# Frameworks for designing and implementing dependable systems using Coordinated Atomic Actions: a comparative study

Alfredo Capozucca [a] Nicolas Guelfi [a] Patrizio Pelliccione [b]
Alexander Romanovsky [c] Avelino Zorzo [d]

[a] *LASSY, University of Luxembourg, Luxembourg*

[b] *Department of Computer Science, University of L'Aquila, L'Aquila, Italy*

[c] *Center for Software Reliability, University of Newcastle upon Tyne, England*

[d] *Faculty of Informatics, Pontifical Catholic University of RS, Brazil*

**Abstract**

This paper [1] presents ways of implementing dependable distributed applications designed using the Coordinated Atomic Action (CAA) paradigm. CAAs provide a coherent set of concepts adapted to fault tolerant distributed system design that includes structured transactions, distribution, cooperation, competition, and forward and backward error recovery mechanisms triggered by exceptions. DRIP (Dependable Remote Interacting Processes) is an efficient Java implementation framework which provides support for implementing Dependable Multiparty Interactions (DMI). As DMIs have a softer exception handling semantics compared with the CAA semantics, a CAA design can be implemented using the DRIP framework. A new framework called CAA-DRIP allows programmers to exclusively implement the semantics of CAAs using the same terminology and concepts at the design and implementation levels. The new framework not only simplifies the implementation phase, but also reduces the final system size as it requires less number of instances for creating a CAA at runtime. The paper analyses both implementation frameworks in great detail, drawing a systematic comparison of the two. The CAAs behaviour is described in terms of Statecharts to better understand the differences between the two frameworks. Based on the results of the comparison, we use one of the frameworks to implement a case study belonging to the e-health domain.

*Key words:* exception handling, fault tolerance, dependable distributed systems

[1] Some preliminary results included in this paper were presented at 17th International Symposium on Software Reliability Engineering (ISSRE'06) (see [22]).

*Email addresses:* `alfredo.capozucca@uni.lu` (Alfredo Capozucca),

## 1 Introduction

Development of modern software systems needs to ensure that such systems meet challenging functional and quality requirements. In the last years a number of instruments and tools have been proposed to drive the software development process to satisfy high quality requirements. Unfortunately, the main trend is to focus on the normal behaviour of software systems, ignoring abnormal behaviour that systems exhibit while facing faults, errors and failures. It is now becoming clear that rigorous methodologies for building dependable software should equally support dealing with such impairments. Fault tolerance is the ultimate technology that can be used to build a system that complies with its specification even when facing the impairments of various types.

Several mechanisms for dealing with system faults have been developed in the past, including, Recovery Blocks (RB) [14], N-Version Programming (NVP) [5], Conversations [14], and Transactions [10]. The last two are specifically targeting fault tolerance of the complex concurrent (distributed) systems.

The general classification of concurrent systems [13, pp. 170-180] identifies three categories of relations between concurrent processes: they can be independent, or disjoint, competing and cooperating. The disjoint active components (e.g., processes and threads) access the disjoint sets of passive components (e.g., data, DBs, and objects). Competitive concurrency between two or more active components exists when these components are designed separately, are not aware of each other, but use the same passive components. Cooperative concurrency exists when several active components cooperate, i.e., do some job together and are aware of this. Hoare [1] explains that while disjoint processes work on disjoint data spaces, competing ones work on the same data. It is however guaranteed that this is done if each of the competing processes had these data at its sole disposal and that processes cooperate if they update common variables by commutative operations and in a disciplined way (to guarantee a consistent access).

*Conversations* were specifically developed to ensure fault tolerance of cooperative systems, whereas *Transactions* are used for ensuring fault tolerance and structuring of the competitive systems. Coordinated Atomic Actions (CAAs) [17,21], on the other hand, were proposed for designing complex distributed systems consisting of components that can cooperate and compete. They allow developers to design systems meeting high availability and reliability requirements. CAAs provide fault tolerance by means of cooperative exception handling and employ, where necessary, the action level abort as part

nicolas.guelfi@uni.lu (Nicolas Guelfi), pellicci@di.univaq.it (Patrizio Pelliccione), alexander.romanovsky@ncl.ac.uk (Alexander Romanovsky), zorzo@inf.pucrs.br (Avelino Zorzo).

of the more general exception handling. CAAs have been successfully used in several case studies [9,15,16,19] that demonstrate high usefulness and general applicability of the approach.

The implementation of systems designed using CAAs was firstly supported by the Dependable Remote Interacting Processes (DRIP) [4] framework. DRIP was initially developed to provide implementation of the "Dependable Multiparty Interactions" (DMIs) abstraction [18]. DMI is a scheme that allows executing a set of participants (objects, threads, or processes) together. These participants join in an interaction to produce a temporal intermediate state, they use this state to execute some activities, and then they leave this interaction to continue their normal execution. In many ways these features are similar to the features of CAAs. As a matter of fact, the DMI concept was developed by the same group that proposed the CAA concept. The main difference between DMI and CAA schemes is in the way they deal with exceptions. DMIs have a more relaxed exception handling semantics than CAAs as they support a chain of recovery levels for dealing with exceptions. CAAs only allow one recovery level. This is why a CAA design can be achieved in terms of DMIs and then implemented using DRIP.

Although CAAs could be implemented using the DRIP framework, this is not straightforward since DRIP components are not mapped directly into CAA concepts. Furthermore, the designer could even forget, or decide to avoid, some of the components that would be needed for the correct implementation of the CAA concept. This could happen because the designer would have to follow some specific patterns, otherwise the CAA semantics could be lost. In developing safety-critical applications, relying on a programmer following specific patterns to implement the application seems to be a dangerous idea. Therefore, a new framework was proposed, named CAA-DRIP [22]. This new framework uses the same terminology as the one used for describing the CAA concepts, therefore the mapping between the design of a CAA and its implementation is straightforward.

In this paper, we present a full comparison between DRIP and CAA-DRIP, showing that the CAA-DRIP framework is an improvement of the DRIP framework when implementing safety-critical applications that use the CAA concepts. This comparison is performed by means of a very simple example. We show that even the performance and the memory consumption of the CAA-DRIP is better than the previous framework. Furthermore, we put in practice CAA-DRIP by presenting the full design and implementation of a safety-critical system, i.e. a Fault-tolerant Insulin Pump Therapy system. The aim of this case study is to show that CAA-DRIP could be applied to a real system.

In order to provide the reader with a clear understanding of the CAAs abstraction, their characteristics are described using Statecharts [11,12]. There has been a considerable work on formal description of CAAs, for example, using Temporal Logic [16], Timed CSP [2] and B [8]. Here a clean formal high level description of the CAA behaviour is offered both to complement previous CAA formalisations and to be used by programmers as a reference to drive the CAA implementation phase. The Statecharts modelling language provides a good approach to express complex behaviours. It enables viewing the description at different levels of details and makes even very large specifications manageable and comprehensible.

Furthermore, some of the CAA concepts were improved in the past years, as well as several new issues were introduced since the first CAA description was given. For example, a new composite type of CAA has been described in [8,9,15] and the way external objects are dealt with is spread throughout several papers [8,17]. In this paper, all these features are collected and described.

After a detailed description of the CAAs mechanism semantics (Section 2), both DRIP and CAA-DRIP frameworks (Section 3) are introduced. The same section explains how programmers have to deal with the frameworks to achieve the implementation of certain CAA design and what the advantages/disadvantages with respect to each other are. Section 4 describes a case study implemented using the CAA-DRIP framework that provides better support according to the comparison previously made. Finally, the paper closes with conclusions and future work.

## 2 Coordinated Atomic Actions

Coordinated Atomic Action (CAA) is a fault-tolerant mechanism that uses concurrent exception handling to achieve dependability in distributed and concurrent systems. The aim of this conceptual framework is to allow engineers/designers to structure (in the sense of software architecture) software in such a way that they comply with their specification in spite of faults having occurred.

As above-mentioned in the introduction, CAAs unify the features of two complementary concepts: *conversation* and *transaction*. For coping with these two concepts, the conceptual framework divides objects in two categories according to their use. Objects used to achieve cooperation and to perform coordinated error recovery (i.e conversation) among the participants are called **shared** objects. On the other hand, those objects that have been designed and implemented separately from the application, which are accessed concurrently (i.e. competitive concurrency) and for which certain properties (like the ACID properties) have to be ensured, are referred as **external** objects. Precisely, the transactional aspects included by CAAs allow us to deal with the external objects.

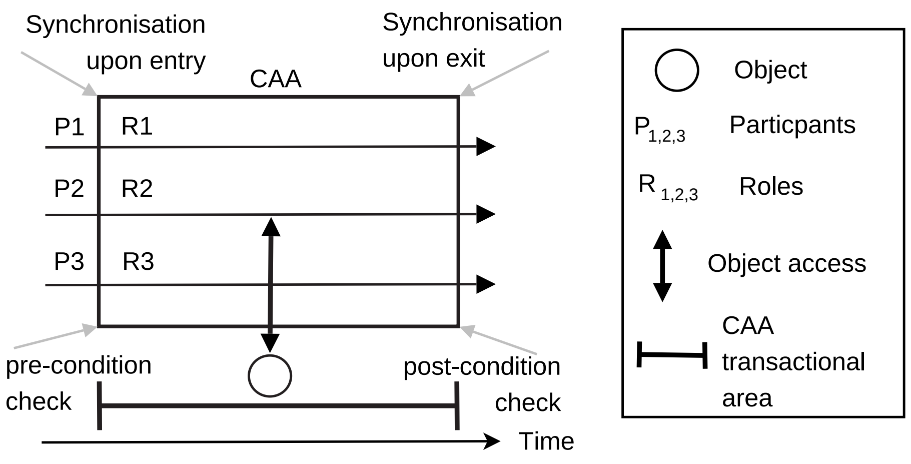


Fig. 1. A simple CAA

As it is shown in Figure 1, one CAA characterises an orchestration of operations executed by a group of roles, which exchange information among themselves through shared objects, and/or access to external objects (concurrently with other CAAs) to achieve a common goal. For the enclosing context where the CAA is embedded, this common goal is seen as the service it provides and which is computed atomically (i.e. there is not any visible intermediate state). To make use of this service, a set of participants gets together by invoking the roles the CAA is composed of. Therefore, in the general case, there is a relationship one-to-one between participants (i.e. CAA outside world) and roles (i.e. internal world) to invoke the service.

Looking into the internal behaviour of a CAA, it starts when all its roles have been activated and they meet a pre-condition. The CAA finishes when all of them have reached the CAA end, and a post-condition is satisfied. This behaviour returns a **normal** outcome to the enclosing context. If for any reason an exception has been raised in at least one of the roles, appropriate recovery measures have to be taken. Facing this situation, a CAA provides a quite general solution for fault tolerance based on exception handling. It consists of applying both forward error recovery (FER) and backward error recovery (BER) techniques.

Basically, the CAA exception handling semantics says that once an exception has been raised, the FER mechanism has to be started. At this point the CAA can finish normally if FER can fulfil the original request (**normal** outcome) or exceptionally if the original request is partially satisfied (**exceptional** outcome). Otherwise, if the same or another exception is raised during FER, then the FER mechanism is stopped and BER is started. BER has a main task to recover every external object to its latest visited error-free state (roll back).

If BER succeed, then the CAA returns the **abort** outcome. If for any reason BER cannot be completed, then the CAA has failed and the **failure** outcome is signalled to the enclosing context.

Every external object that is accessed in a CAA must be able to be restored to its latest visited error-free state (if BER is activated) and it provides its own error recovery mechanism [17]. Therefore, when BER takes place, it restores these external objects using their own recovery mechanisms. However, sometimes the designer/programmer may want to or have to use an external object that does not provide any recovery mechanism (due to reasons of cost or physical constraints [13, pp. 146-149]). Therefore, it would be necessary to allow designers/programmers to specify/implement a hand-made roll back inside the CAA. This can be achieved by refining the classic BER to deal with external objects that are restored using their own mechanism (called **AutoRecoverable** external objects -*AR*-) and also to deal with those that have to be restored by a hand-made roll back (called **ManuallyRecoverable** -*MR*-).

One of the problems of using BER concerns the objects (particularly **external** objects) that cannot be restored from their latest known state. According to the previous information and the CAA semantics, there would be two different places to handle *AR* objects (FER and BER) and only one to handle *MR* objects (FER). Thus, when FER fails because an exception has been raised, potentially any external object could have been left in an unacceptable (non-specified) state. Then, the BER is executed. If it is successful, the **Abort** exception is returned. This outcome corresponds to say that the system has been left at the same state it had before calling the CAA.

As the BER would only undo effects on *AR* objects, it is possible that an *MR* object is still in an inconsistent state. Thus, it would not be true that the system would be in the same state that the one before calling the CAA (**Abort**). Basically, the ACID (Atomicity, Consistency, Isolation, and Durability) properties would not be met.

The BER refinement consists of splitting it between *automatic* abort (classical **roll back**) and a *hand-made* recovery (**compensation**). Compensation must be used to specify the explicit manipulation when a CAA has to abort and there is at least one *MR* object. Compensation cannot be automatically executed since only the designer/programmer knows what the necessary steps to compensate a particular *MR* object are. This compensation can even need the acting of an external agent to help in the recovery (e.g., to call an operator or a maintenance person to fix something). Compensation is not the perfect solution to assure the ACID properties, but at least drives the designers/programmers in that direction.

Originally, the CAA semantics did not clearly distinguish between $AR$ and $MR$ objects. This distinction is made in order to know what are the external objects that will be managed by the transactional support when the CAA has to abort (the $AR$) and those requiring explicit manipulation to be left in a consistent state (the $MR$).

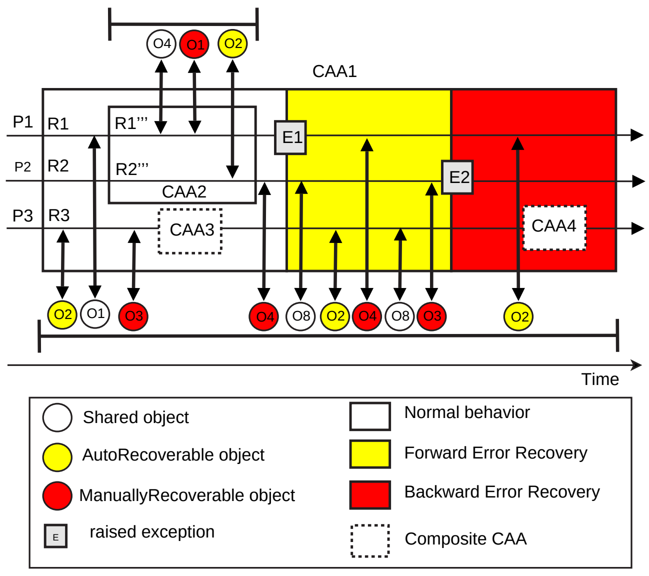


Fig. 2. Coordinated Atomic Actions.

