Modeling Exception Handling: a UML2.0 Platform Independent Profile for CAA

Alfredo Capozucca, Barbara Gallina, Nicolas Guelfi, and Patrizio Pelliccione

Software Engineering Competence Center
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359 -Luxembourg

Abstract. Complex fault-tolerant distributed systems have a growing need of new functional and quality requirements. An immediate consequence of this is an increasing need of new methods for developing complex fault-tolerant distributed applications.
Coordinated Atomic Actions (CAAs), making use of exception handling mechanism, offer an approach to ensure the needed requirements of reliability, availability and fault tolerance.
Unfortunately, there is currently no method for the high-level modeling of such systems. In this paper, in order to offer an instrument for modeling exception handling, we propose a UML2.0 Platform Independent Profile for CAAs that allows designers to describe complex systems separating the specification from the implementation on a specific technology platform.

1 Introduction

Implementing complex fault-tolerant distributed applications is labor intensive and error-prone. Such systems have increasingly new functional and quality requirements. An immediate consequence of such evolutions in the requirements of systems is an increasing need of new methods that assist in these systems development. Furthermore, there is a growing interest in the area of application-level fault tolerance and cooperative exception handling as the main paradigm for developing structured fault-tolerant architectures. Unfortunately, there is currently no method that assists the software engineer in developing such systems.

This situation has motivated the investigation of a UML-based, generative and architecture-centric method to support cost-effective, disciplined and high-level development of complex fault-tolerant distributed systems families. This effort is conducted by the Software Engineering Competence Center of the University of Luxembourg (SE2C) in the context of the CORRECT project [2]. The CORRECT project makes use of Coordinated Atomic Actions (CAAs) [9] that offer approaches for error recovery and of the framework called Dependable Remote Interacting Processes (DRIP) [10] that embodies CAAs in terms of a set of Java classes.

UML [6] provides a standard-based high-level specification language as well as the possibility to clearly separate concerns at different development phases.

Model-driven Engineering (MDE) advocates separating the specification of system functionality, the "what", from the implementation of that functionality on a specific technology platform, the "how" [5]. Platform Independent Models (PIMs) specify the structure and functionality of a system while abstracting away technical details. Platform Specific Models (PSMs) specify how the functionality has to be realized on a selected platform. MDE takes benefits of PIMs and PSMs for generating the executable code of a system by deriving PSM specifications from PIMs by (formal) refinement, and then transforming PSMs to code for a specific platform. This approach allows relating design to requirements and analysis, as well as verifying and testing the application, and generating code for different technologies by transforming PIMs.

In this paper we focus on the "what", i.e. on the specification of the system, by proposing a UML Platform Independent Profile for CAAs. This work refers to a previous work [3] where has been proposed the FTT-UML, a CAA-based UML profile for modeling fault-tolerant business processes. We fix some discovered problems and we propose a new UML2.0 profile for CAAs, able to model fault-tolerant systems on any domain. In other words, the proposed profile provides the necessary features for modeling "pure" CAAs.

After an introduction on CAA in Section 2, we present the FTT-UML profile in Section 3 highlighting the problems driving the definition of the proposal profile, that is explained in Section 4. We make use of a simple example to show how all aspects of a CAA can be modeled with the proposed profile. Conclusion and future works close the paper.

2 Coordinated Atomic Actions

Coordinated Atomic Actions (CAAs) is a fault-tolerant mechanism using exception handling to achieve dependability in distributed and concurrent systems [9]. Thus, using CAAs we can develop systems that comply with their specification in spite of faults having occurred or occurring.

This mechanism unifies the features of two complementary concepts: *conversation* and *transaction*. Conversation [7] is a fault-tolerant technique for performing coordinated recovery in a set of participants that have been designed to interact with each other to provide a specific service (cooperative concurrency). These participants are called **roles** in the context of CAAs. Transactions are used in order to deal with competitive concurrency on external objects, which have been designed and implemented separately from the applications that make use of them.

