CO-OPN/2 Specification of the DSGamma System Designed Using 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, NE1 7RU, UK

Abstract

The objectives of this paper are twofold. On the one hand, it aims to show the advantages of Coordinated Atomic actions (CA actions) as a design concept for dependable distributed system development, and on the other hand, it explains how the formal language CO-OPN/2 can be used to express the semantics of CA action design. A fault-tolerant 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 formally express the design phase. The implementation is made in Java based on a library of generic classes implementing the CA action concept. The paper is to serve as a basis for a more general approach aimed at defining CA action semantics.

Keywords: Structuring Complex Concurrent Systems, CO-OPN/2, Formal Development, Design for Validation, Coordinated Atomic Action, Gamma Computation, Java.

*Submitted to IEE Software Engineering Journal

†University of Newcastle, Uk, TR641, May 1998

1 Introduction

The engineering of dependable distributed systems should be based on a development methodology with a design phase relying on well-established design principles and precise enough for a serious verification phase, which is mandatory in this domain, to be defined. In this paper, a design model called Coordinated Atomic action is present through a case study.

The Coordinated Atomic (CA) action model [1] provides structuring primitives and support for error recovery for designing systems composed of several concurrent interacting entities. The model distinguishes between cooperative concurrency which is expressed using *conversations* [2] and competitive concurrency which is expressed using *transactions* [3]. The design of fault-tolerant distributed systems can be based on the use of exception handling mechanisms. The CA action model allows specific fault tolerance mechanisms to cover both hardware and software faults to be designed.

In this paper we propose using formal specification for two different reasons. First of all, we believe that the properties of the design principles we provide should be precisely and fully described. Secondly, we think that the following verification phase cannot be done without a precise understanding of the semantics of the designed system.

The formalism used for such purpose is Concurrent Object-Oriented Petri Nets (CO-OPN/2) [4]. CO-OPN/2 allows concurrent systems to be described in terms of structured Petri Nets for the behaviour part and algebraic specification for the data structures used to define values managed by the Petri nets. This formalism belongs to the class of object-oriented formal methods adapted to the specification of concurrent systems [5]. Some of the fundamental characteristics of CO-OPN/2, such as atomicity of complex synchronisation, concurrency and object structures, make it the right choice for the formalisation of CA actions.

On the one hand, the complete mathematical definition provided by CO-OPN/2 specifications gives precise semantics to the CA action design. On the other hand, the method for test selection that has been defined for CO-OPN/2 [6] provides test tasks for the verification phase. Furthermore, this paper also shows the advantages of CA actions for designing distributed systems as contrasted with the current use of CA actions which is more at the implementation level. In particular, we are trying to show that the CA action model provides an interesting way of designing the system structure on the earlier stages of the life cycle.

The design model and its semantics is present through a case study using CO-OPN/2. The case study is a development of a Gamma [7] system which computes a distributed sum of integers. "Distributed" means here that integers are distributed over several participant hosts, two summands for a partial sum are taken from some hosts, and the result is put randomly in one

of the hosts. Competition consists in accessing the values from the multiset of each participant while cooperation is used to make sure that the summed up integers are removed from the local multisets and that their sum is transferred to a third multiset. We assume that the process in charge of summing the integers can fail and raise an exception saying that the sum could not be completed. Thus, forward error recovery is used to take the system to a new coherent state. This state is different from the state reached if there have been no errors: both summands are moved from the local multisets to the sum multiset.

The structure of this paper is the following: Section 2 describes the CO-OPN/2 formal specifications language and explains the CA actions concept; Section 3 gives an informal description of the requirements of the case study to be developed; Section 4 explains the chosen CA action design, and formally expresses it by the means of CO-OPN/2 specifications; and, finally, Section 5 describes a Java implementation made according to the CA action design.

2 Background

2.1 CO-OPN/2

CO-OPN/2 is an object-oriented formal specifications language [4] that integrates Petri Nets used to describe concurrent behaviours and algebraic specifications [8] of structured data evolving in 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 an encapsulated algebraic net in which places compose the internal state and transitions model the concurrent events of the object. A place consists of a multiset of algebraic values. Transitions are divided into two groups: parameterised transitions, otherwise known as methods, and internal transitions. The former correspond to the services provided to the outside, while the latter describe the internal behaviours of an object. Unlike methods, internal transitions are invisible to the exterior world and spontaneous (they are fired as soon as their preconditions are satisfied). An object method can be fired only if no further internal transition can. A class describes all the components of a set of objects and is considered to be an object template. Thus, all the objects of one class have the same structure. Objects can be dynamically created. Objects are also called class instances. The usual dot notation for method invocations has been adopted.

Constructors. Class instances can be dynamically created. A pre-defined creation method

`create` is provided. Particular creation methods that create and initialise objects can also be defined. In all the specifications of this paper, we use the following convention: a creation method, which is not pre-defined, is called `new-classname`. For a creation method to be actually a constructor, it is necessary to declare it under the `Creation` field. Classes are normally used to dynamically create new instances, but it is also possible to declare static instances. Creation methods may be used only once for a given object.

Object Identity. Each class instance has an identity, which is also called an object identifier, that can 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 can be simulated. Thus, when an object requires a service, it asks to be synchronised with the method (parameterised transition) of the object provider. The synchronisation policy is expressed by means of a synchronisation expression, which can involve many partners joined by three synchronisation operators (one for simultaneity (`//`), one for sequence (`..`), and one for alternative or non-determinism (`+`)). For instance, an object may simultaneously request two different services of two different partners, followed by a service request to a third object.

Graphical Notation. CO-OPN/2 specifications are graphically noted in the following manner: a CO-OPN/2 class is depicted as a rectangle with a circle for each place inside, a white rectangle for each internal transition, and, on its sides, a black rectangle for each method. Labelled arrows between places and internal transitions or between places and methods give the flow relations (what is consumed and what is added to a place when an internal transition or a method is fired).

2.2 Coordinated Atomic Actions

The CA action [1, 9] concept was introduced as a unified approach for structuring complex concurrent activities and supporting error recovery of 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 [2] and transactions [3]. 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 [3] with respect to other sequences. To the outside world, the execution of a CA action looks like an atomic transaction. 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).

CA actions also provide a basic framework for exception handling that can support a variety of fault tolerance mechanisms aimed at tolerating both hardware and software faults. During the execution of a CA action, any of the roles of the action can raise an exception. If that exception cannot be dealt with locally by the role, then it must be propagated to the other roles in the CA action. Since it is possible for several roles to raise exceptions at more or less the same time, a process of exception resolution is necessary in order to agree on the exception to be propagated and handled within the CA action [10]. Once an agreed exception has been propagated to all of the roles involved in the CA action, then error recovery starts. It may still be possible to complete the execution of the CA action successfully using either forward or backward error recovery [11]. If it is not possible to achieve either a normal outcome or an exceptional outcome using these error recovery mechanisms, then the CA action should be aborted and its effects should be undone. Otherwise, a failure exception will be signalled to the external environment.

3 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 to 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 shows a multiset and a possible Gamma computation achieving result 8.

