Formal Development and Validation of the DSGamma System Based on CO-OPN/2 and Coordinated Atomic Actions*

Giovanna Di Marzo Serugendo[1], Nicolas Guelfi[1],
Alexander Romanovsky[2], Avelino Zorzo[2]
[1] LGL-DI, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland
[2] Department of Computing Science, University of Newcastle upon Tyne

Abstract

The objectives of this research are twofold. On the first hand, it aims to show the interest of Coordinated Atomic actions (CA actions) as a design concept and, on the other hand it explains how the formal language CO-OPN/2 can be used to express a CA action design. A real distributed application is developed according to a simple development life cycle: informal requirements, specification, design, implementation. The design phase is built according to the CA action concept. The CO-OPN/2 language is used to express the specification, and design phase. The implementation is made in Java based on a library of generic classes adapted to CA action concepts. The validation phase is briefly addressed, in order to demonstrate the extent to which the development methodology followed in this paper can be useful for proving properties.

Keywords: structuring complex concurrent systems, CO-OPN/2, formal development, design for validation, Java.

1 Introduction

We provide a top-down engineering methodology for the development of a Java application based on both CO-OPN/2 formal specifications and Coordinated Atomic actions (CA actions). The advantage of a formal specification is that it describes a system in a mathematical and thus precise way, which is necessary when complex dependable applications are developed. Thus, formal specifications also provide a useful tool for verification and validation purposes. The advantage of a CA action design is that it helps to guarantee the data consistency and to parallelize the system properly. It is also a useful structuring mechanism that can simplify proofs of the system's correctness.

1.1 CO-OPN/2

CO-OPN/2 (Concurrent Object Oriented Petri Nets) is an object-oriented formal specification language [4] that integrates Petri nets for the description of concurrent behaviors, and algebraic specifications [6] for the specification of structured data evolving in the Petri nets.

Object and Class. An object is considered to be an independent entity composed of an internal state, which provides some services to the exterior. The only way to interact with an object is to invoke one of 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

*This research has been supported by ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa). Avelino Zorzo is also supported by CNPq (Brazil) under grant number 200531/95.6.

place consists of a multi-set of algebraic values. The transitions are divided into two groups: the parameterized transitions, also called the methods, and the internal transitions. The former correspond to the services provided to the outside, while the latter describe the internal behaviors of an object. Unlike methods, the internal transitions are invisible to the exterior world and are spontaneous events (they are fired as soon as their preconditions are satisfied). The internal transitions are fired as long as their pre-condition is fulfilled. An object's method can be fired only if no further internal transition can be fired. 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 for method invocations has been adopted.

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 `create` is provided. In all the specifications of this paper, we use the following convention: a creation method is called `new-classname`. For a creation method to be actually a constructor, it is necessary to declare it under the `Creation` field. Usually classes are used to dynamically create new instances, but it is also possible to declare static instances.

Object Identity. Each class instance has an identity, which is also called an object identifier, that may be used as a reference. An order-sorted algebra of object identifiers is constructed. Since object identifiers are algebraic values, they can be stored in places of the nets.

Object Interaction. In our approach, the interaction with an object is synchronous, although asynchronous communications may be simulated. Thus, when an object requires a service it asks to be synchronized with the method (parameterized transition) of the object provider. The synchronization policy is expressed by means of a synchronization expression, which may involve many partners joined by three synchronization operators (one for simultaneity, one for sequence, and one for alternative or non-determinism). An object may simultaneously request two different services of two different partners, followed by a service request to a third object.

Concurrency. Intuitively, each object possesses its own behavior and concurrently evolves with the others. The Petri net model naturally introduces both inter-object and intra-object concurrency into CO-OPN/2 because objects are not restricted to sequential processes. Moreover, a set of method calls can be concurrently performed on the same object. The step semantics of CO-OPN/2 allows the expression of true concurrency which is not possible using an interleaving semantics.

1.2 Coordinated Atomic Actions

The Coordinated Atomic action [1, 2] concept was introduced as a unified approach for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object-oriented system. This paradigm provides a conceptual framework for dealing with both kinds of concurrency (cooperative and competitive) by extending and integrating two complementary concepts - conversations and transactions. CA actions have properties of both conversations and transactions. Conversations are used to control cooperative concurrency and to implement coordinated and disciplined error recovery while transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

Each CA action has a set of roles that are activated by action participants (external activities such as threads, processes) and which cooperate within the CA action scope. Logically, the action starts when all roles have been activated (though it is an implementation decision to use either a synchronous or an asynchronous entry protocol) and finishes when all of them reach the action end. The action can be completed either when no error has been detected, or after successful recovery, or when the recovery fails and a failure exception is propagated to the containing action.

External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among these actions and that any sequence of operations on these objects bracketed by the start and completion of a CA action has the ACID (atomicity, consistency, isolation and durability) properties with respect to other sequences. The execution of a CA action looks like an atomic transaction to the outside world. One of the ways to implement this is to use a separate transactional support that provides these properties. This support can offer the traditional transactional interface, i.e. operations start, abort and commit transactions that are called (either by the CA action support or by CA action participants) at the appropriate points during CA action execution.

The state of a CA action is represented by a set of local objects; each CA action (either the action support or the application code) deals with these objects to guarantee their state restoration if the action recovery is to be provided. Local objects are the main means for participants to interact and to coordinate their executions (although external objects can be used as well).

1.3 Formal Development Methodology

The proposed formal development methodology is illustrated with an example that consists of: (1) starting with a set of informal application requirements including validation objectives expressed by a set of desired properties; (2) building an initial CO-OPN/2 specification, **I**, of the application, based on the informal requirements that is abstract enough to be as independent as possible of implementation constraints; the initial specification must validate the desired properties; (3) performing two refinement steps **R1** and **R2**. Refinement **R1** provides CO-OPN/2 specifications of the CA action design of the application. Refinement **R2** provides CO-OPN/2 specifications close to the implementation, from which the Java implementation is derived. Refinement **R2** is as close as possible to the real Java implementation and uses the CO-OPN/2 specifications of some Java basic classes defined in [5].

The assessment of the methodology is currently being under work. We intend to use, in addition to the CO-OPN/2 language, a temporal logic. Desired properties will be expressed as temporal logic formulae, and proved over CO-OPN/2 specifications. 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. This paper does not explain the use of temporal logic for expressing and verifying properties.

This methodology is presented through a concrete example running on the Internet. The application is called DSGamma (for Distributed Gamma). It is based on the Gamma paradigm [7] (a programming paradigm based on chemical reaction concepts). A Gamma-style of computation is used to compute the sum of integers distributed in several multisets. The objective is to perform in a distributed way the addition of all the integers which are in a 'dynamic bag' of integers. The term dynamic means that new integers may be inserted by users into the bag during computation.

The plan of this paper is the following: firstly, we give an informal description of the requirements of the application to be developed. Secondly, we give an initial CO-OPN/2 formal specification **I** for the desired application. Thirdly, we explain the chosen CA action design, and formally express it by the means of refinement **R1**. Fourthly, refinement **R2** and the Java implementation are described. Finally, some properties are stated and an informal sketch of the proof is given for each specification.

2 Informal Requirements of the DSGamma System

The Gamma paradigm [7] advocates a style of programming that is based on chemical reactions. The Gamma paradigm consists of applying one or more chemical reactions on a multiset. A chemical reaction usually removes some values from the multiset, computes some results and inserts them into the multiset. 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.

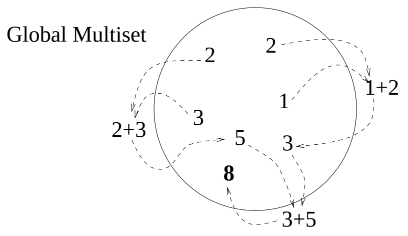

Figure 1: Addition according to the Gamma paradigm

2.1 Informal Requirements

We intend to develop an application allowing several users to insert integers into a possibly distributed multiset. According to the Gamma paradigm, chemical reactions are applied on the multiset, they have to perform the sum of all the integers entered by all the users. The system made of the users, a multiset and chemical reactions is called the DSGamma (Distributed Gamma) system. We present the informal requirements in two parts. The first part presents the system operations that must be provided to the users, and the second part presents the details of the data and of internal computations.

System Operations

(1) A new user can be added to the system at any moment; **(2)** A user may add new integers to the system, at any moment, between his entering time and his exit time; **(3)** A user may exit the system provided he has entered the system.

State and Internal Behavior

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

2.2 Motivations for DSGamma

The DSGamma system assumes that as many reactions as possible can be executed in parallel provided there are enough integers for them in the multiset and that the consistency of these data is hold. Several motivations can be given for distributing the multiset: (a) if the chemical reactions are much more costly than the message passing between computers, then their execution should be distributed; (b) in order to allow as many parallelization as possible in the system, the chemical reaction should be distributed; (c) if the multiset is huge, it makes sense to distribute it and to keep it as a set of local multisets and to distribute the chemical reactions as much as possible.

3 Initial CO-OPN/2 Specification I: Centralized View

The initial CO-OPN/2 specification **I** provides the most abstract specification 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 multiset, several processes (the chemical reactions), and each process, considered separately, is not distributed. The initial CO-OPN/2 specification **I** is given by the `DSGammaSystem` class depicted by Figure 2.

