Formal Development and Validation of Java Dependable Distributed Systems

Giovanna Di Marzo Serugendo[1], Nicolas Guelfi[2],
Alexander Romanovsky[3], Avelino Francisco Zorzo[3,4]

[1] LGL-DI, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland

[2] Institut Supérieur de Technologie, L-1359 Luxembourg-Kirchberg

[3] Department of Computing Science, University of Newcastle upon Tyne, NE1 7RU, UK

[4] Faculdade de Informática, PUCRS, Porto Alegre, 90619-900, Brazil

Abstract

The rapid expansion of Java programs into software market is often not supported by a proper development methodology. Here, we present a formal development methodology well-suited for Java dependable distributed applications. It is based on the stepwise refinement of model-oriented formal specifications, and enables validation of the obtained system wrt the client's requirements.

Three refinement steps have been identified in the case of fault-tolerant distributed applications: first, starting from informal requirements, an initial formal specification is derived. It does not depend on implementation constraints and provides a centralized solution; second, dependability and distribution constraints are integrated; third, the Java implementation is realised. The CO-OPN/2 language is used to express specifications formally; and the dependability and distribution design is based on the Coordinated Atomic action concept. The methodology and the three refinement steps are presented through a very simple fault-tolerant distributed Java application.

Keywords: *Structuring Complex Concurrent Systems, CO-OPN/2, Formal Development, Stepwise Refinement, Design for Validation, Coordinated Atomic Actions, Java.*

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 formal enough for a serious verification phase to be defined.

This paper presents a formal development methodology based on the joint use of synchronized Petri nets and temporal logic formulae. It defines a whole development process (analysis, design, implementation) with a design phase relying on the stepwise refinement of synchronized Petri nets. The correctness of the refinement steps is verified on the basis of the logical formulae.

Synchronized Petri nets are expressed using Concurrent Object-Oriented Petri Nets (CO-OPN/2) [4]. CO-OPN/2 is an object-oriented formal specification language that allows concurrent systems to be described in terms of structured Petri nets for the behaviour part, and algebraic specifications for the data structures used to define values managed by the Petri nets. Temporal logic formulae used for the refinement steps are expressed by means of the Hennessy-Milner logic (HML).

The methodology proposed in this paper is well suited for the development of distributed applications. In the particular case of dependable distributed applications, a design phase has been identified where the dependability and distribution constraints are built according to the *Coordinated Atomic action* concept.

The Coordinated Atomic (CA) action model [1] provides structuring primitives and support for error recovery in 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 CA action model makes it possible to design specific fault tolerance mechanisms to cover both hardware and software faults.

The design phase of dependable distributed applications using the CA action concept is as follows: (1) a set of informal application requirements is stated; it includes validation objectives expressed by a set of desired properties; (2) an initial CO-OPN/2 specification is built; it gives a model of the application, which is abstract enough to be as independent as possible of implementation constraints; (3) a refinement of the initial specification is realised; it provides CO-OPN/2 speci-

fications of the CA action design of the application; (4) finally, the Java implementation is derived. Every specification, as well as the Java program, validates the initial requirements and the desired properties.

The structure of this paper is as follows. Section 2 describes the formal development methodology, presents the formal specification language CO-OPN/2, and explains the CA action concept. Section 3 illustrates the proposed formal development methodology and the design phase using the CA action concept applied to a very simple example. Section 4 presents related work.

2. Development Methodology

The methodology proposed in this paper addresses three classical phases of the development process of distributed applications: the *analysis* phase, the *design* phase, and the *implementation* phase.

After the analysis phase, informal requirements are determined. The design phase consists of the stepwise refinement of CO-OPN/2 specifications. The methodology advocates for this phase the joint use of CO-OPN/2 specifications and HML formulae. The behaviour of a system is specified by means of CO-OPN/2 specifications. Such specifications provide a model of the system, and implicitly define a set of properties corresponding to the behaviour defined by the specification. During a refinement step it is not always necessary to preserve the whole behaviour proposed by the specification. Therefore, essential properties expected by the system are explicitly expressed by means of a set of HML formulae, called *contract*. A contract does *not* reflect the whole behaviour of the system, it reflects only the behaviour part that must be preserved during all subsequent refinement steps. A refinement is then defined as the replacement of an abstract specification by a more concrete one, which respects the contract of the abstract specification, and takes into account implementation constraints.

Finally, the implementation phase is treated in a similar way as the design phase. At the end of the design phase, a concrete CO-OPN/2 specification is reached, it is implemented, and the obtained program is considered to be a correct implementation if it satisfies the contract of the most concrete specification.

Figure 1 shows the three phases. On the basis of the informal requirements, an abstract CO-OPN/2 specification Spec_0 is devised. Its contract Contract_0 formally expresses the requirements. During the design phase, several refinement steps are performed, leading to a concrete CO-OPN/2 specification Spec_n and its contract Contract_n. The implementation phase then provides the program Program and its contract Contract. A refinement step is correct if the concrete contracts contain the abstract contracts.

