# CAA-DRIP: a framework for implementing Coordinated Atomic Actions

A. Capozucca, N. Guelfi, P. Pelliccione
LASSY, University of Luxembourg, Luxembourg
{alfredo.capozucca,nicolas.guelfi,patrizio.pelliccione}@uni.lu

A. Romanovsky
Center for Software Reliability
University of Newcastle upon Tyne - UK
alexander.romanovsky@ncl.ac.uk

A. Zorzo
Faculty of Informatics
Pontifical Catholic University of RS - Brazil
zorzo@inf.pucrs.br

## Abstract

*This paper presents an implementation framework, called CAA-DRIP, that has been defined to allow a straightforward implementation of dependable distributed applications designed using the Coordinated Atomic Action (CAA) paradigm. CAAs provide a coherent set of concepts adapted to the design of fault tolerant distributed systems 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)" which includes a general exception handling mechanism. As DMI has a softer exception handling semantics with respect to CAA semantics, a CAA design can be implemented by DRIP. The aim of the CAA-DRIP framework is to provide a set of Java classes that allows programmers to implement only the semantics of CAAs with the same terminology and concepts at the design and implementation levels. The new framework simplifies the implementation phase and at the same time reduces the size of the final system since it requires fewer number of instances for creating a CAA at runtime. Details of these improvements as well as a precise description of the CAAs behaviour in terms of Statecharts, which is used as a reference model to define the CAA-DRIP framework, are presented in this paper.*

## 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 here is to focus on the normal behaviour of software systems, ignoring abnormal behaviour which 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 which 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) [12], N-Version Programming (NVP) [3], Conversations [12], Transactions [8], Coordinated Atomic Actions (CAAs) [15], and so on. CAAs are intended for designing complex distributed systems with high availability and reliability requirements. More specifically they focus on concurrent systems consisting of cooperative and competitive components and provide fault tolerance by means of cooperative exception handling. CAAs have been successfully used in several case studies [17, 7, 13] that demonstrate high usefulness and general applicability of the approach.

The implementation of systems designed using CAAs is currently supported by the Dependable Remote Interacting Processes (DRIP) [2] framework. DRIP has been initially developed to provide implementation of the "Dependable Multiparty Interactions" (DMIs) abstraction [16]. DMIs is a scheme that allows executing a set of participants (objects, threads, 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 DMIs and CAAs schemes is in the way they deal with exceptions. DMIs have a more relaxed exception handling semantics than the CAAs as they support a

17th International Symposium on Software Reliability Engineering (ISSRE'06)
0-7695-2684-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

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.

However, since working with the DRIP framework is an already complex task for programmers, it is important to provide elements (classes, interfaces, packages, etc.) directly supporting the CAA concepts used at the design level and to facilitate the implementation of the concepts that DRIP does not directly support. Thus, it is very useful to have a framework that guarantees that the design abstraction is going to be implemented correctly. Furthermore, implementing an application using a clean framework will facilitate the testing phase. This will not be the case if a programmer uses the DRIP framework for implementing systems designed using CAAs. Actually, while using DRIP the programmer can forget or even decide to ignore some requirements of the CAAs. Furthermore, to implement CAAs using DRIP programmers need 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 correctly seems to be a dangerous idea.

In this paper, a new framework named CAA-DRIP is presented. It is the result of modifications and extensions made on DRIP to exclusive and completely provide support for implementing the CAA semantics. Improvements respect to performance have been also achieved.

In order to provide the reader with a clear understanding of the CAAs abstraction, their characteristics are described using Statecharts [9, 10]. There has been a lot of work on formal description of CAAs, for example, using Temporal Logic of Actions [14], Stochastic Automata Networks [4] and B [6]. Here a clean formal high level description of the CAA behaviour is offered to, first complement previous CAA formalisations, and second be used for programmers as a reference to drive the CAA implementation phase. Statecharts is a simple description language that 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, the CAAs concept has been improved in the past years and several new issues are provided since its first description. For example, a new composite type of CAA has been described in [7, 13, 6] and the way external objects are dealt with is spread throughout several papers [15, 6]. In this paper, all these features are collected and described.

