Formal Development of Java Based Web Parallel Applications*

Giovanna Di Marzo Serugendo[1,2]
[1] *CUI, University of Geneva Switzerland*
dimarzo@di.epfl.ch

Nicolas Guelfi[2]
[2] *LGL-DI, Swiss Federal Institute of Technology, Switzerland*
guelfi@di.epfl.ch

Abstract

The Java object-oriented programming language has been the subject of an important involvement from programmers and the industry. Especially for applications related to the Web. The problem of such a rapid penetration of Java programs into commercial products is that software engineers do not have any methodology and have to develop complex parallel applications. Here, we present a formal development methodology based on the stepwise refinement of CO-OPN/2 formal specifications, using a real Web parallel application. Starting from a centralized view, we present the following refinement steps: data distribution, behavior distribution, communication layer, and Java program. During the whole refinement process, we study the evolution and the verification of one specific property.

Keywords: *Software Engineering, Petri Nets, Algebraic Specifications, Refinement, Concurrent and Distributed Systems, Java, Web.*

1. Introduction

Java is a recent programming language [1] that 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 apply a top-down engineering methodology for the development of a Java based Web parallel application (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 [3] which is developed by our team and which provides many features adapted to the applications addressed in this paper. More precisely, CO-OPN/2 integrates, in an object-oriented approach, Petri nets for the description of concurrent behaviors, and algebraic specifications [6] for the specification of the structured data evolving in the Petri nets.

The applied formal development methodology consists of: (1) providing a set of informal application's requirements including validation objectives expressed by a set of desired properties; (2) building an initial CO-OPN/2 specification, **I**, of the JPA, based on the informal requirements, and abstract enough to be as independent as possible of the implementation constraints; this specification provides an abstract view of the application, where the problem is neither distributed nor decentralized. This first specification must validate the desired properties; (3) performing four refinement steps, (**R1** to **R4**), concerning both the formal specification and the properties: (**R1**) provides a decentralized view of the application, where the data is distributed but not yet the behavior; (**R2**) provides the application with a client/server architecture where both the data and the behavior are distributed; (**R3**) introduces the communication layer, i.e. the TCP/IP based sockets, the applets, the server, and all necessary threads handling the sockets. In addition, it introduces features of the Java programming language. This step leads to a view *close to the code*, where the problem has been spread on the Web but not yet implemented with Java; (**R4**) produces the Java program directly

*Copyrigth 1998 IEEE. Published in the Proceedings of the Hawaii International Conference on System Sciences, January 6-9, 1997, Kona, Hawaii. 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 Concurrency".

from refinement **R3**. Evidence (by formal or informal proof) must be provided for the properties preservation between two consecutive refinement levels.

This formal development methodology for Java based Web parallel applications is presented trough a concrete example (running at the following address http://lglsun.epfl.ch/Team/GDM/DSGamma.html). The application is based on the Gamma paradigm [2] (a programming paradigm integrating chemical reaction concepts). A Gamma-like addition is realized on distributed multisets of integers. A Java applet maintains locally a multiset of integers, and a user may enter new integers into 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 Java threads that collect pairs of integers, add them and put them into the multiset maintained by one of the applets.

The full description of the formal development of this application, together with properties verification can be found in [4].

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 details the methodology by applying it to the chosen distributed application.

2. The CO-OPN/2 specification formalism

CO-OPN/2 [3] is an hybrid specification formalism based on algebraic specifications [6] and Petri nets that are combined in a way that is similar to algebraic nets [5]. Algebraic specifications are used to describe the data structures and the functional aspects of a system, while Petri nets allow to model the system's concurrent features. To compensate for algebraic Petri nets' lack of structuring capabilities, CO-OPN/2 provides a structuring mechanism based on a synchronous interaction between algebraic nets, as well as notions peculiar to object-orientation such as the notions of class, inheritance, and subtyping.