Every external object should be under the control of the transactional system to guarantee the ACID (atomicity, consistency, isolation and durability) properties. Anyway, there are some external objects that, because of their own nature, could not be rolled back. Therefore, they must be managed in a special way to guarantee the ACID properties on them. The external objects that can be rolled back are called *recoverable* and those that require explicit manipulation to be left in a consistent state are called *unrecoverable*. This categorization of external objects allows us to satisfy the ACID properties on them.

A CAA is the basic element of this technique, which characterizes an orchestration of actions executed by a group of roles that exchange information among them, and/or access to external objects (which at the same time are shared with other CAAs) to get a common goal. A CAA starts when all its roles have been activated and finishes when all of them have reached the CAA end. If for any reason an exception is raised in at least one of the roles belonging to the CAA, appropriate recovery measures have to be taken. Facing this situation, a CAA provides a quite general solution for fault-tolerance that consists of applying both forward and backward error recovery techniques in a complementary or combined manner.

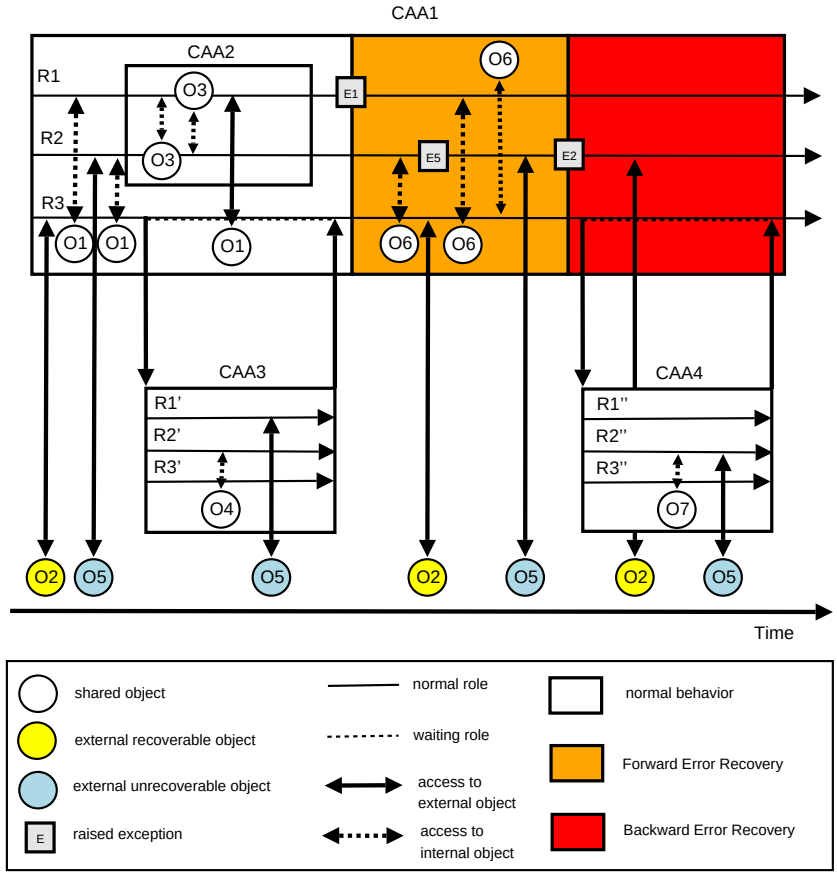

Fig. 1. Coordinated Atomic Actions.

Figure 1 shows how CAAs can be designed in a recursive way using **nesting** and/or **composing**. Nesting is defined as a subset of the roles of a CAA (CAA_1) defining a new CAA (CAA_2) inside the enclosing CAA (CAA_1). The roles of CAA_2 (R_1 and R_2) are the same roles that have been defined for CAA_1, but

the operations that they are doing inside CAA_2 are hidden for the other roles (R_3) (and other nested or composed CAAs) that belong to CAA_1. Accesses to external objects within a nested CAA are performed as nested transactions, so that, if CAA_1 terminates exceptionally, all sub-transactions that were committed by the nested (CAA_2) are aborted as well. Any role of a CAA can only enter one nested CAA at a time. Furthermore, a CAA terminates only when all its nested CAAs have terminated as well. Note that if the nested CAA_2 terminates exceptionally, an exception is signalled to the containing CAA_1.