Another important characteristic of CAAs is that they can be designed in a structured way using **nesting** and/or **composition** (see Figures 2 and 3). Nesting is defined as a subset of the participants used to carry out the roles of a CAA ($CAA_1$). These chosen participants define a new CAA ($CAA_2$) inside the enclosing CAA ($CAA_1$). The participants in $CAA_2$ are a subset of the participants from $CAA_1$, but they play different roles in each CAA. The activities carried out inside of $CAA_2$ are hidden for the other roles ($R_3$) (and other nested or composed CAAs) that belong to $CAA_1$. External object accesses 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. Each participant that is playing a 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 $CAA_1$.

An important consideration to take into account is about the objects that are passed to the nested CAA from the enclosing context (e.g., $O_1$). These objects are considered as *external* for the nested CAA, thus a *shared* object belonging to the enclosing CAA becomes *external* for the nested CAA. This shows that the terms *external* and *shared* are related to the CAA where they are used.

It is also possible that a nested CAA needs to have access to an external object that has not been held by its enclosing CAA. Moreover, a nested CAA may also create new objects ($O_4$) that are persistent after its completion. In any case, it is absolutely necessary to keep a trace on the accessed/created objects by the nested CAA and to pass this information onto the parent CAA. In this way, the enclosing CAA has all the information to leave the system in a safe state if recovery error measures are necessary [17].

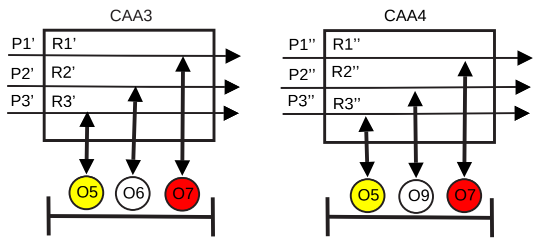


Fig. 3. Composite CAAs.

Composite CAAs [8] are different from nested CAAs in the sense that the use of composite CAAs is more flexible. For example, a nested CAA with two roles can only be used inside an enclosing CAA that is played by at least two participants. Composite CAAs do not have this type of restriction. A composite CAA ($CAA_3$) is an autonomous entity with its own roles (${R_1}'$, ${R_2}'$, and ${R_3}'$) and objects ($O_5$, $O_6$, and $O_7$). The internal structure of a composite CAA ($CAA_3$), i.e., participants, accessed objects, and roles, is hidden from its calling CAA ($CAA_1$).

A role belonging to $CAA_1$ that calls $CAA_3$ synchronously waits for the outcome. Then, the calling role resumes its execution according to the outcome of $CAA_3$. If $CAA_3$ terminates exceptionally, its calling role, which belongs to $CAA_1$, raises an internal exception that 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 $CAA_3$ has terminated with a normal outcome, but the containing $CAA_1$ has to undo its effects (BER has to take place in $CAA_1$), all the tasks that were executed in $CAA_3$ will not be automatically undone by BER in $CAA_1$. Thus, $CAA_1$ needs to carry out a specific handling in order to guarantee the ACID properties on the external objects. The specific handling may include a call to another composite CAA ($CAA_4$) to abort the effects that have been performed by $CAA_3$. Therefore, every time a composite CAA is being used inside a CAA, the *compensation* part of BER must be used. The compensation allows us to specify hand-made recovery during BER, for example to roll back something that a composite CAA has modified.

## 2.1 *Description of the CAAs behaviour*

In this Section we provide a precise description of the CAAs behaviour by means of statecharts [11,12]. This will avoid ambiguities, omissions, and contradictions in understanding the CAAs behaviour.

More precisely, the goal of this section is to describe how each possible kind of CAA outcome (i.e., **normal, exceptional, abort,** and **failure**) can be reached. Therefore, the formalisation provides a high level description of how the CAAs mechanism behaves internally when the service it provides is requested. Both participants and roles do not appear in this description since they have been hidden and abstracted in order to reach a simpler and more understandable description.

The specification is shown in Figure 4. It is composed of a big state called **Enclosing context** that initially is in the state *S0*. The enclosing context contains a CAA, which has been designed to provide a specific service. The CAA is called by an external user (it can be another system where the CAA is embedded). The invocation of the service is represented by the event *runCAA*. This event comes from the enclosing context. The state *Service* represents the execution of the service and it is reached once the *runCAA* has been emitted and the CAA pre-condition (represented by the *preCond* predicate) is true. If the service is able to satisfy its post-condition (*postCond* predicate is true), then the CAA terminates normally. Therefore, the CAA reaches the state *S1* emitting at the same time the *Normal* event. Otherwise, if the post-condition is not met or an exception is raised, the recovery process is started (going to state *Recovery*).

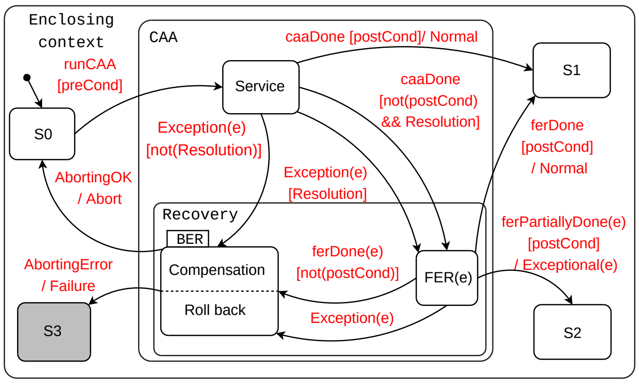

Enclosing context
runCAA [preCond]
S0
CAA
Service
caaDone [postCond]/ Normal
S1
caaDone [not(postCond) && Resolution]
Exception(e) [not(Resolution)]
AbortingOK / Abort
Exception(e) [Resolution]
ferDone [postCond] / Normal
Recovery
BER
Compensation
Roll back
ferDone(e) [not(postCond)]
FER(e)
ferPartiallyDone(e) [postCond] / Exceptional(e)
AbortingError / Failure
S3
Exception(e)
S2

Fig. 4. CAA behaviour.

If an exception is raised (*Exception(e)* event) during the normal execution of the CAA (state *Service*), then a process of exception handling is triggered (state *Recovery*). This exception handling process is defined as a combination of FER and BER. The first step of the exception handling process is the *exception resolution*, which consists of finding a common exception. In fact, due to the concurrent execution of the roles that takes place in the state *Service*, more than one exception could be raised at the same time. An algorithm is used to implement the *exception resolution*. If it succeed (*Resolution* predicate is true), the FER mechanism is started, otherwise the effects of the CAA have to be undone (the BER mechanism is triggered). Besides, every time the post-condition or the resolution do not hold an exception is raised. For example, if the event *caaDone* occurs and neither the *postCond* nor *Resolution* hold, then an exception will be raised and it will be dealt by the BER mechanism (see, for example, *Exception(e) [not (Resolution)]*).

The FER mechanism is represented by the state *FER(e)* and, depending on how successfully it can be executed, the CAA may still terminate normally. The FER finishes normally if it fulfils the original request and the post-condition (represented by *postCond* predicate) is met. Therefore, the state *S1* is reached. Otherwise, if *FER(e)* satisfies the post-condition but the result that FER provides to the enclosing context is partial (or degraded) with respect to the original request (*ferPartiallyDone(e)* event), the CAA finishes exceptionally. Notice that even if the CAA service did not execute according to its specification (to leave the enclosing context in state *S1*), the enclosing context is left in a specified state (*S2*).

Finally, if the post-condition cannot be satisfied by FER or other exceptions have been raised again, the CAA must roll back using BER (state *BER*). If BER is applied successfully (*AbortingOk* event), the CAA publishes the event *Abort* and the enclosing context reaches the same state (*S0*) where it was before calling the CAA service. If BER is unsuccessful (*AbortingError* event) then the CAA must emit the *Failure* event. In this case the enclosing context is left in the state *S3*.

## 3 DRIP and CAA-DRIP frameworks

CAA is a conceptual framework since it establishes concepts and relationships along with their respective semantics. It has been conceived to engineer dependable distributed and concurrent systems. However, considering the fact that it is only a *conceptual* framework, the implementation phase is fully relied on the programmers' knowledge and experience. In order to facilitate the programmers' task, two frameworks have been developed: DRIP [4] and CAA-DRIP [22].

Actually, the CAA-DRIP has been developed using as starting point the DRIP framework. The main improvement brought by CAA-DRIP is that it provides support only for the CAA concepts and behaviour described in Section 2. Therefore, it allows programmers to match every CAA concept with its respective piece of code that implements it. As mentioned in the introduction, DRIP, on the other hand, was developed to support a different abstraction, i.e. DMIs [18].

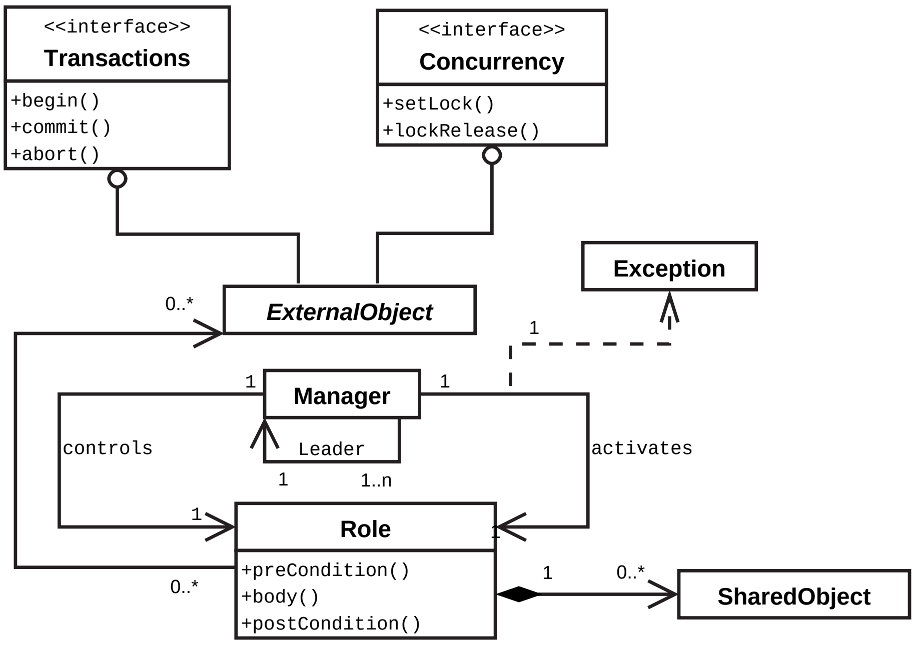


Fig. 5. DRIP UML model

Both frameworks were designed as a set of *Java* [3] classes and interfaces. The instantiation of these classes and the gluing of them together will create the needed CAAs (a CAA is represented as a set of classes). The models in Figures 5 and 6 show the DRIP and CAA-DRIP UML class diagram, respectively. On both frameworks, the *Manager* class is the controller for the *Role* class and has references to external and shared objects. The manager object that is chosen as the *leader* [2] is the responsible for synchronising roles upon entry and upon exit, execution of the exception resolution algorithm and for keeping information about shared objects. Each framework has a set of roles that are responsible for executing the normal behaviour of the CAA. The main difference between the two frameworks resides on the way they handle exceptions.

The DRIP framework deals with exceptions in the following way (remember that DRIP was built for a different abstraction). All exceptions that are raised inside the roles that execute the normal behaviour of the CAAs will be handled by different sets of managers and roles. More precisely, each exception is

[2] A leader is set to avoid the need of distributed algorithms for all synchronisations. If the leader fails, the set of managers can apply an election algorithm to decide which one can act as a new leader. Several different algorithms have already been proposed for this problem and we will not discuss them in the remaining of this paper.

handled by a different set of managers and roles. If another exception is raised during the handling of an exception, then a roll back action has to be performed (see Section 2). Using DRIP this is possible, but a new set of roles and managers have to be created to perform this activity. The programmer has to develop this new set of roles and managers, along with the code to recover all external objects, and to signal the right result to the enclosing action (*abort* if external objects were recovered, or *failure* if they were not). On Figure 5, the arrow *activates* represents the managers for the normal behaviour activating those roles that will perform the FER or the BER, when an exception takes place.

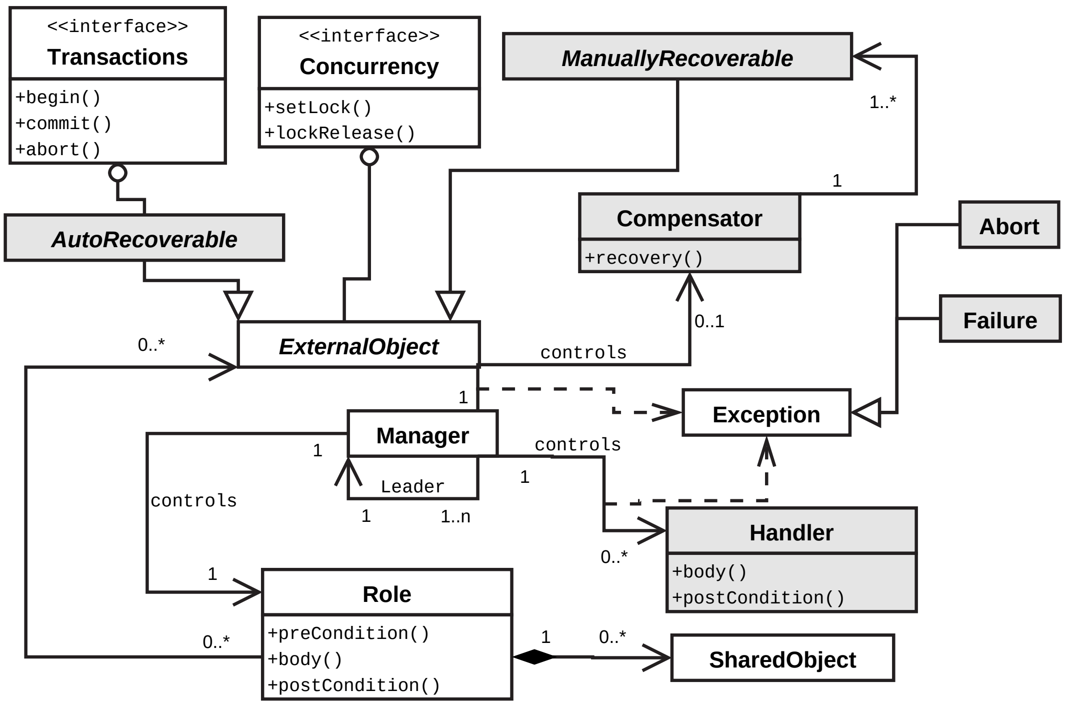


Fig. 6. CAA-DRIP UML model

As mentioned before, to leave some decisions to the programmer could result in a final system that could not respect the correct semantics of CAAs. Furthermore, the recovery of external objects that provide such facility could be performed automatically, alleviating the programmer of such task. The signalling of the correct result, when rolling back, could also be performed automatically.