A system is considered as being a collection of independent objects (algebraic nets) that interact and collaborate together in order to accomplish the various tasks of the system.

Object and class. An object is considered as an independent entity, composed of an internal state, and that provides some services to the exterior. The only way to interact with an object is to ask for its services; the internal state is thus protected against uncontrolled accesses. 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 multiset 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 describe 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. 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. Objects can be dynamically created. The usual dot notation has been adopted.

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 providing the service. The synchronization policy is expressed by means of a synchronization expression (declared after the **`with`** keyword), 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 from two different partners, followed by a service request to a third object.

For each transition (parameterized or not), one or more behavioral axioms are defined, in order to describe: (1) an optional condition imposed on the algebraic values involved in the axiom, (2) an optional synchronization expression, (3) pre- and post-conditions corresponding respectively to what is consumed and what is produced in the different places composing the net, once the transition is executed.

Object identity. Within the CO-OPN/2 framework, each class instance has an identity, that 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 that is established between the classes types. Since object identifiers are algebraic values, they can be stored in places of algebraic nets. Moreover it is possible to define data structures that are built upon object identifiers, e.g. a stack or a queue of object identifiers.

Constructors. Class instances can be dynamically created. Particular creation methods that create and initialize the objects can be defined; these methods may be used only once for a given object. A pre-defined creation method is provided. Usually classes are used to dynamically create new instances, but it is also possible to declare static instances.

Inheritance and subtyping. We believe that inheritance and subtyping are two different notions that are used for two different purposes. Inheritance is considered as being a syntactic mechanism that frees the specifier from the necessity of developing classes from scratch and is mainly utilized 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, and is a semantical mechanism: in any context, any class instance of a type may be substituted to a class instance of its super-type, and the behavior of the whole system remains unchanged. The respective hierarchies generated by these two relationships do not necessarily coincide.

Semantics. The formal semantics of CO-OPN/2 is given in terms of concurrent transition systems expressing all the possible evolutions of objects' states. State changes are associated to a multiset of events that are simultaneously executable. The firing of an object's method causes (internal) transitions to be fired (spontaneously). The transitions are fired as long as their pre-condition is fulfilled. An object's method can be fired only if no transition is firable. The complete semantics of CO-OPN/2 can be found in [3].

3. Formal development: from CO-OPN/2 to Java

This section presents the proposed formal development methodology applied to the concrete application described in section 1.

3.1. Informal requirements and properties

The Gamma paradigm [2] advocates a way of programming that is close to the chemical reactions. One or more chemical reactions are applied on a multiset: a chemical reaction removes some values from a multiset, computes some results and inserts them into the multiset. We consider the following example: computing the sum of the integers present in a multiset. Figure 1 depicts a multiset and a possible Gamma computation achieving the result 8.

3.1.1. Informal requirements.

We intend to develop an application allowing several users to insert integers into a multiset that would be possibly distributed. 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. We call DSGamma (Distributed Gamma) system, the system made of the users, the multiset and the chemical reaction. We present the informal requirements in two parts. The first one presents the system operations that must be provided to the users, and the second one, the details about the data and internal computations.

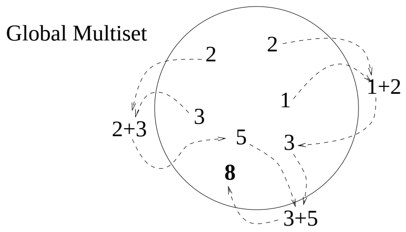

Figure 1. Gamma addition

System operations: [1] A new user can be added to the system at any moment; **[2]** A user may add new integers into 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 it.

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 up, and inserts the sum into the multiset; **[9]** There is only one type of chemical reaction, 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.

3.1.2. Properties.

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

3.2. Initial specification I: centralized view

The initial CO-OPN/2 specification **I** provides the most abstract view of the DSGamma system, that fulfills the informal requirements. There is a global multiset with several chemical reactions occurring concurrently on it. We have a non distributed data (the multiset), several processes (the chemical reactions), and each process, considered separately, is not distributed.

