Using Object-Oriented Algebraic Nets for the Reverse Engineering of Java Programs: A Case Study*

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 problem addressed in this paper is the following: "How to use high-level Petri nets for the reverse engineering of implemented distributed applications ?". We present a reverse engineering methodology applied on a real (simple) Java applet based client/server application. First, starting from the Java program, several abstraction steps are described using the CO-OPN/2 formal specification language. Then, we present a brand new research that studies properties preservations during a refinement process.

Keywords: *Reverse Engineering, Petri Nets, Algebraic Specifications, Concurrent and Distributed Systems, Java, Property Verification.*

1. Introduction

If we consider an already developed distributed application from a software engineering viewpoint, we are interested in having a methodology based on a formal specification language allowing a reverse engineering process that can be used for verification and validation purposes or for re-engineering.

In order to address these aspects, we are working on a methodology based on the joint use of a formal specification language and of a temporal logic. The advantage of formal specifications is that they allow a precise system description necessary for property verification. We have chosen to use the CO-OPN/2 (Concurrent Object-Oriented Petri Nets) specification formalism [3]. CO-OPN/2 integrates, in an object-oriented approach, Petri nets for the description of concurrent behaviors, and algebraic specifications [11] for the specifications of the structured data evolving in the Petri nets. The advantage of a temporal logic is that it allows to express verification and validation requirements as a set of properties. Moreover, temporal logics are well suited for Petri nets because of their operational state-event based approach. We are currently studying several temporal logics in order to choose the one that best fits our needs. Thus, this paper does not explain the use of temporal logic for expressing properties.

In order to apply a real reverse engineering process, first we have implemented a Java application, that we use as a case study. Then, we have performed several abstraction steps (programming language, communication layer, client/server, data distribution), and considered some properties at each step.

The plan of the paper is the following: first, we introduce the basic concepts of the specification formalism CO-OPN/2; second, we present in details several abstraction steps performed on our real Java application; third, we present the methodology we intend to assess concerning the validation of properties during a reverse engineering process.

2. The CO-OPN/2 specification formalism

CO-OPN/2 [3] is an hybrid specification formalism based on algebraic specifications [11] and Petri nets which are combined in a way that is similar to algebraic nets [9]. 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 specific 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) which interact and collaborate together in order to accomplish the various tasks of the system.

 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".

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. 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 corresponds to the services provided to the outside, while the latter composes 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. 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. 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 objects, followed by a service request to a third object.

For each transition (parameterized or not), one or more behavioral axioms are defined using: (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, 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. Since object identifiers are algebraic values they can be stored in the places of algebraic nets. Moreover it is possible to define data structures which 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 which 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.

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 which are simultaneously executable. The firing of an object's method causes internal transitions to be fired spontaneously. The internal transitions are fired as long as their pre-condition is fulfilled. An object's method can be fired only if no internal transition is firable. 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 [3].

3. Reverse engineering: from Java to CO-OPN/2

This section presents (1) the informal requirements and the Java program **P** of a real application, and (2) several abstraction steps (**A1** to **A4**) which lead to even more abstract CO-OPN/2 specifications of the given application.

3.1. Informal requirements

The Gamma paradigm [2] advocates a way of programming which 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 one or more results and inserts them into the multiset.

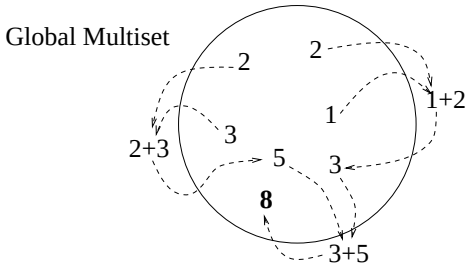

Figure 1. Gamma addition

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.

The application must allow 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. 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 part presents the system operations which must be provided to the users, and the second part, 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 enter 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 computes 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 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.

3.2. The Java program P

The Java [1] program, **P**, of the distributed Gamma-like addition follows a Java applet based client/server architecture, as depicted by figure 2. It is running at the following address `http://lglsun.epfl.ch/Team/GDM/DSGamma.html`. An applet is downloaded and executed by an Internet Web browser, but the applet can communicate only with servers located on the host where the applet comes from.