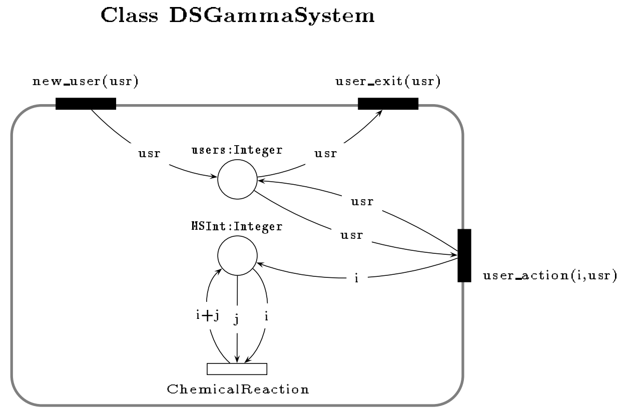

Figure 2: The Initial CO-OPN/2 specification, **I**

System Operations

The three CO-OPN/2 methods, `new_user(usr)`, `user_action(i,usr)`, and `user_exit(usr)` specify the three services, system operations **(1)** to **(3)**, that the system provides to the outside world. The `new_user(usr)` method inserts the user's 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, in the multiset `MSInt`. If the user `usr` has not yet entered the system, the method cannot be fired, thus the `i` value is not inserted in the multiset. If `usr` is in the `users` place, the `user_exit(usr)` method removes `usr`. `usr` and `i` are formal parameters that stand for algebraic terms, the parameter is instantiated when the method is called. The corresponding algebraic term is then inserted/removed into/from a place.

State

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

Internal Behavior

The CO-OPN/2 `ChemicalReaction` transition models the chemical reaction. It takes two integers `i,j` from the `MSInt` place, and inserts their sum `i+j` in `MSInt`. Due to the CO-OPN/2 semantics: (1) the `ChemicalReaction` transition can be fired several times simultaneously if they are sufficient pairs of integers in the `MSInt` place; (2) the transition continues to be fired until only one integer remains in the `MSInt` place.

4 Refinement R1: CA Action Design

The initial specification, **I**, provides a centralized view of the application. As we intend to obtain an implemented application in accordance with the CA action concept, refinement **R1** introduces both distribution (of data and behavior) and CA actions into the specification.

This section first presents informally the CA action design of the application, and then formally expresses it by means of CO-OPN/2 specifications.

4.1 Using CA actions to Design the DSGamma System

4.1.1 General Design

The system is composed of a set of participants (located on different hosts), a CA action scheduler (located on a separate computer) and a set of CA actions (see Figure 3). A participant starts when it is loaded into a client computer and establishes a connection with the CA action scheduler. A participant works on behalf of a user. Each participant has a local multiset, i.e. a queue in which some part of the global multiset is kept. The CA actions are activated dynamically to execute the Gamma computation. Each action has three roles: two producers (each of them provides a number) and a consumer that sums them up (the chemical reaction). The CA actions enclose

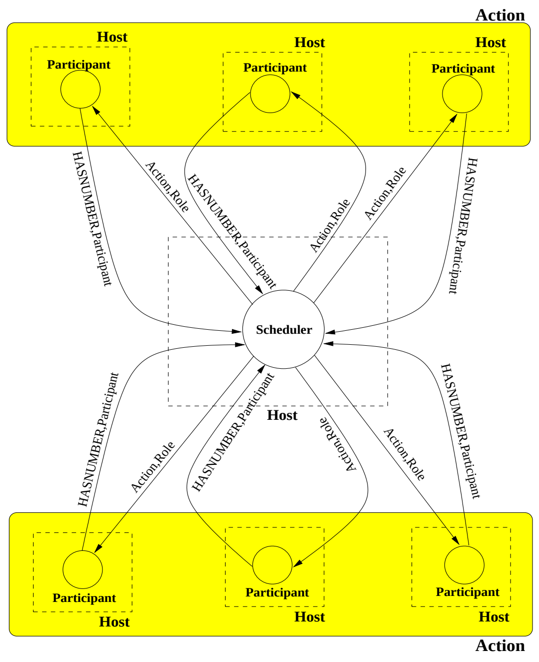

Figure 3: Gamma System

the interactions between participants on the level of the Gamma computation. The CA action scheduler receives information from all participants about any new number they have in their local queues and starts a new action with three roles when there are two new numbers in local multisets. There can be as many actions active concurrently as there are pairs in all local multisets at a given time (but some implementation reasons can restrict this approach). For example, it is allowed to have several active actions in which the same participant takes part (if there are several numbers in its local multiset). This allows a better parallelization of the Gamma computation.

Our system has two levels of design. The first level represents the information exchange between computers (participants and the CA action scheduler). This is the level on which the execution of the CA actions is scheduled (or the actions are glued together); it may well be designed using the CA actions but we have not done this. The second level of our design is the level of the Gamma computation, where the interactions between participants and the access to external objects are executed. On this level the numbers are passed between different local multisets and summed.

Depending on the hardware peculiarities or on some a priori knowledge about the application (e.g. frequency of users joining/leaving the system) other algorithms, for example, less centralized ones, can be used for designing the first level. For example, it would be possible to connect all hosts in a virtual ring or use broadcast to find matching participants ready for the chemical reaction. It was not our intention to investigate deeper this aspect of the problem because we decided not to assume any additional knowledge about the system. The purpose of our research is to design the DSGamma system using CA actions. The design of the second level is general enough to be used with any approach considered for the first level of the system.

4.1.2 Participants

We assume that each participant has two threads. The first thread, *TGetNumbers*, receives numbers typed by the user, and inserts these numbers into *ParticipantQueue*. After the number has been inserted into *ParticipantQueue*, the thread sends a message, *HASNUMBER*, to the CA action scheduler informing it that a new number has been typed and stored in *ParticipantQueue* (this means that the participant is ready to execute a role in an action). The second thread, *TExecuteActions*, receives messages from the CA action scheduler that contains a reference to an action that the participant must join, and the role that the participant must execute in that action.

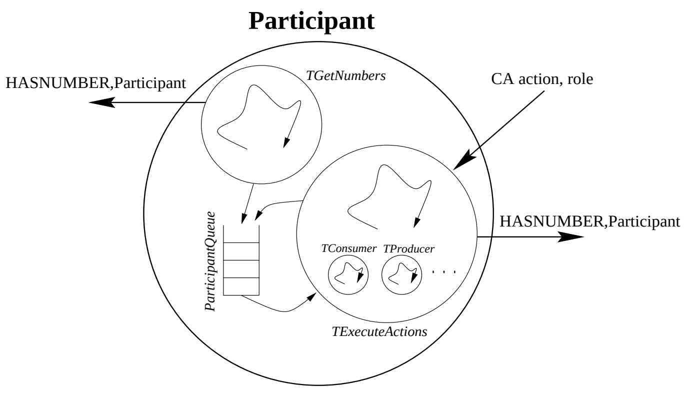

Figure 4: Participant

When *TExecuteActions* receives such message, it starts a new thread to execute the role in an action. This new thread is called *TConsumer*, if the participant has to execute the *Consumer* role in an action, or it is called *TProducer* if it has to execute *FirstProducer* or *SecondProducer* in an action. After *TConsumer* has finished the execution of its role inside the action, it will send a message, *HASNUMBER*, to the CA action scheduler signaling that another number has been inserted into *ParticipantQueue* (see Figure 4). *TConsumer* and *TProducer* threads are destroyed immediately after they have finished their role execution in an action (see Figure 5).

4.1.3 CA action Scheduler

There is a CA action scheduler (one for the entire system) that triggers the creation of all actions, and matches three participants to execute an action. It matches two participants that have numbers (and are ready to take part in an action) and a third participant that will act as the consumer of these two numbers (it sums them and puts the result into its *ParticipantQueue*). The CA action scheduler has the list of all participants, *ParticipantsList*. This list contains two items of information: the address of the participant and the quantity of numbers stored in its *ParticipantQueue*. When the CA action scheduler receives a message, *NEW*, it inserts a new participant into the list. When the CA action scheduler receives a message, *HASNUMBER*, it increases the quantity of numbers that this participant has in its *ParticipantQueue* (see Figure 6).

The CA action scheduler decreases by 1 the number of integers of a participant when this participant has been chosen to be a producer, so that the participant does not have to inform the CA action scheduler that it passes an integer to the consumer and that the producer's multiset has one number less when the action is over. The scheduler decreases this number in an optimistic way

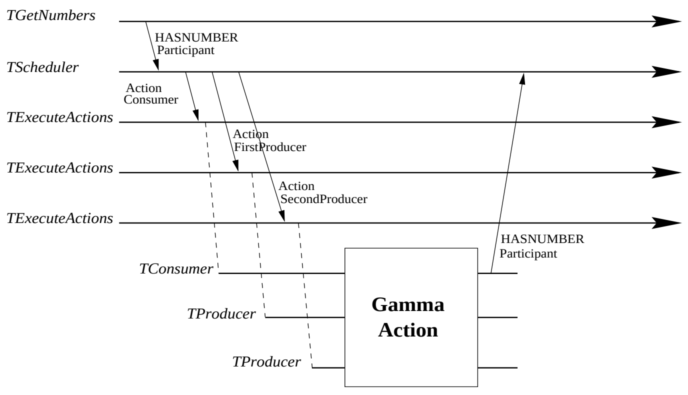

Figure 5: Creation/Destruction of Threads

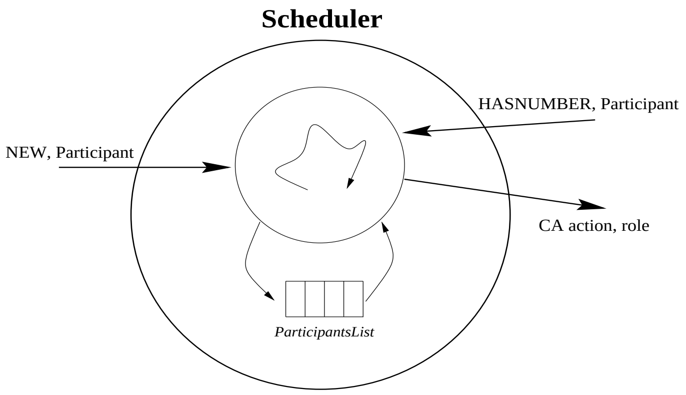

Figure 6: CA action Scheduler

when it chooses the producer and sends the action to it to participate. If a failure happens during the action execution, then the participant is responsible for recovery. This can be achieved by the participant inserting the number back into its multiset and sending a message to the scheduler saying that the participant has a new number.