Composed CAAs are different from the nested in the sense that we can reuse the CAAs designed in other contexts. Thus, composition allows us to develop open distributed systems. A composed CAA (CAA_3) is an autonomous entity with its own roles ($R_1{}', R_2{}'$ and $R_3{}'$) and external objects (unrecoverable O_5). The internal structure of the composed CAA_3 (i.e., set of roles, accessed external objects and behavior of roles) is hidden from the calling CAA_1. A role belonging to CAA_1 that calls the composed CAA_3 synchronously waits for the outcome. Then, the calling role resumes its execution according to the outcome of the composed CAA_3. If the composed CAA_3 terminates exceptionally, its calling role (which belongs to CAA_1) raises an internal exception which is, if possible, locally handled. If local handling is not possible, the exception is propagated to all the peer roles of CAA_1 for coordinated error recovery.

If the composed CAA_3 has terminated with a normal outcome, but the containing CAA_1 has to roll back its effects (abort operation), the tasks that were done in the composed CAA_3 are not automatically undone. Thus the CAA_1, in order to guarantee the ACID properties on the external object, needs to carry out a specific handling, which may have a composed CAA (CAA_4) to abort (or compensate, if there is at least one unrecoverable external object) the effects that have been done by the CAA_3.

In order to formally express the semantics of each possible kind of CAA outcome (**normal, exceptional, abort** and **failure**), we use statecharts [4](Figure 2). The specification is composed of two state machines, which are running in parallel. The machine on the left side represents the *System* which evolves according to the events that are sent (events with a line over them) by the state machine that is on the right side and represents a *CAA*.

A CAA is designed to provide a service, which is called by the users of the System where the CAA is embedded. The invocation of the service is represented by the event *Op*. We can assume that this event comes from the environment. The *W* state represents the execution of the service. If the service is able to satisfy its post-conditions, then the CAA terminates normally. Therefore, the CAA reaches the *OK* state, publishing at the same time the *normal* event, thus the System goes to the well defined *s1* state.

The state of the CAA is defined as the state of the System plus the state variables that are used to deliver the service. Thus, the CAA could be in an unspecified (not well defined or inconsistent) state, but the System is left in a specified (well defined or consistent) state. If the CAA does not normally finalize (because the post-conditions are not met), then each role must signal an

exception to indicate the outcome. The roles should agree about the outcome, and each role should signal the same exception. In this case, the CAA emits the event *exceptional* and it goes to an unspecified state. When the System receives the event *exceptional*, it goes to any well defined state, even if the specific service that is provided by the CAA could not be delivered satisfactorily.

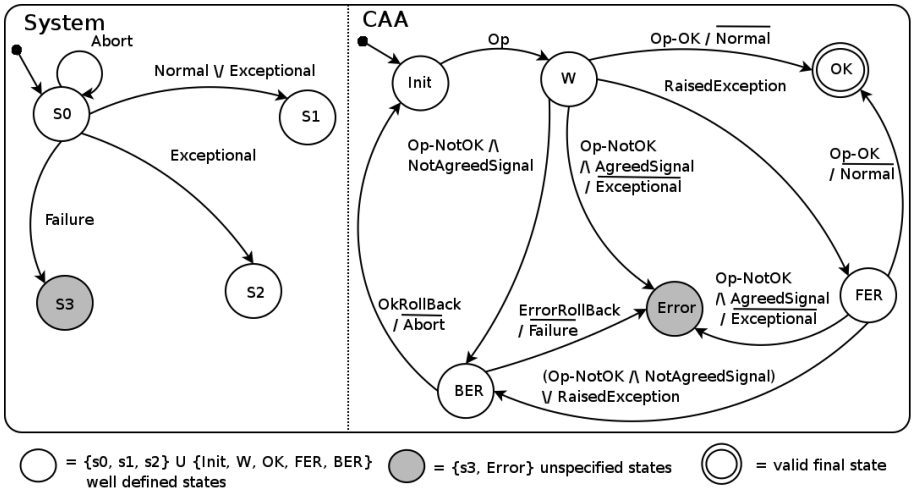

Fig. 2. Outcomes of a CAA and their semantics.