The CAA-DRIP framework is implemented in a way that these problems are avoided. As can be seen in Figure 6, the CAA-DRIP framework provides some extra classes: *Handler, Compensator, ManuallyRecoverable,* and *AutoRecoverable.* The objects of these classes are also managed by a manager object. Using CAA-DRIP, a CAA consists of a set of managers, roles, handlers and compensators linked together via a *leader* manager (represented as an *association* in Figure 6).

Basically, the new classes were developed to handle exceptions using the same concepts as those of CAAs. Therefore, each handler consists of a set of roles to handle one of the possible exceptions that are raised during the normal execution of a CAA. If an exception is raised during the execution of a handler, then a compensator might be called. The compensator is needed only when the application being developed uses *ManuallyRecoverable* objects. If that is the case, then the programmer has to implement the manual recovery of these objects and the compensator is used for that. If the application being developed uses only *AutoRecoverable* objects, then a compensator is not needed because the managers will recover them automatically and will signal the correct exception, i.e. *abort* or *failure*.

## *3.1 Instantiating a CAA*

This section shows how to instantiate a CAA using DRIP and CAA-DRIP. The simple CAA from Figure 2 is taken as an example to show how both frameworks have to be employed to achieve its implementation. We consider that only one exception will be dealt with, i.e. *E1* exception. We also consider that only *AutoRecoverable* objects are used in this simple CAA.

Figure 7 shows the code for instantiating classes using DRIP. In this figure, three sets of managers and roles are created. The first set (lines 1-9) is responsible for recovering the CAA when another exception is raised during the handling of E1. The second set (lines 12-26) is responsible for handling the exception E1. The third set (lines 29-42) is responsible for the normal execution of the CAA. In all three sets, firstly the managers are created and then the roles are created and associated with the manager that will control each role. The manager that will act as leader is informed during the creation of a role. Notice that in lines 12-16 three hashtables are created. These hashtables contain the topmost exception in the *Java* language, i.e. *Exception* class, and the role for the first set. This role will be called when an exception is raised inside the handler for exception E1. Each hashtable will be passed as parameter during the creation of the managers for the handler of E1. The same process is performed for associating the handler for exception E1 with roles that are created for the normal execution of the CAA (see lines 29-32).

Notice, again, that the roll back is constructed in the same way as a handler for an exception raised during the normal execution is constructed. Therefore, there is not any impediment for the programmer to construct another handler for an exception raised during the roll back process. However, this would violate the CAAs semantics, as explained in Section 2.

Figure 8 shows the code for instantiating classes using CAA-DRIP. Basically,

**Set 1**

```
1  //Managers for the roll-back interaction
2  mgr1RB = new ManagerImpl("mgr1RB","CAA1");
3  mgr2RB = new ManagerImpl("mgr2RB","CAA1");
4  mgr3RB = new ManagerImpl("mgr3RB","CAA1");
5
6  //Roles for the roll-back interaction
7  role1RB = new RB1("role1RB",mgr1RB,mgr1RB);
8  role2RB = new RB2("role2RB",mgr2RB,mgr1RB);
9  role3RB = new RB3("role3RB",mgr3RB,mgr1RB);
10 //
```

**Set 2**

```
11
12 //Hashtables to roll-back the CAA when any exception
13 //is raised inside the handling phase
14 Hashtable rb1 = new Hashtable(); rb1.put(Exception.class,role1RB);
15 Hashtable rb2 = new Hashtable(); rb2.put(Exception.class,role2RB);
16 Hashtable rb3 = new Hashtable(); rb3.put(Exception.class,role3RB);
17
18 //Managers to deal with exception E1
19 mgr1E1 = new ManagerImpl("mgr1E1","CAA1",rb1);
20 mgr2E1 = new ManagerImpl("mgr2E1","CAA1",rb2);
21 mgr3E1 = new ManagerImpl("mgr3E1","CAA1",rb3);
22
23 //Roles for the roll-back interaction
24 role1E1 = new R1E1("role1E1",mgr1E1,mgr1E1);
25 role2E1 = new R2E1("role2E1",mgr2E2,mgr1E1);
26 role3E1 = new R3E1("role3E1",mgr3E3,mgr1E1);
27 //
```

**Set 3**

```
28
29 //Hashtables for exceptions raised inside the normal execution
30 Hashtable e1 = new Hashtable(); e1.put(E1.class,role1E1);
31 Hashtable e2 = new Hashtable(); e2.put(E1.class,role2E1);
32 Hashtable e3 = new Hashtable(); e3.put(E1.class,role3E1);
33
34 //Managers for the normal behaviour phase
35 mgr1 = new ManagerImpl("mgr1","CAA1",e1);
36 mgr2 = new ManagerImpl("mgr2","CAA1",e2);
37 mgr3 = new ManagerImpl("mgr3","CAA1",e3);
38
39 //Roles for the normal behaviour phase
40 role1 = new R1("role1",mgr1,mgr1);
41 role2 = new R2("role2",mgr2,mgr1);
42 role3 = new R3("role3",mgr3,mgr1);
```

Fig. 7. CAA1 definition using DRIP

all CAA concepts are used during the construction of a CAA using the CAA-DRIP framework. As a first step, the normal part of the CAA is declared (lines 1-9). Manager objects and their respective roles are created in the same way they are created using the DRIP framework. The second step refers to the definition of the objects that will handle exceptions. The difference between the frameworks is on the way these objects are created. Lines 11-14 show how a handler for exception E1 is created. Notice that the same managers are used to control the handlers. In the DRIP framework, new managers would be necessary to control the handler for each new exception (see Figure 7, lines

**Normal Behaviour**

```
1  //Managers
2  mgr1 = new ManagerImpl("mgr1","CAA1");
3  mgr2 = new ManagerImpl("mgr2","CAA1");
4  mgr3 = new ManagerImpl("mgr3","CAA1");
5
6  //Roles
7  role1 = new R1("role1",mgr1,mgr1);
8  role2 = new R2("role2",mgr2,mgr1);
9  role3 = new R3("role3",mgr3,mgr1);
10 //
```

**Exceptional Behaviour**

```
11 //Handlers
12 hndrE1_R1 = new E1_R1("hndrE1_R1",mgr1);
13 hndrE1_R2 = new E1_R2("hndrE1_R2",mgr2);
14 hndrE1_R3 = new E1_R3("hndrE1_R3",mgr3);
15 //
```

**Binding of Handlers and Managers**

```
16 //Binding Exception-Handler
17 Hashtable ehR1 = new Hashtable();ehR1.put(E1.class,hndrE1_R1);
18 Hashtable ehR2 = new Hashtable(); ehR2.put(E1.class,hndrE1_R2);
19 Hashtable ehR3 = new Hashtable(); ehR3.put(E1.class,hndrE1_R3);
20
21 //Binding Exception-Handler-Manager
22 mgr1.setExceptionAndHandlerList(ehR1);
23 mgr2.setExceptionAndHandlerList(ehR2);
24 mgr3.setExceptionAndHandlerList(ehR3);
```

Fig. 8. CAA1 definition using CAA-DRIP

19-21). The third step to construct the CAA using CAA-DRIP is to link the managers with the handlers and exceptions that will be handled during the normal execution of the CAA (lines 16-24).

The final step for creating a CAA using CAA-DRIP would be the declaration of compensator. In our example, this would not be needed because we assume that the objects are all *AutoRecoverable*, and therefore the framework activates the object recovery. Figure 9 shows how a compensator would be created (for the same example) if at least one of the objects is *ManuallyRecoverable*.

```
25
26 //Compensator
27 cmp1 = new CmpR1("cmp1",mgr1);
28 cmp2 = new CmpR2("cmp2",mgr2);
29 cmp3 = new CmpR3("cmp3",mgr3);
```

Fig. 9. A compensator definition for CAA1 using CAA-DRIP

### *3.2 Defining and Implementing Roles, Handlers, and Shared Objects*

This section presents how to implement the application related code for each role. Both frameworks use the same approach and classes for creating roles and shared objects.

The definition of a role is made by creating a new class that extends the *Role* class (see the *Java* code in Figure 10 - this code is valid for both frameworks). The programmer has to re-implement the *body* method (lines 24-32). This method receives a list of external objects as input parameter and it does not return any value. The defined operations inside this method are executed by the participant that activated the role.

```
1  public class RoleName extends RoleImpl {
2     //defining shared object
3     SharedObject so;
4
5     public RoleName(String roleName, Manager mgr,Manager leader)
6       throws RemoteException {
7        //set role with name, manager and leader
8        super (roleName,mgr,leader);
9
10       //creating shared object
11       so = new SharedObject();
12
13       //exporting shared object
14       mgr.sharedObject("soName",so);
15    }
16
17    public boolean preCondition(ExternalObjects eos)
18      throws Exception,RemoteException {
19       boolean guard;
20       // checking pre-condition
21       return guard;
22    }
23
24    public void body(ExternalObjects eos)
25      throws Exception,RemoteException,
26             InterruptedException{
27       try{
28           //here goes the code to be executed by the role
29       } catch (Exception e) {
30           //here goes the code for handling local exceptions
31       }
32    }
33
34    public boolean postCondition(ExternalObjects eos)
35      throws Exception,RemoteException {
36       boolean assertion;
37       // checking post-condition
38       return assertion;
39    }
40 }
```

Fig. 10. Creating a new Role class

When the role class is instantiated, the role constructor needs the role name, the manager object that drives its execution and the leader manager object used for coordinating the CAA execution (lines 5-15 of Figure 10 for the declaration, and lines 7-9 of Figure 8, for instance, for the instantiation).

Coming back to Figure 10, shared objects for coordinating the CAA roles are defined inside the new class that extends the *Role* class (line 3). Once a shared object has been created (line 11) it can be exported to be used by other roles of the CAA. In order to export a shared object the programmer has to use the *sharedObject* method of the *Manager* class (line 14). The manager has this information so other roles can ask their managers for a reference to these objects.

The other methods that have also to be redefined by the programmer are *preCondition* (lines 17-22) and *postCondition* (lines 34-39). They return a *Boolean* value and are used as guard and assertion of the role, respectively. This set of new classes defines the CAA normal behaviour and their execution corresponds to state *Service* in Figure 4.

The second step is to define the CAA behaviour for dealing with exceptions (state *Recovery* in Figure 4). This is, from an implementation point of view, different from the DRIP framework. If an exception has to be handled by FER, then it is necessary to define a handler for each CAA role. In the DRIP framework this is performed by creating a new *Role* class in the same way a *Role* class is created for the normal execution.

In the CAA-DRIP framework, on the other hand, the new handler has to be created similarly to the way a new *Role* class is created, but instead of extending the *Role* class, the new class has to extend the *Handler* class. Line 1 from Figure 10 changes to "**public class** HandlerName **extends** HandlerImpl". The rest of the code is similar to the way a new *Role* class is created.

The last step is to create a roll back process. In the DRIP framework the programmer has to create a new *Role* class in the same way he creates a role for dealing with exceptions. In CAA-DRIP, the programmer has to create a new *Compensator* class, when *ManuallyRecoverable* objects are used in the CAA. Again, in CAA-DRIP, the differences are: a) the new class has to extend the *Compensator* class instead of extending the *Role* class or the *Handler* class; and, b) the name of the method that has to be re-written is called *recovery* instead of *body*. This method receives, as input parameter, a list with the external objects that need hand-made recovery. The method has to contain the operations to leave these external objects in a consistent state (to keep the ACID properties).

## *3.3 Managing a CAA at runtime*

So far we have shown how the classes in the framework are instantiated and extended to create a CAA. Now we explain how these objects behave when the CAA is activated. The CAA activation process begins when each participant starts the role that it wants to play. The *execute* method (belonging to the *Role* class) has to be used by a participant to start playing a role. When the *execute* method is called, the role passes the control to its manager by executing *controller* method (which belongs to the *Manager* class -see Figure 11). The execution of such a method implies the activation of the manager. Once each manager has been activated the CAA starts the execution of its life cycle.

```java
protected void controller(Role role, Object list[]) throws Exception {
  ...
  // create a new thread to execute the role managed by this manager
  executingThread = new Thread(this,Thread.currentThread().getName());
  // start the execution of the role
  executingThread.start();
  ...
}
```

Fig. 11. Activation the manager

The CAA life cycle is coded in the *Manager* class as a sequence of operations that it executes after its activation. The *Java* code on Figure 12 shows these operations. This figure is divided in different parts according to the framework employed. The first part corresponds to the steps that are common for both frameworks and which are used to perform the service that the CAA provides (i.e. normal behaviour - Figure 13) shows a sequence diagram for the normal behaviour of a CAA). The other parts correspond to the recovery phase. This phase is coded in different ways in DRIP and CAA-DRIP. In the following we explain in details those steps that are common to both framework, as well as those that are specific for each of them.

### *3.3.1 Common part: Normal Behaviour Execution*

The first activity a manager executes is to synchronise itself with all other managers that are taking place in the CAA. This is done by calling the *syncBegin* method (line 3). Remember that there is a leader manager that is responsible for this task. This method blocks until the *leader* determines that all the managers have synchronised and the CAA is ready to begin. Once the *syncBegin* method returns, the manager checks if the pre-condition of the role is valid (line 5). The *preCondition* method receives all the external objects that will be passed to the role managed by this manager as parameters. If the pre-condition is not satisfied, then a *PreConditionException* will be thrown (line 6) and caught by the *catch(Exception e)* block.

If the pre-condition is met, then the manager will execute the role that is under its control by calling the *bodyExecute* method of the *Role* object (line 9). The invocation of this method by each manager can be seen as the implementation of the arrow that goes from state *S0* to state *Service* in Figure 4.