3.2.1. CO-OPN/2 specification. The initial CO-OPN/2 specification **I** is given by the `DSGammaSystem` class depicted by figure 2.

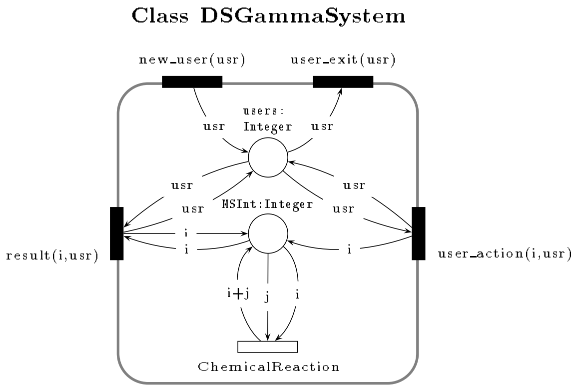

Figure 2. The initial CO-OPN/2 specification

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 provides to the outside.

The `new_user(usr)` method inserts the users' identity, `usr`, into the `users` place. 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, into 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 into 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`. If `usr` is in the `users` place, the `user_exit(usr)` method removes `usr`.

State and internal behavior: 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 CO-OPN/2 place `users` of type `Integer` stores the identity of the users as integers.

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

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`.

3.2.2. Properties. We now consider the property stated in subsection 3.1.2., and we check how it has evolved during this step. We will informally show, 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. 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, $m = n - [n/2]$ integers will remain in `MSInt`. The firings of `ChemicalReaction` proceed similarly on these m integers, and stop when $m = 1$. The initial specification **I** 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 the 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`.

3.3. First refinement R1: 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. Refinement **R1** is concerned with data distributivity.

3.3.1. Refinement process. 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 *global multiset* the multiset obtained by the union of all the local multisets. Figure 3 gives an illustration of chemical reactions over the distributed multisets MS_i, that compute the result 8.

The refinement process has to preserve the system operations. Thus, the overall specification of the DSGamma system must provide the same four methods `new_user(usr)`, `user_action(i,usr)`,

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

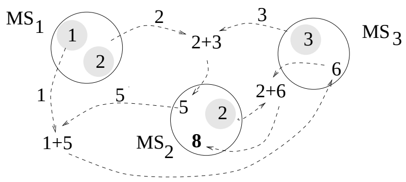

Figure 3. Distributed Gamma-like addition

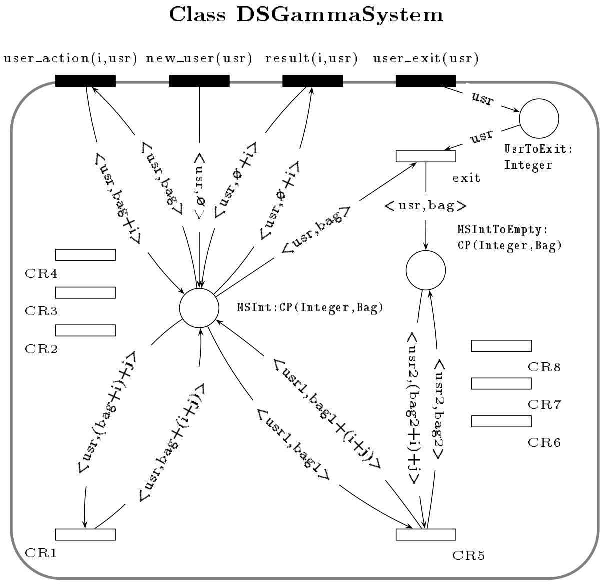

Figure 4. Refinement R1: data distribution

`result(i,usr)`, and `user_exit(usr)`[3] than the initial specification **I**. As the global multiset is split over several local multisets (one for each user), we must redefine the system operations and internal behaviors such that: (1) each user is mapped to a local multiset specified with a bag[4], (2) the chemical reactions have to remove integers from one or more local multisets, (3) the integers present in the local multiset of a user who wants to leave the system must be properly dispatched to the other local multisets.