If an exception is raised during the normal execution of the CAA (*s0* state), then a process of exception handling is triggered. This exception handling process is defined as a combination of Forward (FER) and Backward Error Recovery (BER). The FER mechanism is represented by the *FER* state and, depending on how successfully it can recover from the exception, the CAA may still terminate normally. In any case (*W* or *FER* states), if the CAA finalizes not normally and the roles disagree about the outcome, then the CAA attempts to abort the action by undoing the effects through the BER mechanism.

If the post-conditions cannot be reached through the FER or an exception is raised again thought this recovery process, the CAA must attempt to roll back the state through BER (*BER* state). If BER is applied successfully, the CAA publishes the event *abort* and it goes to the initial state (*Init*). The System receives this event and moves to the same state (*s0*) where it was before calling the CAA.

If the BER is unsuccessful, then the CAA must publish the event *failure*. In this case, both the CAA and the System are left in a unspecified state.

3 FTT-UML: a UML-profile for Fault-Tolerant Transactions

The FTT-UML Profile has been presented at OTM Workshops in 2004 [3] and constitutes an important step towards modeling dependable complex business

processes since it provides means in order to manage exceptions and in order to describe nested transactions in e-business distributed contexts.

FTT-UML privileges Activity Diagrams and Class Diagram among all the UML 1.5 diagrams available: Activity Diagrams in order to model the behavior of the business process and Class Diagrams in order to model the business domain data structure. In this section we present a list of FTT-UML elements directly related to CAA concepts. The starting point for this list has been [1]. FTT-UML has been proposed in e-business context and therefore it provides also means in order to model typical e-business situations, such as filling a form, but these aspects are, in our context, not crucial.

- **Activity Diagram:** an Activity Diagram represents a CAA.
- **Partition:** a partition, which is a vertical solid line dividing an Activity Diagram, represents a CAA role. Each partition contains the actions performed by the corresponding role.
- **Object Flow State:** data structures are modeled using UML class diagrams. The corresponding data are used in activity diagrams exploiting object flow states represented by rectangles. Subsection 3.3 of [3] provides more detailed information about the data manipulation primitives provided by this notation.
- **Action state:** a role performs actions, represented by action states (corner-rounded rectangles). A state has pre- and post-conditions (first paragraph in section 3 of [3]). An action is executed when all its pre-conditions are satisfied, and then it offers a token to all its outgoing edges. Action states represent method calls on objects. The name of this state refers to the following topology: objectName.method() where objectName is an accessible object (from the role point of view) and method is a visible method of this object. Arguments of this call are represented by the name of the objects inside the parenthesis, or by linking objects to the action state (subsection 3.4 of [3]).
- **Transition:** a transition represents the execution flow (solid lines) or the data flow (dotted lines).
- **Final PseudoState:** represents both a normal outcome (if the activity does not produce outputs) and an implicit synchronization point among roles. In CAA, in fact, roles have to synchronize themselves in the exit and agree about the outcome.
- **Initial PseudoState:** represents the only starting point of the CAA in the case of synchronous roles. It means that CAA starts when all the role are activated. In cases in which asynchronous roles are present, more than one initial node has to bes used in order to model the asynchronous activation of the roles themselves.
- **ObjectFlowState with stereotype ≪output≫:** represents a normal outcome with an output.
- **Final PseudoState with stereotype ≪except≫ and labeled with the exception name:** represents an exceptional outcome without parameter.
- **ObjectFlowState with stereotype ≪except≫ and labeled with the exception name:** represents an exceptional outcome with a parameter.