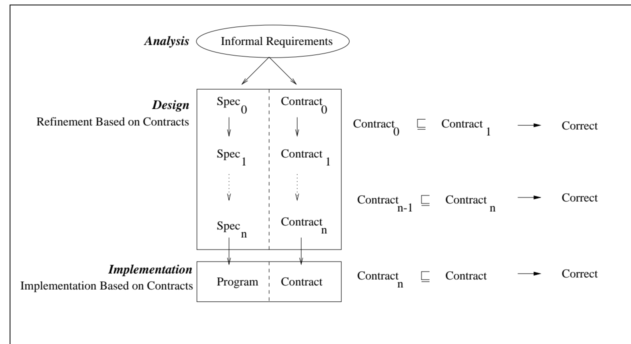

Figure 1. Development Methodology

In the case of dependable distributed systems, a design phase with particular refinement steps has been identified. An *initial* specification provides a centralized system that is as much as possible independent of implementation constraints. The corresponding contract expresses the requested functionality of the system. A *second* specification, which integrates dependability and distributivity constraints, is then built. The CA action concept is used to devise this specification. Indeed, CA actions provide built-in features to achieve fault tolerance. Therefore, the design and the verification of dependable systems become easier. The *third* and last step is provided by the Java implementation.

The rest of this section briefly defines CO-OPN/2 specifications, HML formulae, and the CA action concept.

2.1. CO-OPN/2

CO-OPN/2 is an object-oriented formal specification language [4] that integrates Petri nets used to describe concurrent behaviours and algebraic specifications [7] of structured data evolving in Petri nets. 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, also known as methods, and internal transitions. The former correspond to the services provided to the outside, while the latter describe the internal behaviour of an object. Unlike methods, internal transitions are in-

visible to the exterior world and spontaneous (they are fired as soon as their pre-conditions are satisfied). An object method can only be fired 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. Each class instance has an identity, which is also called an object identifier, that can be used as a reference.

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 nondeterminism (+)). For instance, an object may simultaneously request two different services of two different partners, followed by a service request to a third object.

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. Hennessy-Milner Logic

HML formulae are expressed on CO-OPN/2 specifications. An HML formula is a sequence (or a conjunction ($\wedge$), or an alternative ($+$)) of observable events. Such an event is either the firing of a single method of a CO-OPN/2 object, or the parallel firing of several methods. An HML formula is satisfied by the model of a CO-OPN/2 specification if the sequence of events defined by the formula correspond to a possible sequence of events of the model of the specification.

2.3. Dependable Design: Coordinated Atomic Actions

The CA action [1, 8] 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), which cooperate within the CA action scope. Logically, a CA action starts when all roles have been activated and finishes when all of them have reached the CA action end. A CA action can be completed either when no error has been detected, after successful recovery, or when the recovery fails and a failure exception is propagated to the containing CA action (CA actions can be nested).

External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among these CA 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 way of implementing these semantics is to use a separate transactional support mechanism that provides these properties. This support mechanism can offer the traditional transactional interface, i.e. operations *start, abort* and *commit*, that are called (either by the CA action support or by CA action participants) at the appropriate points during the CA action execution.

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

The CA action mechanism also provides a basic framework for exception handling, which can support a variety of fault tolerance mechanisms aimed at tolerating both hardware and software faults [9, 10].

3. A Complete Example

In order to present the proposed methodology and the CA action design, we consider the following simple example: computing the sum of the integers present in a multiset. The computing of the sum follows the Gamma paradigm [6]: a chemical reaction removes two values from a multiset, computes their sum and inserts the result into the multiset. Figure 2 shows a multiset and a possible Gamma computation achieving result 8.

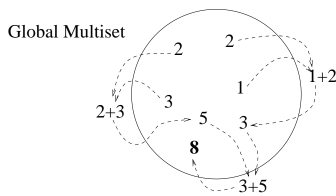

Figure 2. Addition according to the Gamma paradigm

3.1. Informal Requirements

We intend to develop an application allowing several users to insert integers into a distributed multiset according to the Gamma paradigm. 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, the second part describes the details of data and of internal computations, and the third part describes the fault model.

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; **(4)** A user may see the computed result at any moment.

State and Internal Behaviour: **(5)** The integers put in by users are stored in a multiset; **(6)** The application computes the sum of all integers put in by all users; **(7)** The sum is calculated by chemical reactions in accordance to the Gamma paradigm; **(8)** A chemical reaction removes two integers from the multiset, adds them up, and inserts the sum into the multiset; **(9)** There is only one type of chemical reaction, but several of them can occur simultaneously and concurrently in the multiset; **(10)** A chemical reaction may occur provided there are at least two integers in the multiset.

Fault Tolerance: **(11)** The addition operation can fail; **(12)** Storing integers in the multiset can fail; **(13)** Removing integers from the multiset can fail.

Because these operations are executed only inside CA actions, the entire system fault tolerance is provided within the CA action framework. Computer hosts and network do not fail.

3.2. Initial Specification: Centralized View

The initial CO-OPN/2 specification is given by two classes: the `DSGammaSystem` class specifying the services offered by the system to users; and the `Users` class, specifying the users behaviour.

Figure 3 shows the `DSGammaSystem` class. The four system operations **(1)** to **(4)** provided to the users are specified by four CO-OPN/2 methods. The state and internal behaviour (**(5)** to **(10)**) are specified using: place `users` for storing users identity; place `MSInt` for storing integers entered by the users; and internal transition `ChemicalReaction` that actually performs the Gamma chemical reactions: it removes two integers from place `MSInt`, sums them up, and inserts the result into the place.