The execution of the CA action scheduler consists of receiving those messages and of randomly choosing a consumer when it has two producers ready to take part in an action. It sends a message to each participant and tells it that it should take part in a particular action (the name is passed) either as a producer or as a consumer. The CA action scheduler uses the same list to choose two participants that have numbers to be summed (that means that they have numbers in their *ParticipantQueue*).

4.1.4 GammaAction

GammaAction is the CA action used to perform each step in the DSGamma chemical reaction. It has three roles: *FirstProducer*, *SecondProducer*, and *Consumer*. *Producers* take the numbers from their *ParticipantQueues* and send them to the *Consumer*. The *Consumer* sums the numbers and stores the result into its *ParticipantQueue* (see Figure 7). Local multisets *ParticipantQueue* are external objects in our design. They can be accessed only within CA actions. Their consistency and integrity is guaranteed by the CA action support in such a way that several actions can take numbers from the same multiset and add new numbers in it (after the Gamma reaction) without interference. Our particular implementation will use some simplified approach to provide this guarantee (e.g. locking one number but not the entire queue, etc.).

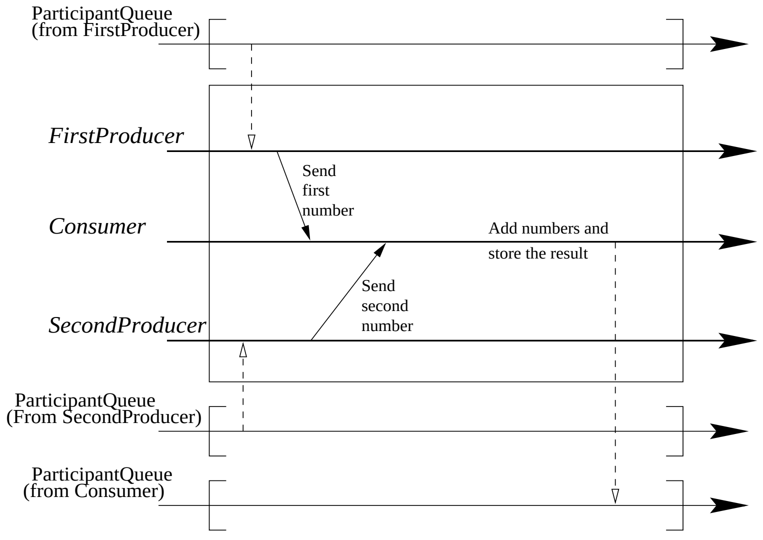

Figure 7: GammaAction

Many actions can be active at the same time in the system and each participant can be involved in several actions at once playing the roles of producer and/or consumer. The CA action scheduler creates an instance of a *GammaAction* whenever there are two new integers in the system and does not wait for this instance to finish execution before creating another if required.

The CA action scheduler involves participants in CA actions and triggers the creation of the instances of these actions. This design is very general, so we assume that there is a set of hosts on which the instances of the CA actions can be instantiated and that the scheduler knows their location. We will use a centralized CA action scheme, so each action has an action manager [3].

4.1.5 InsertNumberAction and FinishAction

Apart from *GammaAction*, we have introduced two other CA actions: *InsertNumberAction* and *FinishAction*. These actions are executed by the *TGetNumbers* thread. When a new number is entered, *TGetNumbers* executes a role inside *InsertNumberAction*. When the user wants to finish its participation in the Gamma computation *TGetNumbers* enters *FinishAction*.

InsertNumberAction has just one role and it is responsible for inserting the number into the *ParticipantQueue*. This CA action has the properties of a simple transaction. Inside this action the user enters a number that is passed to the local multiset. One action inserts one number.

FinishAction has two roles: the first one is executed by *TGetNumbers* thread and the second one is executed remotely by another participant. This participant is chosen randomly by the CA action scheduler from amongst all the participants that are present in the system (except for participants that are wanting to leave the system). *FinishAction* will transfer all numbers from the *ParticipantQueue* of the participant that wants to finish its execution, to the queue of another participant. When a participant decides to finish, it informs the scheduler of this, and the scheduler will choose another participant to execute a *FinishAction* together with the first one. When *FinishAction* is completed, the participant that has received new numbers informs the scheduler about new integers in its multiset. As we explained before, when a participant decides to finish it informs the scheduler, so it will not be selected by the scheduler to execute *GammaAction* again. Moreover the participant waits until all active *GammaActions* in which it is involved are completed and only afterwards it will enter into *FinishAction*.

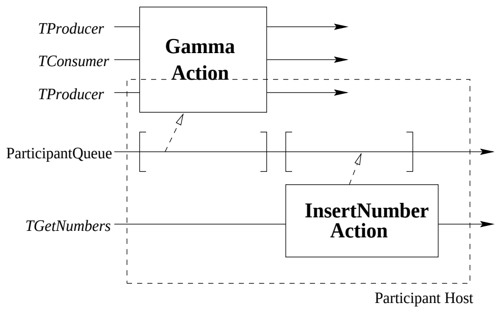

Figure 8: Competition for Accessing External Object

Figure 8 shows the competition between two actions for external *ParticipantQueue* object. Although in our implementation the access to these objects will overlap, at the logical level, access is serialized and consistency is guaranteed.

4.2 CO-OPN/2 Specification

We present now the CO-OPN/2 specifications of the CA action design presented above. The refinement process preserves the system operations. Thus, the overall specification of the DSGamma system provides the same three methods as the initial specification **I**. The internal behavior is specified by several classes, one for each item of the CA action design.

System Operation

Figure 9 gives a graphical representation of the CO-OPN/2 `DSGammaSystem` class, which specifies the overall DSGamma system.

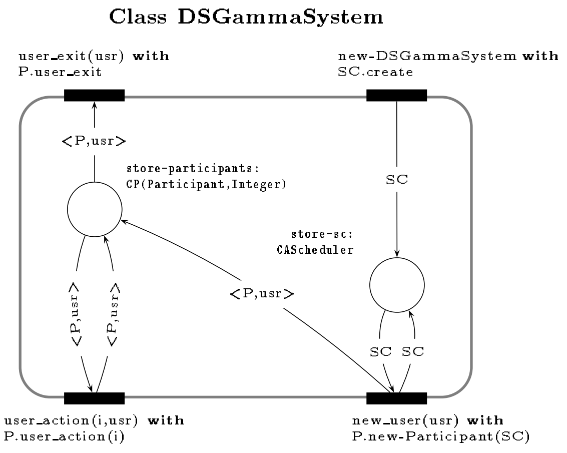
Class DSGammaSystem
user_exit(usr) with P.user_exit
new-DSGammaSystem with SC.create
<P,usr>
store-participants: CP(Participant,Integer)
SC
store-sc: CAScheduler
<P,usr>
<P,usr>
<P,usr>
SC SC
user_action(i,usr) with P.user_action(i)
new_user(usr) with P.new-Participant(SC)

Figure 9: Refinement **R1**: External Behavior

The CO-OPN/2 constructor `new-DSGammaSystem` requires that as soon as a DSGamma system exists, a CAAScheduler is created (calling `SC.create`), where `SC` is the identity of a CO-OPN/2 object of class `CAAScheduler`, and `create` is the default constructor. The `SC` reference is stored in the place `store-SC`. Only one `CAAScheduler`, `SC`, is created during the whole life of the `DSGammaSystem`. All the participants will work with the same `CAAScheduler`.

The `new_user(usr)` method implies the dynamic creation of a new participant `P` (calling `P.new-Participant(SC)`), where `P` is the identity of a CO-OPN/2 object of class `Participant`. The DSGamma system stores pairs of users and participants. Each new user is associated with its own participant, and it is the participant that actually creates the *ParticipantQueue*. The place `store-participants` stores pairs of participants and users. `CP(Participant,Integer)` stands for the Cartesian product of an object identity of class `Participant` and an `Integer`. The `user_action(i,usr)` method checks if the pair `<P,usr>` already exists and if so forwards the action to the participant `P` (calling `P.user_action(i)`). The `user_exit(usr)` method removes the pair `<P,usr>` from the place `store-participants` and forwards this information to participant `P` (calling `P.user_exit`).

State

The *ParticipantQueue* is given by the `Queue(Integer)` class (actually a FIFO of integers); it is created by the `Participant` (one per participant). The global multiset is given by the union of these participant queues.

Internal Behavior

The internal behavior is specified by: (1) the `Participant` class; (2) the `CAAScheduler` class; (3) by 4 thread classes: `TGetNumbers` class, `TExecuteActions` class, `TProducer` and `TConsumer` classes; (4) by 3 classes specifying the CA actions: `GammaAction` class, `InsertNumberAction` class, and `FinishAction` class; (5) the `Queue(Integer)` class that is used to specify both the participant queues (external object) and some local objects used by the CA actions; (6) the `EndChannel` class specifies one of the local objects used by the CA actions.

4.2.1 Participant

There is one object of class `Participant` per user. It creates the *ParticipantQueue* for that user. It handles messages incoming from the `CAAScheduler`. It creates an `InsertNumberAction` when a user inserts a new integer. It forwards to the `CAAScheduler` the information that the user wants to exit. It creates one `TGetNumbers` thread and one `TExecuteActions` thread. It informs all the necessary actions of the roles that are to enter into the action. It maintains the number of CA actions in which that participant is involved. Figure 10 gives the graphical representation of the CO-OPN/2 `Participant` class.

The arrival of a new user in the `DSGammaSystem` causes the creation of a new `Participant`. The CO-OPN/2 `new-Participant(SC)` constructor: (1) stores the `CAAScheduler` object's identity `SC`; (2) informs `SC` that there is a new participant (calling `SC.newParticipant(self)`); (3) creates the participant queue `Q` of class `Queue(Integer)` (calling `Q.create`); (4) creates and stores a CO-OPN/2 object, `TGN`, of class `TGetNumbers` (calling `TGN.new-TGetNumbers(SC,self,Q)`); (5) creates and stores a CO-OPN/2 object, `TEA`, of class `TExecuteActions` (calling `TEA.new-TExecuteActions(SC,self,Q)`). The constructor performs all these operations simultaneously.