After the role has finished its execution, the manager tests the CAA post-condition (line 12). If the post-conditions is satisfied, then the manager, once again, synchronises with all the other managers (line 20) to finish the CAA

**Common code**

```
 1 try{
 2       // synchronising upon entry
 3       syncBegin();
 4       // if pre-condition is not true
 5       if(!roleManaged.preCondition(extObjs))
 6          throw new PreConditionException();
 7       // executing the role
 8
 9       roleManaged.bodyExecute(this,extObjs);
10
11       // if post-condition is not true
12       if(!roleManaged.postCondition(extObjs))
13          throw new PostConditionException();
14       // exiting synchronously
15       syncEnd();
16 }catch (Exception exRole) {
17       try{
18          // applying exception resolution algorithm
19          exRole = exceptionResolution(exRole);
```

**Code in the DRIP framework**

```
20          // launching role to handle the found exception.
21          Role handlerRole = (Role)exceptionRoleList.get(exRole.getClass());
22          handlerRole.execute(listOfParameters);
23     } catch (Exception ex) {
24          roleException = ex;
25     }
26 }
```

**Code in the CAA-DRIP framework**

```
20          // launching FER for the found exception.
21          Handler handler = (Handler)exceptionHndList.get(exRole.getClass());
22          handlerExecution();
23       }catch (Exception exFER) { //BER
24          try{
25             // executing compensation and roll back
26             restoreExecution();
27             // returning ABORT
28             roleException = new AbortException();
29          }catch(Exception exBER){
30             // there was a problem in the BER execution
31             // returning FAILURE
32             roleException = new FailureException();
33          }
34       }
35 }
```

Fig. 12. Manager execution in DRIP and CAA-DRIP

execution synchronously. Thus, this sequence of steps corresponds to the arrow that goes from state *Service* to state *S1* in Figure 4 and which represents the success in the CAA execution.

The *catch(Exception e)* block (from line 16 to 26, if using DRIP; or until 35 if using CAA-DRIP) is executed when an exception is raised during the execution of any role belonging to the CAA. In such situation, the role where the exception was raised notifies its manager. This manager passes the control

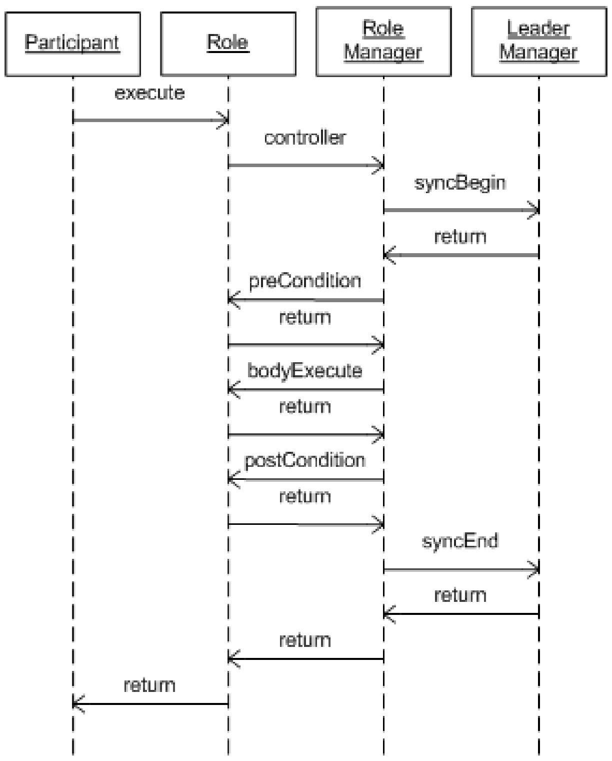


Fig. 13. Sequence diagram for the normal behaviour of a CAA

to the *leader* manager for interrupting[3] all the other roles that have not raised an exception (exceptions can be raised concurrently). Once all the roles have been interrupted the *leader* executes an exception resolution algorithm to find a common exception[4] from those that have been raised (line 19). When such exception is found, the *leader* informs all managers about it. The recovery phase for such exception then has to be started.

### *3.3.2 Recovery phase in the DRIP framework*

The recovery phase performed by DRIP consists of the activation of the roles created to act as handler. Each handler deals with the exception returned by the resolution algorithm. Thus, the first step is to look for the role object that has been defined (if any) to deal with the exception returned by the resolution algorithm (line 21). If the role exists, then it is activated and executed. As mentioned above, the activation of the role starts when the *execute* method is called (line 22). Through the execution of this method, the *controller* method (see Figure 11) is performed, producing in the end the execution of the role that behaves as handler.

[3] Notice that a role will be interrupted only if the role is ready to be interrupted, i.e. the role is in a state in which it can be interrupted.
[4] In the worst case, the common exception is *Exception*.

The set of roles behaving as handlers will perform the FER phase, since their actions can still allow the CAA to finish successfully. However, there are certain situations that will lead to the execution of the BER phase. Thus, BER is executed if[5]:

- the exception resolution algorithm could not find a common exception (arrow that goes from the state *Service* to the state *BER* on Figure 4);
- the post-condition associated to the role[6] is not met (arrow that goes from the state *FER(e)* to the state *BER* with label *ferDone(e)[not(postCond)]* on Figure 4), which causes the exception *PostConditionException()* to be raised[7];
- other exceptions were raised in the FER (arrow that goes from the state *FER(e)* to the state *BER* with the label *Exception(e)* on Figure 4).

The BER phase is defined in the same way as the FER. It means that a set of roles for such a purpose has to be created by the programmer to undo the effects produced by the CAA. Thus, when one of the above-mentioned situations occurs, the roll back of the CAA will be performed by activating and executing such roles. The activation and execution are exactly the same as for the FER phase. Therefore, the DRIP framework makes possible to have as many handling phases as the programmers want. However, according to CAA semantics only a two-level nesting is allowed: the first one to perform the FER phase, and the second one to perform the BER phase.

Figure 14 shows a sequence diagram for the DRIP framework when one exception is raised inside a role of a CAA.

### *3.3.3 Recovery phase in the CAA-DRIP framework*

CAA-DRIP starts the recovery phase by executing the FER phase. In this phase the first step is to look for the handler associated to the exception to deal with (line 21). Once the handler was found, the *handlerExecution* method (line 22) is called. This method executes the specific behaviour coded in the *body* method of the handler and then it checks if the handler's post-condition holds. If the post-condition is held, then the CAA finishes successfully.

The value in *roleException* variable defines how successfully the FER execution was. A *null* value in this variable means that FER has finished normally

[5] These situations will also lead to the execution of the BER phase when CAA-DRIP framework is employed

[6] the precondition for a role acting as a handler must be *true*, so that in CAA-DRIP there not exist a way to define a precondition for a handler

[7] Using the CAA-DRIP framework this exception would be raised by the *handlerExecution()* method

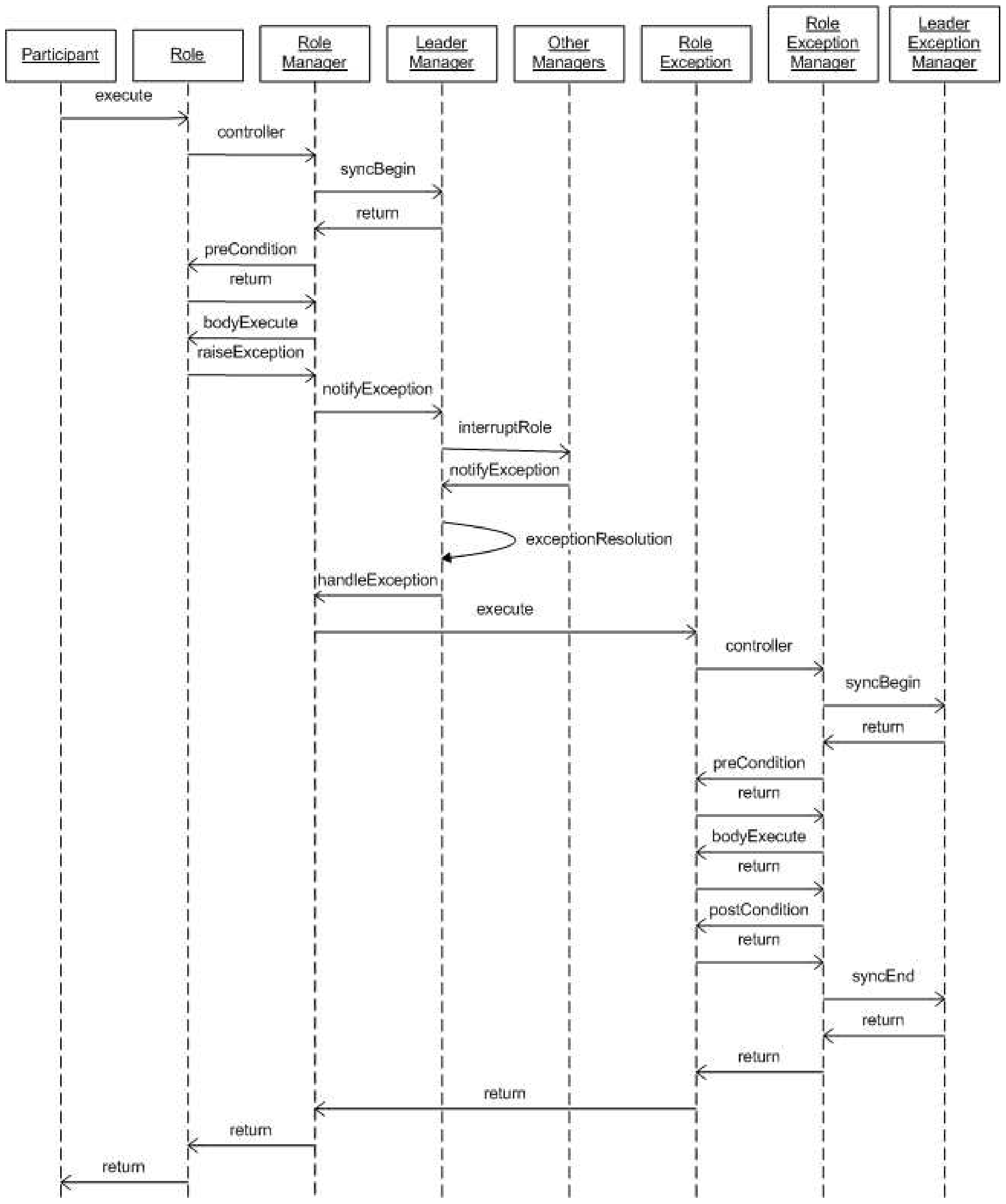


Fig. 14. Sequence diagram of an exceptional behaviour of a CAA (DRIP framework)

(arrow that goes from the state *FER* to the state *S1* in Figure 4). Otherwise, it has achieved a partial solution and the exceptional result (contained in the *roleException* variable) has to be returned to the enclosing context (arrow that goes from state *FER* to state *S2* in Figure 4).

The BER mechanism (lines 25-28) is responsible for calling the *restoreExecution* method (line 26) to undo the CAA effects. Once this method has been executed, the *roleException* variable is set with *Abort* value (line 28) and the CAA can finish (arrow that goes from state *BER* to *S0* in Figure 4).

If for any reason the BER process could not complete its execution, the CAA will be finished returning *Failure* (line 32). This statement corresponds to the arrow that goes from state *BER* to state *S3* in Figure 4.

Figure 15 shows a sequence diagram for the CAA-DRIP framework when one exception is raised inside a role of a CAA.

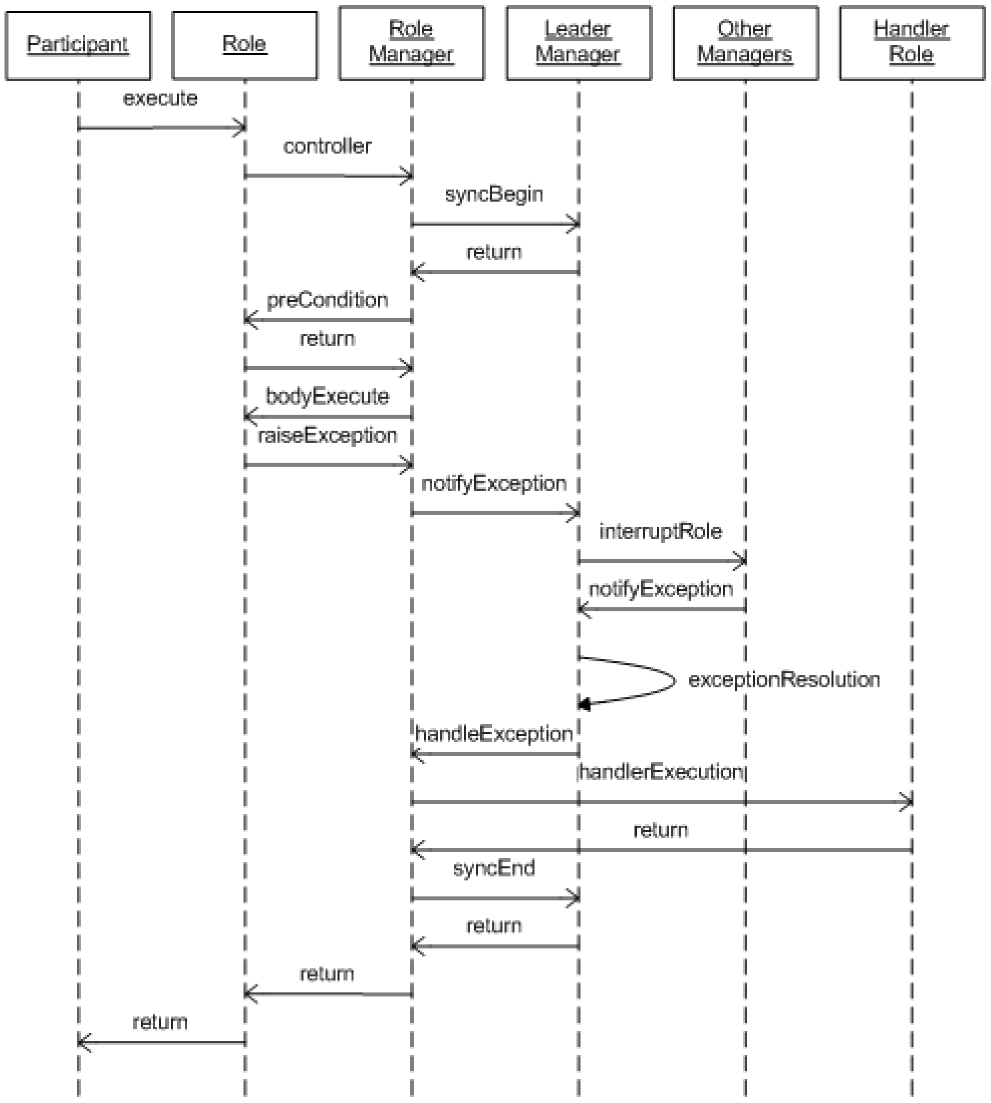


Fig. 15. Sequence diagram of an exceptional behaviour of a CAA (CAA-DRIP framework)

### *3.4 Comparison between DRIP and CAA-DRIP*

The aim at developing CAA-DRIP was to ease the problem of implementing designs based on CAAs. Thus, the former implementation support to carry out this step, i.e., DRIP, was fully analysed to figure out its main weaknesses from both a qualitative and quantitative point of view. This section then first summarises the main differences between DRIP and CAA-DRIP, and secondly, shows what are the consequences of such differences in terms of execution time and memory consumption.