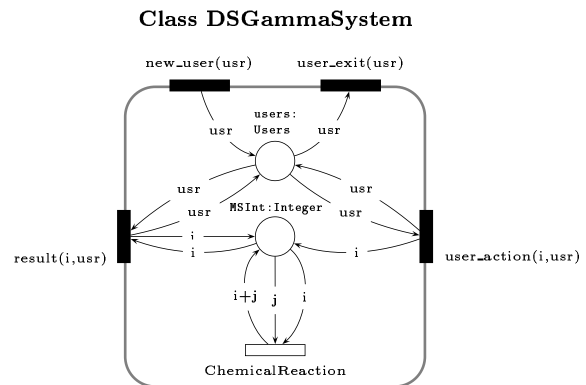

Figure 3. Underlying System

Figure 4 depicts the `Users` class. At creation time a user registers itself to the DSGamma system; then it is able to insert integers, get the result or leave the system. Every action performed by the user is forwarded to `DSG` - a static instance of the DSGamma system.

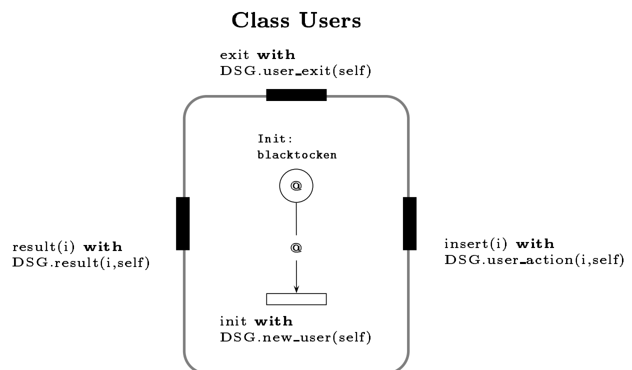

Figure 4. Users

3.2.1. Contract. In order to remain concise, we present a contract $\phi_{\mathbf{I}}$, corresponding to the initial specification, made of only two HML formulae: $\phi_{\mathbf{I}_1}$ and $\phi_{\mathbf{I}_2}$. It is obvious that a larger contract is necessary to ensure all the informal requirements **(1)** to **(10)**.

Formula $\phi_{\mathbf{I}_1}$ states that after the creation of a DS-GammaSystem DSG, it is possible to create a first user usr_1, and a second user usr_2, after that usr_1 inserts integer i in the system, usr_2 inserts integer j, and usr_2 can see the correct result $i + j$. This formula addresses requirement **(4)**.

Formula $\phi_{\mathbf{I}_2}$ is similar but usr_1 leaves the system before usr_2 sees the result. The exit of usr_1 does not affect the computing of the sum. This formula addresses requirements **(3)** and **(4)** simultaneously.

$$
\begin{aligned}
\phi_{\mathbf{I}_1} = {} & <DSG.\,\text{create}><usr_1.\,\text{create}><usr_2.\,\text{create}> \\
& <usr_1.\,\text{user_action}(i)><usr_2.\,\text{user_action}(j)> \\
& <usr_2.\,\text{result}(i+j)> \\
\phi_{\mathbf{I}_2} = {} & <DSG.\,\text{create}><usr_1.\,\text{create}><usr_2.\,\text{create}> \\
& <usr_1.\,\text{user_action}(i)><usr_2.\,\text{user_action}(j)> \\
& <usr_1.\,\text{user_exit}><usr_2.\,\text{result}(i+j)>\ .
\end{aligned}
$$

Contract $\phi_{\mathbf{I}}$ is actually satisfied by the model of the initial specification.

3.3. Refinement R1: CA Action Design

As mentioned in Section 2.3., the CA action mechanism provides a well-structured way of dealing with faults that may happen during a cooperative activity. In this section, we use CA actions to design the DS-Gamma system. Fault tolerance is provided by the CA actions.

3.3.1. System 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 5). 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 *ParticipantQueue*, i.e. a queue in which part of the global multiset is kept.

There are three types of CA action: *GammaAction* (executes the Gamma computation - chemical reaction); *FinishAction* (enables a user to leave the system); and, *InsertNumberAction* (enables a user to insert a new integer in the system).

A CA action scheduler is responsible for receiving information from all participants about any new number they have in their local queues. The CA action scheduler also starts a new *GammaAction* with three roles

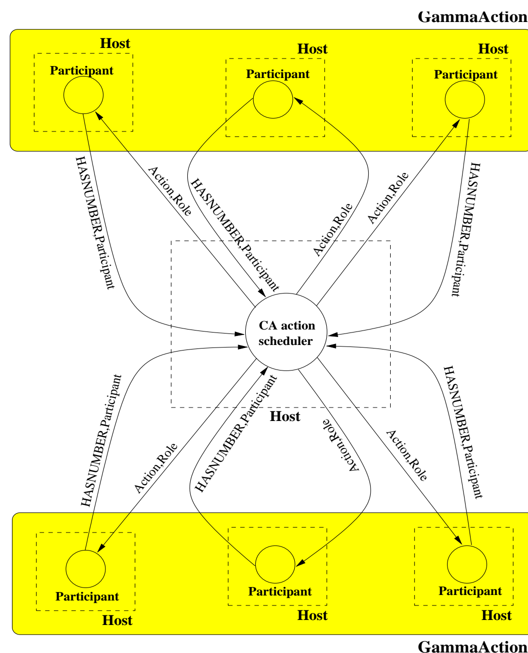

Figure 5. DSGamma System

whenever there are at least two new numbers in the local multisets. There can be as many *GammaActions* active concurrently as there are pairs of integers in all local multisets at a given time. 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). Each participant can be involved in several actions at once playing different roles. This approach allows a better parallelisation of the Gamma computation.

