : Thread;
      cnt    : Integer;
End ServerSocket;


;; DataInputStream class
;; ---------------------
;; A DataInputStream is a fifo buffer of bytes, with the ability
;; to retrieve different kinds of data inside the buffer
;; (integers, chars, etc)
Class DataInputStream;
        ;; public class DataInputStream extends FilterInputStream
Inherit JavaObject; ;; (FilterInpuStream < InputStream)
Interface
  Use Integer, FIFO(Bytes), CP(Integer, Thread);
  Methods
    start_readInt   _ : CP(Integer, Thread);
    end_readInt   _ _ : Integer CP(Integer, Thread);
        ;; public final int readInt() throws IOException;
  Creation
    start_new-DataInputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
    end_new-DataInputStream     _ : CP(Integer, Thread);
Body
  Use CP(Integer,CP(Integer, Thread));
  Places
        ;; Global Variables
    inputstream : FIFO(Bytes);
        ;; Local Variables
    incoming-int      : CP(Integer,CP(Integer, Thread));
    p11,
    p21: CP(Integer, Thread);

  Axioms
    start_new-DataInputStream(in, <cnt,t>)
        :: -> inputstream in,
              p11 <cnt,t>;
    end_new-DataInputStream(<cnt,t>)
        :: p11 <cnt,t> -> ;
    start_readInt(<cnt,t>)
        :: -> p21 <cnt,t> ;
    next with (in.read(b,<cnt1,t>)
        :: p21 <cnt,t>, inputstream in
```

```
         -> p22 <cnt,t>, inputstream in,
            incoming-int (BytesToInt(b), <cnt,t>) ;
    end_readInt(i,<cnt,t>)
         :: p22 <cnt,t>, incoming-int(i,<cnt,t>)-> ;
    Where
      i, cnt, cnt1 : Integer;
      in           : FIFO(Bytes);
      t            : Thread;
      b            : Bytes;
End DataInputStream;
```

```
;; DataOutputStream class
;; ----------------------
;; A DataOutputStream is a fifo buffer of bytes, with the ability
;; insert ih the buffer different kinds of data (integers, chars, etc)
Class DataOutputStream;
        ;; public class DataOutputStream extends FilterOutputStream
Inherit JavaObject; ;; (FilterOutputStream < OutputStream)
Interface
  Use Integer, CP(Integer, Thread), FIFO(Bytes);
  Methods
    start_writeInt _ _ : Integer CP(Integer, Thread);
    end_writeInt     _ : CP(Integer, Thread);
        ;; public final void writeInt(int v) throws IOException;
  Creation
    start_new-DataOutputStream _ _ : FIFO(Bytes) CP(Integer, Thread);
    end_new-DataOutputStream     _ : CP(Integer, Thread);
Body
  Places
        ;; Global Variables
    outputstream : FIFO(Bytes);
        ;; Local Variables
    outgoing-int : CP(Integer,CP(Integer, Thread));
    p11, p21, p22: CP(Integer, Thread);

  Axioms
    start_new-DataOutputStream(out, <cnt,t>)
        :: -> outputstream out,
              p11 <cnt,t>;
    end_new-DataOutputStream(<cnt,t>)
        :: p11 <cnt,t> -> ;
    start_writeInt(i, <cnt,t>)
        :: -> p21 <cnt,t>, outgoing-int <i,<cnt,t>> ;
    next with (out.start_write(intToBytes(i), <cnt1,self>))
        :: p21 <cnt,t>, outgoing-int <i,<cnt,t>>,
           outputstream out
        -> p22 <cnt,t>, outputstream out;
    end_writeInt(<cnt,t>)
        :: p22 <cnt,t> -> ;
    Where
      i, cnt, cnt1  : Integer;
      out           : FIFO(Bytes);
      t             : Thread;
End DataOutputStream;
```

Bibliography

[1] Jacob I. Aizikowitz. Designing distributed services using refinement mappings¿. Technical Report CORNELLCS//TR89-1040, Cornell University, Computer Science Department, September 1989.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[3] J.-P. Banatre and D. Metayer. Gamma and the chemical reaction model. In IC Press, editor, *Proceedings of the Coordination'95 workshop*, 1995.

[4] O. Biberstein, D. Buchs, E. Canver, P. Dauchy, M-C. Gaudel, N. Guelfi, F. von Henke, C. Khoury, B. Marre, D. Schwier, and G. Vidal-Naquet. Comparison of object-oriented formal methods. Technical Report to appear as Technical Report of the Esprit Long Term Research Project 20072 "Design For Validation", University of Newcastle Upon Tyne, Department of Computer Science, 1997.

[5] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, July 1997. To appear.

[6] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Coopn/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Canterbury ,UK, March 1997. Chapman and Hall, London. to appear.

[7] Didier Buchs and Nicolas Guelfi. CO-OPN: 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.

[8] David Flanagan et al. *Java in a Nutshell*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, deluxe edition, 1997.

[9] J. Jacob. The varieties of refinements. In J. M. Morris and R. C. Shaw, editors, *4th Refinement Workshop*, Workshops in Computing, pages 441–455. Springer-Verlag, 1991.

[10] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, 1997.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[12] W. Reisig. Correctness proofs of distributed algorithms. *Lecture Notes in Computer Science*, 938:164–177, 1995.

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

[14] N. Uchihira and S. Honiden. Verification and synthesis of concurrent programs using petri nets and temporal logic. *The transaction of the institute of electronics, information and communication engineers*, E73(12):2001–2009, December 1990.

[15] P. J. Whysall and J. A. McDermid. Object-oriented specification and refinement. In J. M. Morris and R. C. Shaw, editors, *Proc. 4th Refinement Workshop*, Workshops in Computing, pages 151–184. Springer-Verlag, 1991.

[16] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

[17] P. Yurkowski and C. M. Laucht. Combining petri nets and temporal logic to model and analyse distributed systems. In *Proc. of the Fifteenth Manitoba Conf. on Numerical Mathematics and Computing*, pages 211–227, 1986. NewsletterInfo: 30.