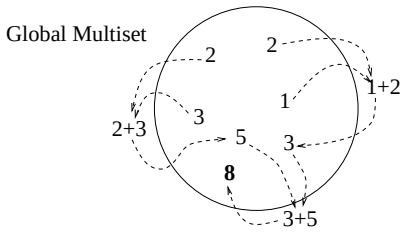

Figure 1: Addition according to the Gamma paradigm

3.1 Informal Requirements

We intend to develop an application allowing several users to insert integers into a possibly distributed multiset. Within the Gamma paradigm, chemical reactions are applied to the multiset; they have to sum all integers put in by all users. We call Distributed Gamma (DSGamma) system, the system composed of users, a distributed multiset and chemical reactions. The informal requirements are as follows: the first part describes the system operations to be provided to users, and the second part the details of 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/her entering and exit time; **(3)** A user may exit the system provided he/she has entered the system.

State and Internal Behaviour

(4) The integers put in by users are stored in a multiset; **(5)** The application computes the sum of all integers put in by all users; **(6)** The sum is calculated by chemical reactions in accordance with 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 in the multiset; **(9)** A chemical reaction may occur provided there are at least two integers in the multiset.

Fault Tolerance

The original Gamma paradigm [7] assumes that there are no faults in the system. We have chosen to include some fault tolerance in the system in order to demonstrate how CA actions can be used to increase dependability and to make the system more realistic. Our fault assumption is that only the addition operation can fail, and, because these operations are executed only inside CA actions, the entire system fault tolerance is provided within the CA action framework.

3.2 Reasons for Introducing DSGamma

The DSGamma paradigm assumes that reactions can be executed in parallel provided there are enough integers for them in the multiset and the data are consistent. Several reasons can be given for distributing the multiset: (a) if chemical reactions are much costlier than message passing between computers, then their execution should be distributed; (b) in order to allow as much parallelisation in the system as possible, 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 execution of chemical reactions as much as possible.

4 Using CA actions and CO-OPN/2 to Design DSGamma System

This section presents the CA action design of the DSGamma system whose requirements are given above. For each component of the CA action design, a CO-OPN/2 specification is provided. In this paper, we present some of these CO-OPN/2 classes; the complete set of CO-OPN/2 specifications together with some informal proof of properties can be found in [12].

4.1 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 2). 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.

There are three kinds of CA actions: *GammaActions*, *FinishActions* and *InsertNumberActions*. *GammaActions* are activated dynamically to execute the Gamma computation. Each *GammaAction* has three roles: two producers (each of which provides a number) and a consumer that sums

them up (the chemical reaction). *GammaActions* enclose the interactions between participants

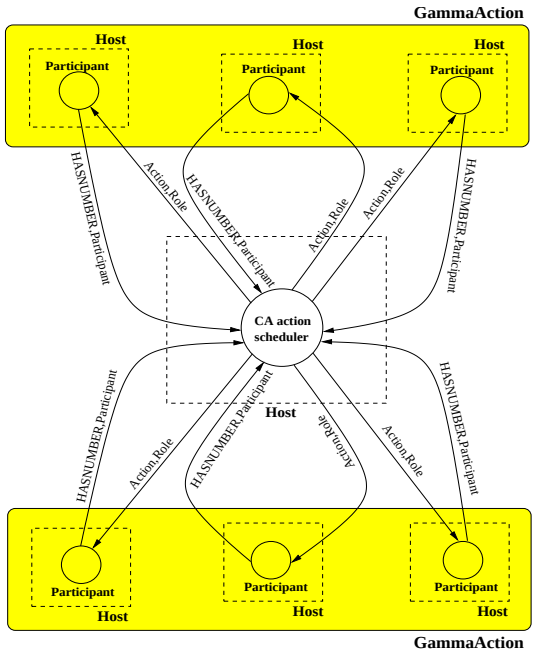

Figure 2: DSGamma System

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 *GammaAction* with three roles when there are two new numbers in local multisets. There can be as many *GammaActions* active concurrently as there are pairs in all local multisets at a given time (but some implementation factors can restrict this approach). For example, it is allowed to have several active *GammaActions* in which the same participant takes part (if there are several numbers in its local multiset). This allows a better parallelisation of the Gamma computation. The other two kinds of CA actions, *FinishActions* and *InsertNumberActions*, are used to enable a user to leave the system and to enable a user to enter a new integer in the system respectively.

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 CA actions is scheduled (or actions are glued together); it may well be designed using CA actions but we have not done this. We have chosen to use a CA action scheduler which creates new CA actions. 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 centralised

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 is not our intention to investigate this aspect of the problem any deeper because we have decided not to assume any additional knowledge about the system. The purpose of our research is to design a 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.

4.2 Participants

4.2.1 Informal Description

We assume that each participant has two threads. The first thread, *TGetNumbers*, receives numbers typed by the user, and inserts these numbers into *ParticipantQueue* by means of an *InsertNumberAction*. The participant starts a new *InsertNumberAction* each time the user types a new integer, and informs the *TGetNumbers* thread that it has to enter this action. 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 a *GammaAction*). The second thread, *TExecuteActions*, receives messages from the CA action scheduler that contains a reference to the *GammaAction* that the participant must join, and the role that the participant must execute in that action.

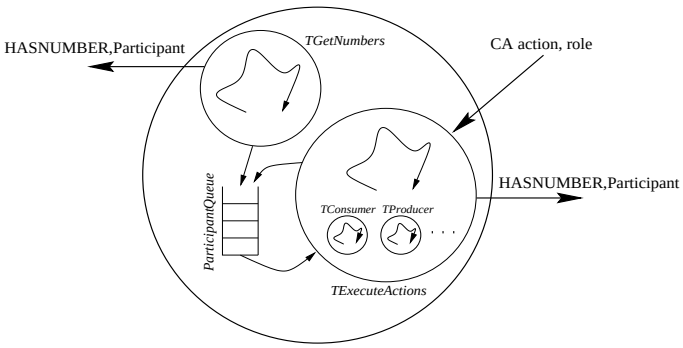

Figure 3: Participant

When *TExecuteActions* receives such message, it starts a new thread to execute the role in the action. This new thread is called *TConsumer* if the participant has to execute the *Consumer* role in the action, or *TProducer* if it has to execute the *FirstProducer* or *SecondProducer* role in the action. After *TConsumer* has finished the execution of its role inside the action, it will send a message, *HASNUMBER*, to the CA action scheduler signalling that another number has been

inserted into *ParticipantQueue* (see Figure 3). *TConsumer* and *TProducer* threads are destroyed immediately after they have finished their role execution in an action (see Figure 4).

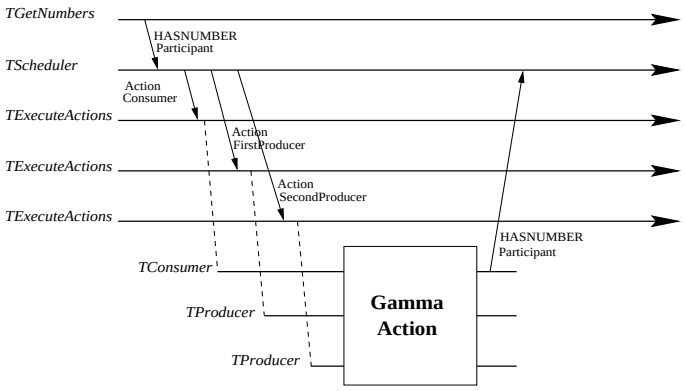

Figure 4: Creation/Destruction of Threads

If a user wants to leave the system, the participant informs the CA action scheduler, which will inform two participants (the participant that wants to leave the system and another participant) which *FinishAction* the *TGetNumbers* thread must join. *FinishAction* will transfer all numbers from the *ParticipantQueue* of the participant that wants to finish its execution to the queue of another participant. The participant waits until all active *GammaActions* in which it is involved have been completed and only afterwards it will enter *FinishAction*.

