An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling

D.M. Beder 1,2, A. Romanovsky 2, B. Randell 2, C.R. Snow 2 and R.J. Stroud 2

1 University of Campinas (UNICAMP) P.O. Box 6176, 13083-970, Brazil
2 University of Newcastle upon Tyne NE1 7RU, United Kingdom

Developing and applying advanced approaches for system structuring is vital for fighting ever-increasing complexity of modern and future software systems. The concept of Coordinated Atomic (CA) actions has been developed at Newcastle University for designing and structuring complex concurrent and distributed applications. Certain successful experience has been gained in applying them in several application areas. The purpose of the research, some initial results of which we report here, is twofold: to show how CA actions can be used in a new application area (a railway control system) and to analyse how the design patterns which have been developed using our previous experience can help in designing such system using CA actions.

1 Introduction

Modern concurrent and distributed applications are getting more complex so that providing fault tolerance requires the use of special structuring mechanisms to help in reducing this complexity. Recently we have developed a concept of Coordinated Atomic (CA) actions [14] to be used for structured design of such systems and for providing fault tolerance using various techniques (including, rollback, design diversity and exception handling). In the subsequent research we have gained significant experience in designing several applications using CA actions; in particular, a series of Production Cell case studies [17], including a fault tolerant one [16] and a real time one [9]. In another experiment we have designed a distributed internet Gamma computation [10]. In spite of the evident success of these experiments, we realise that our experience is still limited to one or two application areas and feel that there is a need for more case studies to demonstrate general applicability of the concept. In this paper we examine the applicability of the CA action concept to a new application area with different types of activity coordination, resources, requirements, faults and real time constraints. We report here our first results on designing a subsystem of the railway control system which deals with train control and coordination in the vicinity of a station. It is our belief that this new case study will help us in developing further the CA action concept using the experience gained.

Another active area of research in fault tolerance is developing design patterns which can assist in applying different fault tolerance techniques. Recently several patterns [1,5] have been proposed to help system designers to use CA actions and to develop bespoke CA action mechanisms which best suit particular applications. Within this new case study our intention is to apply such patterns and to analyse the results gained.

One challenge we face in this new case study is to be as realistic as possible but at the same time to be able to check our design: we are using a simulation system and a trainset which are available at University of Newcastle to implement our ideas. Real railway systems are used for providing realistic scenarios and problems to be addressed.

Our analysis shows that the initial case reported in this paper can be extended in several directions and can provide us with a family of useful case studies based on more complicated and more realistic scenarios to allow us to apply CA actions and corresponding patterns in a wide context. Some of the examples are: designing crossing control using CA actions, considering different types of trains and of items to be transported (mails, bulky items, freight, etc.), softening fault assumptions. Another way of extending the initial case study is by introducing autonomous subsystems which are controlled separately but are still connected by the railway: plants producing parts, plants with assembly lines, airports, stores, cities, etc. which effectively are glued together by a railway system and which interface it via station interfaces. The challenge for future research would be to develop a railway control system supporting complex functionalities of such systems of systems.

2 Coordinated Atomic Actions

The Coordinated Atomic action (CA action) concept was introduced as a unified general approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed
Each CA action is designed as a stylised multi-entry unit with roles which are activated by action participants and which cooperate within the CA action. Logically, the action starts when all roles have been activated and finishes when all of them reach the action end. The action can be completed either when no error has been detected, or after successful recovery or when a failure exception has been propagated to the containing action. If an error is detected inside an action all roles are involved in recovery. External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the CA action start and completion has the ACID properties with respect to other sequences. A CA action execution looks like an atomic transaction to the outside world. The state of the CA action is represented by a set of local and external objects; the CA action (either the action support or the application code) deals with these objects to guarantee their state restoration (which is vital primarily for backward error recovery). Participants can only cooperate (interact and coordinate their executions) through local objects.

The CA action concept allows designers to deal with system complexity by encapsulating several state transitions and an activity of multiple components into one atomic unit with a clearly-defined interface. Systems are designed recursively using action nesting. Fault tolerance features are always associated with such units confining all errors. When an action is not able to tolerate an error a failure exception is propagated to the containing action passing the responsibility for recovery to the higher system level and leaving the objects involved is the action execution in well-defined states which facilitate the following recovery at the higher level.

3 Design Patterns

As the size and complexity of systems increase, software designers have recognised the importance of exploiting and reusing knowledge in the definition of their overall system architecture. This kind of reuse can be achieved by applying design patterns [4]. Design patterns describe solutions for specific problems that occur over and over again in software design. Recently several patterns [1,5] have been proposed to help system developers in designing dependable concurrent object-oriented systems. These patterns are based on the Reflection architectural pattern [2] that employs a meta-level architecture. The architecture is composed of a base level implementing the application logic and a meta level consisting of meta components which are responsible for providing the application dependability requirements in a way that is transparent to application designers. Because of such separation of concerns the application designers concentrate their attention on the functional requirements, abstracting from the dependability requirements.

The Reflective Contract pattern provides a design solution for detecting errors. Application designers structure a set of contract classes, which check the pre- and post-conditions of each service, returning exceptions if these conditions are not satisfied. The Handler pattern provides the explicit separation between the normal and error-handling activities. Application designers structure their components by creating normal and exceptional classes. The normal classes consist of methods implementing component normal services, while the methods of exceptional classes are the handlers for the exceptions raised during the execution of the normal services.