A server, the `RandomRelayServer` thread, acts as a random relay between the applets, the server maintains a FIFO of integers, `GlobalRelay`, and waits for applet's connections (position 1 on figure 2). The `DSGammaClientApp` applet maintains a graphical user interface and a local multiset of integers `MSInt` (implemented as a Java `Vector`). The user can enter integers directly into the local multiset by the means of the graphical user interface. As soon as an applet is started, a socket is created between the applet and the server. In addition, two threads `InputRelay`, `OutputRelay` are created at the server side in order to handle integers incoming from and going to the socket linking the server and the applet. Similarly, two more threads `TakeoffGlobal` and `TakeoffLocal` are created at the applet side (position 2 on figure 2). The `TakeoffLocal` thread is responsible for taking integers off the local multiset and for sending them to the server. The `InputRelay` thread receives these integers and forwards them to the `GlobalRelay` FIFO (positions 3 on figure 2). The `OutputRelay` thread takes integers at the head of the `GlobalRelay` FIFO and sends them to the applet. The `TakeoffGlobal` thread is responsible for waiting for two integers incoming from the server, making their sum and inserting this sum into the local multiset maintained by the applet (positions 4 on figure 2).

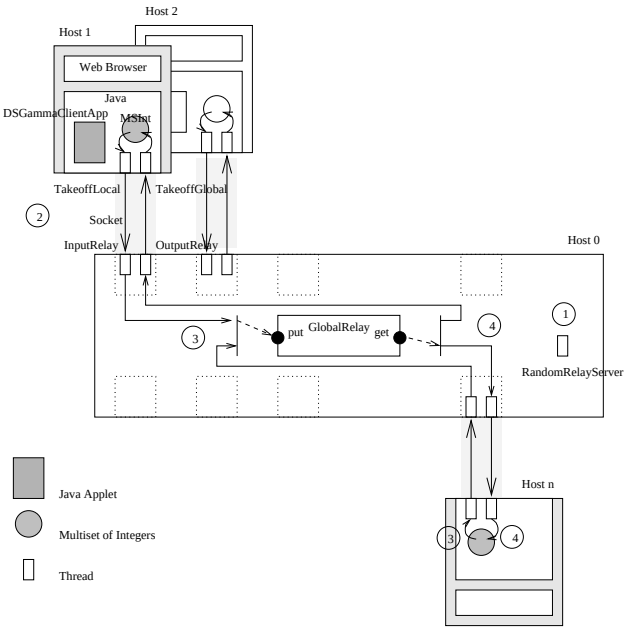

Figure 2. DSGamma implemented architecture

A user who wants to leave the system informs the applet by the means of the graphical user interface. The `TakeoffLocal` and `TakeoffGlobal` threads properly send to the server all the integers remaining in the local multiset, and stop receiving any new integer from the server. For that purpose a two-way handshake protocol is used.

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 applets (which is also the number of local multisets). 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 without timeout, each of these two applets would be blocked indefinitely waiting for a second integer. Consequently, the whole system would be in a deadlock state. The `TakeoffGlobal` thread uses a timeout in order to avoid that deadlock.

3.3. First abstraction A1: from Java to CO-OPN/2

Abstraction **A1** *translates* the program, written in the Java programming language, into specifications expressed with CO-OPN/2.

3.3.1. Abstraction process. Abstraction **A1** leads to CO-OPN/2 specifications which take into account both the semantics of the Java programming language and the application's behavior (the given program). A Java program is built upon existing classes, i.e. the basic classes provided by the Java programming language. Similarly we build the CO-OPN/2 formal specifications of the Java application upon CO-OPN/2 formal specifications of the Java basic classes. In this manner, we cope with the problem of expressing both the Java semantics and the application's behavior: a first layer of CO-OPN/2 specifications of Java basic classes is provided (as building blocks), and the CO-OPN/2 specifications of the application is built on top of this layer.

3.3.2. Building blocks. We have specified a dedicated CO-OPN/2 class for each Java basic class. The inheritance tree of these CO-OPN/2 classes reproduces exactly the inheritance tree of the Java classes. The `Object` Java class is the superclass of all Java classes. The corresponding CO-OPN/2 class is called the `JavaObject` class and is the superclass of all the CO-OPN/2 classes related to Java. The CO-OPN/2 `JavaObject` class specifies the `wait`, `notify`, `notifyall` methods and the way they affect a thread's execution, as well as the locks associated to each object. For the needs of the application described in this paper, we have specified the Java `Thread`, `Applet` and `Socket` classes. The complete CO-OPN/2 specification of these Java basic classes is given in [6].

3.3.3. CO-OPN/2 specification. We have specified a dedicated CO-OPN/2 class for each Java class defined by the program **P**. These CO-OPN/2 specifications are constructed using the CO-OPN/2 specifications of the Java basic classes, either by sub-classing them or by using them. The graphical user interface has not been specified. Except for the socket that has been specified simply with two buffers (for the two streams), any other implementation detail is fully specified. Every data structure and algorithm has been specified in order to reflect the Java semantics. Abstraction **A1** provides the complete specifications of the Java program. It is fully described in [6].