3.3.2. CO-OPN/2 specification. The CO-OPN/2 specification of the application with distributed multisets is given by the `DSGammaSystem` class depicted by figure 4.

System operations: The `new_user(usr)` method inserts `<usr,∅>` into the `MSInt` place. A new user joins the system with an empty bag, representing an empty local multiset. 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` 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 one local bag.

The `user_exit(usr)` method inserts the `usr` value in the place `UsrToExit`. The `exit` transition then removes the pair `<usr,bag>` from the `MSInt` place and inserts it into the `MSIntToEmpty` place. As the user is tightly coupled with a local multiset, it is necessary to introduce at this point a treatment for dispatching his values. After having exited the system, a user may no longer enter a new integer, nor get the result, nor exit the system, unless it reenters the system, and the system itself cannot add integers into the user's local multiset.

[3] this will be the case for all the next refinements.
[4] the specification of the type `Bag` is made using an algebraic specification which defines an empty bag and the usual operations.

State and internal behavior: 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. They are of type `CP(Integer,Bag)`, an algebraic specification for Cartesian products of `Integer`s and `Bag`s; they store pairs `<usr,bag>`. Thus, we handle a multiset as a whole, and not through its elements as it was the case in the initial specification **I**.

Four chemical reactions (`CR1` to `CR4`) have been defined on `MSInt` only. They describe the four possible ways of removing two integers from one or two bags and inserting their sum into a (possibly other) bag. Four chemical reactions (`CR5` to `CR8`) have been defined on both `MSInt` and `MSIntToEmpty`. They are basically the same as the four chemical reactions defined on `MSInt` only, except for 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 chemical reactions 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 into his local multiset. For simplicity purpose, figure 4 depicts only the behavior of chemical reactions `CR1` and `CR5`: for `CR1` two integers `i,j` are removed from the same local multiset, their sum is inserted into this local multiset; for `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`.

3.3.3. Properties.

The computation of the sum is realized by the transitions CR1 to CR8. Due to the CO-OPN/2 semantics, the 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 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 (one integer per bag), because each occurrence of user_action(i,usr) has to access a whole bag in order to insert one integer. It is not possible for a bag to be accessed by two or more methods or transitions. In the initial specification **I**, $[n/2]$ pairs of integers are immediately involved in the 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 the CRi transitions only one integer remains in 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.

3.4. Second refinement R2: behavior distribution

Refinement **R1** provides a distributed view of the application at the data level. As we intend to obtain a Java application distributed over the Web, it is necessary to think about applets storing the local multiset related to the user who starts the applet. 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, refinement **R2** introduces a server. This leads to a behavior distribution.

3.4.1. Refinement process.

The server, called GlobalRelay, acts as a buffer between all the Applets. The server is only able to receive integers from a set of applets, and to send these integers to this same set of applets, such that an integer goes randomly from one applet to another via the server.

The system operations and internal behaviors are specified such that: (1) the GlobalRelay server is specified as a FIFO buffer, (2) each user is mapped to an applet, (3) the applets are responsible to maintain a local multiset of integers, (4) an applet has to insert integers entered by the user into its local multiset, (5) an applet has to collect pairs of integers, to make their sum, and to insert this sum into its local multiset, (6) an applet has to send integers to the server, (7) the applet has to correctly send its local multiset of integers to the server, once the user wants to leave the system, (8) the applets have to avoid a deadlock situation that would occur when the number of integers present in the whole system is less than the number of applets.

3.4.2. CO-OPN/2 specification.

The CO-OPN/2 classes of the application viewed with a client/server architecture is given by figures 5, 6 and 7.