4.2.2 CO-OPN/2 Specification

The CO-OPN/2 `Participant` class, shown in Figure 5, is a specification of the participant of the CA action design.

The arrival of a new user in the DSGamma system causes the creation of a new instance of the `Participant` class. The CO-OPN/2 `new-Participant(SC)` constructor: stores the CA action scheduler identity `SC` (of class `CAAScheduler`); informs the CA action scheduler that there is a new participant (by calling `SC.newParticipant(self)`); creates the participant queue `Q` of class `Queue(Integer)` (by calling `Q.create`); creates and stores a CO-OPN/2 object identity, `TGN`, of class `TGetNumbers` (by calling `TGN.new-TGetNumbers(SC,self,Q)`); creates and stores a CO-OPN/2 object identity, `TEA`, of class `TExecuteActions` (by calling `TEA.new-TExecuteActions (SC,self,Q)`). The constructor performs all these operations simultaneously. The `Queue(Integer)` class specifies a FIFO queue of integers. `TGetNumbers` and `TExecuteActions` classes specify the *TGetNumbers* and *TExecuteActions* components of our design respectively.

The DSGamma system calls the `user_exit` method of a participant in order to inform the participant that the user wants to leave the system. The participant just forwards this information

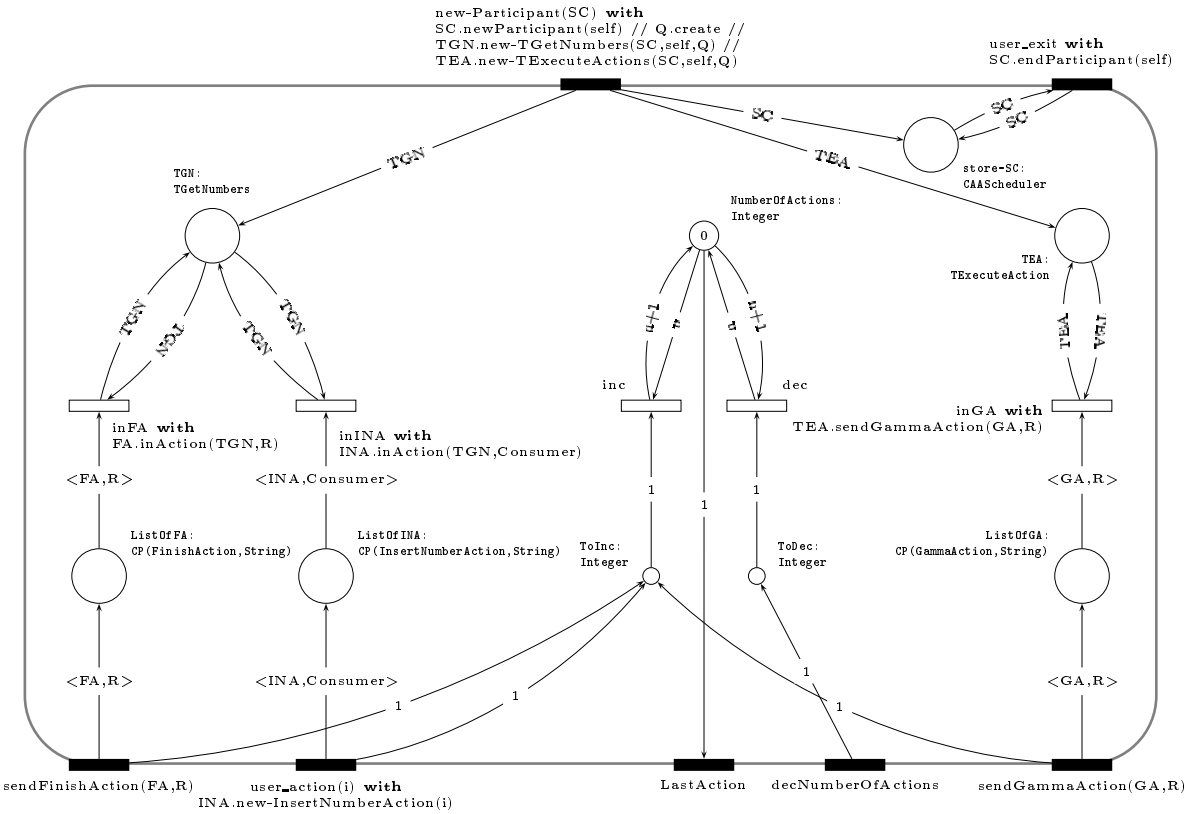

Figure 5: `Participant` Class

to the CA action scheduler (by calling `SC.endParticipant(self)`).

The DSGamma system calls the `user_action(i)` method once the user enters integer `i` into the system. The `user_action(i)`: creates a CA action, `INA`, of class `InsertNumberAction`; increments by one the number of CA actions in which the participant is involved, the place `ToInc` receives token `1`; and, 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 the 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 (by calling `INA.inAction(TGN,Consumer)`).

The CA action scheduler sends messages to the participant by means of two methods: `sendFinishAction(FA,R)` and `sendGammaAction(GA,R)`. The participant has to provide role `R` for the `FA` action of class `FinishAction` and role `R` for the `GA` action of class `GammaAction`. These two methods increase by one the number of CA actions that the participant is involved in, and insert the pairs `<FA,R>` and `<GA,R>` into the places `ListOfFA` and `ListOfGA` respectively. As soon as the pair `<FA,R>` is in the `ListOfFA` place, the `inFA` transition informs the `FA` action that the `TGN` thread will perform role `R` in that CA action. As soon as the 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 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 calling the decNumberActions method. This method inserts 1 into place ToDec. The two transitions, dec and inc, decrements and increments by one the total number of actions for each token 1 they find in places ToDec and ToInc respectively. The LastAction method is called by the Producer role that has to enter a FinishAction. That role enters FinishAction only if LastAction finds that there is just one action remaining to be executed by the participant. This ensures that all the other actions involving that participant are finished.

4.2.3 CO-OPN/2 Specification: TExecuteActions and TGetNumbers

The TExecuteActions class, shown in Figure 6, is a specification of the *TExecuteActions* thread. The creation of a new participant P causes the creation of a new instance of the TExecuteActions class. The new-TExecuteActions(SC,P,Q) constructor stores the CA action scheduler identity SC, the participant identity P, and the participant queue Q.