In addition to the CO-OPN/2 specifications of the Java classes of program **P**, we have specified the overall system with the CO-OPN/2 `DSGammaSystem` class of figure 3. This class specifies the beginning of the system (constructor `new-DSGammaSystem(port)` creates the `RandomRelayServer` waiting on `port`), and the interaction of the users and the system.

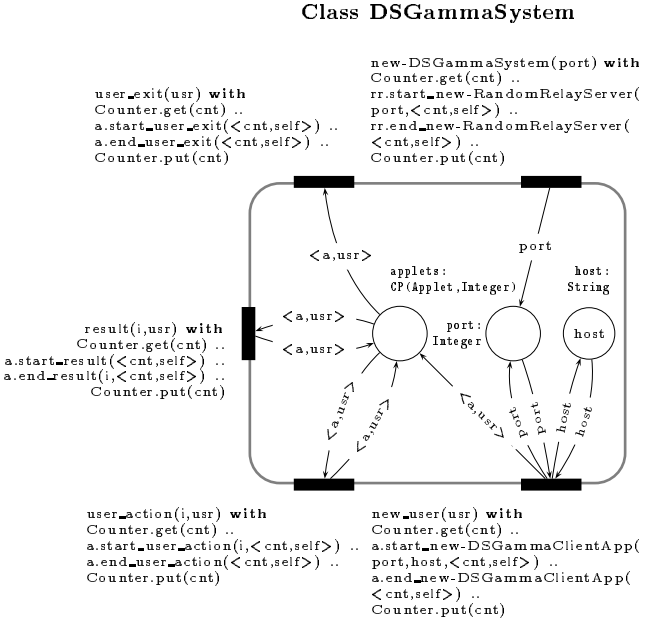

Figure 3. Abstraction A1: overall system

Informal requirements [1] to [4] are specified each with a dedicated CO-OPN/2 method of the `DSGammaSystem` class. `new_user(usr)` inserts the new user `usr` into the system and creates an applet `a` dedicated to this user. The identity of a user is specified as an integer. `user_action(i,usr)` enables `usr` to enter integer `i` in the system, this method informs the applet `a`, dedicated to `usr`, that integer `i` enters the system. `result(i,usr)` enables `usr` to obtain a partial view of its local multiset, this method informs the applet `a`, dedicated to `usr`, that `usr` wants a result. `user_exit(usr)` removes `usr` from the system and forwards this information to the corresponding applet. Further abstraction steps keep the class `DSGammaSystem` and its four methods.

3.4. Second abstraction A2: communication layer abstraction

Abstraction **A1** provides CO-OPN/2 specifications very close to the Java program and its semantics.

Abstraction **A2** removes the programming language and the socket layer. This step provides the most abstract specification of the application viewed as a *client/server* application.

3.4.1. Abstraction process. We throw away all specific constructs required by the target programming language, here Java. The notions of Java object, Java thread, Java socket, and Java applet disappear. We keep only the skeleton of the application, i.e. we keep the (distributed) architecture and behavior which are specific to the application but not specific to the Java programming language.

Besides the abstraction from the programming language, we abstract the communication layer provided by the sockets. The applets, instead of reading and writing data from and to sockets, directly receive and send data from and to the server. Thus, we keep a client/server architecture with one server and several applets, but without sockets and threads dedicated to the socket's handling.

The server has become very simple, it has been shrunk to the `GlobalRelay` functionality, i.e. the server acts as a FIFO buffer, where every applet directly deposits integers, and from where every applet directly takes off integers. Similarly, at the user's side, the applet and the chemical reactions, become more simple. The applet handles a local multiset in the following manner: (1) new integers coming in from the user are inserted into the local multiset, (2) integers stored in the local multiset are taken off the local multiset and sent to the server, and (3) pairs of integers coming from the server are collected, their sum is computed, and inserted into the local multiset, (4) the applet has to correctly send its local multiset of integers to the server, once the user wants to leave the system, (5) the applet has to avoid a deadlock situation occurring when the number of integers present in the whole system is smaller than the number of applets.

3.4.2. CO-OPN/2 specification. Abstraction **A2** is given by three CO-OPN/2 classes, (1) the `DSGammaSystem`, (2) the `GlobalRelay`, and (3) the `Applet` classes (figures 4, 5 and 6).

System operations: The overall DSGamma system is specified by the `DSGammaSystem` class, it keeps the same CO-OPN/2 methods than abstraction **A1**.

The `new-DSGammaSystem` CO-OPN/2 constructor 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. The `user_action(i,usr)` method checks whether 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`).