After a detailed description of the CAAs mechanism semantics (Section 2), the CAA-DRIP framework (Section 3) is introduced. In the same section is explained how programmers have to deal with CAA-DRIP to achieve the implementation and what are the advantages of using it. Section 4 provides a small description of a case study in which the new framework is applied. 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. Thus, using CAAs systems that comply with their specification in spite of faults having occurred can be developed. This mechanism unifies the features of two complementary concepts: *conversation* and *transaction*. Conversation [12] is a fault-tolerant technique for performing coordinated error recovery in a set of participants that have been designed to interact with each other to provide a specific service (cooperative concurrency). Objects that are used to achieve the cooperation among the participants are called **shared** objects. Transactions are used to deal with competitive concurrency on objects that have been designed and implemented separately from the applications that make use of them. These kind of objects are named **external** objects.

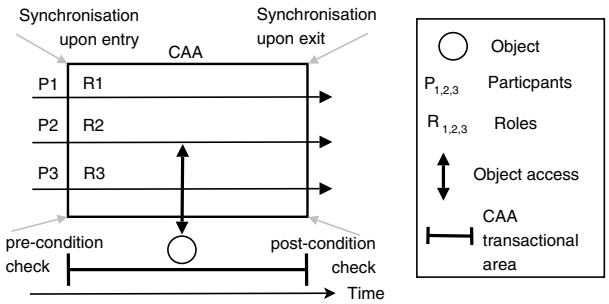


**Figure 1. A simple CAA**

As it is shown in Figure 1, one CAA characterises an orchestration of operations executed by a group of roles that exchange information among themselves, and/or access to external objects (concurrently with other CAAs) to achieve a common goal. The CAA 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 belonging to the CAA, 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 nor-


IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

mally 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 last 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 last visited error-free state (if BER is activated) and it provides its own error recovery mechanism [15]. Therefore when BER takes place, it restores these external objects using their own recovery mechanisms. However, sometimes the designer/programmer might want to use an external object that does not provide any recovery mechanism (due to reasons of cost or physical constraints [11]). 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 problem of using BER concerns objects (particularly **external** objects) which cannot be restored from their last 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 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 before calling the CAA (**Abort**). Basically, the ACID 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 are the necessary steps to compensate a particular *MR* object. 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 drive 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*).

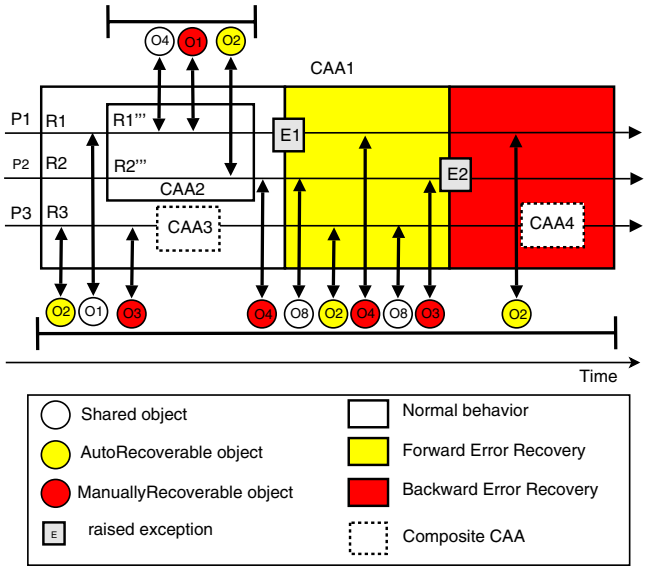


**Figure 2. Coordinated Atomic Actions.**

Another important characteristic of CAAs is that they can be designed in a structured way using **nesting** and/or **composition** (see Figure 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


IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

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 track 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 [15].

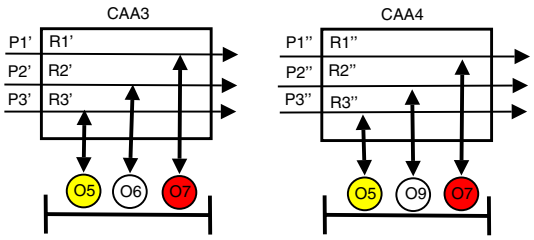


**Figure 3. Composite CAAs.**

Composite CAAs [6] 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 the composite $CAA_3$ (i.e., participants, accessed objects and roles) is hidden from the calling $CAA_1$.

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, $CAA_1$, to guarantee the ACID properties on the external objects, needs to carry out a specific handling, which 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 Formal description of the CAAs behaviour

Statecharts [9, 10] is used to formally express the semantics of each possible kind of CAA outcome (**normal, exceptional, abort** and **failure**).