The first difference appears in the development process. CAA-DRIP allows programmers to deal with the same conceptual concepts (i.e., *Role, Handler, Compensator*, etc.) used in the design phase, whereas DRIP either directly does not support some CAA concepts (e.g., *Compensator*) or it is implemented in a way that does not match CAA concepts of the design phase (e.g., a *Handler* is implemented in the same way as a *Role* is). Having such a direct correlation between design and implementation eases the code production. Furthermore, it helps also the maintainability of the code as well as the inspection of the code to detect defects (testing).

Another difference between both frameworks is related to the number of objects to be instantiated for creating a CAA. Let *caa* be a CAA composed of $n$ roles which have to handle $m$ exceptions without any external object to be recovered manually, CAA-DRIP requires "$2 * n + m * n$" objects, whereas DRIP requires "$2 * n + m * (2 * n) + 2 * n$" objects. In case there exist at least one external object which needs manual recovery, by using CAA-DRIP the number of objects is "$2 * n + m * n + n$", whereas by using DRIP the number remains the same.

Therefore, assuming the worst possible scenario (i.e., manual recovery must be performed) CAA-DRIP requires "$n * (3 + m)$" objects, and DRIP "$2 * n * (2 + m)$". When $n$ and $m$ grow large, the $(3 + m)$ and $(2 + m)$ terms have a very close value, turning the object rate between CAA-DRIP and DRIP to $n/2 * n$ , i.e. 1/2. Therefore, CAA-DRIP requires half number of objects than DRIP to create a CAA. Most of the cut off objects were manager objects. This is due to the fact that in CAA-DRIP the same manager object can control roles, handlers and compensators, whereas in DRIP one set of managers for each set of handlers and compensators would be necessary.

However, the main quantitative advantage of this reduction of the number of manager objects consists in considerably decreasing the number of threads created for executing a CAA (three times less, in the worst case). The *Manager* class has been developed to work as a supervisor in the execution of objects extending from Role, Handler, and Compensator classes. Thus, at runtime, the place (i.e., thread) to carry out the computation (i.e. code programmed into the *body* method) of such instances is created by a manager object. Therefore, as the number of manager objects created at runtime has been considerably decreased, then the number of threads to be created for executing role, handler and compensator instances is also much less. All these aspects are summarised in Table 1.

| **Characteristics/Framework** | **DRIP** | **CAA-DRIP** |
|---|---|---|
| Supported abstractions | Role | Role, Handler, Compensator; *Abort* and *Failure* exceptions |
| Pre-defined outcomes | None | *Abort* and *Failure* |
| Normal behaviour implementation | Instance of *Manager* class and extension of *Role* class | Instance of *Manager* class and extension of *Role* class |
| FER behaviour implementation | Instance of *Manager* class and extension of *Role* class | Extending *Handler* class |
| BER behaviour implementation | Instance of *Manager* class and extension of *Role* class | Combining automatic roll back and hand-made recovery (extending *Compensator* class) |
| Number of objects to create a CAA with $n$ roles, handling $m$ exceptions without manual recovery of external objects | $2*n+m*(2*n)+2*n$ | $2*n+m*n$ |
| Number of objects to create a CAA with $n$ roles, handling $m$ exceptions with manual recovery of external objects | $2*n+m*(2*n)+2*n$ | $2*n+m*n+n$ |
| Number of threads created at runtime to run a CAA with $n$ roles, which deals with an exception, first by FER and then by BER | $3*n$ | $n$ |

Table 1
DRIP vs. CAA-DRIP

Performance evaluations regarding response time and memory consumption were performed to see the impact of the differences between DRIP and CAA-DRIP. The experiments consisted of measuring (1) the elapsed time and (2) the memory use of a CAA implemented both with CAA-DRIP and DRIP frameworks.

The CAA used to run such experiments was intentionally defined very simple to avoid introducing extra overheads, since the goal is to measure the frameworks' overheads. Therefore, the role played for each participant entering into the CAA is a very simple arithmetical operation which takes too slight computation time. Moreover, there is not interaction among roles, which makes the CAA execution even lighter and faster.

Let $Caa_{p_i}$ represent this simple CAA with $p_i$ participants entering into it. Measures of the elapsed time and the memory used[8] were taken for $Caa_{p_i}$ with $p_i$ going from 100 to 3000 and step 100. Both for measuring the timing and memory costs, each $Caa_{p_i}$ was run 50 times under the same hypothesis and environment conditions to avoid introducing random measuring errors. These values were used to calculate the arithmetic mean (here denoted by AVG) of each $Caa_{p_i}$ regarding elapsed time and memory used.

The measures for each $Caa_{p_i}$ were performed over two representative cases: the normal behaviour case and the exceptional behaviour case. The normal behaviour case represents the scenario where the $Caa_{p_i}$ reaches its goal without executing the recovery phase. The exceptional behaviour case represents the execution of the same $Caa_{p_i}$, but where an exception is raised by one arbitrary role. The tasks to be performed for each handler during the recovery phase are the same as the ones performed by the roles in the normal phase.

Therefore, the graphs shown on Figure 16 and Figure 17 are the interpolation of

$$AVG_{p_i} = \overline{Time(Caa_{p_i})} = \tfrac{1}{50}\sum_{j=1}^{50} Time_j(Caa_{p_i}),$$

with $p_i = [100, 3000]$ and step 100 for the normal and exceptional behaviour of $Caa_{p_i}$, respectively, regarding timing costs (in milliseconds).

On the other hand, the graphs on Figure 18 and Figure 19 are the interpolation of

$$AVG_{p_i} = \overline{Memory(Caa_{p_i})} = \tfrac{1}{50}\sum_{j=1}^{50} Memory_j(Caa_{p_i}),$$

[8] The elapsed time was taken using the Linux/Unix command **time**, whereas for measuring the memory used the command employed was **top**.

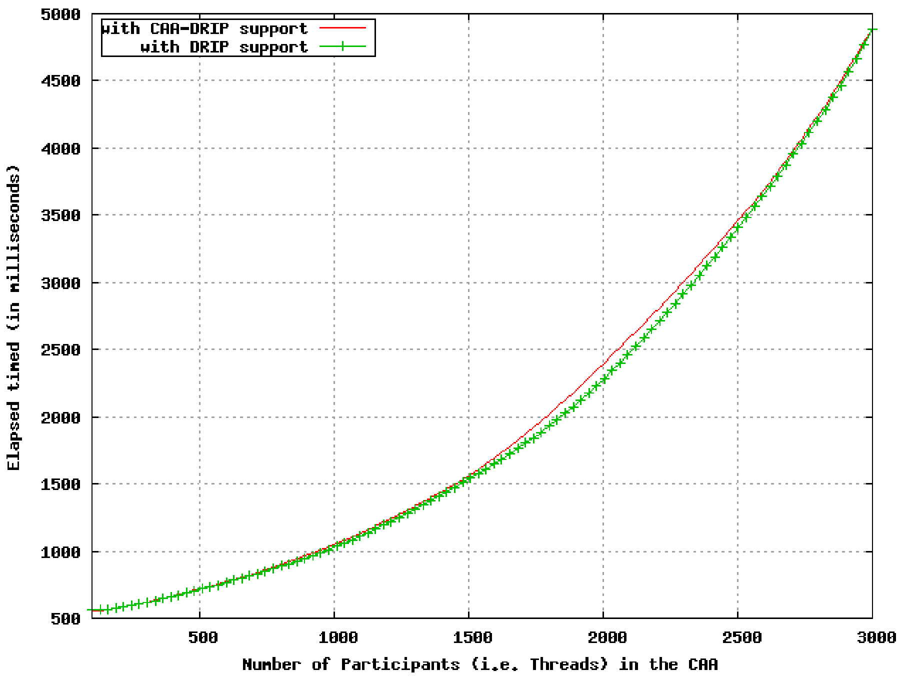


Fig. 16. DRIP vs. CAA-DRIP: time costs for Normal behaviour

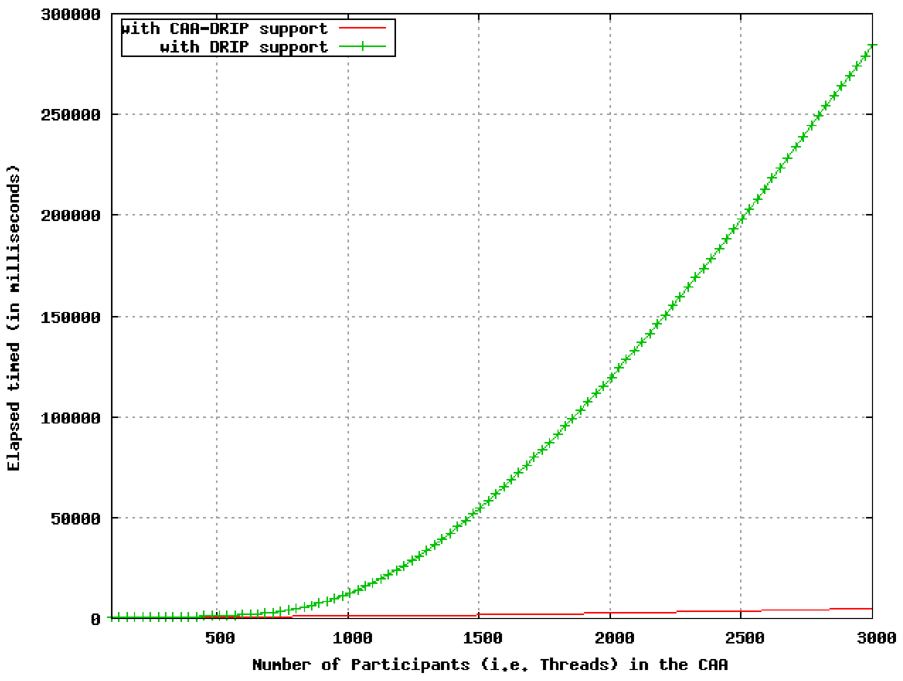


Fig. 17. DRIP vs. CAA-DRIP: time costs for Exceptional behaviour

with $p_i = [100, 3000]$ and step 100 for the normal and exceptional behaviour of $Caa_{p_i}$, respectively, regarding memory costs. On the Appendix can be found the average (AVG) of each $Caa_{p_i}$ regarding timing and memory costs along with their respective standard deviation (SD) and confidence interval (CI) for a confidence of 95%.

It is worth saying that the memory measurements (in Mb.) include both the amount of physical memory assigned to executable code, also known as the "text resident set" size and the amount of physical memory assigned to the "data resident set" [9] .

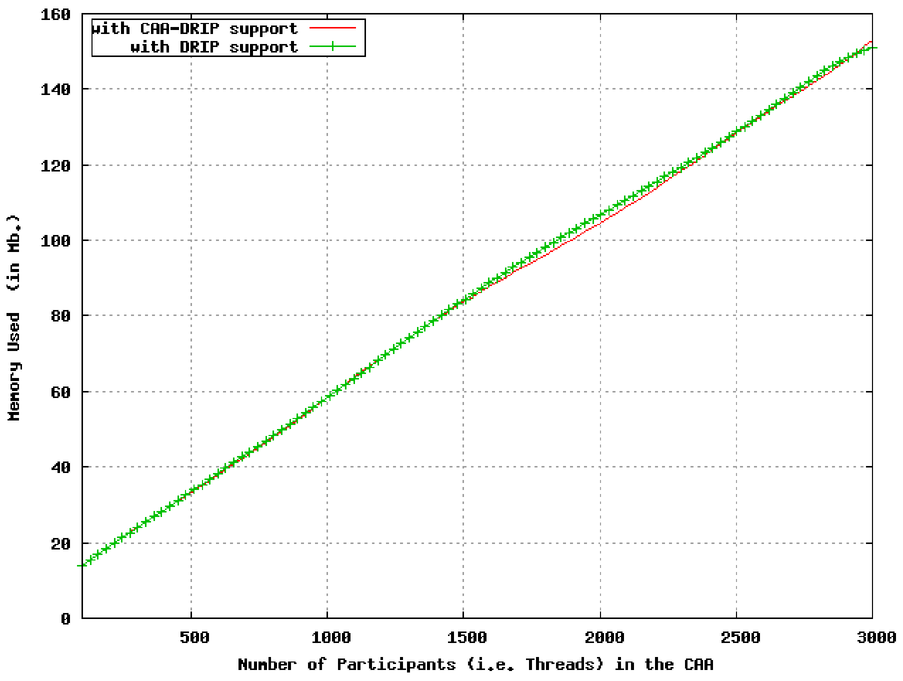


Fig. 18. DRIP vs. CAA-DRIP: memory costs for Normal behaviour

The experiments regarding costs in terms of response time reflect that the difference between the frameworks is negligible when the normal behaviour case is considered (Figure 16). However, when the exceptional behaviour case is considered, the cost for executing the $Caa_{p_i}$ using DRIP increases considerably, whereas it remains quite similar when the $Caa_{p_i}$ is executed using CAA-DRIP (Figure 17). The fact that CAA-DRIP shows similar response times in both cases (i.e., normal and exceptional) means that the overhead introduced by the context switch from the normal to the exceptional phase is negligible.

In respect of memory consumption costs, experiments show a better performance of CAA-DRIP compared with DRIP. This is also a consequence of having reduced the number of threads at runtime during the recovery phase. Thus, it is not surprising that there are almost no differences between the frameworks, when the normal behaviour case is considered (Figure 18). Therefore, the memory performance improvements achieved by CAA-DRIP come out when the exceptional behaviour case is considered (Figure 19).

[9] The JVM was tuned (i.e. heap of 2Gb. and thread stack of 128Kb) to allow the complete execution be run without need of using virtual memory, so that the non-swapped physical memory a task has used (RES) reported by the command top is an accurate value for which was looked for.

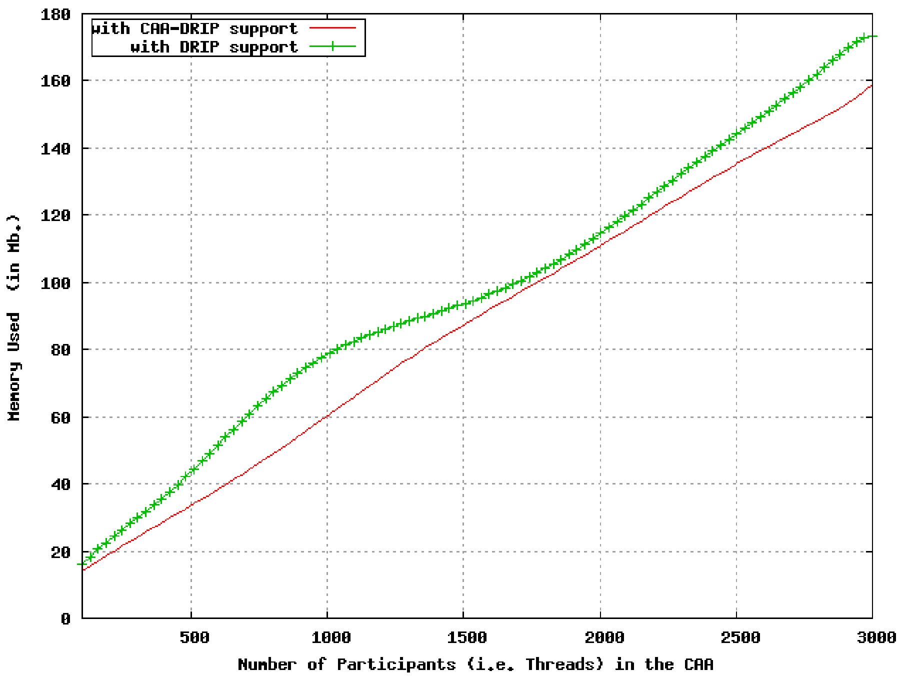


Fig. 19. DRIP vs. CAA-DRIP: memory costs for Exceptional behaviour

In conclusion, CAA-DRIP is not only faster (in terms of response time), but also lighter (in terms of memory consumption) compared to DRIP.