3.3.2. 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 6). 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.

The CA action mechanism provides a framework for dealing with exceptions that happen during the execution of the system. In our DSGamma system, when the addition operation fails inside a *GammaAction*, an

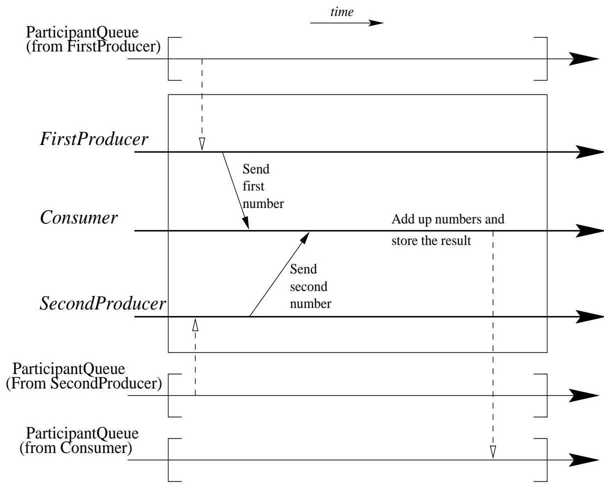

Figure 6. GammaAction

exception, called `ReactionException`, is raised in the thread executing a role in the action. In Figure 7, we represent the `ReactionException` being raised by the *Consumer*.

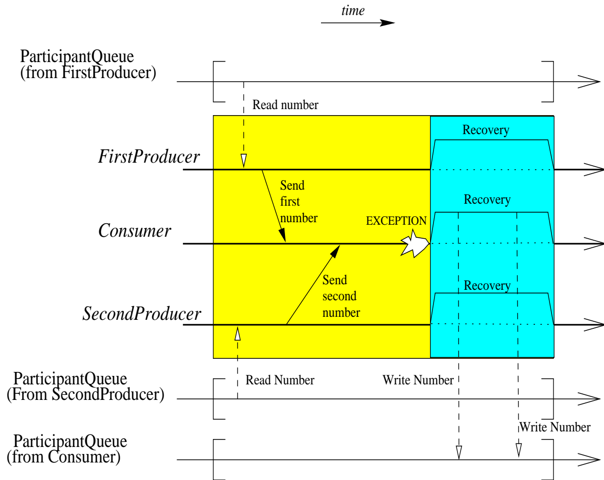

Figure 7. Forward Error Recovery in GammaAction

In accordance to the CA action concept, we attach exception handlers to each role. In Figure 7, for example, after an exception `ReactionException` has been raised, the CA action support mechanims interrupts all the roles in the CA action and calls the handler for this exception in each one of them. Our design decision for the `ReactionException` is to use forward error recovery in the following way: when the reaction (addition) fails, the consumer keeps both integers by inserting them into its local multiset whilst the producers complete the CA action as if nothing had happened. Thus, if the addition operation fails during the CA 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 for *GammaAction* to inform about these new integers in the local multiset of the consumer.

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). Failures in storing, or removing, integers in the multiset are dealt with in a similar way. Due to space limit, we will concentrate, in the rest of this paper, only on the `ReactionException`.

3.3.3. CO-OPN/2 Specification. 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 informal proof of properties can be found in [11].

The `DSGammaSystem` class, shown in Figure 8, depicts the overall system with CA action design. It only creates an instance of the CA action scheduler that participants can use in the sequel.

Class DSGammaSystem

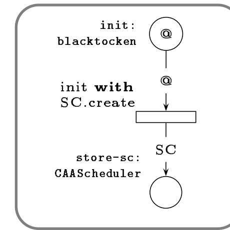

Figure 8. Refinement R1: Overall System

The `Users` class of the initial specification is replaced by the `Participant` class partially given in Figure 9. At creation time, the participant registers itself to the CA action scheduler. It forwards the user's exit request to the CA action scheduler, and furnishes the result by reading in its queue.

The `GammaAction` class, shown in Figure 10, specifies *GammaAction*. 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.