The specification (Figure 4) is composed of a big state called **Enclosing context** that intially is at 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 state *S1*, publishing 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*).

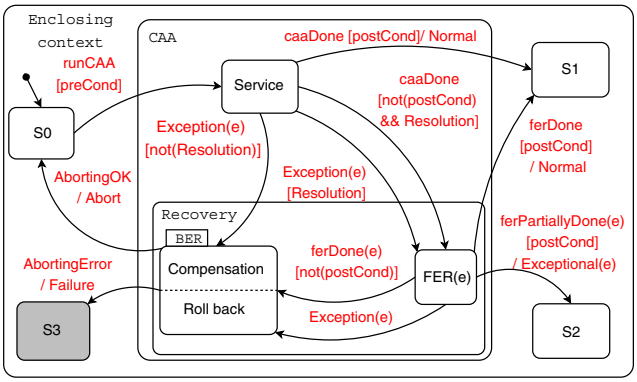


**Figure 4. CAA behaviour.**

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 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).

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


IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

(represented by *postCond* predicate) is met. Therefore 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*).

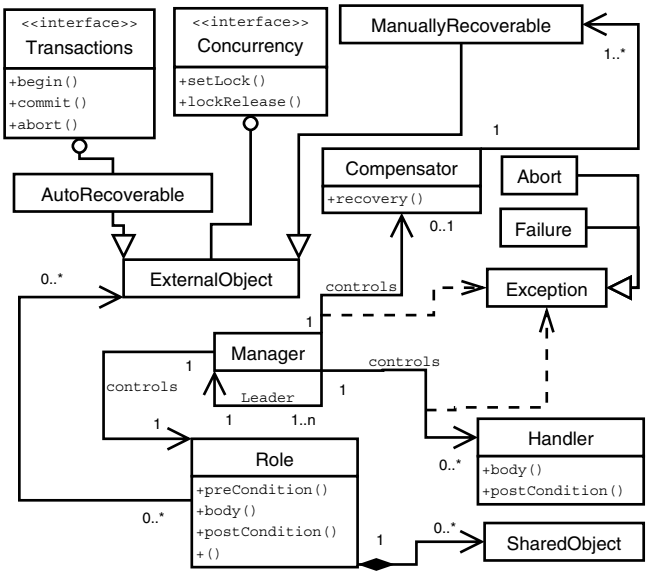


**Figure 5. CAA-DRIP UML model**

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 publish the *Failure* event. In this case the enclosing context is left in an unspecified state (*S3*).

# 3 The CAA-DRIP framework

A set of *Java* [1] classes and interfaces called CAA-DRIP has been defined using as starting point the DRIP framework [2]. CAA-DRIP allows us to implement the concepts and behaviour described in Section 2. The Statecharts description in Figure 4 is used as reference to show how the states that compose the CAA are defined and how each of them can be reached. As mentioned before, a CAA is defined as a set of participants that come together to do "something". This "something" is the service the CAA provides. The service is carried out by the *roles* that the participants will play inside the CAA.

The model in Figure 5 shows a UML diagram of the classes used to implement a CAA. The *Manager* class is the controller for *Role, Handler* and *Compensator* classes. Thus, each role, handler and compensator objects created are managed by a manager object. As shown in Figure 5, there is not a class to represent a CAA. It consists of a set of managers, roles, handlers and compensators linked together via a *leader* manager (represented as an *association* in Figure 5). The manager object that is chosen as the *leader* is the responsible for synchronising roles upon entry and upon exit, the execution of the exception resolution algorithm and for keeping information about shared objects. $CAA_1$, from Figure 2, is used as example to show how the CAA-DRIP framework has to be used to create a CAA.