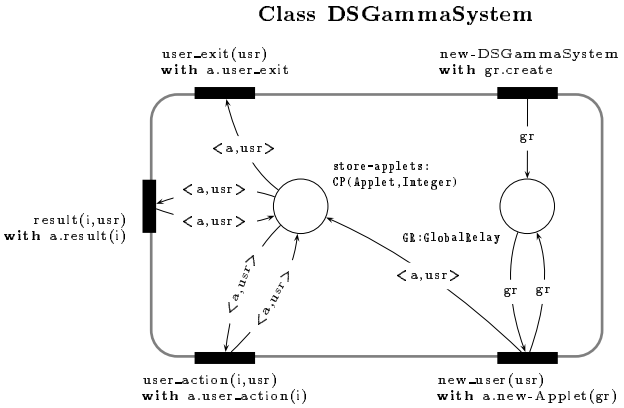

Figure 4. Abstraction A2: overall system

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 the union of 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`. The FIFO of integers is specified with an algebraic specification. An integer goes from an `MSInt` place of an applet directly to `buffer`, and from there it goes to a `first` place of another applet, waiting for a second integer, their sum then goes into the `MSInt` place of this 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 by the means of the `get(i)` method.

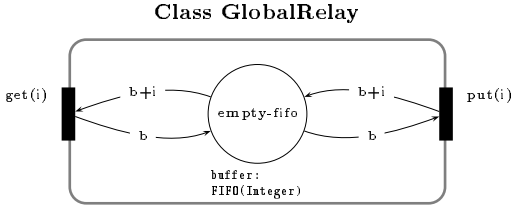

Figure 5. Abstraction A2: server side

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 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 computed. The `user_action(i)` method 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`.

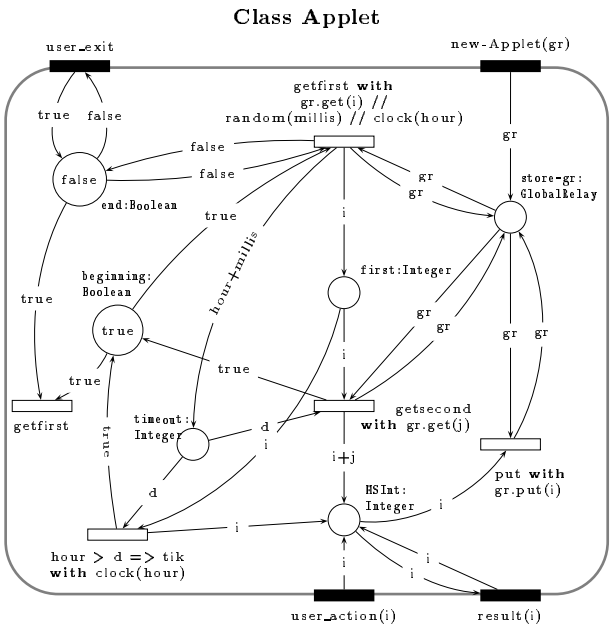

Figure 6. Abstraction A2: client side

The chemical reactions are specified by 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; as soon as it obtains a first integer it enables a timeout. The `getsecond` transition is responsible for removing a second integer from the FIFO `gr`, and for disabling the timeout. The `tik` transition handles a timeout event occurring before a second integer can be obtained by the `getsecond` transition. It is responsible for disabling the timeout and for inserting 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. The `put` transition randomly removes integers from the local multiset, and sends them to the FIFO buffer.

3.5. Third abstraction A3: client/server abstraction

Abstraction **A2** provides a client/server view of the application. Abstraction **A3** abstracts the notion of client/server that had been imposed by the Java programming language, because applets can connect only to the host where they come from. This step provides the most abstract specification of the application viewed as a *distributed* application.

3.5.1. Abstraction process. We remove the server as well as the applets. We keep just the notion of distributed local multisets, each of them related to a user. Data travels directly from one local multiset to another, without traveling through a server.

The chemical reactions are no longer distributed. A chemical reaction is specified as an atomic action which takes two integers from possibly two different local multisets, and which inserts their sum into another local multiset. Several chemical reactions may occur concurrently. There are several types of chemical reactions according to how they remove integers from the local multisets.

The multiset of integers is physically distributed over several different locations. We call *local multiset*, the portion 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.

3.5.2. CO-OPN/2 specification. Abstraction **A3** consists of the `DSGammaSystem` class depicted by figure 7. This class maintains several users and their related multisets. All possible chemical reactions are specified on those multisets.

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 into the place `UsrToExit`. The `exit` transition then removes the pair `<usr,bag>` from the `MSInt` place and inserts it into the `MSIntToEmpty` place. After having exited the system, a user may no longer enter new integers, nor get the result, nor exit the system, unless it reenters the system, and the system itself cannot insert integers into the user's local multiset.