The experiments were performed on a Intel Pentium D-3.2 GHz, with 4 GB DDR-II of memory RAM, running Debian GNU/Linux 4.0. The Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_14-b03) Java HotSpot(TM) Client VM (build 1.5.0_14-b03, mixed mode, sharing) was used as virtual machine.

## 4 The Fault-Tolerant Insulin Pump (FTIP) System

This section presents a complete solution of an application on the e-health domain. This application was designed using the CAA concepts and implemented using the CAA-DRIP framework.

The aim of this section is to put in practice what was presented until now, explaining how to design and implement dependable systems. The case study is designed using CAAs, while the implementation is performed only by means of the CAA-DRIP framework. The previous comparative study demonstrated that CAA-DRIP is the better solution for implementing systems designed by using CAAs and thus it is the implementation framework that we chose.

The “Fault-Tolerant Insulin Pump” (FTIP) therapy is based on the Continuous Subcutaneous Insulin Injection technique that combines devices (a sensor and a pump) and software to make glucose sensing and insulin delivery automatic. The idea is to emulate the way in which the pancreas secretes insulin in the human body. Therefore, the FTIP system defines a closed-loop, which continuously monitors the patient's glucose level through the sensor, and constantly supplies fast-acting insulin into his/her body by the pump.

This tiny sensor is only a piece of hardware without any embedded software. It has a small integrated transmitter which is used to communicate wirelessly the patient's glucose level to the pump. Every $T_{SensorValue}$ units of time the sensor sends an updated glucose value. Unlike the sensor, the pump is a more sophisticated device. This pump includes an on-board computer that carries the embedded software (i.e. FTIP control system) in charge of managing both the sensor and the pump to perform the treatment. Figure 20 depicts the devices taking part in the treatment, and how they are connected to the patient and interact to each other.

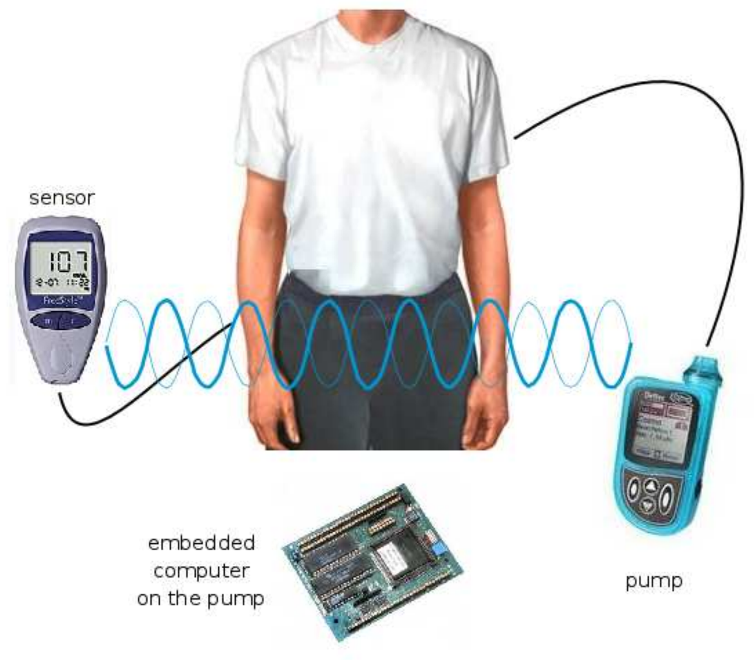


Fig. 20. Case study scenario.

The pump is also composed of a motor, and a cartridge with fast-acting insulin. This motor is connected to a piston rod, which pushes a plunger forward when the motor is working. Once the motor is activated, the insulin is delivered to the patient's body by a cannula that lies under the patient's skin. Therefore, the FTIP control system interfaces with the motor to administrate the dose of insulin to be supplied to the patient. Two infrared motion detectors are also included on the pump to get feedback about the behaviour of the motor and the plunger.

Since the FTIP system is conceived to emulate a complex natural process performed by the human body, it is imperative that the system does not fail. Therefore, both the software and the hardware that compose the system must be built using techniques that allow obtaining products with a high level of reliability. However, it is worth saying that the safety-criticalness level of the FTIP system is much lower compared, for instance, to the one required in air traffic control systems. It means that despite the FTIP system should be operating 24x7, it can perfectly be stopped (intentionally or unintentionally) for a while without compromising the health of the patient.

The core of the FTIP system is the piece of software embedded in the pump, which is referred as FTIP control system. Its main function is to manipulate the pump to deliver the right dose of insulin based on information sent by the sensor. Since there is no physical connection between the sensor and the pump, an exceptional situation can arise due to a problem in the communication between them. The level of glucose dropping outside the allowed range pre-defined by the patient or doctor is another exceptional situation. To mask random measuring errors, which in this particular case are due to transient faults of the sensor, the current patient's glucose level is obtained as the average of 200 samples. Thus, this measuring technique is a step forward to the improvement of the FTIP control system's reliability. Regarding the insulin delivery, the control system relies on the motor and on the plunger infrared motion detectors to determine if the pump is working properly. Therefore, the FTIP control system is able to detect any of the following critical conditions: (i) no values have been received from the sensor for the last $T_{Sensor}$ units of time (**E1**), (ii) the current patient's glucose level is out of the safe range (**E2**), (iii) the insulin to be delivered to keep the glucose in a safe level does not drop into the safe range programmed (**E3**), (iv) the insulin is not being delivered properly (**E4**).

When, at least one of these critical conditions takes place, the FTIP control system stops the insulin delivery and beeps the pump's alarm to alert the patient about the current situation. The alarm remains ringing until the patient switches it off. Instead, when the quantity of insulin in the cartridge (**E5**) is less than the *low limit* configuration parameter, the control system will ring the alarm, as a warning, for only $T_{Warning}$ units of time.

In order to satisfy the previous requirements a set of CAAs that interact cooperatively among them is defined. Figure 21 represents one possible trace of the CAAs behaviours at runtime, in which any exception is raised. In this Figure is also possible to see where exceptions could take place. *CAA_Cycle* is the outermost CAA. It is composed of four roles: *Sensor, Controller, Pump,* and *Alarm*.

Its main task is to perform repetitively a set of operations while the safety conditions of the treatment are kept. These operations are grouped in three basic steps. The first step, which consists of getting the current patient's glucose level, is carried out by *CAA_Checking*. The second step is to calculate how much insulin has to be delivered and it is performed by *CAA_Calculus*. The last step is performed by *CAA_Delivery* and consists of delivering the insulin into the patient.

*Sensor, Pump,* and *Alarm* are the roles used to manage the access to the *SensorDev, MotorDev,* and *AlarmDev* devices, respectively. The *Controller* role coordinates the other roles of *CAA_Cycle* to keep on executing these steps.

As shown earlier, the process to deliver insulin into the patient is achieved by combining several physical devices (e.g. piston rod, plunger, etc.) that are activated when the pump's motor works. Therefore, there is a relationship between "insulin delivering" and "motor working time" that is part of the domain knowledge embedded in the system. Taking into account the "insulin delivering-motor working time" relationship, *CAA_Calculus* has been defined to calculate how long the motor of the pump has to work ($T_{Delivery}$ value). *CAA_Calculus* has been designed to allow the N-version programming technique to be used. In fact, three different algorithms to compute the $T_{Delivery}$ value are employed to discover and mask a possible wrong result produced by one of them. In this manner, the role called *VotingCheck* acts as supervisor, so that it (i) invokes the three different versions of the calculation algorithm; (ii) waits for them to complete their execution; and (iii) compares the results and makes a decision according the majority. Finally, this decision, which is either a numeric value or an exception, is communicated to the enclosing CAA.

Once the *Controller* role has received the $T_{Delivery}$ value from *CAA_Calculus*, *CAA_Checking* and *CAA_Delivery* can be performed in parallel to improve the system performance. Therefore the *Controller* role has to synchronise the CAAs to achieve the execution of the previous described steps. Due to space reasons, Figure 21 only shows the accesses to external objects.

The Java code in Figure 22 shows how the *body* method of the *Controller* role is implemented for the *CAA_Cycle*. *CAA_Cycle* is executed repeatedly until the patient stops manually the delivery (by pressing the *Stop* button) or a critical condition has taken place. The *Controller* role works as a coordinator of the tasks that have to be carried out in *CAA_Cycle*. One of these tasks is to launch the composed *CAA_Calculus* that was described earlier.

The first time that *CAA_Cycle* is called (lines 7-15), the *Controller* role starts to execute *CAA_Checking* (line 10) in order to get the information provided

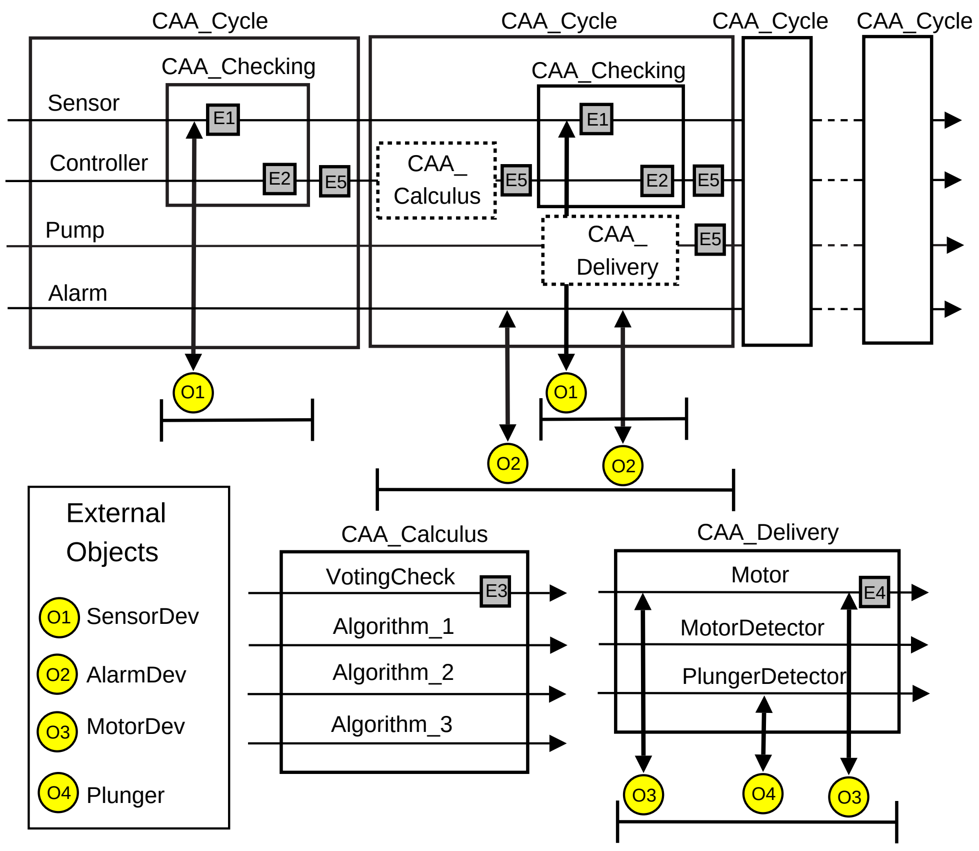


Fig. 21. CAA Design.

by the sensor. Once the role has got the information, it returns the value to the enclosing context (line 15). After *CAA_Cycle* has been executed once, the enclosing context is able to provide the *sv* value that has been taken in the previous execution of *CAA_Cycle*. Therefore, the *Controller* role gets the *sv* value (lines 18-19) and then passes it as an input parameter (line 23) to *CAA_Calculus*. The *CAA_Calculus* execution (line 24) returns the period of time (*tDelivery* value) that the motor has to run (lines 26-27).

When the *tDelivery* value is known, it has to be passed to the *Pump* role (line 29). After that, the *Pump* role receives *tDelivery* and then it can call *CAA_Delivery* to make the delivery of insulin. While *CAA_Delivery* is executing, in order to improve performance, the *Controller* role launches *CAA_Checking* (line 33) that is used to get information from the sensor. This information will be used in the next iteration of *CAA_Cycle*.

When the information returned from *CAA_Checking* and *Pump* roles (lines 35-36, and 38, respectively) has been passed to the enclosing context (lines 40-41) the role can finish its execution and pass control to the enclosing context where *CAA_Cycle* is embedded.