The `DSGammaSystem` calls the `user_exit` method of the `Participant` in order to inform the `Participant` that the user wants to leave the system. The `Participant` just forwards this information to the `CAAScheduler` (calling `SC.endParticipant(self)`).

The `DSGammaSystem` calls the `user_action(i)` method of the `Participant` once the user enters integer `i` into the system. The `user_action(i)`: (1) creates a CA action, `INA`, of class `InsertNumberAction`; (2) increments by one the number of CA actions that the participant is involved in, the place `ToInc` receives a `1` token; (3) inserts the pair `<INA,Consumer>` into the place `ListOfINA`. This means that the participant has to provide a `Consumer` role for the `INA` action. As soon as a pair `<INA,Consumer>` is in the `ListOfINA` place, the `inINA` transition informs the `INA` action that the `TGN` thread will perform the role `Consumer` in that CA action (calling `INA.inAction(TGN,Consumer)`).

The `CAAScheduler` sends messages to the participant by the means of the `sendFinishAction(FA,R)` and the `sendGammaAction(GA,R)` methods. The participant has to provide a `R` role for the `FA` action of class `FinishAction` and a `R` role for the `GA` action of class `GammaAction`. These two

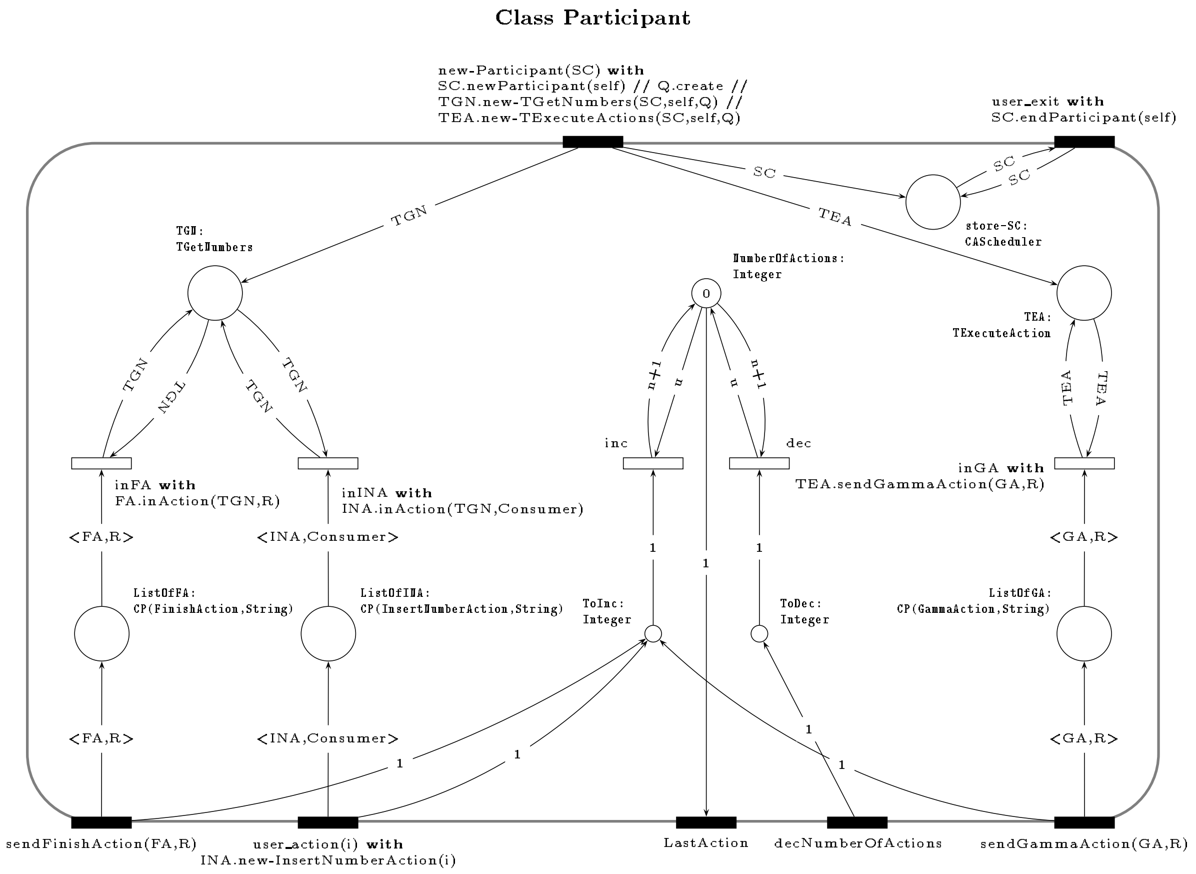

Figure 10: Refinement **R1**: `Participant` Class

methods increase by one the number of CA actions that the participant is involved in, and insert the pair `<FA,R>` respectively `<GA,R>` into the places `ListOfFA` respectively `ListOfGA`. As soon as a pair `<FA,R>` is in the `ListOfFA` place, the `inFA` transition informs the `FA` action that the `TGN` thread will perform the role `R` in that CA action. As soon as a pair `<GA,R>` is in the `ListOfGA` place, the `inGA` transition informs the `TEA` thread that it must provide a thread to perform role `R` in action `GA`.

The `NumberOfActions` place stores the total number of actions that the participant is involved in. As soon as the participant is informed that it has to participate for a role in a CA action it inserts `1` into the `ToInc` place. As soon as that role leaves a CA action it informs the participant by the means of the `decNumberActions` method. This method inserts `1` into the place `ToDec`. The two transitions, `dec` and `inc` respectively decrements and increments by one the total number of actions for each `1` token they found in the places `ToDec` and `ToInc` respectively. The `LastAction` method is called by the `Producer` role that has to enter into a `FinishAction`. That role enters into the `FinishAction` only if `LastAction` finds `1` as the number of actions remaining to be done by the participant. This ensures that all the other actions involving that participant are finished.

4.2.2 CAAScheduler

The `CAAScheduler`, given by Figure 11, maintains the *ParticipantList* as a multiset of pairs `<P,k>`, where `P` is the object's identity of the participant and `k` is the current number of integers present in the *ParticipantQueue* of `P`. The place `ParticipantList` stores these pairs.

A participant `P` calls the `newParticipant(P)` method of the `CAAScheduler` to inform the `CAAScheduler` that it exists. The `newParticipant(P)` method inserts a pair `<P,0>` into the place `ParticipantList`.

Class CAAScheduler

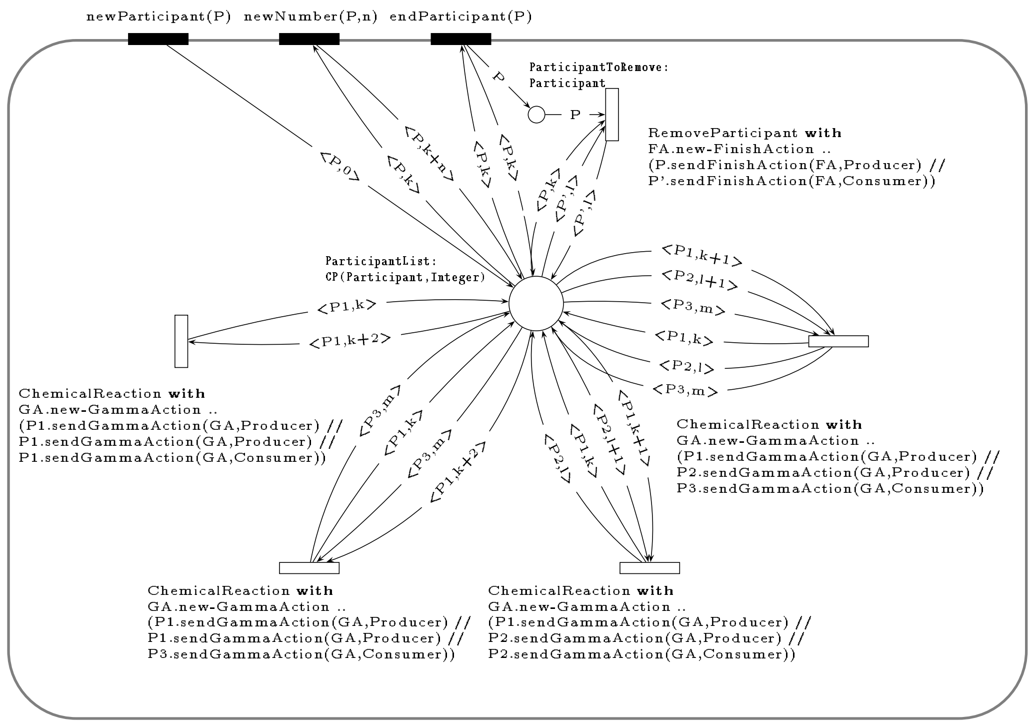
newParticipant(P)
newNumber(P,n)
endParticipant(P)
ParticipantToRemove: Participant
P
RemoveParticipant with FA.new-FinishAction .. (P.sendFinishAction(FA,Producer) // P'.sendFinishAction(FA,Consumer))
<P,0>
<P,k>
<P,k+n>
<P,k>
<P,k>
<P,k>
<P',l>
<P',l>
ParticipantList: CP(Participant,Integer)
<P1,k>
<P1,k+2>
<P1,k+1>
<P2,l+1>
<P3,m>
<P1,k>
<P2,l>
<P3,m>
ChemicalReaction with GA.new-GammaAction .. (P1.sendGammaAction(GA,Producer) // P1.sendGammaAction(GA,Producer) // P1.sendGammaAction(GA,Consumer))
<P3,m>
<P1,k>
<P3,m>
<P1,k+2>
<P2,l>
<P1,k>
<P2,l+1>
<P1,k+1>
ChemicalReaction with GA.new-GammaAction .. (P1.sendGammaAction(GA,Producer) // P2.sendGammaAction(GA,Producer) // P3.sendGammaAction(GA,Consumer))
ChemicalReaction with GA.new-GammaAction .. (P1.sendGammaAction(GA,Producer) // P1.sendGammaAction(GA,Producer) // P3.sendGammaAction(GA,Consumer))
ChemicalReaction with GA.new-GammaAction .. (P1.sendGammaAction(GA,Producer) // P2.sendGammaAction(GA,Producer) // P2.sendGammaAction(GA,Consumer))

Figure 11: Refinement **R1**: `CAAScheduler` Class

At the end of a CA action, the `Consumer` role informs the `CAAScheduler` that `n` new numbers have been added to the `P` participant queue. It calls the `newNumber(P,n)` method, that updates the participant list.