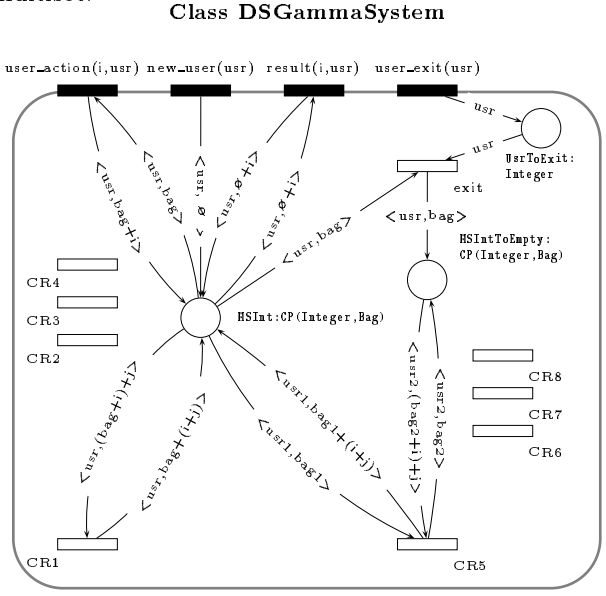

Figure 7. Abstraction A3: DSGammaSystem

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>`. Bags are specified with an algebraic specification.

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. Figure 7 depicts the behavior of chemical reactions `CR1` and `CR5`: (`CR1`) two integers `i,j` are removed from the same local multiset, and their sum is inserted into this local multiset; (`CR5`) two integers `i,j` are removed from the same local multiset in `MSIntToEMpty`, and their sum is inserted into another local multiset in `MSInt`.

3.6. Fourth abstraction A4: data distribution abstraction

Abstraction **A3** abstracts the notion of distributed computing, but the notion of distributed data is kept. Abstraction **A4** abstracts the notion of distributed data. This step provides the most abstract specification of the application when it is *not distributed*.

3.6.1. Abstraction process. There is a global (not distributed) multiset and only one type of chemical reactions. Abstraction **A4** contains only one class, the `DSGammaSystem`, which maintains several users and only one global multiset. One type of chemical reaction is defined on the global multiset, which removes two integers from the multiset and inserts their sum into the multiset.

3.6.2. CO-OPN/2 specification. The class `DSGammaSystem` is depicted by figure 8.

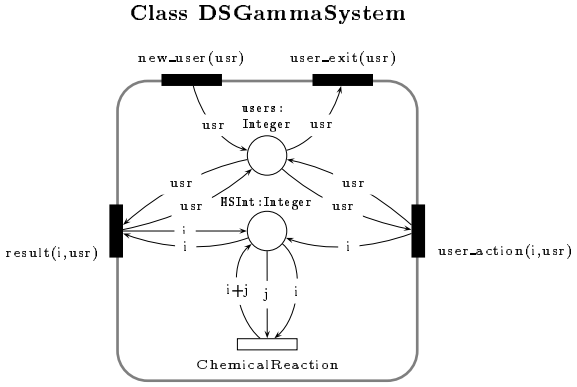

Figure 8. Abstraction A4: DSGammaSystem

System operations: 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. 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.

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` into `MSInt`.

4. Towards a formal verification of stepwise refinements in CO-OPN/2

We intend to develop a methodology which can be used during a development process: a CO-OPN/2 specification is refined into another CO-OPN/2 specification and some desired properties are preserved during the refinement step. In addition, the methodology can be used for validating properties on previously implemented applications: a reverse engineering process is performed and properties are studied during the abstraction process.

In this section, we firstly underline some problems that arise when Petri nets, and more particularly CO-OPN/2 specifications are refined. Secondly we give the lines of the methodology for both the refinement case and the reverse engineering case. Finally, in order to show the interest of this methodology, we list some properties and follow informally their evolution during the abstraction process of the DSGamma application.

4.1. Related work

Refinement of Petri nets. Usually, the refinement of a Petri net consists of the replacement of a transition or a place by a net. The two usual interpretations of refinement of Petri nets are: (1) a net and its refinement have the same behavior wrt safeness or liveness properties (preservation of behavior), or (2) two semantically equivalent nets are refined into two semantically equivalent nets (preservation of behavior equivalence) [4].

Refinement of algebraic specifications. Usually, an algebraic specification $Spec'$ is a refinement of $Spec$ if both specifications have the same signature and if all the models of $Spec'$ are models of $Spec$ [11].

Refinement of CO-OPN/2 specifications. The refinement of an algebraic Petri net combines both the replacement of transitions or places by an algebraic net, and the replacement of algebraic specifications by other algebraic specifications. The CO-OPN/2 language structures algebraic Petri nets with a synchronization mechanism. The refinement of a CO-OPN/2 specification by another CO-OPN/2 specification can be obtained by the refinement of an algebraic net, or more generally by the replacement of a CO-OPN/2 specification by another CO-OPN/2 specification.