The `inAction` method is used to instruct the CA action about which thread will perform which role in that CA action. The CA action is actually performed by the `Action` transition that first calls all the roles

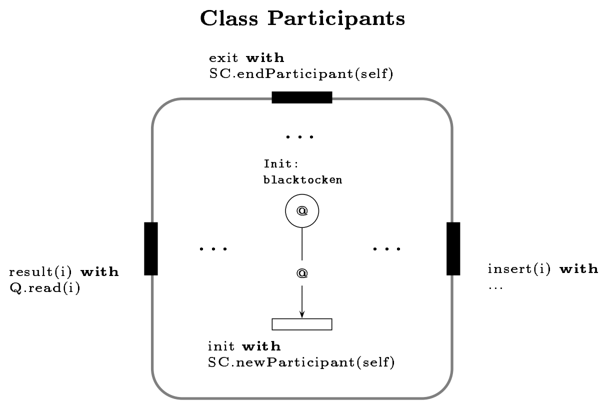

Figure 9. Refinement R1: Participant

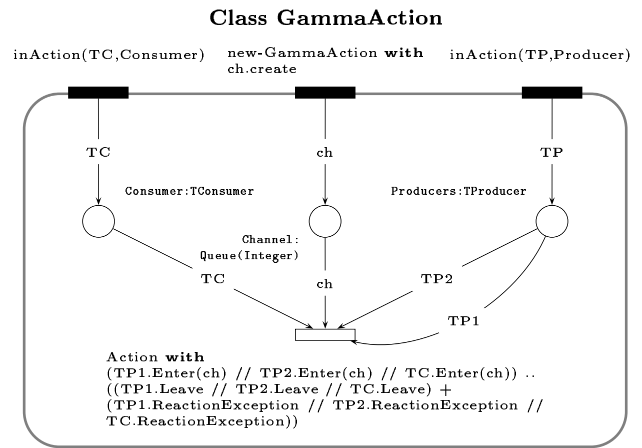

Figure 10. Refinement R1: `GammaAction` Class

in order to let them enter (by calling `Enter`) the CA action simultaneously, and then sequentially calls all the roles in order to let them leave (by calling `Leave`) the CA 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 (hence none of its required synchronisations are).

This CA 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.

The `TConsumer` class, shown in Figure 11, specifies the thread performing the `Consumer` role of *GammaAction*. `TConsumer` has to collect two integers from a channel provided by the CA action (received as a local object), sums them up and inserts the sum into its participant queue. It receives the reference of the queue (at creation time).

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`, e.g. `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 `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,

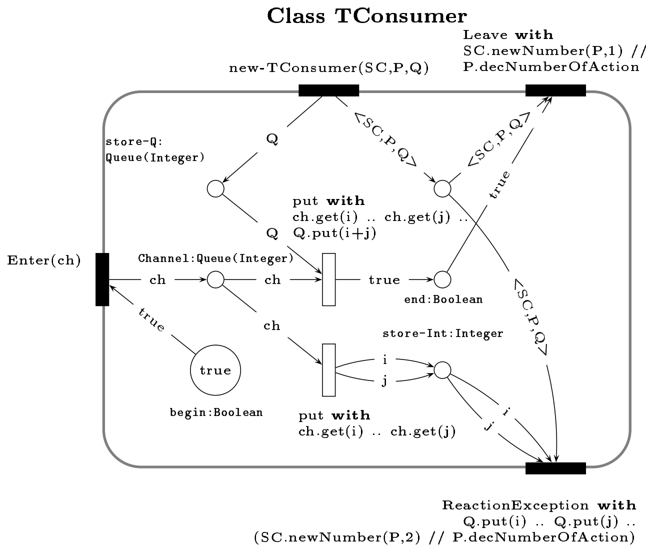

Figure 11. Refinement R1: `TConsumer` **Class**

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 12, specifies the thread performing 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).

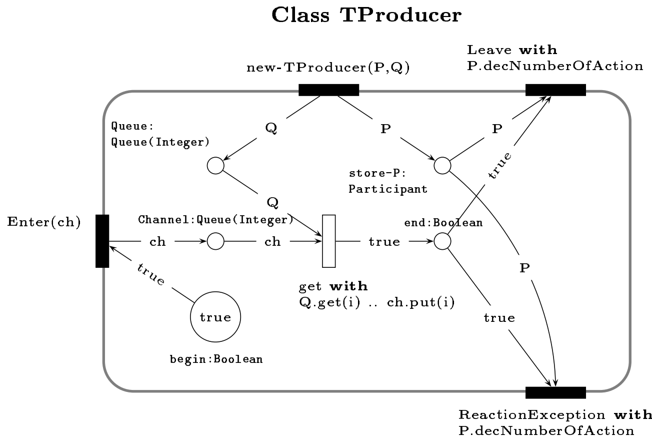

Figure 12. Refinement R1: `TProducer` **Class**

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.

3.3.4. Contract. Contract $\phi_{\mathbf{R1}}$ corresponding to refinement **R1** is made of four formulae. The first two $\phi_{\mathbf{R1}_1}$, $\phi_{\mathbf{R1}_2}$, are similar to those of Contract $\phi_{\mathbf{I}}$, except that the users instances are replaced by participant instances.

Formula $\phi_{\mathbf{R1}_3}$ states that: (1) three participants p_i $(1 \leq i \leq 3)$ and their respectives queues q_i are created; (2) two `TProducer` threads instances TP_1, TP_2 are created, and a `TConsumer` thread instance TC is created; (3) integer i is put into queue q_1, and integer j into queue q_2; (4) a `GammaAction` instance GA is created; (5) the roles are announced with the respective threads; (6) finally there are two possible outcomes after the CA action ends: either the consumer queue q_3 contains the sum $i+j$, or there has been some problem and the queue q_3 contains both i and j.