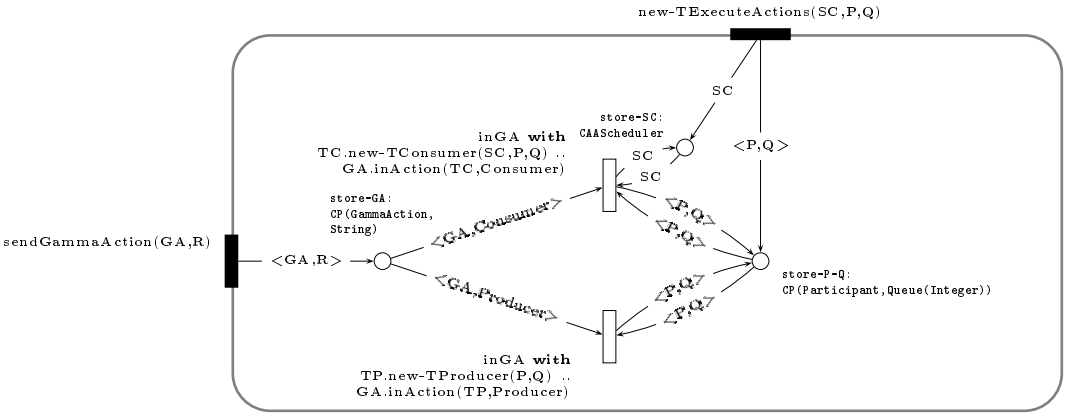

Figure 6: TExecuteActions Class

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

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

The CO-OPN/2 `TGetNumbers` class specifies the *TGetNumbers* thread. This thread can enter either *InsertNumberAction* or *FinishAction* and execute roles `Consumer` or `Producer`. When it executes a `Consumer` role in *InsertNumberAction*, it inserts a new integer into the *ParticipantQueue*. When it executes a `Producer` role in *FinishAction*, it removes all the integers from its *ParticipantQueue* and sends them to the `Consumer` role. When it executes a `Consumer` role in *FinishAction*, it inserts all the integers received from the `Producer` role into its *ParticipantQueue*.

4.3 CA Action Scheduler

4.3.1 Informal Description

There is a CA action scheduler (one for the entire system) that triggers the creation of all *GammaActions*, and picks three participants to execute an action. It picks 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 number of integers 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 number of integers that this participant has in its *ParticipantQueue* (see Figure 7).

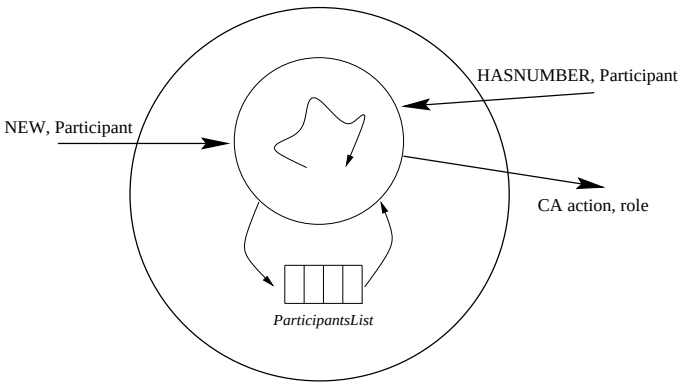

Figure 7: CA Action Scheduler

The CA action scheduler decreases the number of integers of a participant by one 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 multiset has one integer fewer when the action is over. The scheduler decreases this number in an optimistic way 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, i.e. they have numbers in their *ParticipantQueue*.

When a participant decides to finish, it informs the scheduler of this, and the scheduler chooses another participant to execute a *FinishAction* together with the first one.

4.3.2 CO-OPN/2 Specification

The CO-OPN/2 `CAAScheduler` class, shown in Figure 8, is a specification of the CA action scheduler. It maintains *ParticipantList* as a set of pairs `<P,k>`, where `P` is the object identity of the participant and `k` is the current number of integers present in the *ParticipantQueue* of `P`. The place `ParticipantList` stores these pairs.

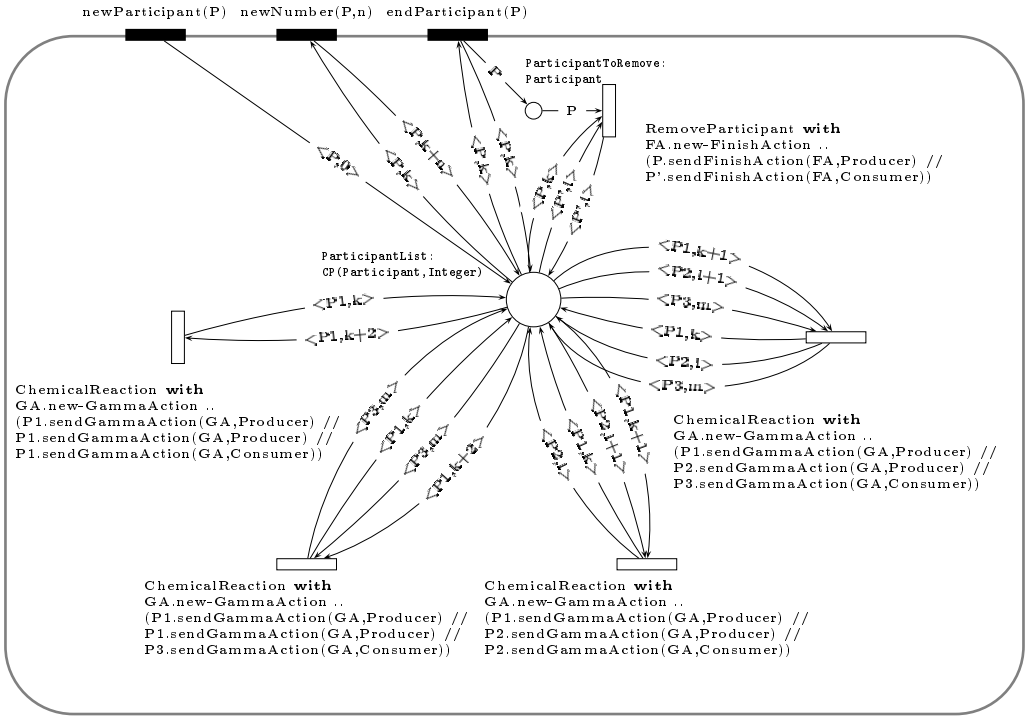

Figure 8: `CAAScheduler` Class

The DSGamma system creates exactly one instance of the `CAAScheduler` class.

Participant P calls the newParticipant(P) method of the CA action scheduler to inform the CA action scheduler that it has joined the system. The newParticipant(P) method inserts a pair <P,0> into place ParticipantList.

Participant P informs the CA action scheduler that n new numbers have been added to its queue. Participant P calls the newNumber(P,n) method, that updates the participant list.

As soon as a user wants to exit, the corresponding participant informs the CA action scheduler by calling the endParticipant(P) method. This method checks if participant P is already present in the participant list, and simply adds participant identity P to place ParticipantToRemove.

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

The ChemicalReaction transition finds, on the basis of ParticipantList, three participants (two producers and one consumer), creates a GammaAction, GA, and informs the participants.