Temporal logic and Petri nets. Several approaches [10, 12] combine Petri nets and temporal logic [7] in order to define and verify properties of distributed systems described with Petri nets. Z and VDM have many results concerning refinement and proofs, including proofs of temporal logic formulae over the specifications [8].

4.2. A refinement methodology

The proposed refinement methodology is based on the joint use of the CO-OPN/2 formal specification language and of a temporal logic. The behavior of a system is specified by means of CO-OPN/2 specifications, while properties expected by the system are expressed by means of temporal logic formulae. These properties do *not* reflect the whole behavior of the system, they only reflect the behavior part that must be preserved during all subsequent refinement steps. The range of properties we are interested to verify covers functional local properties of a CO-OPN/2 object and global properties involving several parts of the system. A refinement is then defined as the replacement of a specification by a new one which respects the properties required by the replaced specification and which takes into account implementation constraints.

The refinement process starts with a pair (S_0, P_0), where S_0 is an abstract CO-OPN/2 specification of the system, and P_0 is a set of temporal logic formulae expressed on the basis of S_0. The set of temporal logic formulae has to be proven on S_0. This set of temporal logic formulae expresses the minimal set of properties that the desired system has to verify during the whole refinement process. At each refinement step, both the formal specification and the set of temporal logic formulae of the previous step are refined. Thus, each refinement step i is given by a pair (Ref_{S_i}, Ref_{P_i}) which produces, from a pair (S_{i-1}, P_{i-1}), a pair (S_i, P_i). The refinement process stops when the specification S_i is expressed by the means of predefined building blocks. These building blocks are CO-OPN/2 components that take into account the targeted programming language. In addition, each refinement step has to provide the proof that P_i is satisfied by S_i. Indeed, given the proof $Proof_{i-1}$ that P_{i-1} is satisfied by S_{i-1}, and the pair of refinements (Ref_{S_i}, Ref_{P_i}), then the methodology has to bring the proof $Proof_i$ that P_i is satisfied or not by

S_i. (See figure 9.)

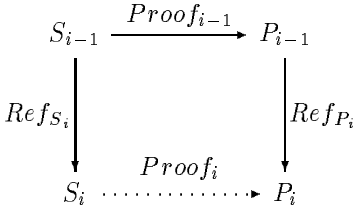

Figure 9. A refinement step

If $Proof_i$ brings the proof that S_i satisfies P_i, then we say that S_i is a refinement of S_{i-1} wrt P_{i-1}, because the properties required by P_{i-1} are preserved. If the sequence of specifications $S_0, \ldots, S_n$ is such that $\forall i \in \{1, \ldots, n\}$, S_i is a refinement of S_{i-1}, then S_n preserves at least the initial set of properties P_0.

It is important to note that we do not require the preservation of the whole behavior of a refined specification during the refinement process. We only require the preservation of the behavior that is described by the temporal logic formulae expressing the properties.

The methodology provides: (1) a *formal specification language* (CO-OPN/2) for expressing the behavior of a system; (2) a *temporal logic* for expressing the properties expected by the system; (3) a set of *building blocks* suitably specified for the targeted programming language; (4) *guidelines* for refining abstract specifications into concrete specifications built exclusively with the building blocks; (5) a *refinement process* that leads to concrete specifications (close to a program) satisfying the desired properties expected by the system.

4.3. The methodology and the reverse engineering

The proposed methodology can be applied to the development of an application. It can also be applied to prove that an already implemented application satisfies some desired properties. Indeed, we propose the following reverse engineering process: starting from the application's program, a very concrete CO-OPN/2 specification is derived which uses exclusively the building blocks, we can call it S_n; several abstraction steps are then performed, leading to more and more abstract CO-OPN/2 specifications. Once a sufficiently abstract CO-OPN/2 specification, called S_0, is reached, the desired properties are expressed for this abstract specification. The set of properties is called P_0. The refinement methodology is then applied, starting from the pair (S_0, P_0). The refinement path for the CO-OPN/2 specifications is given by the CO-OPN/2 specifications obtained during the reverse engineering process. During the refinement process, the set of properties P_i is computed from P_{i-1} and from the transformation of S_{i-1} to S_i. If it is possible to prove that S_n satisfies P_0 then, we say that the program satisfies the set of the desired properties.

Section 3. describes the reverse engineering process performed on the CO-OPN/2 specifications. Abstractions **A4** to **A1** can be used as the refinement path for the specification part of the methodology. The building blocks are given by the CO-OPN/2 classes specifying the Java basic classes. Starting from **A4**, and performing several (other) refinement steps, a new implementation of the DSGamma system, based on Coordinated Atomic Actions (CAAs) has been provided [5].