Formula $\phi_{\mathbf{R1}_4}$ is similar to formula $\phi_{\mathbf{R1}_3}$. It states that it is *not possible* that after the end of `GammaAction` GA the three queues are empty, (two integers i and j lost), or queue q_3 contains only i (integer j lost).

$$\begin{aligned}
\phi_{\mathbf{R1}_1} = {} & <DSG.\text{create}><p_1.\text{create}><p_2.\text{create}> \\
& <p_1.\text{user_action}(i)><p_2.\text{user_action}(j)> \\
& <p_2.\text{result}(i+j)> \\
\phi_{\mathbf{R1}_2} = {} & <DSG.\text{create}><p_1.\text{create}><p_2.\text{create}> \\
& <p_1.\text{user_action}(i)><p_2.\text{user_action}(j)> \\
& <p_1.\text{user_exit}><p_2.\text{result}(i+j)> \\
\phi_{\mathbf{R1}_3} = {} & _{\psi}\ (<q_3.\text{get}(i+j)> + (<q_3.\text{get}(i)><q_3.\text{get}(j)>)) \\
\phi_{\mathbf{R1}_4} = {} & _{\psi}\ \neg((<q_1.\text{isEmpty}><q_2.\text{isEmpty}><q_3.\text{isEmpty}>) + \\
& (<q_1.\text{isEmpty}><q_2.\text{isEmpty}> \\
& <q_3.\text{get(i)}><q_3.\text{isEmpty}>))
\end{aligned}$$

where:

$$
\begin{aligned}
\psi = & <DSG.\,\mathrm{create}> \\
& <p_1.\,\mathrm{create}><p_2.\,\mathrm{create}><p_3.\,\mathrm{create}> \\
& <q_1.\,\mathrm{create}><q_2.\,\mathrm{create}><q_3.\,\mathrm{create}> \\
& <TP_1.\,\mathrm{new\text{-}TProducer}(p_1, q_1)> \\
& <TP_2.\,\mathrm{new\text{-}TProducer}(p_2, q_2)> \\
& <TC.\,\mathrm{new\text{-}TConsumer}(SC, p_3, q_3)> \\
& <q_1.\,\mathrm{put}(i)><q_2.\,\mathrm{put}(j)><GA.\,\mathrm{new\text{-}GammaAction}> \\
& <GA.\,\mathrm{inAction}(TP_1, \mathrm{Producer})> \\
& <GA.\,\mathrm{inAction}(TP_2, \mathrm{Producer})> \\
& <GA.\,\mathrm{inAction}(TC, \mathrm{Consumer})>\ .
\end{aligned}
$$

Refinement **R1** actually satisfies this contract. It is a correct refinement of specification **I** since contract $\phi_{\mathbf{I}}$ is included (modulo renaming) in contract $\phi_{\mathbf{R1}}$.

3.4. Java Implementation

In this section we show how we have implemented the DSGamma system using the Java programming language [13] (Java ORB Remote Method Invocation - RMI is used to distribute objects).

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.

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 it participates in the `GammaAction` as `consumer` (the CA action scheduler will set that).

Figure 13 shows the `GammaAction` object and its roles. The CA action object is composed of a set of internal objects, which are used only by the CA action roles in order to exchange values, i.e. to communicate; a set of external objects that the roles will access in an atomic way; a manager that is responsible for recovering the CA action from possible errors, and for pre-synchronising and post-synchronising the participants; and the roles that the participants will execute. In order to execute a role in a CA action, the participants must be informed about the action and the role they have to execute; such information will be provided by the CA action scheduler using the `sendGammaAction` method of the participants. Once

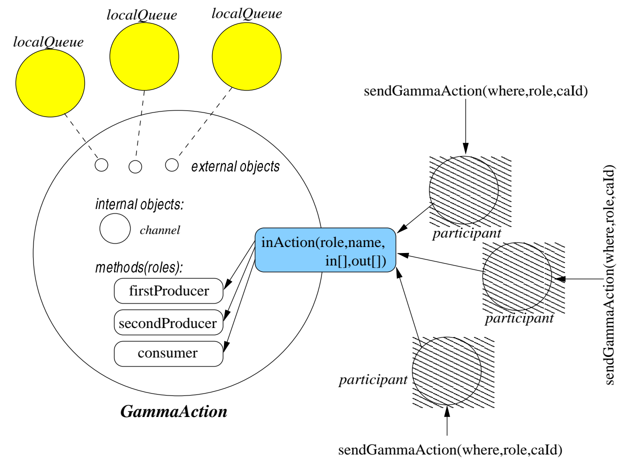

Figure 13. The GammaAction Object

the participants know the action and the 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, and enables the participants to perform their role by calling the methods `consumer`, `firstProducer`, and `secondProducer`.

All CA 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). Different approaches for implementing CA actions can be found in [19].