The ChemicalReaction transition has four possible behaviours: the same participant P1 is chosen to be Producer twice and Consumer; participant P1 is chosen to be Producer twice, and participant P3 is chosen to be Consumer; participant P1 is chosen to be one of the Producers, participant P2 is chosen to be both the other Producer and Consumer; three different participants, P1, P2, P3, are chosen for each role. The two first behaviours are possible only if participant P1 has at least two integers in its participant queue.

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

4.4 CA Actions

4.4.1 Informal Description: GammaAction

GammaAction is a CA action used to perform a DSGamma chemical reaction. It has three roles: *FirstProducer*, *SecondProducer*, and *Consumer*. *FirstProducer* and *SecondProducer* take integers from their *ParticipantQueues* and send them to *Consumer*. *Consumer* sums the integers up and stores the result in its *ParticipantQueue* (see Figure 9). 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 integers from the same multiset and add new integers in it (during 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.).

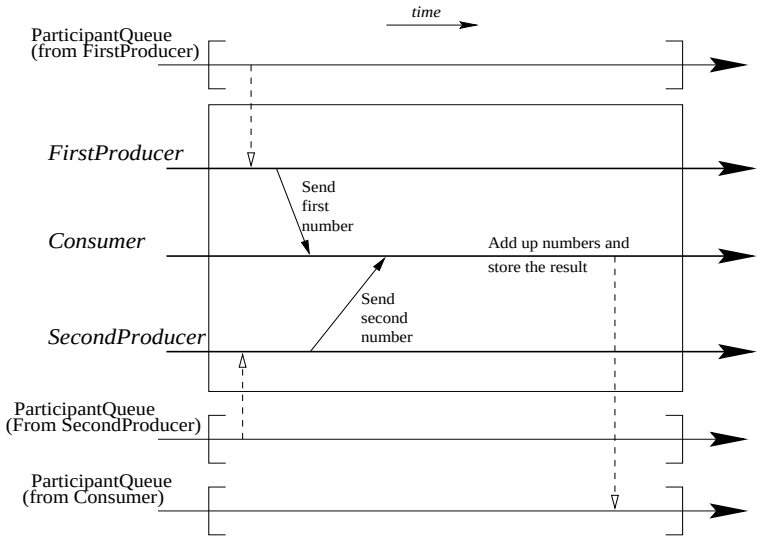

Figure 9: GammaAction

Many actions can be active in the system at the same time, and each participant can be involved in several actions at once playing the roles of producers and consumers. The CA action scheduler creates an instance of *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 CA actions can be instantiated, and that the scheduler knows their location. We use a centralised CA action scheme, so each action has an action manager [13].

As we have mentioned before, CA actions provide a framework for dealing with exceptions that happen during the execution of the system. In our DSGamma system, when a fault happens inside a *GammaAction*, a predefined exception `ReactionException` is raised in the thread executing a role in the action (see Figure 10).

In accordance with the CA action concept, we attach exception handlers to each role. After exception `ReactionException` has been raised, the CA action support interrupts all the roles in the action and calls the handler for this exception in each of them (see Figure 10). Our design decision is to use forward error recovery in the action in the following way: when the reaction fails, the consumer keeps both integers by inserting them into its local multiset whilst the producers

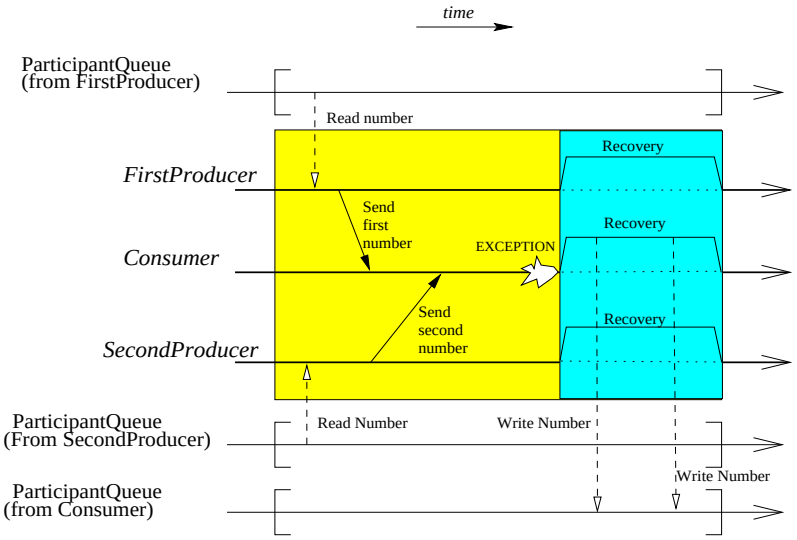

Figure 10: Forward Error Recovery in *GammaAction*

complete the action as if nothing had happened. Thus, if a fault happens during the action execution, the consumer recovers the system, but in this case two new integers appear in the consumer local multiset. We use a special outcome of action *GammaAction* to inform the scheduler about these new integers.

GammaActions are atomic with respect to faults in the chemical reaction: exception handlers guarantee "nothing" semantics for the global multiset (although local multisets are modified during this recovery).

4.4.2 CO-OPN/2 Specification

CO-OPN/2 classes specifying CA actions are all similar: an `inAction` method is used to instruct the action about which thread will perform which role in the action. The action is actually performed by the `Action` transition that first calls all the roles in order to let them enter (by calling `Enter`) the action simultaneously, and then sequentially calls all the roles in order to let them leave (by 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 the `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 that CA actions have the ACID properties. Indeed, CO-OPN/2 semantics ensures that either the `Action` transition together with its required synchronisations is completely fired, or the `Action` transition is not fired at all (and hence none of its required

synchronisations are).

CO-OPN/2 classes specifying the CA actions presented in this paper *do not* specify the effect of CA actions on global objects, they specify *what* roles are in 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 that of its roles.

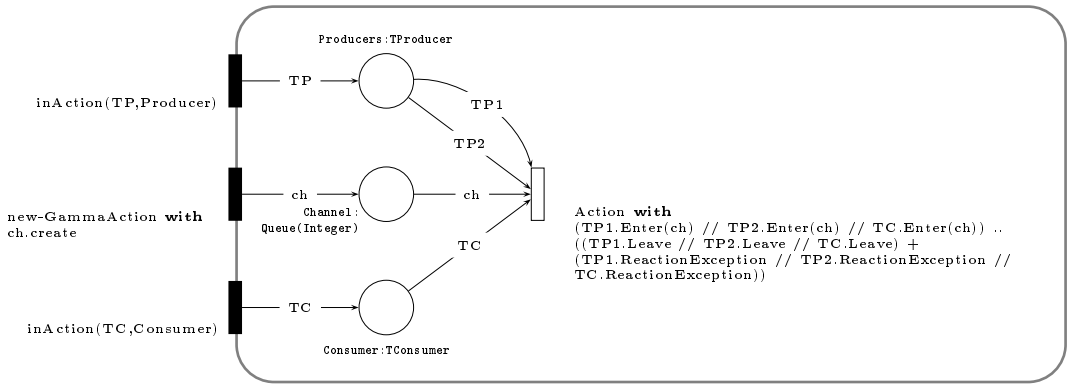

Figure 11: `GammaAction` Class

The `GammaAction` class, shown in Figure 11, specifies *GammaAction*. Gamma chemical reactions are performed in this action. 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 a new instance of the `GammaAction` class is triggered by the CA action scheduler. A `TExecuteActions` object 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 the action simultaneously (by calling the `Enter` methods) and then calls them to leave the action simultaneously (by calling the `Leave` methods). This action is also able to cope with one exception, `ReactionException`, incoming from roles. If `ReactionException` has been raised, the `Action` transition does not call the `Leave` methods of the roles but the `ReactionException` method. This will cause the exception handler of the roles to be activated.