4.4. The case of the DSGamma system

We present some properties and we follow them during the abstraction process described in section 3. Some of these properties are true for all abstractions, while others are true for some of them only. We explain informally the evolution of the properties on the basis of the transformation of the Petri nets, the algebraic specifications, and the CO-OPN/2 specifications.

Subsequently, we assume the following:

> "There exists a time T, such that after T, no new integer is entered in the system."

4.4.1. Abstraction A1. Abstraction **A1** is the immediate translation of the Java program into CO-OPN/2 specifications. It is constructed on the basis of the building blocks (CO-OPN/2 classes specifying the Java basic classes). Among others we are interested in the following properties:

> P1. "After T, the system computes the sum of the remaining integers distributed among (1) the `MSInt` of each `DSGammaClientApp`, (2) the `GlobalRelay` FIFO, (3) the two buffers of each `Socket`."

Property P1 expresses the fact that abstraction **A1** has to be able to compute the correct sum of all the integers remaining in the whole system after T. Property P1 is true if at least one user remains in the system: the `TakeoffGlobal` thread of each applet removes two integers from the global multiset and inserts the sum into the local multiset `MSInt` of the `DSGammaClientApp`. Property P1 is not true if all the users leave the system at the same time: the `GlobalRelay` FIFO buffer will store their integers. The sum will continue to be computed only when a new user enters the system.

> P2. "Every integer received by the `GlobalRelay` FIFO has to be taken by exactly one of the `OutputRelay` threads."

Property P2 expresses the fact that the server defined by abstraction **A1** must neither lose an integer nor duplicate an integer. This property is true, because just one `OutputRelay` thread can call method `get(i)` of the `GlobalRelay` FIFO at once. Indeed, the `get(i)` method cannot be fired twice simultaneously, because each firing requires the buffer `b` in the place `buffer`. The `GlobalRelay` FIFO of abstraction **A1** is specified as that of abstraction **A2** (figure 5).

> P3. "Every user can see at any moment an integer that is in the `MSInt` maintained by its `DSGammaClientApp` applet."

Property P3 expresses the fact that the `result(i,usr)` method of the `DSGammaSystem` class can always be fired. This method forwards to the `DSGammaClientApp` the information that the user wants to see an integer. In order to read an integer in `MSInt`, the `DSGammaClientApp` needs an access to the whole `Vector` specifying the local multiset (the CO-OPN/2 class for the Java `Vector` class). Thus, property P3 is not true: the user can read an integer in its local multiset provided he is not currently inserting a new integer with the `user_action(i,usr)` method, or if the `DSGammaClientApp` is not involved in a chemical reaction.

4.4.2. Abstraction A2. The abstraction step from abstraction **A1** to abstraction **A2** preserves the same algebraic specification for the `GlobalRelay` FIFO (`FIFO(Integer)`). On the contrary, the `MSInt` place of the `Applet` class in abstraction **A2** stores `Integer`s, while the `MSInt` place of the `DSGammaClientApp` class in abstraction **A1** stores an algebraic specification of type `Vector`. Thus, in abstraction **A1**, an access to an integer in the `MSInt` place implies an access to the `Vector` storing the whole multiset of integers. In abstraction **A2**, each integer in the `MSInt` place can be accessed separately. Properties P1, P2, and P3 are expressed now on the basis of the specifications provided by abstraction **A2**.

> P1. "After T, the system computes the sum of the remaining integers distributed among (1) the `MSInt` of each `Applet`, and (2) the `GlobalRelay` FIFO."

The socket layer has disappeared, thus the integers remain in the `Applet`s or in the `GlobalRelay`. Property P1 is true if at least one user remains in the system: the `getfirst` and `getsecond` transitions will remove one integer each from the global multiset, and insert the sum into the local multiset. Property P1 is not true if all the users leave the system: all the integers present in the `MSInt` place of the `Applet`s will be moved to the `GlobalRelay` FIFO and will stay there until a new user enters the system.

> P2. "Every integer received by the `GlobalRelay` FIFO has to be sent to exactly one of the `Applet`s."

In abstraction **A1** the `DSGammaClientApp`s do not require themselves integers from the `GlobalRelay` FIFO. An `OutputRelay` thread is responsible for that. In abstraction **A2**, the `Applet`s themselves require integers from the `GlobalRelay` FIFO. As for abstraction **A1**, this property is true because of the specification of the `GlobalRelay` FIFO: two or more accesses to the FIFO are not allowed at the same time.

> P3. "Every user can see at any moment an integer that is in the `MSInt` maintained by its `Applet`."