The code for the `consumer` role of the `GammaAction` class is shown below. As shown in Figure 6, the consumer first receives an integer from each of the two producers. After the consumer has received both numbers, it sums them up and then stores the result in its local multiset. If anything goes wrong during the consumer activity, then an exception is raised and stored in the `exception` variable (see `catch (Exception e)` block below). Before the consumer can finish its activity, it executes the `finally` block. In this block it notifies the CA action manager about any exception that has been raised during the execution of its code by calling the `exceptionResolution` method. If the consumer has not raised any exception, then the calling of the `exceptionResolution` method is used by the CA action manager to notify the consumer about possible exceptions raised by the producers. If no-one has raised any exception in the `GammaAction`, then the consumer can finish its execution. In the code below we also show how the exception `ReactionException`

would be handled, i.e. the two integers are stored in the queue of the participant performing the consumer role. Any exception raised during the handling of an exception will cause the CA action mechanism to raise a failure exception to the enclosing context [8].

```
private void consumer(Participant p) throws Exception {
  Exception exception = null;
  Integer num1, num2;
  int sum;
  try {
    num1 = (Integer) p1Channel.receive();
    num2 = (Integer) p2Channel.receive();
    sum  = num1.intValue() + num2.intValue();
    p.remoteQueuePut(sum);
  } catch (Exception e) {
        exception = e;
  } finally {
      try {
           exceptionResolution(exception);
      } catch (ReactionException e) {
          p.remoteQueuePut(num1);
          p.remoteQueuePut(num2);
      } // handling for other exceptions
  }
}
```

The code for the `firstProducer` role of the `GammaAction` class is shown below. Notice that any exception raised in the role, or in the `GammaAction`, is dealt with in the same way as in the `consumer` role.

```
private void firstProducer(Participant p) throws
Exception {
  Exception exception = null;
  int num;
  try {
    p1Channel.send(new Integer(p.remoteQueueGet()));
  } catch (Exception e) {
        exception = e;
  } finally {
      try {
           exceptionResolution(exception);
      } catch (ReactionException e) {
          // do nothing.
      } // handling for other exceptions
  }
}
```

3.4.1. Contract. Contract $\psi_{\mathbf{Prog}}$, corresponding to the Java program, is made of four formulae corresponding to the four formulae of contract $\phi_{\mathbf{R1}}$. The syntax of the formulae is slightly different since it is adapted to the program. The creation of the system is represented by the start of the `main` method of the program (provided by Class `CASchedulerServer`). The creation of instances is noted by i.`ClassName`. For instance instruction: p_1 = new ParticipantApplet() is noted in HML by p_1. ParticipantApplet. The interaction with the user by means of the GUI is expressed by means of the Java `action` method that intercepts the events (e.g. a user entering an integer in a TextField). In HML it is noted with the name of the corresponding GUI component.

$$\begin{aligned}
\psi_{\mathbf{Prog_1}} = &<CASserver.\mathrm{main}><p_1.\mathrm{ParticipantApplet}>\\
&<p_2.\mathrm{ParticipantApplet}><p_1.\mathrm{action_newNumber}(i)>\\
&<p_2.\mathrm{action_newNumber}(j)><p_2.\mathrm{action_result}(i+j)>\\
\psi_{\mathbf{Prog_2}} = &<CASserver.\mathrm{main}><p_1.\mathrm{ParticipantApplet}>\\
&<p_2.\mathrm{ParticipantApplet}><p_1.\mathrm{action_newNumber}(i)>\\
&<p_2.\mathrm{action_newNumber}(j)>\\
&<p_1.\mathrm{action_Finish}><p_2.\mathrm{action_result}(i+j)>\\
\psi_{\mathbf{Prog_3}} = &\psi\ (<q_3.\mathrm{get}(i+j)> + (<q_3.\mathrm{get}(i)><q_3.\mathrm{get}(j)>))\\
\psi_{\mathbf{Prog_4}} = &\psi\ \neg((<q_1.\mathrm{isEmpty}><q_2.\mathrm{isEmpty}><q_3.\mathrm{isEmpty}>) +\\
&(<q_1.\mathrm{isEmpty}><q_2.\mathrm{isEmpty}>\\
&<q_3.\mathrm{get}(i)><q_3.\mathrm{isEmpty}>))
\end{aligned}$$

where:

$$\begin{aligned}
\psi = &<CASserver.\mathrm{main}><p_1.\mathrm{ParticipantApplet}>\\
&<p_2.\mathrm{ParticipantApplet}><p_3.\mathrm{ParticipantApplet}>\\
&<q_1.\mathrm{PQueue}><q_2.\mathrm{PQueue}><q_3.\mathrm{PQueue}>\\
&<TP_1.\mathrm{Thread}(p_1,q_1)><TP_2.\mathrm{Thread}(p_2,q_2)>\\
&<TC.\mathrm{Thread}(SC,p_3,q_3)><q_1.\mathrm{put}(i)><q_2.\mathrm{put}(j)>\\
&<GA.\mathrm{GammaAction}><GA.\mathrm{inAction}(TP_1,\mathrm{Producer})>\\
&<GA.\mathrm{inAction}(TP_2,\mathrm{Producer})>\\
&<GA.\mathrm{inAction}(TC,\mathrm{Consumer})>.
\end{aligned}$$

This contract is actually satisfied by the program, since it relies on several properties guaranteed by the CA actions (ACID property, the recovery when the Consumer cannot make the sum, and the fact that no number is lost during the computation). This contract includes (modulo renaming) contract $\phi_{\mathbf{R1}}$. Therefore, the Java program is a correct implementation of CO-OPN/2 specification **R1**, and hence of the initial specification.

4. Related Work

Formal methods traditionally use a single formal specification language for expressing both the requirement specifications, and the system specifications. When the chosen formal specification language is a logical language, the specification task is more difficult, but the verification task is reduced to showing logical implications. When the chosen formal specification language is model-oriented, specifications are more easily and powerfully expressed, but the verification task is difficult and usually follows an informal way (e.g. simulation).