4.4.3 CO-OPN/2 Specification: TConsumer and TProducer

The `TConsumer` class, shown in Figure 12, specifies the `Consumer` role of *GammaAction*. `TConsumer` has to collect two integers from a channel provided by the action (received as a local object), sum them up and insert the sum into its participant queue. It receives the reference of the queue (at creation time). Instances of the `TConsumer` class are created by `TExecuteActions` objects.

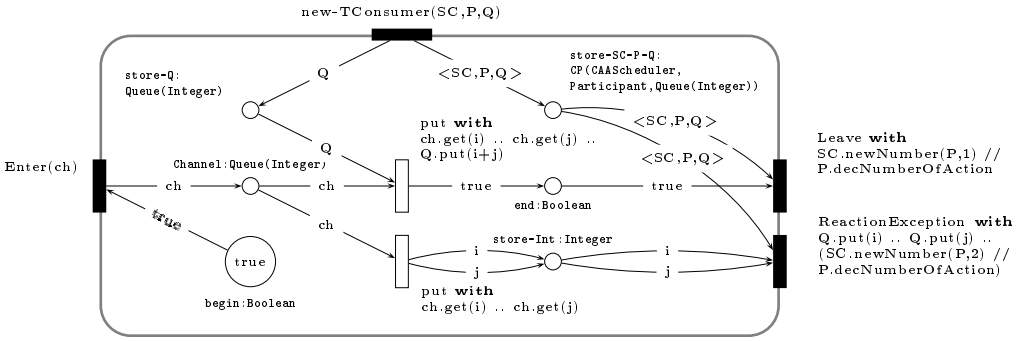

Figure 12: `TConsumer` Class

The `new-TConsumer(SC,P,Q)` constructor stores the CA action scheduler identity `SC`, the participant identity `P`, and the participant queue identity `Q`.

A `GammaAction`, `GA`, calls the `Enter(ch)` method in order to enable the role to begin its execution. The `ch` object is a 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. This transition has two possible behaviours: either it correctly does the sum, and enables the role to correctly end, or it does not do the sum and causes the role to raise `ReactionException`.

In the first case, the `put` transition takes a first integer from the channel (by calling `ch.get(i)`), a second integer from the channel (by calling `ch.get(j)`), and stores their sum into its participant queue (by calling `Q.put(i+j)`); finally it makes the firing of the `Leave` method possible by inserting the `true` token into the `end` place.

In the second case, the `put` transition takes a first integer from the channel (by calling `ch.get(i)`), a second integer from the channel (by calling `ch.get(j)`), and stores them into the `store-Int` place, making it possible to fire the `ReactionException` method.

Only one of these `put` can be fired at a time, and the choice between them is non-deterministic. Depending on which of them has been fired, either the `Leave` method or the `ReactionException` is firable. This will cause the `GammaAction GA` to call the firable one.

When no exception has occurred, the `Leave` method is called by `GammaAction GA` in order to let the role leave the action. The `Leave` method informs the CA action scheduler `SC` that the participant has one new integer in its queue (by 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 (by calling `P.decNumberOfAction`).

When an exception has occurred, the `ReactionException` method is called by `GammaAction GA`. This method performs the recovery of the error and enables the role to leave the action: it removes the two integers from the `store-Int` place and stores them, *without* adding them up, in the participant queue `Q`. After that it informs the CA action scheduler `SC` that the participant has *two* new integers in its queue(by calling `SC.newNumber(P,2)`), and decrements the number of actions the participant is involved in.

The `TProducer` class, shown in Figure 13, specifies the `Producer` role of *GammaAction*. The `TProducer` has to remove one integer from its participant queue and send it to the channel provided by the action (received as a local object). Instances of the `TProducer` class are created by `TExecuteActions` objects.

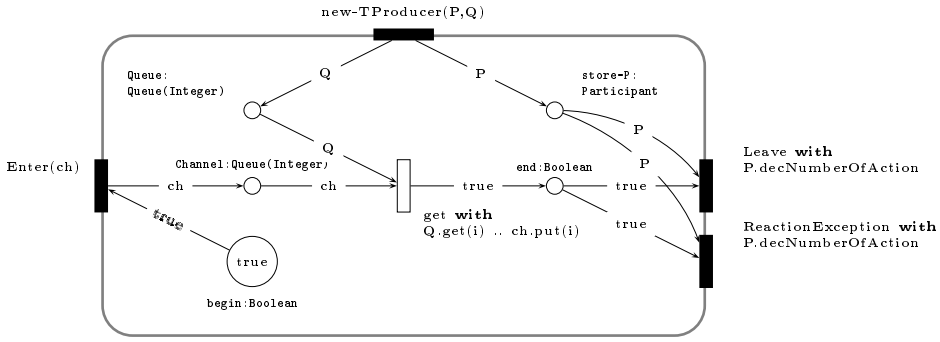

Figure 13: `TProducer` Class

The `new-TProducer(P,Q)` constructor stores the participant identity `P`, and the participant queue identity `Q`.

A `GammaAction`, `GA`, 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 it in the channel (by calling `Q.get(i) .. ch.put(i)`), finally it makes the firing of both `Leave` and `ReactionException` methods possible by inserting the `true` token into the `end` place.

When no exception has been raised by the `TConsumer` role, the `Leave` method is called by `GammaAction GA` 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.

When an exception has been raised by the `TConsumer` role, the `ReactionException` method is called by `GammaAction GA` in order to let the role perform some error recovery. In the case of the `TProducer` role, there is no need to perform error recovery, and the `ReactionException` method

behaves just like the `Leave` method.

4.4.4 Informal Description: 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 of *InsertNumberAction*. When the user wants to finish its participation in the Gamma computation, *TGetNumbers* enters *FinishAction*.

InsertNumberAction has just one role and is responsible for inserting the number into *ParticipantQueue*. This CA action has the properties of a simple transaction. Within 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 is executed by the *TGetNumbers* thread and the second 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 those that want 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 chooses another participant to execute a *FinishAction* together with the first one. When *FinishAction* is completed, the participant that has received new integers informs the scheduler about the new integers in its multiset. As we have 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 is delayed until all active *GammaActions* in which it is involved are completed and only afterwards it will enter *FinishAction*.

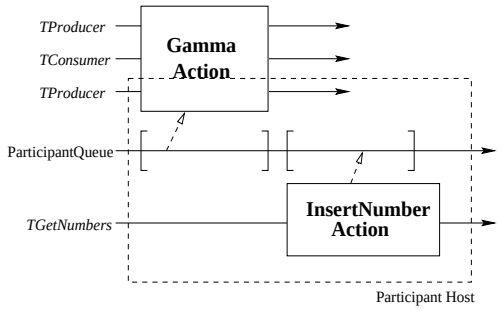

Figure 14: Competition for Accessing External Object

Figure 14 shows the competition between two actions for an external *ParticipantQueue* object. Although in our implementation the access to these objects can overlap, at the logical level access is serialised and consistency is guaranteed.