System operations: The overall DSGamma system is specified by the DSGammaSystem class, it keeps the same methods than refinement **R1**.

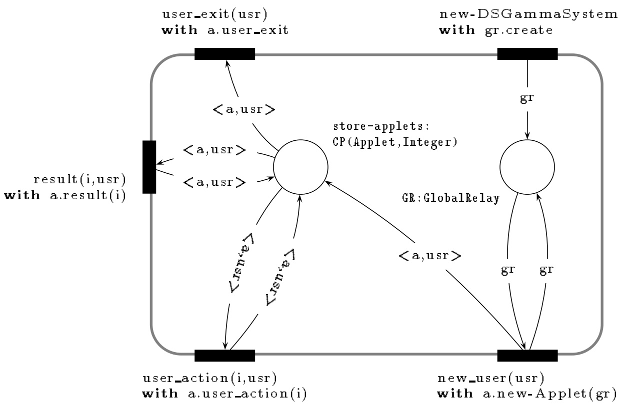

Figure 5. Refinement R2: overall system

A constructor new-DSGammaSystem has been added (a default constructor is no longer sufficient). It 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. The object identity gr is then stored in the GR place. The new_user(usr) method implies the dynamic creation of a new applet a (calling 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. 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 (calling 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` (calling `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` (calling `a.user_exit`).

State and internal behavior: 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 the `Applet` class, and by `buffer` of `FIFO(Integer)` type in the class `GlobalRelay`. Formally the FIFO of integers is specified with an algebraic specification. An integer travels from an `MSInt` place of an applet directly to `buffer`, and from there it goes to a `first` place of another applet, later it is summed with a second integer, the sum then goes into the `MSInt` place of this applet. 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 objects of class `Applet`.

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

Class GlobalRelay

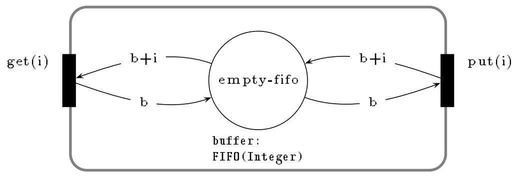

Figure 6. Refinement R2: server side

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 an 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 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 computed. 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 transitions: `getfirst`, `getsecond`, `tik`, `put`. The `getfirst` transition is responsible for obtaining the first integer being involved in a sum; as soon as it obtains a first integer it enables a timeout. The `getsecond` transition is responsible to remove a second integer from the FIFO `gr`, and to disable the timeout. 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. 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 smaller than or equal to the number of users, because all integers are blocked by different applets. The `put` transition randomly removes integers from the local multiset, and sends them to the FIFO buffer.

Class Applet

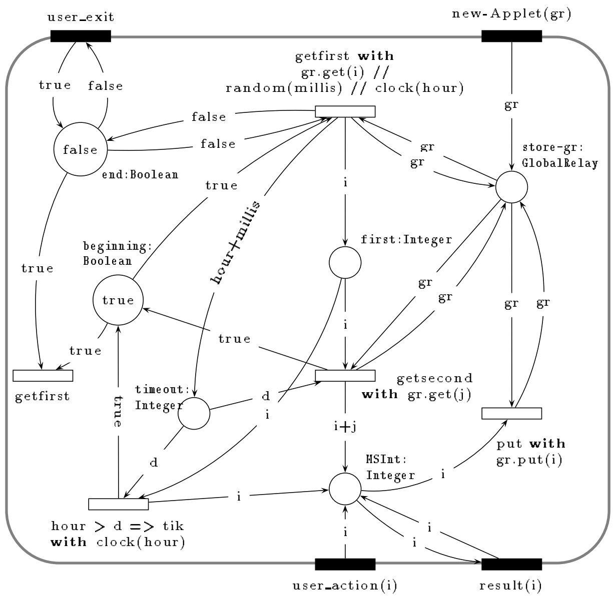

Figure 7. Refinement R2: client side

3.4.3. 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 long 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 several applets, say b. Note that b can be greater or smaller than n because a user may enter several integers, while another one may enter none. For simplicity purpose, we suppose that the users will not add new integers into the system. From now on, it is possible to interleave `put`, `getfirst`, `getsecond`, and `tik` internal transitions inside an applet object and between applet objects.