Once a user wants to exit, the corresponding participant informs the `CAAScheduler` calling the `endParticipant(P)` method. This method checks if the participant `P` is already present in the participant list, and simply adds the participant's identity `P` to the place `ParticipantToRemove`.

For each participant `P` present in the `ParticipantToRemove` place, the `RemoveParticipant` transition removes the pair `<P,k>` and searches for a pair `<P',l>` in the `ParticipantList`. It then creates a `FinishAction`, `FA`, and informs the participant `P` that it must enter into the `FA` action with a `Producer` role, and the participant `P'` that it must enter into the `FA` action with a `Consumer` role. Participant `P` will be no longer chosen by the `CAAScheduler` to participate in a CA action.

The `ChemicalReaction` transition is responsible to find, on the basis of the `ParticipantList`, three participants (two producers and one consumer), to create a `GammaAction` and to inform that participants.

The `ChemicalReaction` transition has four possible behaviors: (1) the same participant `P1` is chosen to be `Producer` twice and `Consumer`; (2) a participant `P1` is chosen to be `Producer` twice, and a participant `P3` is chosen to be the `Consumer`; (3) a participant `P1` is chosen to be one of the `Producer`s, a participant `P2` is chosen to be both the other `Producer` and the `Consumer`; (4) three different participants, `P1`, `P2`, `P3`, are chosen for each role. A `Producer` participant may be chosen *once* if it has at least one integer in its participant queue, or *twice* if it has at least two integers in its participant queue.

The `ChemicalReaction` transition immediately updates the `ParticipantList` for the `Producer` participant, but not for the `Consumer` participants. The `ChemicalReaction` transition decrements by one the number of integers present in the participant queue of `P` if it has been chosen to be

`Producer` once, and by two if `P` has been chosen to be `Producer` twice.

4.2.3 TGetNumbers

The `TGetNumbers` class, given by Figure 12, is able to perform, once a time, a role in several CA actions. It can enter into an `InsertNumberAction` with the `Consumer` role, or in a `FinishAction` action with the `Consumer` or with the `Producer` role.

Class TGetNumbers

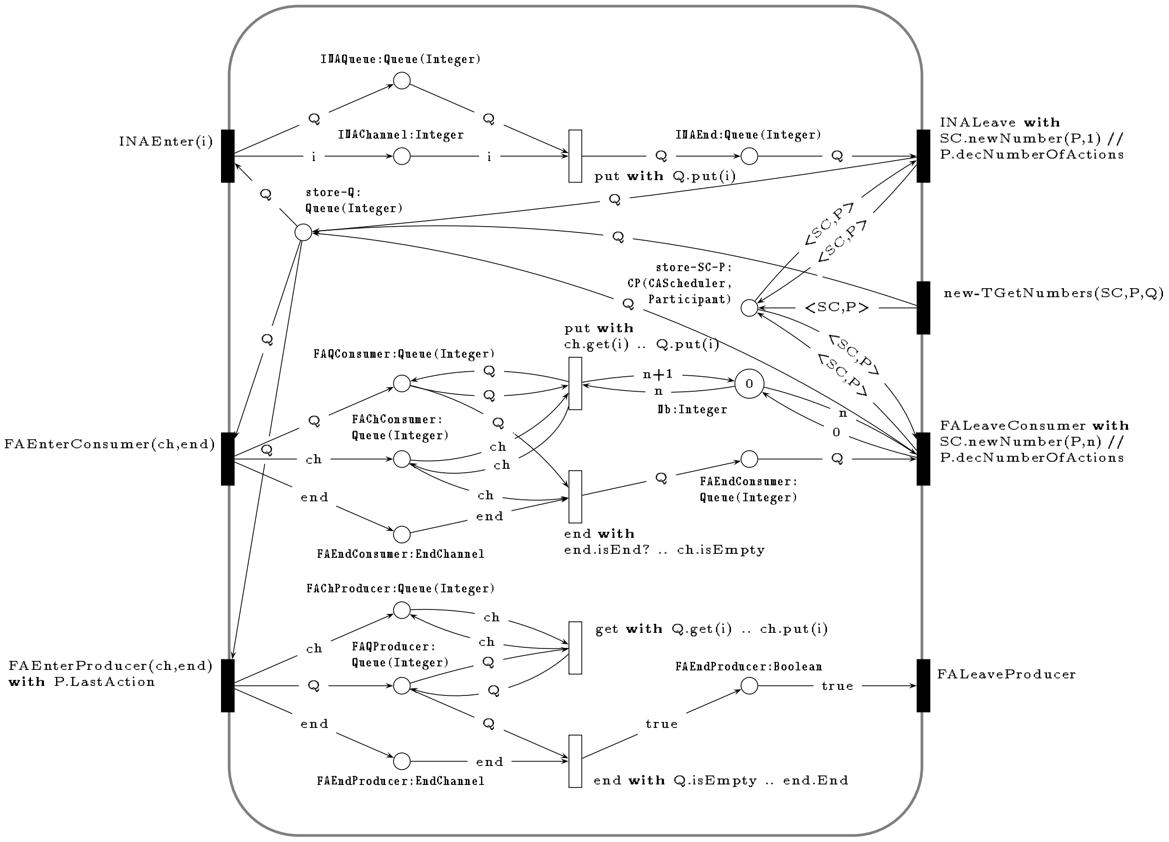
INAEnter(i)
INAQueue:Queue(Integer)
INAChannel:Integer
INAEnd:Queue(Integer)
put with Q.put(i)
INALeave with SC.newNumber(P,1) // P.decNumberOfActions
store-Q: Queue(Integer)
new-TGetNumbers(SC,P,Q)
store-SC-P: CP(CAScheduler, Participant)
<SC,P>
put with ch.get(i) .. Q.put(i)
FAQConsumer:Queue(Integer)
FAChConsumer: Queue(Integer)
Nb:Integer
n+1
FAEnterConsumer(ch,end)
FALeaveConsumer with SC.newNumber(P,n) // P.decNumberOfActions
FAEndConsumer: Queue(Integer)
FAEndConsumer:EndChannel
end with end.isEnd? .. ch.isEmpty
FAChProducer:Queue(Integer)
FAQProducer: Queue(Integer)
get with Q.get(i) .. ch.put(i)
FAEnterProducer(ch,end) with P.LastAction
FAEndProducer:Boolean
FALeaveProducer
FAEndProducer:EndChannel
end with Q.isEmpty .. end.End

Figure 12: Refinement **R1**: `TGetNumbers` Class

The CO-OPN/2 constructor `new-TGetNumbers(SC,P,Q)` stores the `CAAScheduler`, `SC`, identity, the participant `P` identity and the participant queue `Q` identity.

An `InsertNumberAction` calls the `INAEnter(i)` method in order to let the thread enter its role in the action. The `INAEnter(i)` method is firable if the `TGetNumbers` thread is not currently involved in another CA action. The firing of this method moves the object `Q` (from the `store-Q` place to the `INAQueue` place, and stores the local object `i`. The `put` transition is then firable, it inserts `i` into the participant queue `Q` (calling `Q.put(i)`). It enables the `INALeave` method by inserting the `true` token into the `INAEnd` place. The `INALeave` method is called by the `InsertNumberAction` in order to let the role leave the action. The role leaves the action provided it has ended its work, i.e. the value `Q` is present in the `INAEnd` place. This method causes the role to inform the `CAAScheduler` that one new number has been added to the participant queue `Q` (calling `SC.newNumber(P,1)`), and to inform the participant that a role in an action has been performed, and thus the number of actions involving that participant has to be decreased by one (calling `P.decNumberActions`). Finally, this method enables the thread to perform a new role in another action, by inserting the `Q` value into the `store-Q` place.

A `FinishAction` calls the `FAEnterConsumer(ch,end)` method in order to let the thread enter the `Consumer` role into the action. The firing of this method stores the two local objects (channels) `ch` and `end` in two separate places. Channel `ch` stores the integers incoming from the `Producer` role and that have to be inserted into the `Q` queue, and channel `end` is used to know when the `Producer` role has finished to give all the integers of its participant queue. The `put` transition is then firable. This transition removes each integer `i` from the channel `ch` and then inserts `i` into the participant queue `Q` (calling `ch.get(i) .. Q.put(i)`), and for each `i` it increments a counter `Nb`. The `end` transition can be fired only when the `Producer` role has finished to furnish integers, and when the `ch` channel is empty, (calling `end.isEnd? .. ch.isEmpty`). The `FALeaveConsumer` method is called by the `FinishAction` in order to let the role leave the action. This method causes the role to inform the `CAAScheduler` that `n` new numbers have been added to the participant queue `Q` (calling `SC.newNumber(P,n)`). The number of new integers is found in place `Nb`. This method also informs the participant that a role in an action has been performed, and thus the number of actions involving that participant has to be decreased by one.

A `FinishAction` calls the `FAEnterProducer(ch,end)` method in order to let the thread enter its `Producer` role into the action. The `FAEnterProducer(ch,end)` method is firable if the thread is not currently involved in another CA action, *and* if all the other actions in which the participant was involved are finished (calling `P.LastAction`). As for the `Consumer`, channel `ch` stores the integers `i` that the `Producer` removes from the participant queue, and channel `end` is used to inform the `Consumer` role that the `Producer` has given all the integers of its participant queue. The `get` transition removes each integer `i` from the participant queue `Q` and then inserts `i` into the channel `ch` (calling `Q.get(i) .. ch.put(i)`). When the participant queue `Q` is empty, the `Producer` role enables the `end` channel (calling `Q.isEmpty .. end.End`). The transition then provides a `true` token in the `FAEndProducer` place. The `FALeaveProducer` method is called by the `FinishAction` in order to let the role leave the action. This method causes `TGetNumbers` to stop working anymore, as it does not insert the `Q` value into the `store-Q` place. Indeed, the `FinishAction` where the participant has to provide a `Producer` role has to be the last action of that participant.