- **Sub-activityState:** a sub-activity state models a nested CAA.
- **ActionState with the stereotype ≪invited≫:** represents a role in an enclosing CAA that does not contain the sub-activity node (representing a nested CAA), but participates in that nested CAA.
- **ObjectFlowState with an edge to the sub-activity node:** represents providing an input to a nested process.
- **Transition with the stereotype ≪compensate≫ going to the nested process sub-activity state:** represents undoing the effects of a nested process.
- **Activity diagram having the stereotype ≪compensate≫:** represents a compensation, it undoes the effects of a nested process already successfully terminated, whose effects are requested to be undone inside the running CAA.
- **Action state with stereotype ≪raise≫ and named by the exception to raise:** represents a role of a CAA that wants to raise an internal exception. Such a state must not have outgoing transitions.
- **Partition with the stereotype ≪ExceptionResolution≫:** represents a graph whose purpose is the resolution of concurrent exceptions into a single exception to raise. Such a partition contains an oriented graph, where action states represent exceptions.
- **Action state with the stereotype ≪handler≫ and named by the exception name:** indicates the starting point of a subset of a role activities that represents a handler for one of the exceptions of the exception resolution graph. (variants: for a parameterized exception, the action ≪raise≫ has an incoming edge from an object flow state and the action ≪handler≫ has an outgoing transition to an object flow state representing the parameter).

3.1 The Discovered Problems

Inside a CAA three phases may be distinguished: one in order to describe the normal behavior, one in order to capture the exception handling behavior and finally, one in order to capture the roll back behavior. It should, anyway, be clear that the last behavior is a special case of exception handling.

In FTT-UML these three behaviors are not well defined. Only the normal one is well defined and detailed, while the other two are only sketched and never illustrated. By reading the profile explanation, it comes out that there are two particular stereotypes, one called *compensate* which stereotypes an Activity Diagram and should be used in order to undo effects when roll back is needed, and one, called *handler*, in order to describe exception handling behavior.

These two stereotypes however are never illustrated and therefore it is not clear if they have to be used in a particular partition or not. Suppose, for example, to put the ≪handler≫ stereotype in all the partitions (roles) that may have a handler in order to manage an exception; the resulting diagram would be overloaded.

Moreover CAAs could be composed and nested and this difference does not appear in the FTT-UML profile.

Roles of an enclosing CAA may, in FTT-UML, change their name by entering in a nested process. This possibility in CAA theory is not present. In nested CAAs in fact still appear roles that were already part of the enclosing CAA. Different roles are only possible in the case of CAA composition.

Logically, a CAA starts when all roles have been activated (though it is an implementation decision to use either synchronous or asynchronous entry protocol) and finishes when all of them reach the action end (see [8]). In FTT-UML profile the user may also model implementation details by defining synchronous/asynchronous roles. A CAA model, however, in order to ensure a good separation of implementation and business concerns, should not contain implementation details.

4 Improving the FTT-UML profile

Taking the FTT-UML Profile as basic reference to represent CAA by UML 1.5, we have defined a new one (called CAA-UML), which has as main improvement the fact that it supports completely and purely CAA semantics. Supporting only the basic features of CAAs avoids embedding characteristics for any specific domain, which would be a bias of our target.

Using the last version of UML (2.0) as our notation language also allows us to find a better representation for some basic CAAs aspects, i.e. detecting and handling exceptions, as well as the definition of context where an exception may happen.

In the following, we give the detailed list of elements that compose the CAA-UML Profile explaining improvements and solutions with respect to the problems found on its predecessor profile. To better explain the new proposed profile we provide an example of its use (Figure 3), which corresponds to the CAAs described in Figure 1.