In order to bring a solution to the problem of the choice between a model-oriented and a property-oriented formal specification language, some model-oriented specification languages have acquired a property-oriented specification language. This is

known as the *two languages framework* described, among others, by Pnueli in [17]: a logical language is used for expressing requirements, and a model-oriented language is used for describing models or implementations. In addition, the logical language is also used for translating the system specification into logical properties, and the verification task is then realized in the logical framework. Among others, the VDM^{++} [16] language and the temporal Petri nets [14] use this approach. The methodology proposed in the current paper follows the two languages framework. Its particularity is that it goes a step further, since the contracts explicitly point out the essential properties to be verified.

The verification that a program is correct wrt system specifications is a problem similar to the one of verifying that system specifications are correct wrt the requirement specifications. Thus, the use of a logical language in addition to a programming language should help the verification task. In the last decades, only few attempts have been undertaken to consider the idea of integrating assertions into programs. More recently, Meyer [15] has promoted this idea, and even goes a step further. Indeed, he advocates that, in order to face the problem of correctness, every program operation (instruction or routine body) should be systematically accompanied by a pre- and a post-condition.

5. Conclusions

This paper presents a methodology for developing distributed programs based on the stepwise refinement of formal specifications. It advocates the use of specific temporal properties for guiding and verifying refinement steps. In addition, for dependable applications, a design phase using the CA action concept is promoted.

The complete development of a small system is described: starting from informal requirements a Java implementation is reached, and every step is formally proved.

We think that the combined use of CA action design and CO-OPN/2 specification makes it easier to prove formally that the system has certain properties. Indeed, CO-OPN/2 specifications providem a mathematical framework, and each CA action guarantees a set of properties. These can be used to construct the proof of global system properties.

The methodology presented here, in the special case of the CO-OPN/2 language, is part of a more general theory [12] that adds a contract (made of logical properties) to a model-oriented formal specification, and that enables to prove the correctness of refinement steps formally.

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 [18];
- the verification of distributed systems designed using the CA actions by applying the test method defined for CO-OPN/2 [5];
- the Hennessy-Milner logic used to express the contracts is a very simple logic that enables to build tools for verification easily. However, it lacks expressivity, since any invariant needs an infinite set of formulae to be described. We are now considering improving this logic with some temporal operators, or even using another logic in the framework of CO-OPN/2;
- this paper only shows proofs 'made by hand'; automated verification, and construction of contracts are being investigated.

6. 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 is supported by CNPq (Brazil) under grant number 200531/95.6.

References

[1] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. J. Stroud, and Z. Wu: '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] B. Randell: 'Systems structure for software fault tolerance', *IEEE Transactions on Software Engineering*, 1975, **1**(2) pp. 220-232.

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

[4] O. Biberstein, D. Buchs, and N. Guelfi: 'CO-OPN/2: A concurrent object-oriented formalism', in Proc. Second IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). Chapman and Hall, 1997.

[5] C. Péraire, S. Barbey, and D. Buchs: '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).

[6] J.-P. Banâtre, and D. Le Métayer: 'Gamma and the chemical reaction model: ten years after', in J.-M. Andréoli, C. Hankin, and D. Le Métayer (Eds): 'Coordination Programming: Mechanisms, Models and Semantics', IC Press, 1996.

[7] M. Wirsing: 'Algebraic specification', in J. Van Leeuwen (Ed.): 'Handbook of Theoretical Computer Science, volume B: Formal Methods and Semantics', North-Holland, Amsterdam, 1990.

[8] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo: '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)

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

[10] P. A. Lee, and T. Anderson: 'Fault tolerance: principles and practice', Springer-Verlag, Berlin, 1990.

[11] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. F. Zorzo: '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.

[12] G. Di Marzo Serugendo: 'Stepwise Refinement of Formal Specifications Based on Logical Formulae: from CO-OPN/2 Specifications to Java Programs', Swiss Federal Institute of Technology in Lausanne, Phd thesis no 1931, 1999.

[13] J. Gosling, J. Bill, and G. Steele: 'The Java language specification' (The Java Series, Addison-Wesley, Massachusetts, 1996).

[14] M. Felder, D. Mandrioli, and A. Morzenti: 'Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models', *IEEE Transactions on Software Engineering*, 1994, **20** (2), pp.127-141.

[15] B. Meyer: 'Object-Oriented Software Construction', Prentice Hall, 1997.

[16] K. Lano: 'Formal Object-Oriented Development', Springer-Verlag, 1995.

[17] A. Pnueli: 'System Specification and Refinement in Temporal Logic', LNCS 652, pp. 1-38, 1992.

[18] J. Vachon, D. Buchs, M. Buffo, G. Di Marzo Serugendo, B. Randell, A. Romanovsky, R. Stroud, and J. Xu, 'COALA - A Formal Language for Coordinated Atomic Actions', DeVa Third Year Report, Deliverables: Part 2 (Papers), December 1998.

[19] A. F. Zorzo: 'Dependable Multiparty Interactions: A Case Study', 29th Conference on Technology of Object-Oriented Languages and Systems - TOOLS29-Europe, Nancy, France, June, 1999.