4.4.5 CO-OPN/2 Specification

The `FinishAction` class, shown in Figure 15, specifies *FinishAction*. This action is 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. The creation of new instances of the `FinishAction` class are triggered by the CA action scheduler. This action has two roles, the `Producer` role and the `Consumer` role. They are both of class `TGetNumbers`. A `TExecuteActions` object calls the `inAction` method, for announcing either 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 (by calling the `FAEnterProducer` and `FAEnterConsumer` methods) and then calls them to leave the action simultaneously (by calling the `FALeaveProducer` and `FALeaveConsumer` methods).

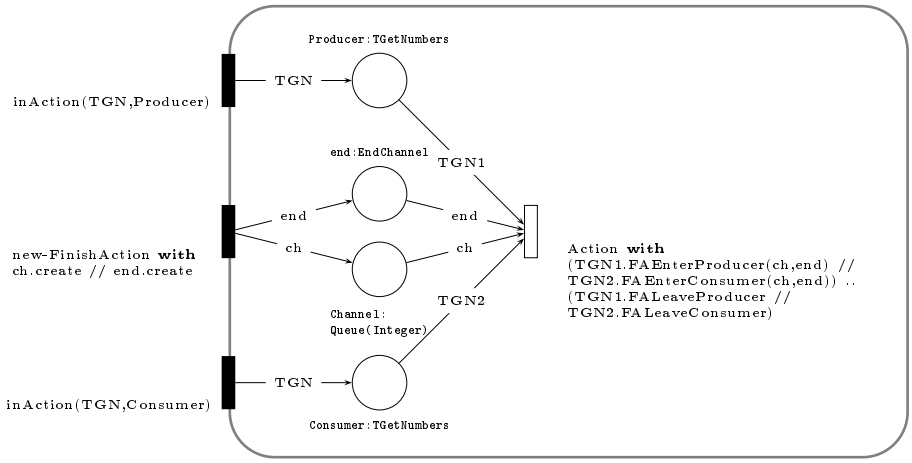

Figure 15: `FinishAction` Class

The `InsertNumberAction` class, shown in Figure 16, specifies *InsertNumberAction*. This action is used to insert an integer (incoming from the user) into a participant queue. The `new-InsertNumberAction(i)` constructor just stores integer `i`. The participant creates new instances of the `InsertNumberAction` class. This action has one role, the `Consumer` role of class `TGetNumbers`. 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 (by calling the `TGN.INAEnter(i)` method) and then calls it to leave the action (by calling the `TGN.INALeave` method).

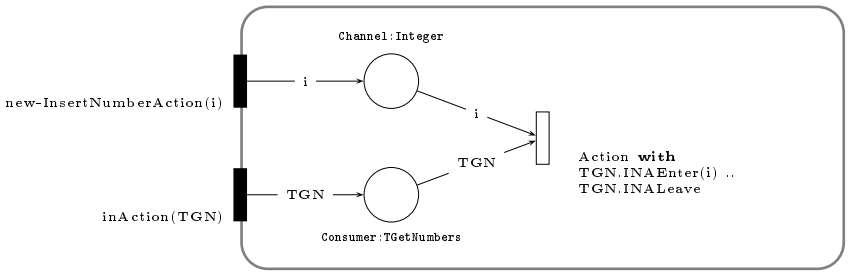

Figure 16: `InsertNumberAction` Class

4.4.6 CO-OPN/2 Specification: Queue(Integer)

Figure 17 shows the participant queue. It is accessed by three methods: the `put(i)` method is used to insert a new integer of value `i` at the end of the queue, the `get(i)` method is used to remove integer `i` from the head of the queue; the `isEmpty` method is used to find out if the queue is empty or not. 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 algebraic term `q`. This property is the fundamental property that enables our specification to satisfy the ACID properties: only one object can request synchronisation with the `get(i)` or `put(i)` method, and not simultaneously, either.

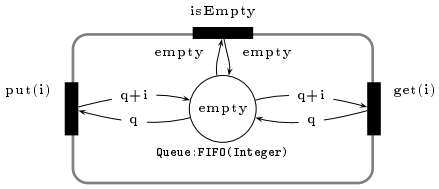

Figure 17: `Queue(Integer)` Class

5 Java Implementation

We have implemented the DSGamma system using the Java programming language [14] and the Remote Method Invocation (RMI) API [15].

The CA action scheduler has been implemented 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 get 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 `CAAScheduler` list. The caller has to send a reference to its remote object that can be accessed by `CAAScheduler`. `newNumber` is used by the participants to inform `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 `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 participant willing to finish.

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 Java `synchronized` methods). 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 `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 the identifier of the action the participant has to execute; it guarantees that the chosen participants will execute the same CA action. `remoteGammaAction` is used by other participants that want to execute `role` in the action located in this participant. The participant willing to execute 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 *FinishAction* together with another participant. `remoteFinishAction` is used by another participant to execute *FinishAction* in this participant. `remoteQueuePut/remoteQueueGet/remoteQueueIsEmpty` are used by a participant when executing the *GammaAction* remotely. It provides access to the local queue of this participant.

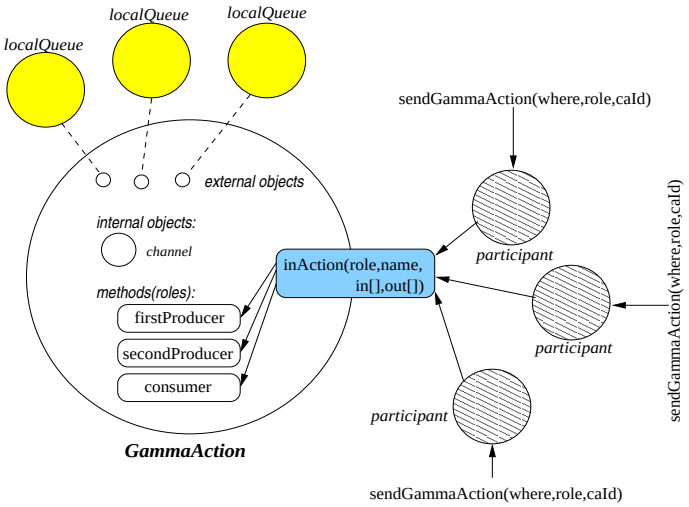

Figure 18: GammaAction Object

Figure 18 shows the `GammaAction` object and its roles. The CA action object is composed of a set of internal objects, used only by the action roles in order to exchange values, i.e. 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-synchronising and post-synchronising the participants; and the roles that the participants will execute. In order to execute a role in an action, the participants must be informed 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 bound to CA actions

dynamically. The `inAction` method handles the tasks of the CA action manager as described above.

When implementing CA actions, we have decided to attach them to 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).

6 Conclusions

We have used the CA action concept as a model for designing a fault-tolerant DSGamma system. It has been shown that the CA action concept helps to obtain a well-structured system, where we separate the concurrent interaction activities (that cooperate and compete) from the scheduling of those activities. These two levels of design allowed us to use CA actions for critical activities in the DSGamma system, i.e. the chemical reactions. The scheduling of the CA actions could be provided by an underlying system (e.g. a specialised operating system).