- **CAA:** this is represented by an *Activity Diagram*. In Figure 3, CAA_1 and the nested CAA_2 are examples of how an Activity Diagram describes a CAA.
- **Role:** this is represented by a *Activity Partition*. In Figure 3, R_1 and R_2 and R_3 are partitions that represent roles of CAA_1, *Nested* CAA_2 and *FER for exception E1*.
- **Nested CAA:** as shown in the Figure 1, several roles of CAA_1 can enter into a nested CAA_2, which defines an atomic operation inside the enclosed CAA_1. This is represented by one, and only one, *Call Behavior Action* with stereotype ≪Nesting≫. This stereotyped *Call Behavior Action* has an incoming control flow arrow for each role that is participating in the nested CAA. These incoming flow arrows do not need labels like in the previous profile, because it is not required to change the names of the participating roles in the nested CAA.

  The *Nesting Call Behavior Action* must be in one of the roles belonging to the nested CAA. Since there is no defined semantics in order to choose where to put the nested CAA, the designer must put it in one of the involved roles (it does not matter in which one).

The nesting of CAA_2 in CAA_1, represented in Figure 1, becomes, in the profile, *Call Behavior Action* with the stereotype ≪Nesting≫, called CAA_2, that is inside R_1 belonging to CAA_1 (see Figure 3).

- **Composed CAA:** composition is the other kind of relationship that we can use to design CAAs. This is denoted by a *Call Behavior Action* with the stereotype ≪Composition≫ and represents the creation of a completely independent CAA (CAA_3 in Figure 1) with its own roles and objects. The composed CAA is called by a role belonging to the enclosing CAA (CAA_1 in Figure 1). In Figure 3, inside R_3 of CAA_1, this representation can be found. Thus, according to the last two elements presented, there are two different well defined ways to describe nesting and composing of CAAs. This feature is not present in the previous profile.
- **Outcomes:** the profile must be able to represent the four kinds of outcome that a CAA can return. The **normal** outcome is represented by a *Final Activity Node*. The **exceptional** outcome is represented by an *Final Activity Node* with the stereotype ≪Exception≫. The name of the stereotyped *Final Activity Node* represents the exception that is signalled to the enclosing context. It can happen when the post-conditions of the CAA are not met and every role agrees with the exception to signal. The representation of these elements can be found in Figure 3 in each CAA and in the FER as well.

  The outcome **abort** and **failure** are used to notify how successfully the BER has been ("abort" if the effects can be undo, otherwise it must be "failure"). The BER is part of the Coordinated Error Recovery mechanism, so these outcomes are generate automatically according to how well this mechanism could be applied with respect to the handled exception. The enclosing context (CAA_1 in Figure 3), from where the nested (CAA_2) or composed (CAA_3) CAA has been called, receives the corresponding not normal outcome ("exception", "abort" or "failure") through an exception, which is represented by an *Accept Event Action Object Node* with an outgoing *Interrupting Edge*. The node must be stereotyped with ≪Abort≫ (AbortEx, in Figure 3) or ≪Failure≫ (FailureEx, in Figure 3) according to the outcome. The *Interrupting Edge* must be incoming to the associated handler (*Call Behavior Node* called CAA_3 with stereotype ≪BER≫) for the exception that has been detected (details about the handler can be found in the next point).
- **Coordinated Error Recovery mechanism:** this mechanism allows designers to describe the substitution of the normal behavior execution by an exceptional behavior. This exceptional behavior starts to execute when an exception is detected. Firstly, the designer has to declare the area in which an exception could be detected and then specify the exceptional behavior. This area is represented by an *Interruptible Activity Region*. This region must have associated a set of exception handlers, one of which is called when one of its related exception is raised.

  As explained in the representation of the outcomes, an exception is represented by an *Accept Event Action Object Node* with an outgoing *Interrupting*