`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 the whole FIFO in order to insert or remove one integer. Thus, one chemical reaction occurs in parallel with another chemical reaction. However, the smaller steps (transitions) that realize one chemical 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 requiring an exclusive access to the FIFO buffer, thus the transitions occur in an interleaved way. The chemical reactions stop 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. When the number of integers present in the global multiset is smaller than the number of applets, a short period of deadlock occurs, before the `tik` transitions are firable. After a possibly long (but finite) time, only one integer will remain 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, only one integer remains 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.

3.5. Third refinement R3: communication layer

Refinement **R2** provides a client/server view of the application, with applets communicating with each other through a relay server. The applets communicate directly with the server. As the 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. The specification provided at this stage is also intended to be the last one before the Java program. For this reason, refinement **R3** takes into account some features of the Java programming language, and specifies all the Java components that will be part of the final program.

3.5.1. Refinement process. The informal view of both the specification and the implementation of the DSGamma system is given by figure 8. The relay server

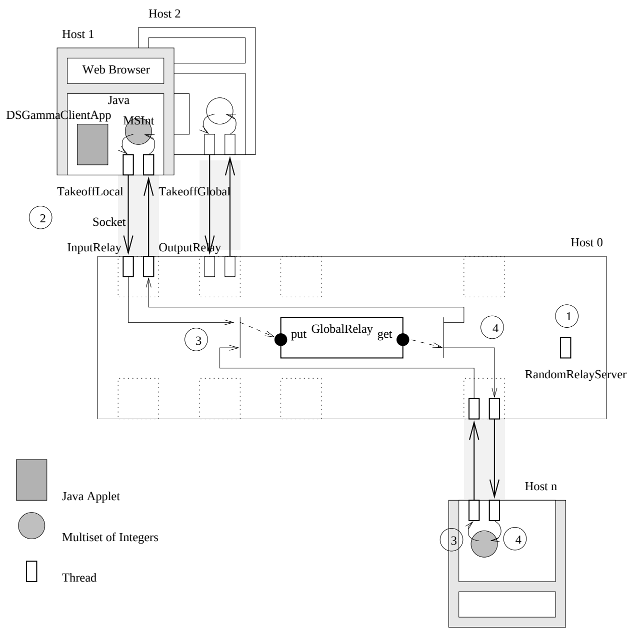

Figure 8. DSGamma implemented architecture

is bigger than it was in refinement **R2**, it is now given by class `RandomRelayServer` (position 1 on figure 8). 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`, is dedicated to the handling of the communication with an applet (position 2 on figure 8).

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)`, and `DataInputStream`, `DataOutputStream` classes. As soon as an applet connects to the `RandomRelayServer`, a `Socket` is created together with two streams. The first stream goes from the server to the applet, it is made of one DataInputStream at the applet 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 applet side working upon another FIFO(Bytes).

The applets are given by class DSGammaClientApp. They are more complex than what they are at the refinement **R2**. As soon as an applet is created, two threads of classes TakeoffLocal, TakeoffGlobal are created. These threads are responsible for the handling of the chemical reactions, the timeout and the quitting protocol (position 2 on figure 8). The applet also handles the local multiset MSInt.

3.5.2. CO-OPN/2 specification. The CO-OPN/2 specification of the application viewed as a Java applet based application is given by several CO-OPN/2 classes. Firstly, a set of basic Java classes has been specified into CO-OPN/2 classes, respecting the same inheritance tree than the Java programming language. A Java scheduler and an Appletviewer have been also specified. The Java Object class has been specified by the JavaObject CO-OPN/2 class, it is the same 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 the future program.