## 3.1 Instantiating a CAA

When a *Manager* object is created it has to be informed of its name and the name of the CAA (lines 2-4 in Figure 6). Once the managers have been created it is necessary to create the role objects. Each role upon creation is informed of its name, which manager will be its controller and the manager that will act as the *leader* (lines 7-9).

```
1  //Managers
2  mgr1 = new ManagerImpl("mgr1","CAA1");
3  mgr2 = new ManagerImpl("mgr2","CAA1");
4  mgr3 = new ManagerImpl("mgr3","CAA1");
5  //Roles
6  role1 = new R1("role1",mgr1,mgr1);
7  role2 = new R2("role2",mgr2,mgr1);
8  role3 = new R3("role3",mgr3,mgr1);
9  //Handlers
10 hndrE1_R1 = new E1_R1("hndrE1_R1",mgr1);
11 hndrE1_R2 = new E1_R2("hndrE1_R2",mgr2);
12 hndrE1_R3 = new E1_R3("hndrE1_R3",mgr3);
13 //Binding Exception-Handler
14 Hashtable ehR1 = new Hashtable();
15 ehR1.put(E1.class,hndrE1_R1);
16 Hashtable ehR2 = new Hashtable();
17 ehR2.put(E1.class,hndrE1_R2);
18 Hashtable ehR3 = new Hashtable();
19 ehR3.put(E1.class,hndrE1_R3);
20 //Binding Exception-Handler-Manager
21 mgr1.setExceptionAndHandlerList(ehR1);
22 mgr2.setExceptionAndHandlerList(ehR2);
23 mgr3.setExceptionAndHandlerList(ehR3);
24 //Compensators
25 cmp1 = new CmpR1("cmp1",mgr1);
26 cmp2 = new CmpR2("cmp2",mgr2);
27 cmp3 = new CmpR3("cmp3",mgr3);
```

**Figure 6. CAA1 definition**

For each *e* exception that the CAA has to handle by FER the user of the framework has to: *(1)* create a handler for each role that the CAA has; *(2)* link each defined handler with the exception to handle; *(3)* pass the link *exception-handler* to the object that controls the handler execution.

Now, with this information in mind, how these steps are applied on $CAA_1$ to handle the *E1* exception is shown. Each handler upon creation (step 1) needs to know its name and the manager that will control this handler (lines 12-14). The link between the *E1* exception and each handler (step


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

2) is implemented by a hashtable that has as key the exception and as value the handler object that has been defined in the step 1 (lines 17-22). The method *setExceptionAndHandlerList* is used to inform the manager about the relationships *exception-handler* that have been set in the step 2 (lines 25-27).

If the CAA has to handle manually recoverable objects (see Section 2), a compensator has to be created also. This is shown on lines 30-32. Analogously to a handler creation, each compensator upon creation is informed of its name and the handler that will drive its execution. This kind of compensator was not part of the original DRIP framework.

### 3.2 Extending and Implementing the CAA-DRIP framework classes

The definition of a role is made by creating a new class that extends the *Role* class (see the *Java* code in Figure 7). The programmer has to re-implement the *body* method (line 19). 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.

When the programmer wants to instantiate the created new class, it has to be informed of its name, the manager object that drives its execution and the leader manager object used for coordinating the CAA execution (lines 3-4).

Shared objects for coordinating the CAA roles are defined inside the new class that extends the *Role* class (line 2). Once a shared object has been created (line 9), 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 11).

The other methods that have also to be redefined by the programmer are *preCondition* (line 13) and *postCondition* (line 28). 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 previous framework, i.e. the DRIP framework. If an exception has to be handled by FER, then it is necessary to define a handler for each CAA role. This task is made by creating a new class that extends the *Handler* class. The programmer has to re-implement the *body* and the *postCondition* methods. The operations to deal with only raised exceptions take place inside the *body* method. Thus, for each exception that has to be handled by the CAA, its corresponding handlers has to be defined (one for each CAA role). State *FER(e)* in Figure 4 corresponds to the execution of the handlers for dealing with exception *e*.

If the raised exception cannot be handled by FER, then the CAA has to undo all its effects on the external objects. This task is done by BER.

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

**Figure 7. Creating a new Role class**

As shown in Figure 4, BER is composed of two substates, *Roll back* and *Compensation*. Compensation has to execute a specific task if there is at least one *external* object that needs manual recovery or a *composite* CAA has been called being at state *Service* (normal behaviour) or at state *FER*. Compensation is achieved by defining a compensator for each CAA role. A compensator is made by creating a new class that extends the *Compensator* class. The *recovery* method has to be re-implemented by the programmer. 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 Executing a CAA

Notice that so far, how the classes in the framework are instantiated to create a CAA has been shown. Now how these objects behave when the CAA is activated will be explained. 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. The *Java* code in Figure 8 represents the sequence of operations that each manager will execute once it is activated. The first activity a manager executes is to synchronise itself with all


IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

other managers that are taking place in the CAA. This is done by calling the *syncBegin* method (line 2). Remember that there is a leader manager that is responsible for this task.

```
1  try{
2        syncBegin(); // synchronising upon entry
3        // if pre-condition is not true
4        if(!roleManaged.preCondition(extObjs))
5           throw new PreConditionException();
6        // executing the role
7        roleManaged.bodyExecute(this,extObjs);
8        // waiting for everyone before
9        // checking post-conditions
10       syncEnd();
11       // if post-condition is not true
12       if(!roleManaged.postCondition(extObjs))
13          throw new PostConditionException();
14       syncEnd(); // exiting synchronously
15 }catch (Exception exRole) { //FER
16       try{
17         // applying exception resolution algorithm
18         exRole = exceptionResolution(exRole);
19         // launching FER for the found exception.
20         handlerExecution();
21       }catch (Exception exFER) { //BER
22         try{
23            // executing compensation and roll back
24          restoreExecution();
25            // returning ABORT
26            roleException = new AbortException();
27          }catch(Exception exBER){
28            // there was a problem in the BER execution
29            // returning FAILURE
30          roleException = new FailureException();
31         }
32       }
33 }
```

**Figure 8. Manager execution**

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 4). 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 5) and caught by the *catch(Exception e)* block (how an exception will be dealt with, is explained later).

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 7). 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 synchronises with all the other managers (line 10) before testing its post-condition (line 12). If the post-conditions are satisfied, then the manager will synchronise with all the other managers (line 14) and the CAA will finish successfully. Executing this sequence of steps corresponds to the arrow that goes from state *Service* to state *S1* in Figure 4.

The *catch(Exception e)* block (lines 15-33) will be executed if 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 to *leader* manager for interrupting[1] all the 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[2] from those that have been raised (line 18). When such an exception is found, the *leader* informs all managers about that exception and FER (for the found exception) is activated (line 20). If the *handlerExecution* method completes its execution, then its post-condition has been satisfied and the CAA can finish. The value set in *roleException* variable defines how successfully was the FER execution. If this variable has a *null* value means that FER has finished normally (arrow that goes from state *FER* to state *S1* in Figure 4). Otherwise, it has achieved a partial solution and *roleException* variable that contains the exceptional result that has to be returned to the enclosing context (arrow that goes from state *FER* to state *S2* in Figure 4).

Now, if the exception resolution algorithm could not find a common exception (arrow that goes from state *Service* to state *BER* in Figure 4), or the *handlerExecution()* method raised *PostConditionException()* exception because FER did not satisfy the post-condition (arrow that goes from state *FER(e)* to state *BER* with label *ferDone(e)[not(postCond)]* in Figure 4) or other exceptions were raised in the FER (arrow that goes from state *FER(e)* to state *BER* with label *Exception(e)* in Figure 4), then the BER mechanism will be started (lines 21-32).

BER calls the *restoreExecution* method (line 24) to undo the CAA effects. Once this method has executed, the *roleException* variable is set with *Abort* value (line 26) 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 30). This statement corresponds to the arrow that goes from state *BER* to state *S3* in Figure 4.

### 3.4 Frameworks comparison

This section summarises the main differences between DRIP [2] and CAA-DRIP frameworks. As said, the aim of CAA-DRIP is to provide an easier way to implement designs based on CAAs, therefore that is the reason why the main differences are on aspects that take place in the implementation phase. CAA-DRIP allows programmers to deal with the same abstractions (*Role, Handler, Compensator* and so on) managed in the design phase and which are written on the documentation that drives the implementation phase. Thus, this direct correlation between design

[1] Notice that a role will be interrupted only if the role is ready to be interrupted, i.e. the role is in a state that it can be interrupted.

[2] In the worst case, the common exception is *Exception*.



IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

| Characteristics/Framework | DRIP [2] | 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, that handles $m$ 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, that handles $m$ 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. Comparison between DRIP and CAA-DRIP frameworks**

and implementation helps both in the code production and the modification of already existent code as well as the inspection of the code to detect defects (testing).

Other advantage of using CAA-DRIP instead of DRIP is that the first one requires less number of objects to instantiate a CAA. This reduction in memory use has been achieved by requiring only one *Manager* object for each *Role, Handler* and *Compensator* object.

But, the main advantage of had reduced the number of *Manager* objects consists in requiring less threads (three time less, in the worst case) for running a CAA. It is because the Manager class is implemented to manage the execution of a Role, and part of this management corresponds to create a new thread to perform the tasks programmed into the *body* method for each class that extends from *Role* class.