4.2.4 TExecuteActions

The `TExecuteActions` class, given by Figure 13, receives from its participant the information that the participant has to enter into a `GammaAction` class. Depending on the role, `Producer` or `Consumer`, the `TExecuteActions` creates a `TProducer` or a `TConsumer` role respectively.

Class TExecuteActions

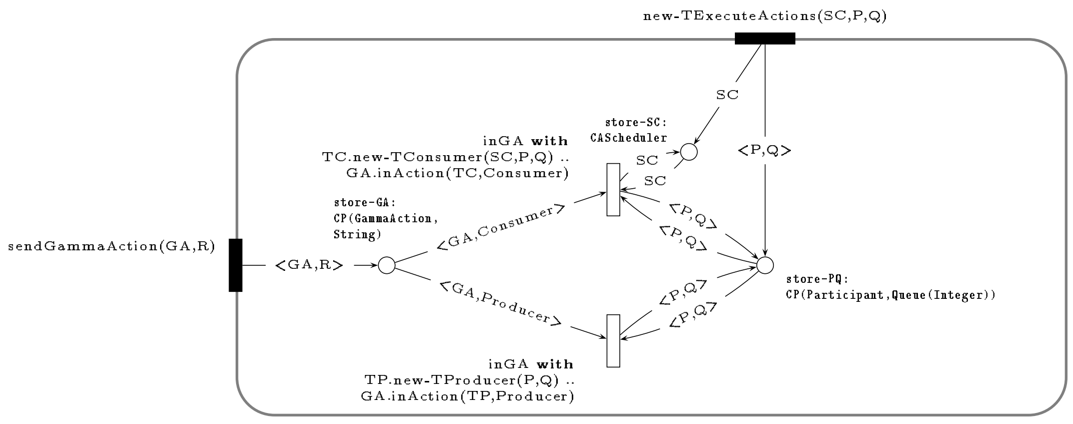
new-TExecuteActions(SC,P,Q)
SC
<P,Q>
store-SC: CAScheduler
SC
SC
inGA with TC.new-TConsumer(SC,P,Q) .. GA.inAction(TC,Consumer)
store-GA: CP(GammaAction, String)
<GA,Consumer>
<P,Q>
<P,Q>
sendGammaAction(GA,R)
<GA,R>
store-PQ: CP(Participant,Queue(Integer))
<GA,Producer>
<P,Q>
<P,Q>
inGA with TP.new-TProducer(P,Q) .. GA.inAction(TP,Producer)

Figure 13: Refinement **R1**: `TExecuteActions` Class

The `sendGammaAction(GA,R)` method is called by the participant in order to inform the `TExecuteActions` that it has to enter into the `GA` action with role `R`.

For each pair `<GA,R>` in the place `store-GA`, the `inGA` transition is fired: (1) it creates a `TConsumer` thread (calling `TC.new-TConsumer(SC,P,Q)`) and informs the `GA` action that this thread is ready to enter into the action (calling `GA.inAction(TC,Consumer)`), or it creates a `TProducer` thread (calling `TP.new-TProducer(P,Q)`) and informs the `GA` action that this thread is ready to enter into the action (calling `GA.inAction(TP,Producer)`).

4.2.5 TConsumer and TProducer

The `TConsumer` class, given by Figure 14, specifies the `Consumer` role of a `GammaAction`. The `Consumer` has to collect two integers from the two `Producer`s of the action, has to make their sum and to insert the sum into its participant queue. It receives the reference of the queue (at creation time), and collects integers from a channel provided by the action (received as a local object).

Class TConsumer

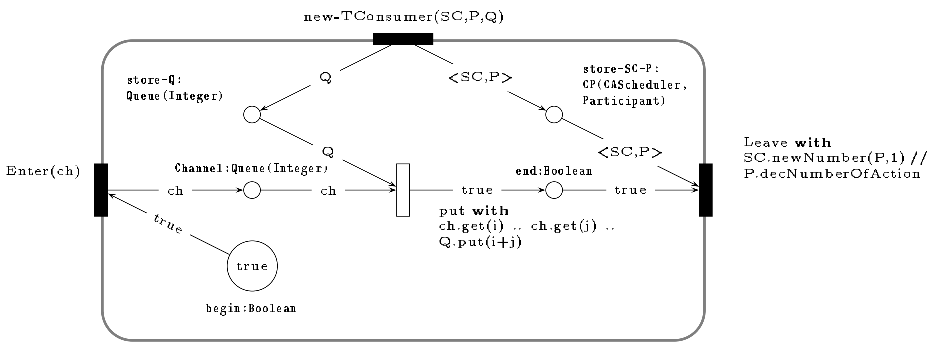

Figure 14: Refinement **R1**: `TConsumer` Class

A `GammaAction` calls the `Enter(ch)` method in order to enable the role to begin its execution. The `ch` object is the local object used to communicate with the `Producer` roles. The `Enter(ch)` method is firable only once (the token `true` is removed from the `begin` place and is never inserted into that place).

The `put` transition is then firable, it takes a first integer from the channel (calling `ch.get(i)`), it takes a second integer from the channel (calling `ch.get(j)`), it stores their sum into its participant queue (calling `Q.put(i+j)`), finally it enables the firing of the `Leave` method by inserting the `true` token into the `end` place.

The `Leave` method is called by the `GammaAction` in order to let the role leave the action. The `Leave` method informs the `CAAScheduler` that the participant has one new integer in its queue (calling `SC.newNumber(P,1)`), and informs the participant that the action is finished and the number of actions involving the participant has to be decremented by one (calling `P.decNumberOfAction`).

The `TProducer` class, given by Figure 15, specifies the `Producer` role of a `GammaAction`. The `Producer` has to remove one integer from its participant queue and sends it to the channel provided by the action (received as a local object).

A `GammaAction` calls the `Enter(ch)` method in order to enable the role to begin its execution. The `ch` object is the local object used to communicate with the `Consumer` role.

The `get` transition is then firable, it takes an integer from the participant queue and stores this integer in the channel (calling `Q.get(i) ..  ch.put(i)`), finally it enables the firing of the `Leave` method by inserting the `true` token into the `end` place.

The `Leave` method is called by the `GammaAction` in order to let the role leave the action. The `Leave` method informs the participant that the action is finished and the number of actions involving the participant has to be decremented by one.

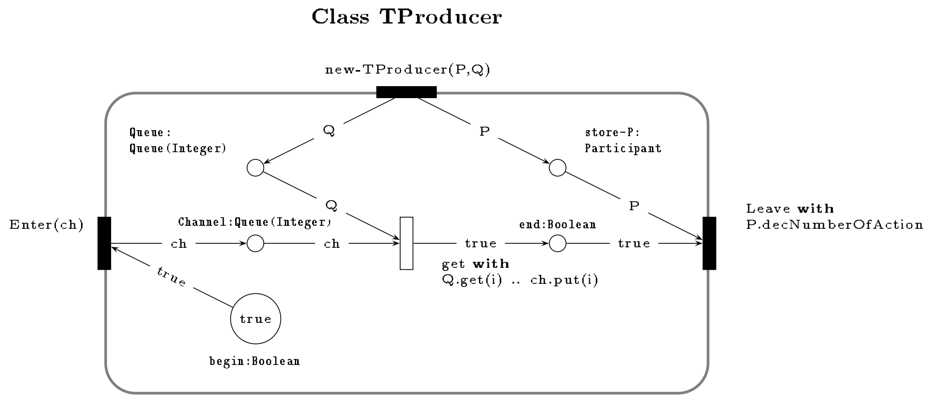

Figure 15: Refinement **R1**: `TProducer` Class

4.2.6 GammaAction, FinishAction, InsertNumberAction

The CO-OPN/2 classes specifying the CA actions are similar: an `inAction` method is used by the participant or the `TExecuteActions` to instruct the action about which thread will perform which role in the action. The action is actually performed by the `Action` transition that firstly calls all the roles in order to let them enter (calling `Enter`) into the action simultaneously and then sequentially calls all the roles in order to let them leave (calling `Leave`) the action simultaneously.

The call to the `Enter` method of a role causes that role to perform some work; the end of this work causes the enabling of the `Leave` methods. The roles work *between* the calls to `Enter` methods and the calls to the `Leave` methods. If the `Leave` method of one role cannot be fired, then the entire `Action` transition is not fired at all. The `Action` transition together with the specification of the participant queue ensures the specifications of the CA actions to have the ACID properties.

The CO-OPN/2 classes specifying the CA actions presented in this paper *do not* specify the effect of the CA actions on global objects, they specify *what* roles are in the CA actions and *how* the actions coordinate them. The effect of a CA action on global objects is derived from the specification of the CA action and by the specification of its roles.

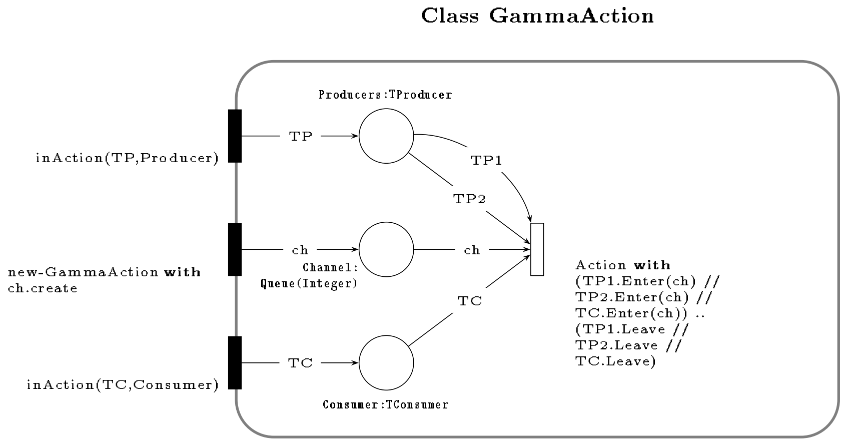

Figure 16: Refinement **R1**: `GammaAction` Class

The `GammaAction` class, given by Figure 16, is the action in which the Gamma chemical reactions are performed. The `new-GammaAction` constructor causes the creation of a channel `ch`, used as a local object. The channel is a queue of integers. The creation of the action is triggered by the `CAAScheduler`. The `TExecuteAction` calls the `inAction` method, either for announcing a `Consumer` role or a `Producer` role. The `Action` transition removes two producers from place `Producers`, and one consumer from the `Consumer` place. It calls the roles to enter into the action simultaneously (calling the `Enter` methods) and then calls them to leave the action simultaneously (calling the `Leave` methods).