System operations: The overall DSGamma system is specified with the DSGammaSystem class, it keeps the same methods (with exactly the same behavior) than the refinement **R2**, except the constructor new-DSGammaSystem(port) that takes into account the fact that the server waits for applets connections on a given port. Thus, the constructor new-DSGammaSystem(port) creates an object of the class RandomRelayServer waiting on the port.

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

Server side: The CO-OPN/2 constructor of the RandomRelayServer class is new-RandomRelayServer(port). It creates the GlobalRelay FIFO buffer, and a ServerSocket on port port. A RandomRelayServer is a thread, whose run method waits indefinitely for connections on the ServerSocket, and as soon as an applet connects, it creates two threads of class OutputRelay, InputRelay respectively connected to the applet's socket.

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 (positions 3 on figure 8). It is also responsible for the handling of end signals incoming from the applet.

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 (positions 4 on figure 8). It is also responsible to handle end signals.

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

Applet side: The constructor new-DSGammaClientApp(port,remotehost) of the DSGammaClientApp applet class just stores the remote host and the port of the server. An init method has been added that creates: (1) the Socket, (2) the DataInputStream and DataOutputStream at the applet side, (3) the local multiset: MSInt vector (specifying a Java vector); (4) two threads, TakeoffLocal, TakeoffGlobal that realize the chemical reaction, the timeout, and the quitting protocol.

An applet keeps the same methods than refinement **R2**: the user_action(i) method just 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.

The TakeoffLocal thread permanently checks for integers in MSInt, removes randomly one and writes it to DataOutputStream at the applet's side. It also handles end signals.

The TakeoffGlobal thread reads a first integer from the DataInputStream at the applet's 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 to the first one, and inserted into MSInt. Otherwise, a tik transition prevents a deadlock, by inserting the first integer into MSInt. It also handles end signals.

In refinement **R2**, the timeout had been already specified, it is specified exactly in the same manner in refinement **R3**. The quitting protocol of refinement **R2** was more simple, because there were no intermediate buffers storing integers. It has been enhanced in refinement **R3**, in order to (1) notify the server that the user wants to exit; (2) receive from the server eventual integers present in the `DataOutputStream` at the server's side; and finally (3) empty the local multiset `MSInt` a last time before stopping.

Communication layer: The `DataOuputStream` and `DataInputStream` are used to insert or remove integers into or from a FIFO buffer of bytes, realizing the conversions. 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.

3.5.3. 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 split into two transitions each: one in `TakeoffGlobal` and one in `OutputRelay`, the `put` transition has been split over `TakeoffGlobal` and `InputRelay`, and the `tik` has been relocated to the `TakeoffGlobal` thread. Provided this extension of the chemical reaction to the communication layer, the same conclusion than refinement **R2** applies: (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.

3.6. Fourth refinement R4: the Java program

The Java program has exactly the same classes than refinement **R3** with exactly the same behavior.

3.6.1. Refinement process. The only differences with refinement **R3** are the following: firstly, in refinement **R3** we assumed that the streams work on `FIFO(Bytes)`, whereas in the concrete program, we need to be aware of the fact that the `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.

3.6.2. Properties. As the result of this paper, the Java program is derived immediately from refinement **R3**. The program and refinement **R3** show very few differences. Thus, we obtain the same justification for the property verification.

4. Conclusion

We have applied a formal development methodology for Java Web based parallel applications. Our methodology is based on several refinement steps of formal specifications and of 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. We have explained how these refinement steps were influencing the system properties. Currently, we express only informally these properties, but we envisage to use temporal logic for expressing, verifying and validating them.

References

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

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

[3] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindio, editors, *Advances in Petri Nets on Object-Orientation*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[4] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Formal development of java programs. Technical Report 97/248, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1997.

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

[6] 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.