Table 1 surveys the comparison between both frameworks with respect to the previous discussed aspects.

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

The "Insulin Pump" 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, with this therapy, is to copy the way in which the pancreas secretes the insulin. Therefore the pump constantly supply fast-acting insulin to the patient's body according to information that it receives from the sensor.

The tiny sensor used is only a piece of hardware without embedded software that has an integrated small transmitter that communicates wirelessly and continually the patient's glucose level to the pump. Every $T_{SensorValue}$ units of time the sensor sends an updated glucose value.

The pump has an internal clock that provides the current time at any instant. Based on this clock, the FTIP system can control the other parts that compose the pump, as for instance the motor. This motor works for $T_{Delivery}$ units of time. The $T_{Delivery}$ value is defined by the FTIP system according to the patient's glucose value sent by the sensor and the individual settings defined by a doctor. The pumps also contain a cartridge with fast-acting insulin that is supplied to the patient's body by a cannula that lies under the patient's skin. The motor is connected to a piston rod that sends a plunger forward so that the insulin is delivered to the body. Therefore, the $T_{Delivery}$ units of time that the motor works will deliver (by the cannula) the necessary amount of insulin (**deliveredInsulin(m)**) in a non-diabetic person. The plunger only moves to its initial position (by pressing the **Stop** button) when the pump user needs to fill the cartridge with insulin.

Because the pump delivers insulin almost continuously, and as said, any interruption or excess in the supplying may result in serious problems to the patient, it is reasonable to make checks to be sure that the pump is working properly.

The system makes sure that each amount of insulin that has to be delivered does not drop out of the safe range programmed. The pump has an *infrared motion detector* that is used by the system to check the correct movement of the plunger and another one to check the status of the motor. Both detectors help to determine if the pump is delivering the insulin properly.

Therefore, the FTIP system, using the internal clock and the embedded detectors into the pump, 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 that has 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 system full stops the pump works and it sounds an alarm to alert the patient about the


IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

current situation. The alarm will remain ringing till the patient switch it off (**E5**). Instead, when the quantity of insulin in the cartridge is less than the *low limit* parameter, the system will ring the alarm, as a warning, for only $T_{Warning}$ units of time.