Property P3 is not true, because due to the CO-OPN/2 semantics, the `result(i)` method of the `Applet` class can be fired only if no internal transition of that class is firable. As the internal transitions are used to compute chemical reactions, it may happen that the `result(i)` method is firable only after several chemical reactions have been computed. The user cannot see at any moment an integer in its local multiset.

4.4.3. Abstraction A3. The abstraction step from abstraction **A2** to abstraction **A3** changes the algebraic specification for the `MSInt` place. In abstraction **A2** the `MSInt` place stores `Integer`s. In abstraction **A3** the `MSInt` place stores algebraic specifications for Cartesian products of users and bags of integers. Thus, in abstraction **A2**, each integer in the `MSInt` place can be accessed separately, while in abstraction **A3** an access to an integer in the `MSInt` place implies an access to the `<usr,bag>` value storing the user identity and the whole multiset of integers.

> P1. "After T, the system computes the sum of the remaining integers distributed among the `bag` of each `usr`."

The server layer has disappeared, thus integers remain only in the bags of the users. Property P1 is true if at least one user does not want to exit: the `CRi` transitions are fired until only one integer remains in the union of all the bags, i.e. in the global multiset. Each `CRi` transition removes two integers from the global multiset and inserts their sum into the global multiset.

Property P1 is false if all the users want to leave the system, then all the pairs `<usr,bag>` will be moved to the `MSIntToEmpty` place, and none of the `CRi` can be fired because they need a pair in the `MSInt` place. The system is blocked until a new user enters the system.

P2 is no longer necessary, indeed P2 is intended to verify if the server does not lose or duplicate integers received from the applets. Abstraction **A3** skips the notion of server, thus this property disappears.

P3. "Every user can see at any moment an integer that is in its `bag`."

This property is not true, because the user can see the final result (the sum), *and* only if the result is in its bag. Indeed, due to the CO-OPN/2 semantics, the `result(i,usr)` method can be fired only if none of the `CRi` transitions is firable. These transitions are firable as long as there are at least two integers in the union of all bags. The sum of all integers is in exactly one of the bags, the other bags are empty.

4.4.4. Abstraction A4. In abstraction **A3** the `MSInt` place stores algebraic specifications for Cartesian products of users and bags of integers. In abstraction **A4** the notion of distributed multiset disappears, only a global multiset remains. The `MSInt` place of abstraction **A4** stores `Integer`s. In abstraction **A3** an access to an integer in the `MSInt` place implies an access to the `<usr,bag>` value, while in abstraction **A4** each integer is accessed separately.

P1. "After T, the system computes the sum of the integers present in the `MSInt` place of the `DSGammaSystem`."

This property is true, because the `ChemicalReaction` transition removes two integers from the `MSInt` place and inserts their sum into that place. The `ChemicalReaction` transition stops being fired when only one integer remains in the place. This integer *is* the result. Property P1 is true even if all the users wants to leave the system, because the sum is computed independently of the users.

P3. "Every user can see at any moment an integer that is in `MSInt`."

Due to the CO-OPN/2 semantics, the sum is completely computed before one of the methods of the `DSGammaSystem` class of abstraction **A4** can be fired. As there is one copy of the sum in the `MSInt` place, one user only can see it at once. In abstraction **A3** only the user whose bag contains the sum can see the sum. In abstraction **A4** all the users can see the sum but not at the same time.

5. Conclusion

We have presented the following case study: starting from a real Java program, we have performed several abstraction steps using the CO-OPN/2 formal specification language. Several properties expected by the implementation have been informally studied during the reverse engineering process. This case study is a preliminary work towards the assessment of a refinement methodology for distributed applications.

References

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

[2] J.-P. Banâtre and D. Le Métayer. Gamma and the Chemical Reaction Model: Ten Years After. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 3–41. Imperial College Press, 1996.

[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] Wilfried Brauer, Robert Gold, and Walter Volger. A survey of behaviour and equivalence preserving refinement of Petri nets. In *Advances in Petri Nets 1990*, volume 483, pages 1–46. Lecture Notes in Computer Science, 1992.

[5] G. Di Marzo Serugendo, N. Guelfi, A. Romanovski, and A. Zorzo. Formal development and validation of the DSGamma system based on CO-OPN/2 and Coordinated Atomic Actions. Technical Report of the Esprit Long Term Research Project 20072 'Design For Validation', University of Newcastle Upon Tyne, Department of Computer Science, 1997.

[6] 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.

[7] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Centre, 25 December 1991.

[8] K. Lano. *Formal Object-Oriented Development*. Springer-Verlag, London, 1995.

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

[10] 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.

[11] 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.

[12] 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.