Class FinishAction

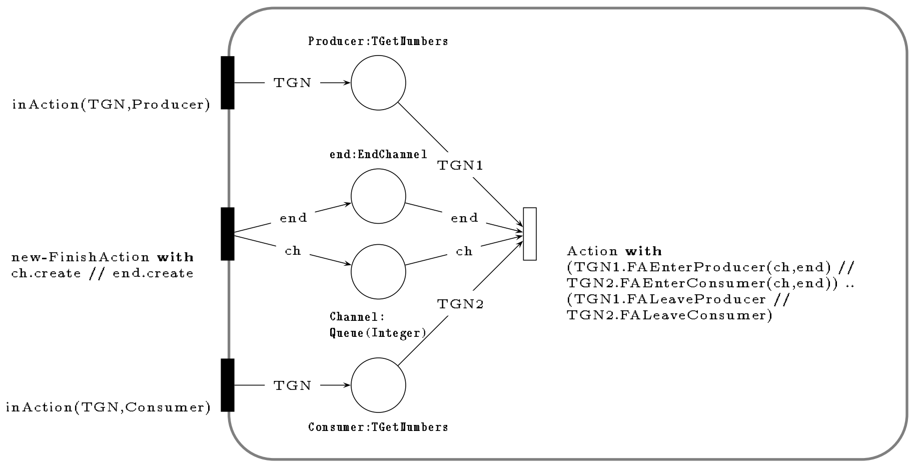

Figure 17: Refinement **R1**: `FinishAction` Class

The `FinishAction` class, given by Figure 17, is the action used to dispatch the participant queue, of a participant that wants to leave to system, to a participant queue of a participant that is still in the system. The `new-FinishAction` constructor causes the creation of two channels, `ch` and `end`, used as two local objects. This action has two roles, the `Producer` role and the `Consumer` role. The `FinishAction` is triggered by the `CAAScheduler`. The `TExecuteAction` calls the `inAction` method, either for announcing a `Consumer` role or a `Producer` role. The `Action` transition removes the producer and the consumer from their respective places. It then calls the roles to enter into the action simultaneously (calling the `FAEnterProducer` and `FAEnterConsumer` methods) and then calls them to leave the action simultaneously (calling the `FALeaveProducer` and `FALeaveConsumer` methods).

Class InsertNumberAction

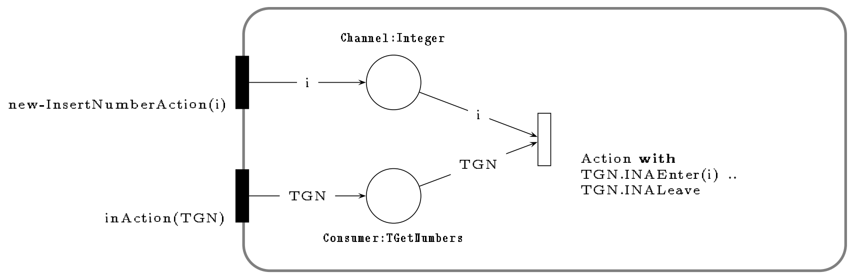

Figure 18: Refinement **R1**: `InsertNumberAction` Class

The `InsertNumberAction` class, given by Figure 18, is the action used to insert an integer (incoming from the user) into a participant queue. The `new-InsertNumberAction(i)` constructor just stores the integer `i`. The `InsertNumberAction` is created by the `Participant` itself. This action has one role, the `Consumer` role. The `Participant` calls the `inAction` method for announcing the `Consumer` role. The `Action` transition removes the consumer from place `Consumer`. It then calls the role to enter into the action (calling the `TGN.INAEnter(i)` method) and then calls it to leave the action (calling the `TGN.INALeave` method).

4.2.7 Queue(Integer) and EndChannel

The left part of Figure 19 depicts the participant queue. It is accessed by three methods: (1) the `put(i)` method is used to insert a new integer of value `i` at the end of the queue, (2) the `get(i)` method is used to remove integer `i` from the head of the queue; (3) the `isEmpty` method is used to know if the queue is empty or not. The place `Queue` contains one algebraic term, `q`, (of type `FIFO(Integer)`). The `get(i)` and `put(i)` methods cannot be fired simultaneously because each of them requires the algebraic term `q`. This property is the fundamental property that enables refinement `R1` to satisfy the ACID properties.

The right part of Figure 19 depicts the channel used by `FinishAction` to let the `Consumer` role wait upon the `Producer` role. The `End` method is used by the `Producer` role to let the `isEnd?` method become firable. The `Consumer` role waits on the `isEnd?` method to be firable. This method becomes firable as soon as the `Producer` role calls the `End` method.

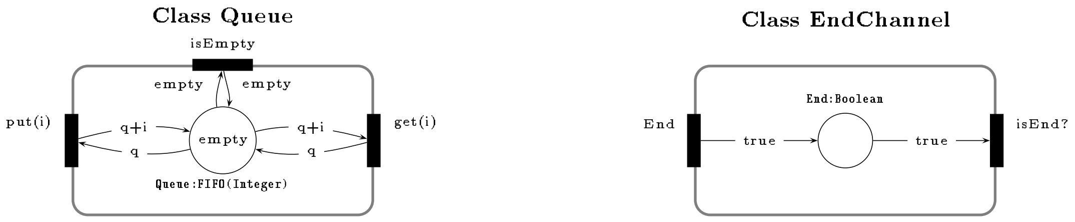

Figure 19: Refinement **R1**: `Queue(Integer)` and `EndChannel` Classes

5 Refinement R2: Java Implementation

We present now our Java implementation of the DSGamma system and give some hints on the CO-OPN/2 specifications of this implementation.

5.1 Java Implementation

We have implemented the DSGamma system using the Java programming language [8] and the Remote Method Invocation (RMI) API [9].

On the CA action scheduler side, we have implemented the scheduler as a remote object that can be accessed by the participants to inform the scheduler when they are joining the system, when they are willing to leave the system, and every time they got a new number in their local queue. The CA action scheduler object has the following interface:

```
public interface CAAScheduler extends java.rmi.Remote
{
public void newParticipant(Participant newPart) throws RemoteException;
public void newNumber (Participant newPart)     throws RemoteException;
public void endParticipant(Participant newPart) throws RemoteException;
}
```

`newParticipant` includes a new participant in the CAAScheduler list. The caller has to send a reference to its remote object that can be accessed by the CAAScheduler. `newNumber` is used by the participants to inform the CAAScheduler every time a new number is inserted into their local queue. When a participant has got numbers in its local queue, then the scheduler can select that participant to perform a role in a GammaAction as a producer; and, `endParticipant` is used by a participant to inform the scheduler that it is willing to finish its execution. The scheduler will choose another participant to get the numbers from the willing to finish participant.

The participants are implemented as remote applets that can be accessed by the CA action scheduler or by other participants. Each participant has a local queue (local object) that stores the numbers of its local multiset. This local queue implements its operations (`put, get`) using a monitor style approach (all methods are `synchronized` methods in Java). Each participant has also a list of GammaAction objects in which it always performs the `Consumer` role, i.e. when a CA action in that participant is activated, then the participant that contains that action will participate in the action as the `Consumer` (the CA action scheduler will set that). The `Participant` applet has the following interface:

```
public interface Participant extends java.rmi.Remote
{
boolean sendGammaAction(Participant where, String role, Integer caId)
                                             throws RemoteException;
boolean remoteGammaAction(String role, Participant part, Integer caId)
                                             throws RemoteException;
boolean sendFinishAction(Participant where, String role)
                                             throws RemoteException;
boolean remoteFinishAction(String role, Participant part)
                                             throws RemoteException;
void remoteQueuePut(int num) throws RemoteException;
int  remoteQueueGet()        throws RemoteException;
boolean remoteQueueIsEmpty() throws RemoteException;
}
```

`sendGammaAction` is used by the CA action scheduler to inform the participant that it has to execute the `role` in `where` (`caId` is an identifier of the action the participant has to execute, this caId guarantees that the right designated participants will execute the same CA action); `remoteGammaAction` is used by the other participants that want to execute a `role` in the action located in this participant (the willing to execute participant will send a reference to itself then the action can access its local queue); `sendFinishAction` is used by the CA action scheduler to inform the participant that it has to execute a finish action together with another participant; `remoteFinishAction` is used by another participant to execute the finish action in this participant; and `remoteQueuePut/remoteQueueGet/remoteIsEmpty` are used by a participant when executing the GammaAction remotely. It provides the access to the local queue of this participant.

Figure 20 represents the `GammaAction` object and the roles of this action. The CA action object is composed of a set of internal objects, used only by the roles of the action in order to exchange values, e.g. communicate; a set of external objects that the roles will access in an atomic way; a manager that is responsible for recovering the action from possible failures, and for pre-synchronizing and post-synchronizing the participants; and the roles that the participants will execute. In order to execute a role in an action, the participants must be informed of which action and role they have to execute, such information will be provided by the CA action scheduler using the `sendGammaAction` method of the participants. Once the participants have received the information about which action and role they have to execute, they activate the action by calling the `inAction` method in the action object sending information about their local queues. These local queues are external objects for the CA action manager, and they are bound to CA actions dynamically. The `inAction` method handles the tasks of the CA action manager as mentioned before.