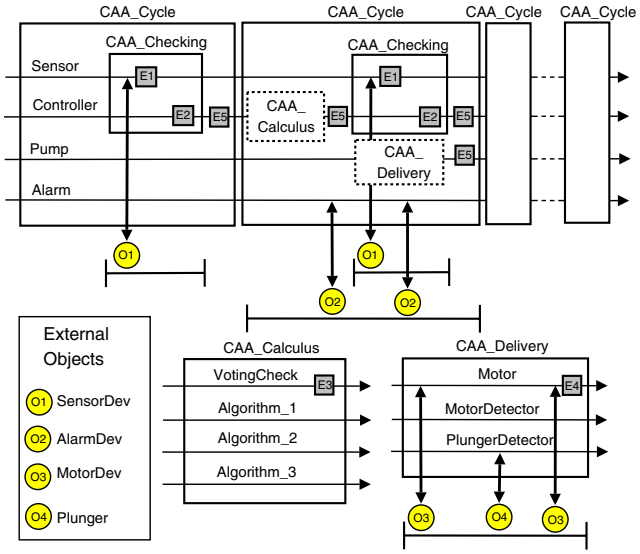


**Figure 9. CAA Design.**

In order to satisfy the previous requirements a set of CAAs that interact cooperatively among them is defined. Figure 9 represents one possible trace of the CAAs behaviour in runtime, in which any exception is raised. In this Figure is also possible to see where exceptions could take place. *CAA_Cycle* is the outmost 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, that consists of getting the current patient's glucose level, which 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 (piston rod, plunger) that are activated by a motor working. Thus, 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 be working ($T_{Delivery}$ value). 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, only the accesses to external objects is shown in Figure 9

```
1  public void body(ExternalObjects eos)
2    throws Exception, RemoteException {
3     try{
4       //Getting information from the enclosing context
5       Loop loop = (Loop)eos.getExternalObject("loop");
6       if(loop.isfirst()){
7        //launching nested CAA_Checking
8        ExternalObjects checking =
9         new ExternalObjects("checking");
10       roleControllerChecking.execute(checking);
11       //getting outcome from CAA_Checking
12       SensorValue sv =
13        (SensorValue)checking.getExternalObject("sv");
14       //Sending information to the enclosing context
15       eos.setExternalObject("sv",sv);
16      }else{
17        //getting sensor value
18        SensorValue sv =
19        (SensorValue)eos.getExternalObject("sv");
20        //launching composed CAA_Calculus
21        ExternalObject calculusREOs =
22        new ExternalObjects("calculus");
23        calculus.setExternalObject("sv",sv);
24        roleVotingCheck.executeAll(calculus);
25        //getting outcome from CAA_Calculus
26        Time tDelivery =
27        (Time)calculus.getExternalObject("tDelivery");
28        //passing information to Pump role
29        pumpQueue.put(tDelivery);
30        //launching nested CAA_Checking
31        ExternalObject checking =
32        new ExternalObject("checking");
33        roleControllerChecking.execute(checking);
34        //getting outcome from CAA_Checking
35        SensorValue sv =
36        (SensorValue)checking.getExternalObject("sv");
37        //getting values from CAA_Delivery by Pump role
38        Status st = (Status)pumpQueue.get();
39        //Sending information to the enclosing context
40        eos.setExternalObject("sv",sv);
41        eos.setExternalObject("st",st);
42      }
43    }catch (Exception e) {
44      //Local handling for Controller exception;
45      throw e;
46    }
47 }
```

**Figure 10. Body method of Controller class**

The Java code in Figure 10 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 by the sensor. Once


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.

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*. Thus 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 be running (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, to improve the performance, the *Controller* role launches *CAA_Checking* (line 33) to get information from the sensor that 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 the control to the enclosing context where *CAA_Cycle* is embedded.

## 5 Conclusions and Future Work

In this paper all the different concepts that concern CAAs have been put together. Based on these concepts, a formal description of the CAAs behaviour based on the *Statecharts* language has been provided. Both the *Statecharts* description as well as the DRIP framework were used as reference to drive the implementation of a new framework. This framework, named CAA-DRIP, is an object-oriented solution which provides separated classes (roles, handlers and compensators) for implementing, by a one-to-one mapping, each CAA concept. This separation of concepts, useful for programmers to carry out the implementation phase, together with the performance improvements, are the main reasons why CAA-DRIP represents a better cost-effective solution with respect to its ancestor, DRIP. Cheaper solutions are clearly possible when the CAA support is implemented as a special part of the middleware/run-time. It is future work to investigate these solutions.

Building a tool to become CAA-DRIP a *black-box* framework is part of the future work, too. In fact, even if the abstraction level of CAA-DRIP is the same of the CAA design, to use the framework requires a deep knowledge of its internal composition. Based on this characteristic this kind of framework can be called *white-box*. On the contrary, the *black-box* term means that programmers will not need anymore to know the framework details to implement a CAA design. This work is being carried out in the context of the *CORRECT* project [5].

## 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, P. Periorellis, and R. Razavi. A. Zorzo is partially funded by CNPq Brazil. A. Romanovsky is supported by IST RODIN project.

## References

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

[2] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99*, pages 435–446. ACM Press, 1999.

[3] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Sofi. Eng*, pages 1491–1501, 1985.

[4] C. Bertolini, L. Brenner, P. Fernandes, A. Sales, and A. F. Zorzo. Structured Stochastic Modeling of Fault-Tolerant Systems. In *MASCOTS*, pages 139–146, 2004.

[5] Correct Web Page. http://lassy.uni.lu/correct, 2006.

[6] Ferda Tartanoglu and Nicole Levy and Valerie Issarny and Alexander Romanovsky. Using the B Method for the Formalization of Coordinated Atomic Actions. In *Proc. ICSE 2003 Workshop on Software Architectures for Dependable System*.

[7] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, A. Zorzo. Formal Development and Validation of Java Dependable Distributed Systems. In *ICECCS'99*, pages 98–108, Nevada, USA, October, 1999.

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

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

[10] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

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

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

[13] A. Romanovsky, P. Periorellis, and A. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. *Proceedings of ISADS 2003*, pages 99–106, 2003.

[14] 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 Trans. on Computers*, pages 164–179, 2002.

[15] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery. In *Symposium on Fault-Tolerant Computing*, 1995.

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

[17] A. F. Zorzo. A Production Cell Controlled by Dependable Multiparty Interactions. In *Technical Report Series No. 667*. University of Newcastle, March 1999.



IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 16:52:10 UTC from IEEE Xplore. Restrictions apply.