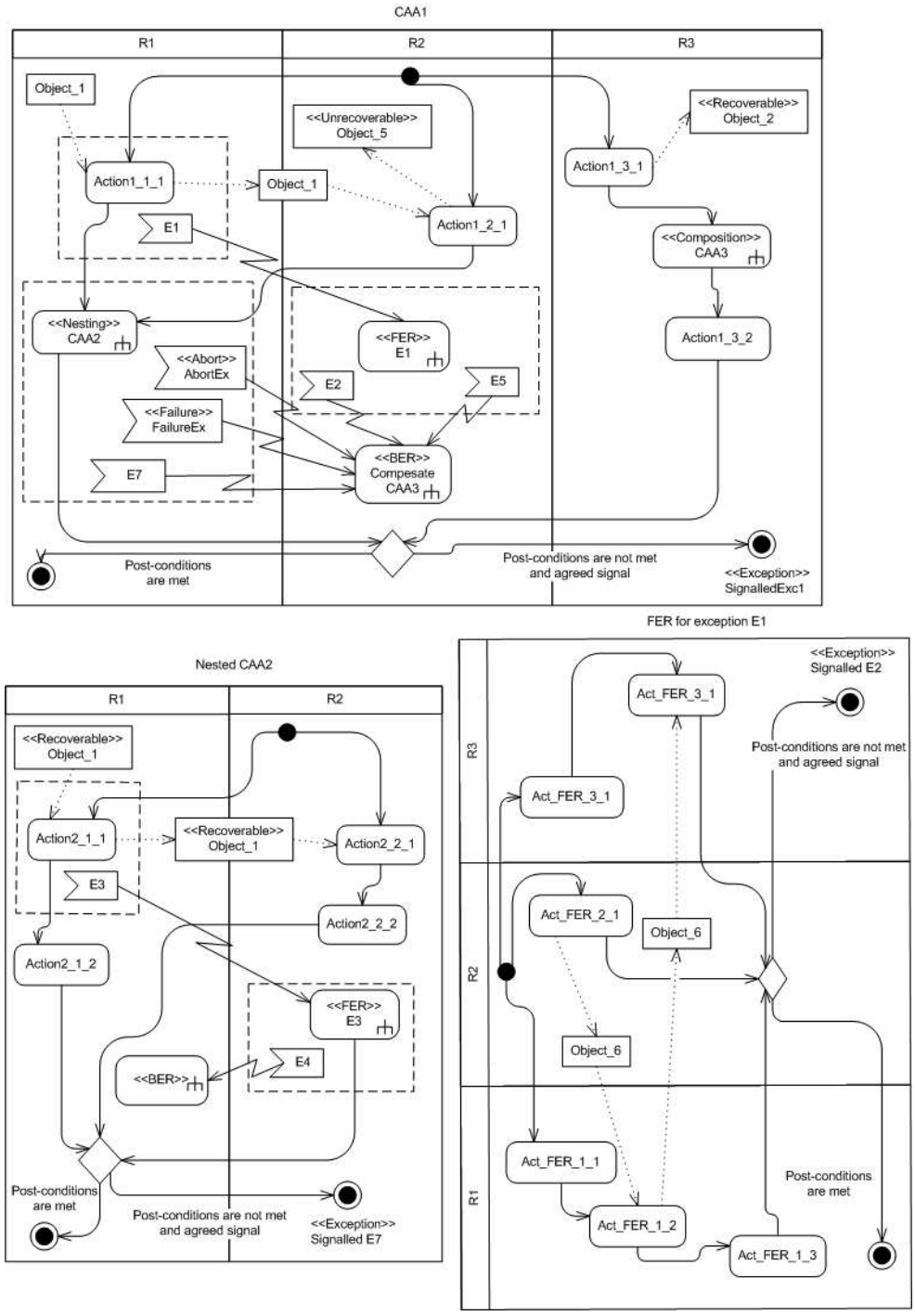

Fig. 3. UML representation of CAAs.

Edge. The exception handler, that is attached by the *Interrupting Edge*, is represented by a *Call Behavior Action*. The Coordinated Error Recovery mechanism allows us to combine FER and BER (*Call Behavior Action* with stereotype ≪FER≫ and ≪BER≫, respectively). Since there is no defined semantics in order to choose where to put the nested CAA, the designer puts it in one of the CAA roles (it does not matter in which one).

According to the exception semantics of CAA, if an exception is raised when the FER is executing (E_5, in Figure 3), the BER mechanism must be started.