Regarding verification (i.e. the process to ensure the implementation meets its

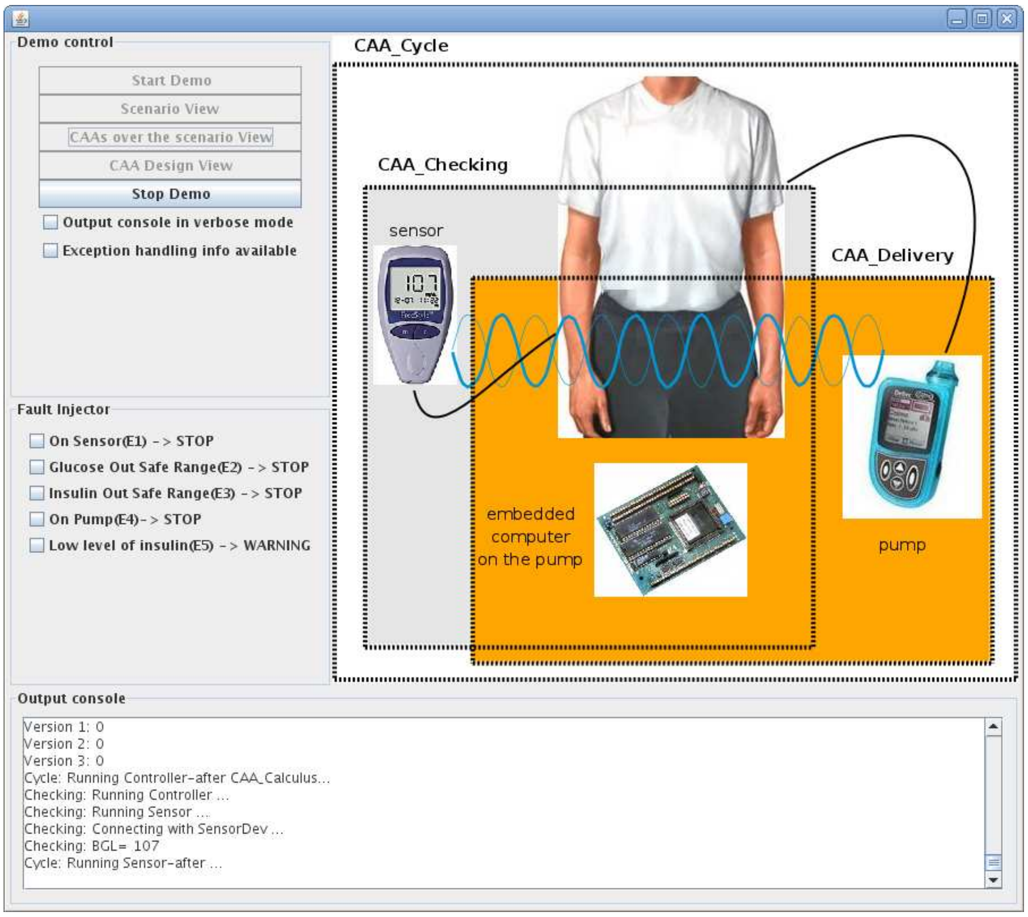
```
public void body(ExternalObjects eos)
  throws Exception, RemoteException {
    try{
      //Getting information from the enclosing context
      Loop loop = (Loop)eos.getExternalObject("loop");
      if(loop.isfirst()){
       //launching nested CAA_Checking
       ExternalObjects checking =
        new ExternalObjects("checking");
       roleControllerChecking.execute(checking);
       //getting outcome from CAA_Checking
       SensorValue sv =
        (SensorValue)checking.getExternalObject("sv");
       //Sending information to the enclosing context
       eos.setExternalObject("sv",sv);
      }else{
        //getting sensor value
        SensorValue sv =
        (SensorValue)eos.getExternalObject("sv");
        //launching composed CAA_Calculus
        ExternalObject calculusREOs =
        new ExternalObjects("calculus");
        calculus.setExternalObject("sv",sv);
        roleVotingCheck.executeAll(calculus);
        //getting outcome from CAA_Calculus
        Time tDelivery =
        (Time)calculus.getExternalObject("tDelivery");
        //passing information to Pump role
        pumpQueue.put(tDelivery);
        //launching nested CAA_Checking
        ExternalObject checking =
        new ExternalObject("checking");
        roleControllerChecking.execute(checking);
        //getting outcome from CAA_Checking
        SensorValue sv =
        (SensorValue)checking.getExternalObject("sv");
        //getting values from CAA_Delivery by Pump role
        Status st = (Status)pumpQueue.get();
        //Sending information to the enclosing context
        eos.setExternalObject("sv",sv);
        eos.setExternalObject("st",st);
      }
    }catch (Exception e) {
      //Local handling for Controller exception;
      throw e;
    }
}
```

Fig. 22. Body method of Controller class

specification) the chosen approach was testing. Among the different techniques existing nowadays to test the dependability of a system, fault injection is one of the most practical and effective [20]. It provides a way to test the dependability of the system with respect to certain fault classes that may need hard-to-reach preconditions in order to allow them to be manifested. Thus, fault injection in this particular case has been used to evaluate the fault-tolerant aspects included in the FTIP system to deal with the above-mentioned exceptional situations (i.e. $E_{1..5}$).

These exceptional situations are occasioned by natural faults, i.e. they are caused without any human participation. Therefore, the states manipulated by the fault injector to allow faults to manifest, are those belonging to the environment where the FTIP control system is embedded. This environment is only composed of the sensor and the pump. It does not include the patient as he/she does not interact directly with the control system. This interaction is achievable through the devices. Therefore, the fault injector needs to modify the sensor and pump's states to make the exceptional situations of interest effective. This has as advantage that the fault injector does not represent an intrusion into the FTIP control system.

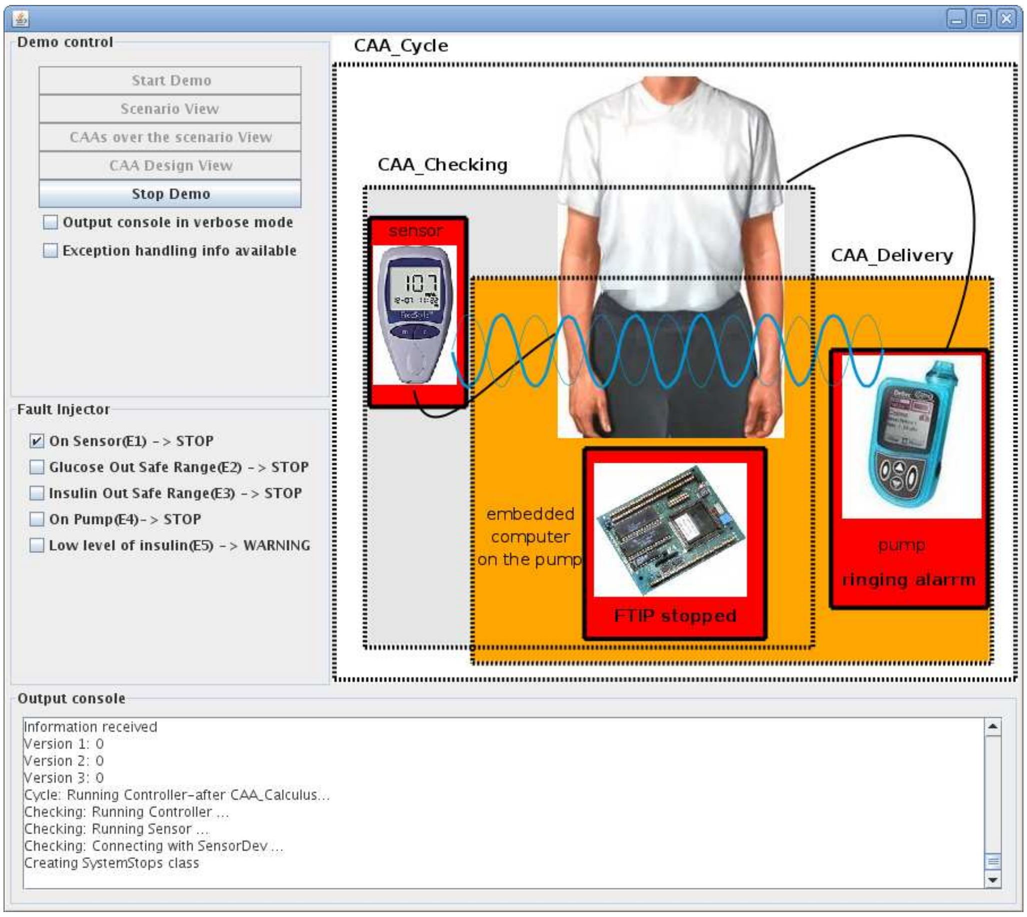
Since both the sensor and the pump used to achieve this type of closed-loop system do not exist in the market yet, they were implemented by software to allow the testing process to be performed. Thus, as these devices represent the environment for the FTIP control system, and the fault injector has to operate on it, a full-integrated tool (called FTIP simulator [7]) was developed to show the progress of the control system and to check its reaction when a fault is injected.


Demo control
Start Demo
Scenario View
CAAs over the scenario View
CAA Design View
Stop Demo
Output console in verbose mode
Exception handling info available
Fault Injector
On Sensor(E1) - > STOP
Glucose Out Safe Range(E2) - > STOP
Insulin Out Safe Range(E3) - > STOP
On Pump(E4)- > STOP
Low level of insulin(E5) - > WARNING
CAA_Cycle
CAA_Checking
sensor
CAA_Delivery
embedded computer on the pump
pump
Output console

```
Version 1: 0
Version 2: 0
Version 3: 0
Cycle: Running Controller-after CAA_Calculus...
Checking: Running Controller ...
Checking: Running Sensor ...
Checking: Connecting with SensorDev ...
Checking: BGL= 107
Cycle: Running Sensor-after ...
```

Fig. 23. The Fault-Tolerant Insulin Pump simulator.

Figure 23 shows a screen dump of the FTIP simulator. It has a main console (top-left corner) from where the simulation can be started/stopped. From this console it is also possible to decide the form (simple or verbose mode) and the type of information (e.g. internal CAA-DRIP framework steps regarding the exception handling manipulation) to be displayed on the output console (bottom part). An outline of the scenario appears on the background of the simulator's central display. In fact, the outline of the scenario is used as reference to show the execution progress of the involved CAAs. When a CAA is executing it is coloured. Figure 24 shows the simultaneous execution of *CAA_Checking* and *CAA_Delivery* (two different colours were used to show such effect).



- [ ] Output console in verbose mode
- [ ] Exception handling info available

- [x] On Sensor(E1) -> STOP
- [ ] Glucose Out Safe Range(E2) -> STOP
- [ ] Insulin Out Safe Range(E3) -> STOP
- [ ] On Pump(E4)-> STOP
- [ ] Low level of insulin(E5) -> WARNING

```
Information received
Version 1: 0
Version 2: 0
Version 3: 0
Cycle: Running Controller–after CAA_Calculus...
Checking: Running Controller ...
Checking: Running Sensor ...
Checking: Connecting with SensorDev ...
Creating SystemStops class
```

Fig. 24. Injecting a fault on the sensor.

The fault injector is another console situated in-between the main control and the output console. Figure 24 shows the result of injecting a fault on the sensor device (i.e. the exceptional situation E1). Facing to this situation, the FTIP control system performs an insulin delivery safe-stop and starts ringing the pump's alarm.

## 5 Conclusions and Future Work

This paper puts together all concepts concerning CAAs and discusses two CAAs implementation frameworks (DRIP and CAA-DRIP). Based on the analysis of the concepts it provides a formal description of the CAAs life cycle using the *Statecharts* language. This *Statecharts* description is used to demonstrate how each life cycle step of applying CAA is conducted depending on the framework employed to develop an application.

Both DRIP and CAA-DRIP are object-oriented solutions. The former was conceived to implement systems designed in terms of DMIs, whereas the later was developed to implement systems designed using CAAs. The only difference between DMIs and CAA is in the exception handling semantics. The CAA exception handling semantics is more restricted than the one of DMIs. Despite that, the DRIP framework can be used to implement a CAA design. However, the fact that DRIP was not directly targeted for CAAs makes its use no straight for programmers. CAA-DRIP was developed to bridge this gap, as it provides only the elements (i.e. roles, handlers, and compensators) necessary for implementing each CAA concept in a straightforward manner. Furthermore, this one-to-one mapping between the design model and its respective implementation allows a reverse engineering process (i.e. getting the design from the source code).

The separation of concepts, helping programmers to carry out the implementation, along with the performance improvements, are the main reasons why CAA-DRIP represents a better solution comparing with its ancestor. However, our analysis shows that there are situations in which the features provided by CAA-DRIP are not sufficient, e.g. for developing real-time systems. In such systems, the framework should include features for specifying time requirements of CAA execution, i.e. starting time, processing time and deadline. Scheduling policies to make the execution of the CAAs more predictable would be necessary as well. The required features that should be included in the CAA-DRIP framework is still an ongoing research area.

Another issue that is being addressed in this paper is to understand how much details programmers need to know about the CAA-DRIP framework. Even if the abstraction level of CAA-DRIP is the same as in the CAA design, programmers still have to invest time for learning it. This means that they need to go through a training process to get experience and gain confidence in using CAA-DRIP. A framework which is easy to learn and to get confidence in is known as a *white-box framework* [24,25]. Therefore, it would be useful to make the framework use more straightforward, in order to speed up the implementation phase. Currently, the development of a tool to become CAA-DRIP a *black-box framework* (i.e. programmers would not need to know its details to use it) is being performed. This tool will receive as input a CAA design described in a domain specific language [23] and it will produce automatically the code that implements such design.

## 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 B. Gallina, A. Campéas, H. Muccini, and P. Periorellis. A. Zorzo is partially funded by CNPq Brazil. A. Romanovsky is supported by ICT DEPLOY project.

## References

[1] Hoare, C. A. R.. Parallel Programming: an Axiomatic Approach. In Goos, G. and Hartmaur, J. (eds), Languages Hierarchies and Interfaces, LNCS 46. Springer-Verlag. 1976.

[2] S. Veloudis and N. Nissanke. Modelling Coordinated Atomic Actions in Timed CSP. In Proceedings of 6th International Symposium on Formal Techniques in Real-Time Fault Tolerant Systems (FTRTFT), LNCS 1926, pages 228-239. Springer-Verlag. 2000.

[3] Java 2 Platform, Standard Edition (J2SE). http://java.sun.com, 2007.

[4] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 435–446. ACM Press. 1999.

[5] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transaction on Software Engineering*, SE-11(12):1491–1501, 1985.

[6] C. Bertolini, L. Brenner, P. Fernandes, A. Sales, and A. F. Zorzo. Structured Stochastic Modeling of Fault-Tolerant Systems. In Proceedings of IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pages 139–146. IEEE Press. 2004.

[7] Correct Project's Web Page. http://lassy.uni.lu/correct, 2007.

[8] F. Tartanoglu, N. Levy, V. Issarny, and A. Romanovsky. Using the B Method for the Formalization of Coordinated Atomic Actions. Technical Report CS-TR: 865, School of Computing Science, University of Newcastle upon Tyne, UK, October 2004.

[9] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. Zorzo. Formal Development and Validation of Java Dependable Distributed Systems. In Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pages 98–108. IEEE Press. 1999.

[10] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. *Morgan Kaufmann*, 1993.

[11] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[12] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.

[13] P. Lee and T. Anderson. Fault Tolerance: Principles and Practice, Second Edition. *Prentice-Hall*, 1990.

[14] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[15] A. Romanovsky, P. Periorellis, and A. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. Proceedings of International Symposium on Autonomous Decentralized Systems (ISADS), pages 99–106. IEEE Press. 2003.

[16] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, E. Canver, and F. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. *IEEE Transactions on Computers*, 51(2):164–179, 2002.

[17] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery. In Proceedings of IEEE International Symposium on Fault-Tolerant Computing (FTCS), pages 499–508. IEEE Press. 1995.

[18] A. Zorzo. Multiparty Interactions in Dependable Distributed Systems. *PhD Thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK*, 1999.

[19] A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud and I. Welch Using Coordinated Atomic Actions to Design Safety-Critical Systems: a Production Cell Case Study. *Software: Practice and Experience*, 29(8):677–697, 1999.