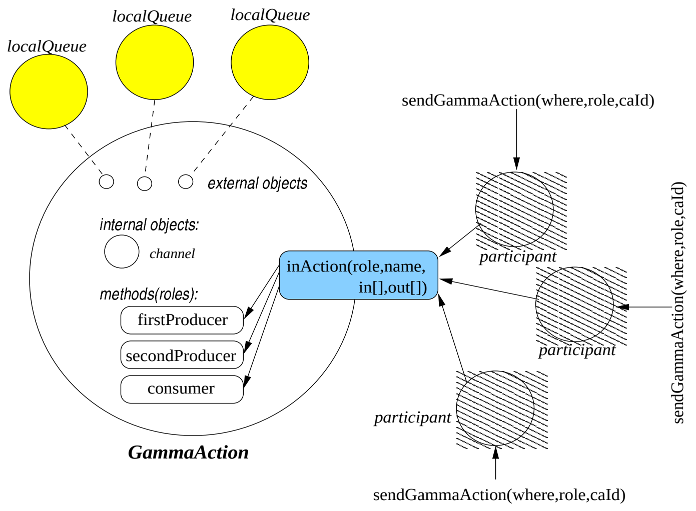

Figure 20: GammaAction Object

When implementing the CA actions, we have decided to attach the CA actions to the participants objects rather than to the scheduler. This prevents the CA action scheduler host from becoming overloaded with too many actions. All actions in our system are implemented as objects and the roles of such actions are implemented as methods of these objects (*action* is an *object*; *role* is a *method* of this object).

5.2 CO-OPN/2 Specification

Refinement **R2** leads to CO-OPN/2 specifications that take into account both the semantics of the Java programming language and the application's behavior (the implementation).

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 both expressing the Java semantics and the application's behavior. Indeed, a first layer of CO-OPN/2 specifications of Java basic classes is provided (as building blocks), and then the CO-OPN/2 specifications of the application are built on top of this layer.

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 with each object. We have specified the Java `Thread`, `Applet` and `Socket` classes. The complete CO-OPN/2 specification of these Java basic classes is given in [5].

Specifications of the Java program. Each Java class of the program is specified with a dedicated CO-OPN/2 class. These CO-OPN/2 classes are constructed using the CO-OPN/2 specifications of the Java basic classes, either by sub-classing them or by using them. Every data structure and algorithm used by the program is specified. In addition to the CO-OPN/2 specifications of the Java classes of the program, the overall system is fully specified with a CO-OPN/2 `DSGammaSystem` class in a similar way to refinement **R1**.

6 Validation of the DSGamma System

We present some properties and we informally show how they are validated by the DSGamma system at each step of the development process. In addition, we assume that there are no faults.

- **P1**: no number is lost in the computation process;
- **P2**: the sum of all numbers is correct;
- **P3**: the exit of a participant does not affect the sum of all numbers;
- **P4**: if participants stop inserting new numbers then the system execution eventually stops and only one number is left in the global multiset.

6.1 Initial Specification I

Property **P1** is true because the `ChemicalReaction` transition inserts the sum of the pair of integers it removes from the place `MSInt`. It is not possible for an integer to be removed without being added up to some other integer.

Property **P2** 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 **P3** is true because the sum is computed independently of the users.

Property **P4** is true because no integer is lost and because the `ChemicalReaction` transition is fired as long as there are at least two integers in the `MSInt` place.

6.2 Refinement R1

The informal proof that refinement **R1** validates the properties is based on several properties guaranteed by the CA actions.

Properties Guaranteed by CA actions

GammaAction has the following pre-condition: there are at least two numbers, $n_1$and n_2, in the global multiset. The post-condition is: those two numbers disappear from the multiset but there is a third new number equal to $n_1 + n_2$ that is inserted into the global multiset. The amount of numbers in the global multiset after the execution of the *GammaAction* is N-1, where N is the amount of numbers in the global multiset before the execution of the *GammaAction.*

FinishAction has the following pre- and post-conditions. Pre-condition: there is a local multiset in the participant that is willing to finish; post-conditions: all numbers from this local multiset are moved to another local multiset, the global multiset remains the same. *FinishAction* does not change either the amount of numbers in the global multiset, or the numbers themselves.

InsertNumberAction has the following pre- and post-conditions. A new number is inserted into the global multiset, all other numbers remain unchanged. The amount of numbers after the execution of the *InsertNumberAction* is N+1.

As we said CA actions have ACID properties, that is why concurrently executed actions are serializable in such a way that their effect on the multiset is equal to the effect of these actions executed in some sequential order (in which case the post-conditions of each of them are hold if the pre-conditions were hold when the action started). The CA actions support guarantees that the execution of each action is atomic, so the actions cannot interfere on each other. One more reason for this is that the action results are seen/assessable from the outside system only when the action is over.

Note that only these three actions operate with the global multiset. There is no other activity in the system which can access these data.

Validation of the DSGamma Expected Properties

Let us consider what can happen with the numbers in the global multiset. A number can disappear only when it has been summed by *GammaAction*. A new number can appear in the multiset when either *InsertNumberAction* is executed or *GammaAction* is executed (the sum of two numbers from the global multiset is a new number).

Let us consider what can happen with a number in a local multiset. A new number can appear in a local multiset only in the following situations: *i)* when *FinishAction* moves it from another local multiset; *ii)* when *InsertNumberAction* is executed in the participant that holds that local multiset; or, *iii)* when the new number is a sum inserted by a *GammaAction*. A number can disappear from a local multiset only in the following two events: when it is taken by a *GammaAction* to be summed, or when it is moved to another local multiset by *FinishAction*.

P1: if a number has been inserted into a local multiset, it can either be summed with another number in a *GammaAction*, or be moved to another local multiset by *FinishAction*.

P2: we do not lose numbers, that is why the sum of all numbers is correct; if new numbers are not inserted for some period of time then eventually all of them will be summed, because the scheduler will eventually be notified about each of them, and about each new number which is a sum inserted as a result of *GammaAction*. The number of the steps is finite (the amount of number decreases after each *GammaAction*). Another reason for the sum to be correct is that our considerations show that new numbers can be inserted into the multiset only by *GammaAction* (if no new numbers are inserted).

P3: the consideration above show that if a participant leaves the system, its local multiset is moved to another multiset. In our design this participant cannot be chosen for *GammaActions* or other *FinishActions* after it decides to leave and *FinishAction* starts (*FinishAction* only starts when the participant has no *GammaAction* to execute).

P4: the execution stops when there is no pair of numbers, to be more precise when there is only one number left in the global multiset. There is one and only one number when the systems stops, because of the conditions of the *GammaAction*: it always produces one new number from two existing ones from the global multiset (so, it cannot happen that none numbers left). This number is the sum of all numbers. Even though *GammaActions* can happen in parallel with *FinishActions*, the execution of *FinishAction* do not affect the execution of *GammaActions* because *FinishAction* does not change the amount of numbers in the global multiset, neither the numbers themselves.

6.3 Refinement R2

Refinement **R2** keeps the same CA action design than refinement **R1**. If the Java implementation of the CA actions provides the properties expected by the CA actions, then the same conclusions than refinement **R1** can be reached. Indeed, for any of the CA actions if the pre-conditions hold then the post-conditions hold also, after the action has been completed. Moreover, our implementation guarantees that if the pre-conditions are true then the action will be working with the same numbers that were checked in the pre-conditions and that the sequence <checking pre-conditions, execution CA action> cannot be pre-empted.

7 Conclusion

We have applied a top-down engineering methodology to develop a real application. Three development phases have been described: an abstract specification of the desired system, a CA action design, and the implementation. Each phase is formally specified by the means of the CO-OPN/2 language. Based on the CO-OPN/2 specifications and on the CA action design (after it has been formally described using CO-OPN/2), we have implemented the whole system using the Java language, an object-oriented language, that allowed us to build applications over the Internet, making the system very dynamic because several participants can join and leave the system at different times. The implementation of the DSGamma system was a clerical task due to the good mechanisms used to specify and design the system, i.e. CO-OPN/2 and CA actions, and due to re-use of CA action support system.

We think that the combined use of a CA action design and of CO-OPN/2 specifications will simplify the formal proofs that properties are validated by the system. Indeed, CO-OPN/2 specifications provide a mathematical framework, and each CA action provides its own set of properties. These properties are used to construct the proof that global properties are validated by the system.

Moreover, this paper shows how to use both the CO-OPN/2 specification language and the CA action concept and how they complement each other, particularly for validation and verification purposes. Moreover, this paper gives a complete CO-OPN/2 specification of an application developed according to the CA action design. It is a preliminary work towards the formalization of CA actions with CO-OPN/2 [10].

References

[1] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. http://www.cs.ncl.ac.uk/research/trs/papers/595.ps Computing Dept., University of Newcastle upon Tyne, TR 595, 1997.

[2] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing*, IEEE CS Press, Pasadena, USA, 1995, pp. 450-457.

[3] A. Romanovsky, B. Randell, R. Stroud, J. Xu, and A. Zorzo. "Implementation of Blocking Coordinated Atomic Actions Based on Forward Error Recovery". In *Journal of System Architecture - Special Issue on Dependable Systems.* July/97.

[4] 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.

[5] 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.

[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.

[7] Jean-Pierre Banâtre and Daniel Le Métayer, Gamma and the Chemical Reaction Model. In IC Press, editor, *Proceedings of the Coordination'95 Woorkshop*, 1995.

[8] The Java Language Specification. Sun Microsystems, Inc., 1996.

[9] Java Remote Method Invocation Specification. Sun Microsystems, Inc., 1996.

[10] Didier Buchs, Giovanna Di Marzo Serugendo, Nicolas Guelfi, Mathieu Buffo and Julie Vachon. Formal Specifications of Coordinated Atomic Actions using CO-OPN/2. Technical Report to appear, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1997.