The CA action design of the system has been formally and completely specified using the CO-OPN/2 specification language, thus giving semantics to the CA action model. Fundamental characteristics of CO-OPN/2, such as atomicity of complex synchronisation, concurrency and object structures, have been shown suitable for the formalisation of CA actions.

According to CO-OPN/2 specification of the CA action based design, we have implemented the whole system using the Java language. 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 the re-use of CA action support system.

Our design clearly separates different system levels. CA actions have distinct important properties supported by the CA action scheme used. For these reasons, we believe that the use of CA actions makes the reasoning about the DSGamma system properties simpler and clearer, and gives us more evidence that this system is correct. We think that the combined use of a CA action design and of CO-OPN/2 specification will make it easier to formally prove that the system has certain properties. Indeed, CO-OPN/2 specification provides a mathematical framework, and each CA action has a set of properties. These can be used to construct the proof of global system properties. Those of interest for the DSGamma system are, for instance: *i*) "no number is lost in the computation process"; *ii*) "the sum of all numbers is invariable and always correct"; *iii*) "the exit of a participant does not affect the sum of all numbers".

Further research will consider:

- a general definition of semantics for the CA action model by giving a denotational semantics of a core CA action language to CO-OPN/2 formal specification [16];
- the verification of distributed systems designed using the CA actions by applying the test method defined for CO-OPN/2 [6];
- the verification of properties of the DSGamma system using formal proofs. A formal refinement strategy based on the preservation of temporal properties expressed over CO-OPN/2 formal specification is being developed [17]. As the development of the DSGamma system presented in this paper is part of a re-engineering process of an application already developed and specified using CO-OPN/2, we will apply this refinement methodology to the CA action new design and follow the properties which were initially studied for the first implementation of this system [18, 19] ;
- an implementation of a set of CO-OPN/2 specification modules and the development of rules and templates which would help to specify and implement, in a reliable semi-automatic way, fault tolerant distributed systems designed using CA actions. We expect that the proof of all main CA action properties, made on these basic CO-OPN/2 specification units, will help to prove interesting properties for the overall implemented system.

7 Acknowledgements

We are grateful to Robert Stroud at the University of Newcastle upon Tyne for his fruitful comments. This research has been supported by ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa)(http://www.newcastle.research.ec.org/deva). Avelino F. Zorzo, lecturer at PUCRS (on leave), is also supported by CNPq (Brazil) under grant number 200531/95.6.

References

[1] XU, J., RANDELL, B., ROMANOVSKY, A., RUBIRA, C., STROUD, R. J., and WU, Z.: 'Fault tolerance in concurrent object-oriented software through coordinated error recovery', 25th International Symposium on Fault-Tolerant Computing, 1995, Pasadena, USA, 1995, pp. 450-457

[2] RANDELL, B.: 'Systems structure for software fault tolerance', *IEEE Transactions on Software Engineering*, 1975, **1**(2) pp. 220-232

[3] GRAY, J., and REUTER, A.: 'Transaction processing: concepts and techniques' (Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993) 2nd edn.

[4] BIBERSTEIN, O., BUCHS, D., and GUELFI, N.: 'Object-oriented nets with algebraic specifications: the CO-OPN/2 formalism', in AGHA, G., and DE CINDIO, F. (Eds): 'Advances in Petri Nets on Object-Orientation' (Springer-Verlag, Lecture Notes in Computer Science to appear, 1997)

[5] GUELFI, N., BIBERSTEIN, O., BUCHS, D., CANVER, E., GAUDEL, M.-C., VON HENKE, F., and SCHWIER, D.: 'Comparison of object-oriented formal methods.' Second Year Report of Esprit Long Term Research Project 20072 "Design For Validation" (DeVa), Department of Computing Science, University of Newcastle Upon Tyne, 1997

[6] PERAIRE, C., BARBEY, S., and BUCHS, D.: 'Test selection for object-oriented software based on formal specifications.' Second Year Report of Esprit Long Term Research Project 20072 "Design For Validation" (DeVa), Department of Computing Science, University of Newcastle Upon Tyne, 1997 (also in PROCOMET'98)

[7] BANATRE, J.-P., and LE METAYER, D.: 'Gamma and the chemical reaction model: ten years after', in ANDREOLI, J.-M., HANKIN C., and LE METAYER, D. (Eds): 'Coordination Programming: Mechanisms, Models and Semantics' (IC Press, 1996)

[8] WIRSING, M.: 'Algebraic specification', in VAN LEEUWEN, J. (Ed.): 'Handbook of Theoretical Computer Science, volume B: Formal Methods and Semantics' (North-Holland, Amsterdam, 1990)

[9] RANDELL, B., ROMANOVSKY, A., STROUD, R. J., XU, J., and ZORZO, A. F.: 'Coordinated atomic actions: from concept to implementation.' Technical report TR 595, Department of Computing Science, University of Newcastle upon Tyne, 1997 (http://www.cs.ncl.ac.uk/research/trs/papers/595.ps)

[10] CAMPBELL, R. H., and RANDELL, B.: 'Error recovery in asynchronous systems', *IEEE Transactions on Software Engineering*, 1986, **12**(8), pp. 811-826

[11] LEE, P. A., and ANDERSON, T: 'Fault tolerance: principles and practice' (Springer-Verlag, Berlin, 1990)

[12] DI MARZO SERUGENDO, G., GUELFI, N., ROMANOVSKY, A., and ZORZO, A. F.: 'Formal development and validation of the DSGamma system based on CO-OPN/2 and coordinated atomic actions.' Technical Report 98/265, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998

[13] ROMANOVSKY, A., RANDELL, B., STROUD, R. J., XU, J., and ZORZO, A. F.: 'Implementation of blocking coordinated atomic actions based on forward error recovery', *Journal of System Architecture - Special Issue on Dependable Parallel Computing Systems*, 1997, **43**(10), pp. 687-699

[14] GOSLING, J., BILL, J., and STEELE, G.: 'The Java language specification' (The Java Series, Addison-Wesley, Massachusetts, 1996)

[15] Java remote method invocation specification. Sun Microsystems, Inc., 1996 (http://java.sun.com:80/products/jdk/rmi/index.html)

[16] BUCHS, D., DI MARZO SERUGENDO, G., GUELFI, N., BUFFO, M., and VACHON, J.: 'Formal specifications of coordinated atomic actions using CO-OPN/2.' Technical Report to appear, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998

[17] DI MARZO SERUGENDO, G., and GUELFI, N.: 'Refinement of synchronized Petri nets based on temporal logic formulae.' Technical report to appear, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998

[18] DI MARZO SERUGENDO, G., and GUELFI, N.: 'Formal development of Java programs.' Technical Report 97/248, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1997

[19] DI MARZO SERUGENDO, G., and GUELFI, N.: 'Using object-oriented algebraic nets for the reverse engineering of Java programs: a case study,' First International Conference on Application of Concurrency to System Design, CSD'98, March 1998, Fukushima, Japan, pp. 166-176