[20] C. Ramesh, R.M. Lefever, K.R. Joshi, M. Cukier, and W.H. Sanders A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, 2004.

[21] B. Gallina, N. Guelfi, and A. Romanovsky. Coordinated Atomic Actions for Dependable Distributed Systems: the Current State in Concepts, Semantics and Verification Means. In Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE), pages 29–38. IEEE Press. 2007

[22] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, and A. Zorzo, CAA-DRIP: a framework for implementing Coordinated Atomic Actions, In Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE), pages 385–394. IEEE Press. 2006.

[23] J. Vachon, COALA: a Design Language for Reliable Distributed Systems, *Ph.D Thesis, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Suisse*, 2000.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns Elements of Reusable Object-Oriented software, Fifth edition. *Addison-Wesley Publishing Company*, 1995.

[25] K. Beck, and R. Johnson, Patterns generate architectures, In Proceedings of European Conference on Object-Oriented Programming (ECOOP), pages 139–149, 1994.

## Appendix: Experimental results

### *5.1 Measuring the elapsed time used by a CAA in milliseconds (ms.)*

#### *5.1.1 Normal behaviour of a CAA using CAA-DRIP and DRIP frameworks*

| | *CAA-DRIP* | | | *DRIP* | | |
|---|---|---|---|---|---|---|
| *Part.* | *AVG* | *SD* | *CI* | *AVG* | *SD* | *CI* |
| 100 | 553.2 | 4.71 | 1.31 | 565.6 | 108.87 | 30.18 |
| 200 | 559.4 | 3.14 | 0.87 | 554.6 | 5.03 | 1.4 |
| 300 | 611.6 | 10.17 | 2.82 | 608.2 | 3.88 | 1.08 |
| 400 | 660.2 | 1.41 | 0.39 | 660 | 0 | 0 |
| 500 | 715.8 | 8.35 | 2.32 | 715.8 | 10.32 | 2.86 |
| 600 | 765.6 | 5.41 | 1.5 | 746 | 22.22 | 6.16 |
| 700 | 818.4 | 5.1 | 1.41 | 808.6 | 17.96 | 4.98 |
| 800 | 871.4 | 4.52 | 1.25 | 875.4 | 5.03 | 1.4 |
| 900 | 929 | 10.15 | 2.81 | 953.8 | 27.4 | 7.59 |
| 1000 | 989.8 | 14.36 | 3.98 | 983.8 | 8.55 | 2.37 |
| 1100 | 1151.2 | 11 | 3.05 | 1053.8 | 18.61 | 5.16 |
| 1200 | 1258.2 | 6.61 | 1.83 | 1159.8 | 15.97 | 4.43 |
| 1300 | 1321.4 | 18.52 | 5.13 | 1250.4 | 19.37 | 5.37 |
| 1400 | 1346.4 | 16.63 | 4.61 | 1433.2 | 31.97 | 8.86 |
| 1500 | 1456 | 22.59 | 6.26 | 1600.2 | 19.95 | 5.53 |
| 1600 | 1576.4 | 26.86 | 7.45 | 1652.2 | 26.29 | 7.29 |
| 1700 | 1838.2 | 21.82 | 6.05 | 1693.8 | 42.03 | 11.65 |
| 1800 | 2033.2 | 30.33 | 8.41 | 1802.2 | 28.09 | 7.79 |
| 1900 | 2157.6 | 25.6 | 7.1 | 1951 | 33.52 | 9.29 |
| 2000 | 2377 | 40.47 | 11.22 | 2265.4 | 34.54 | 9.57 |
| 2100 | 2553 | 28.09 | 7.78 | 2473 | 35.24 | 9.77 |
| 2200 | 2736.8 | 35.13 | 9.74 | 2661.8 | 40.24 | 11.15 |
| 2300 | 2981 | 27.87 | 7.72 | 2879.2 | 56.74 | 15.73 |
| 2400 | 3273.6 | 43.79 | 12.14 | 3119 | 54.52 | 15.11 |
| 2500 | 3445.4 | 47.13 | 13.06 | 3350.8 | 54.05 | 14.98 |
| 2600 | 3623.2 | 55.6 | 15.41 | 3728.6 | 52.68 | 14.6 |
| 2700 | 3886.8 | 100.92 | 27.97 | 3881.2 | 66.44 | 18.42 |
| 2800 | 4251.8 | 81.83 | 22.68 | 4182.4 | 96.69 | 26.8 |
| 2900 | 4546.8 | 98.67 | 27.35 | 4487.2 | 104.26 | 28.9 |
| 3000 | 4887.8 | 74.57 | 20.67 | 4885.4 | 116.36 | 32.25 |

### *5.1.2 Exceptional behaviour of a CAA using CAA-DRIP and DRIP frameworks*

| | *CAA-DRIP* | | | *DRIP* | | |
|---|---|---|---|---|---|---|
| *Part.* | *AVG* | *SD* | *CI* | *AVG* | *SD* | *CI* |
| 100 | 606.6 | 9.17 | 2.54 | 760.15 | 1.23 | 0.3 |
| 200 | 660.2 | 1.41 | 0.39 | 798.18 | 24.49 | 5.91 |
| 300 | 662.2 | 15.56 | 4.31 | 827.27 | 21.38 | 5.16 |
| 400 | 809.2 | 20.49 | 5.68 | 981.06 | 20.47 | 4.94 |
| 500 | 820.6 | 3.14 | 0.87 | 1073.33 | 19.16 | 4.62 |
| 600 | 888.6 | 48.49 | 13.44 | 1141.06 | 31.43 | 7.58 |
| 700 | 978.6 | 9.26 | 2.57 | 1388.48 | 45.28 | 10.92 |
| 800 | 998.8 | 25.92 | 7.19 | 1652.12 | 133.58 | 32.23 |
| 900 | 1126.4 | 20.97 | 5.81 | 2948.48 | 955.53 | 230.53 |
| 1000 | 1218 | 21.09 | 5.85 | 7829.39 | 2753.41 | 664.29 |
| 1100 | 1327.8 | 14.47 | 4.01 | 13637.27 | 4660.27 | 1124.33 |
| 1200 | 1463.4 | 21.44 | 5.94 | 19579.7 | 7784.68 | 1878.13 |
| 1300 | 1535.6 | 39.85 | 11.05 | 27750.15 | 9469.74 | 2284.66 |
| 1400 | 1554.4 | 18.86 | 5.23 | 40307.88 | 10747.42 | 2592.92 |
| 1500 | 1675.6 | 18.09 | 5.01 | 50435.91 | 12673.61 | 3057.63 |
| 1600 | 1828.2 | 30.35 | 8.41 | 65625.3 | 9672.99 | 2333.7 |
| 1700 | 2056.6 | 23.44 | 6.5 | 76103.94 | 10229.24 | 2467.9 |
| 1800 | 2317.4 | 49.48 | 13.71 | 87731.97 | 10908.8 | 2631.85 |
| 1900 | 2390.6 | 44.56 | 12.35 | 102946.31 | 22150.15 | 5384.88 |
| 2000 | 2691.2 | 61.13 | 16.95 | 121299.51 | 23143.64 | 5807.95 |
| 2100 | 2831 | 44.69 | 12.39 | 129872.54 | 17233.88 | 4255.68 |
| 2200 | 2982.2 | 69.08 | 19.15 | 145301.13 | 29256.46 | 7282.53 |
| 2300 | 3316.8 | 62.71 | 17.38 | 165615.41 | 48616.42 | 12200.4 |
| 2400 | 3518.8 | 61.5 | 17.05 | 179458.85 | 45005.91 | 11294.34 |
| 2500 | 3664.2 | 88.67 | 24.58 | 199284.26 | 41827.1 | 10496.61 |
| 2600 | 3837.2 | 76.13 | 21.1 | 211009 | 44218.39 | 11188.8 |
| 2700 | 4146 | 80.18 | 22.22 | 229506.67 | 39968.04 | 10113.31 |
| 2800 | 4426.4 | 89.78 | 24.89 | 258589.83 | 62623.08 | 15845.82 |
| 2900 | 4730.8 | 106.96 | 29.65 | 262802.54 | 42845.35 | 10932.86 |
| 3000 | 5127 | 109.47 | 30.34 | 284491.23 | 45682.67 | 11859.6 |

## *5.2 Measuring the memory used by a CAA in Mb.*

### *5.2.1 Normal behaviour of a CAA using CAA-DRIP and DRIP frameworks*

| | *CAA-DRIP* | | | *DRIP* | | |
|---|---|---|---|---|---|---|
| *Part.* | *AVG* | *SD* | *CI* | *AVG* | *SD* | *CI* |
| 100 | 14 | 0 | 0 | 14 | 0 | 0 |
| 200 | 19 | 0 | 0 | 19 | 0 | 0 |
| 300 | 24 | 0 | 0 | 24 | 0 | 0 |
| 400 | 29 | 0 | 0 | 28.5 | 0.51 | 0.14 |
| 500 | 33 | 0 | 0 | 33 | 0 | 0 |
| 600 | 38 | 0 | 0 | 39.04 | 0.2 | 0.05 |
| 700 | 42.5 | 0.5 | 0.14 | 43 | 0 | 0 |
| 800 | 47.52 | 0.5 | 0.14 | 47.48 | 0.5 | 0.14 |
| 900 | 51.73 | 0.45 | 0.12 | 53.6 | 0.49 | 0.14 |
| 1000 | 55.25 | 1.22 | 0.33 | 58.6 | 1.44 | 0.4 |
| 1100 | 65.21 | 0.89 | 0.24 | 60.32 | 3.4 | 0.94 |
| 1200 | 71.42 | 0.64 | 0.17 | 70.52 | 0.89 | 0.25 |
| 1300 | 75.77 | 0.83 | 0.23 | 76.36 | 0.72 | 0.2 |
| 1400 | 81.08 | 0.76 | 0.21 | 78.66 | 1.1 | 0.3 |
| 1500 | 84.29 | 0.82 | 0.22 | 83.6 | 1.65 | 0.46 |
| 1600 | 88.42 | 1.75 | 0.48 | 89.96 | 1.95 | 0.54 |
| 1700 | 93.73 | 0.92 | 0.25 | 95.82 | 1 | 0.28 |
| 1800 | 93.73 | 0.78 | 0.21 | 100.1 | 0.95 | 0.26 |
| 1900 | 100.65 | 2.86 | 0.78 | 99.56 | 0.7 | 0.2 |
| 2000 | 101.71 | 1.36 | 0.37 | 108.48 | 1.43 | 0.4 |
| 2100 | 109.33 | 1.41 | 0.39 | 110.28 | 1.44 | 0.4 |
| 2200 | 115.61 | 2.94 | 0.81 | 114.08 | 2.23 | 0.62 |
| 2300 | 113.06 | 1.63 | 0.45 | 120.92 | 2.32 | 0.64 |
| 2400 | 124.8 | 3.14 | 0.86 | 122.54 | 2.3 | 0.64 |
| 2500 | 130.39 | 2.12 | 0.58 | 125.4 | 1.95 | 0.54 |
| 2600 | 132.63 | 2.44 | 0.67 | 134.6 | 3.16 | 0.87 |
| 2700 | 139.2 | 2.9 | 0.8 | 140.64 | 2.55 | 0.71 |
| 2800 | 140.57 | 3.19 | 0.88 | 142.8 | 2.62 | 0.73 |
| 2900 | 147.71 | 2.51 | 0.69 | 149.74 | 2.01 | 0.56 |
| 3000 | 152.86 | 4.42 | 1.21 | 150.98 | 3.13 | 0.87 |

### 5.2.2 *Exceptional behaviour of a CAA using CAA-DRIP and DRIP frameworks*

| | *CAA-DRIP* | | | *DRIP* | | |
|---|---|---|---|---|---|---|
| *Part.* | *AVG* | *SD* | *CI* | *AVG* | *SD* | *CI* |
| 100 | 14 | 0 | 0 | 16 | 0 | 0 |
| 200 | 19 | 0 | 0 | 24 | 0 | 0 |
| 300 | 24 | 0 | 0 | 31 | 0 | 0 |
| 400 | 29.48 | 0.5 | 0.14 | 34.57 | 2.1 | 0.58 |
| 500 | 33 | 0 | 0 | 37.08 | 3.5 | 0.96 |
| 600 | 38 | 0 | 0 | 52.65 | 1.32 | 0.36 |
| 700 | 42.52 | 0.5 | 0.14 | 62.45 | 1.79 | 0.49 |
| 800 | 47.58 | 0.5 | 0.14 | 71.16 | 0.99 | 0.27 |
| 900 | 53 | 0 | 0 | 77.69 | 0.68 | 0.19 |
| 1000 | 60.36 | 0.53 | 0.15 | 83.31 | 1.2 | 0.33 |
| 1100 | 67.66 | 0.59 | 0.16 | 85.35 | 10.26 | 2.82 |
| 1200 | 73.64 | 0.48 | 0.13 | 90.22 | 12.75 | 3.5 |
| 1300 | 77.66 | 0.63 | 0.17 | 86.98 | 12.59 | 3.46 |
| 1400 | 83.5 | 1.04 | 0.29 | 89.67 | 10.92 | 3 |
| 1500 | 89.04 | 1.38 | 0.38 | 91.39 | 7.15 | 1.96 |
| 1600 | 91.72 | 2.34 | 0.65 | 94.51 | 5.42 | 1.49 |
| 1700 | 98.34 | 1.36 | 0.38 | 98.59 | 1.34 | 0.37 |
| 1800 | 98.42 | 0.81 | 0.22 | 102.51 | 2.35 | 0.64 |
| 1900 | 106.74 | 2.32 | 0.64 | 102.49 | 0.51 | 0.14 |
| 2000 | 107.94 | 0.79 | 0.22 | 111.16 | 3.36 | 0.92 |
| 2100 | 117.02 | 2.06 | 0.57 | 123.34 | 12.98 | 3.6 |
| 2200 | 120.86 | 1.78 | 0.49 | 125.98 | 15.83 | 4.39 |
| 2300 | 125.12 | 1.47 | 0.41 | 133.78 | 14.45 | 4 |
| 2400 | 131.54 | 1.25 | 0.35 | 138.16 | 15.61 | 4.33 |
| 2500 | 135.1 | 1.04 | 0.29 | 143.04 | 19.63 | 5.44 |
| 2600 | 140.54 | 2.12 | 0.59 | 153.5 | 19.65 | 5.45 |
| 2700 | 143.5 | 1.05 | 0.29 | 152.86 | 22.09 | 6.19 |
| 2800 | 148.9 | 2.17 | 0.6 | 158.9 | 17.81 | 5.04 |
| 2900 | 150.66 | 1.44 | 0.4 | 173.63 | 23.48 | 6.64 |
| 3000 | 158.94 | 2.75 | 0.76 | 173.08 | 20.9 | 5.91 |