A *Call Behavior Action* with stereotype ≪BER≫ and without name represents the classic roll back (the System is restored to its previous state). Otherwise, if the BER has a name, it means that a specific behavior must be executed to undo the effects that have been done by the CAA. In Figure 1, the BER uses the composed CAA_4 to leave the $Object_4$ in a consistent state. In Figure 3, it is represented by the *Call Behavior Action* with stereotype ≪BER≫ and name CAA_3. Since BER is a special case of FER the modeling way does not change therefore there is no need here to introduce another figure to represent it.

It is important to stress that an exception (represented by an *Accept Event Action Object Node*) with a stereotype ≪Abort≫ or ≪Failure≫ could only be found inside an *Interruptible Activity Region* where there is at least a nested or composed CAA (R_1 of CAA_1 has the nested CAA_2).

- **External unrecoverable object:** in this case we want to represent an external object that cannot be restored to its initial state after the operations applied inside the CAA. This kind of object is represented by an *Object Node* with the stereotype ≪Unrecoverable≫ ($Object_5$, in Figure 3). We must use a special handling to be sure that the ACID properties on this kind of objects are satisfied.
- **External recoverable object:** this is represented by an *Object Node* with the stereotype ≪Recoverable≫ (O_2, in Figure 3), and the meaning is the complement of the previous kind of object. Thus, the state for every object with this stereotype, can be restored to the last well defined state known just before the CAA starts. This kind of objects has full transactional support, therefore ACID properties are satisfied.
- **Local shared object:** this is represented by an *Object Node* ($O_{1,6}$, in Figure 3). This kind of object is used by the roles of a CAA to exchange information among them. A local shared object can become either a recoverable or unrecoverable external object depending on its recoverable or unrecoverable characteristics. In Figure 1, O_1 is local to CAA_1, but it becomes external recoverable for the nested CAA_2 (the fact that is recoverable could be seen in Figure 3).
- **Exception Resolution tree:** this is represented through a class diagram in which every class represents an exception and it is related to other exceptions by a generalisation relationship. By moving from the leaf to the root it is possible to find out how to manage exceptions in case of concurrency.

5 Conclusion

In this paper we presented a UML 2.0 profile for CAAs. The work is based on a previous work and, by putting in evidence the weak points of that approach, we have justified the introduction of a new one, based on UML 2.0. This profile is completely platform independent, separating the specification of system functionalities from the implementation of them on a specific technology platform. Thus, this paper represents an important contribution for high-level design of fault-tolerant complex distributed systems. Finally, through a theoretical example, we showed how a software engineer can model these systems detailing normal and exceptional scenarios.

On the future work side we are interested in integrating this profile in a development process from a high-level system description to the system implementation, through suitable refinements, able to preserve fault tolerance properties.

Acknowledgments This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04. The authors gratefully acknowledge help from A. Campéas, A. Romanovsky and R. Razavi.

References

1. A. Capozucca, N. Guelfi, and R. Razavi. Towards a UML-based Notation for CAAs. *TR-SE2C-04-05, SE2C-University of Luxembourg, Luxembourg*, 2004.
2. Correct Web Page. http://se2c.uni.lu/tiki/tiki-index.php?page=correctdoc.
3. N. Guelfi, G. L. Cousin, and B. Ries. Engineering of dependable complex business processes using uml and coordinated atomic actions. In *OTM Workshops*, pages 468–482, 2004.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
5. Object Management Group (OMG). OMG/Model Driven Architecture - A Technical Perspective, 2001. OMG Document: ormsc/01-07-01.
6. Object Management Group (OMG). Unified Modeling Language (UML): Superstructure version 2.0, final adopted specification (02/08/2003). http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2003.
7. B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):pp. 220–232, 1975.
8. A. B. Romanovsky and A. F. Zorzo. A distributed coordinated atomic action scheme. *Comput. Syst. Sci. Eng.*, 16(4):237–247, 2001.
9. J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.
10. A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 435–446. ACM Press, 1999.