The Competitive Collaboration pattern offers a design solution for implementing the Coordinated Atomic action concept. This pattern separates objects into two well-defined levels. The base level provides designers with classes for creating the CA actions and for defining nested actions (to allow better structuring of normal and error handling activities of the enclosing action). The meta level implements the management mechanism based on reification of method invocations. This pattern introduces five types of objects: Action, Participant, MetaAction, MetaParticipant and MetaAtomic (Figure 1). Designers extend the Action and Participant classes by adding application-specific information. Instances of the Action have references to action participants, internal and external action exceptions, the enclosing action, the nested actions and shared (local) objects used for inter-participant communication. Instances of the MetaAction class are associated with instances of the Action subclasses and are responsible for exception resolution and for synchronising the action participants. Instances of the Participant
subclasses represent the action participants; they hold references to the action and to the object (role) and its method that will be executed during the action. Instances of the MetaParticipant class are associated with instances of the Participant subclasses; they are responsible for executing the application method which is held by its associated participant, for informing the corresponding MetaAction about the end of this method execution, and for invoking the handler associated with a resolved exception.

The Exception pattern allows application designers to deal with simple exceptions and their (concurrent) compositions in a uniform way. Contract classes (the Reflective Contract pattern) can be designed to check the pre- and post-conditions of each role method. The exceptional classes (the Handler pattern) implement handlers for the exceptions which can be raised during the action execution. External (transactional) objects are simple objects which are derived from the Object class and associated with the instances of the MetaAtomic class: this is how their transactional semantics is guaranteed.

Figure 1 – The Competitive Collaboration pattern

4 Case Study

This section describes the Station Case Study that is a subsystem of the railway control system, which deals with train control and coordination in the vicinity of a station. Trains transport passengers from a source to a destination station. Stations usually have several platforms on which trains can stop (no more than one train on each platform at a time). We assume that trains can execute some join cooperative activity when they stop at the same station together, for example, passengers can change trains during this stop to make their connections faster.

4.1 Basic System Requirements

A correct control program must satisfy certain requirements, namely:

- **Safety**: (i) collisions of trains at stations must be prevented, (ii) sufficient distance must be maintained between trains.
- **Fairness**: trains get access to the stations fairly.
- **Liveness**: liveness is the property of the whole railway control system. Stations only provide enough information for a possible component of the railway control system that will do the scheduling. Stations obey it. Trains at stations must take all passengers who have to be in these trains. All passengers who have to change within an action always do this.
4.2 Actuators and Sensors

In this section, we describe actuators that can be used to take influence on the system, and the sensors, which provide the control program with information on the state of the system.

- **Actuators.** The system can be controlled using the following commands: set the direction of railway switches, start/stop trains, decrease/increase the speed of trains and (iv) reverse the direction of trains.

- **Sensors.** We assume that railway tracks contain sensors that report useful information to the control program. These sensors are the only means for determining the position of the trains. Hence, it is extremely important that the integrity of the information provided by these sensors be checked and verified, because this information is not used only for keeping track of the locality of the trains but also for preventing disasters from occurring.

4.3 Failure Definitions

Before defining and analysing various possible failures, we state some assumptions about this case study: (i) the system clock is fault-free and does not fail; (ii) values of sensors and clocks are always transmitted correctly without any loss or error; (iii) all sensor failures are indicated by sensor values; and (iv) only one failure can happen on each track or on the train using this track during the interval of interest.

4.3.1 Time-Related Failures

This case study works correctly, if the following timing constraint is fulfilled:

- A train must not arrive at one station after $t_{\text{entry}}$ and leave it before $t_{\text{exit}}$. If a train fails to arrive before $t_{\text{entry}}$, then the others trains continue their activities but some corrective actions must be taken, for example, leaving some passengers waiting for the late train.

4.3.2 Sensor Failures

The failure modes of sensors are either that they trigger when they should not (recognised by unexpected triggering of the sensor), or do not trigger when they should (recognised by a different but predictable sensor being triggered). These conditions may be intermittent or permanent. Intermittent faults can probably be handled by re-trying (by backing up the train if necessary), and permanent failure to trigger by a sensor could also be handled sensibly, once the system was aware of it. Continuous triggering by a rogue sensor could be problematic, especially if that sensor could at certain times be part of a route.

4.3.3 Actuator Failures

This covers two failures, because it relates to the situation where a train or switch fails.

4.3.3.1 Switch Failures

If the switch is broken, it is most likely that the switch is set in one direction, and refuses to change. It is also possible that the switch does not change in response to a request, but repeating the request will achieve the desired effect (i.e. an intermittent fault).

4.3.3.2 Train Failures

Again the fault may be permanent or intermittent, and re-trying the operation may solve the problem.

- If a train fails to start, there is no actual harm done, except that some of the timing constraints will not be met. This will affect not only the failing train, but also the trains which may subsequently depend on the failing train moving away from where it is, or arriving at a future destination.

- More interesting is the potentially disastrous situation of a train failing to stop - the equivalent of a brake failure. In this case, of course, the failing train will not respect its route. The only recovery is to direct the train to a safe part of the railway, while at the same time ensuring that all other trains are stopped if there is any danger of the their using part of the rogue train route.

5 Newcastle Trainset

The trainset is a digitally controlled model railway that have been used at the University of Newcastle for investigations, experiments and student projects in real time and distributed control [11]. The railway layout is mounted on three separate boards that can be independently controlled by separate computers.
5.1 Layout Hierarchy

The layout boards themselves contain a number of physical entities. These are: segments of continuous track, switches, and crossings. There are two varieties of switch, points and crossovers, which may be set in one of two directions, straight and curved. In addition, a number of sensors are placed at various locations around the layout. Trains move along the tracks, their routes being determined by the settings of the switches. A train passing over a sensor causes the sensor to be triggered. The switches may receive commands to instruct them to change their state (and hence potentially the path to be taken by a train). Trains may also receive commands instructing them to change speed (which includes the operation of stopping and reversing the train). Finally, it is possible to determine which of the sensors has been triggered.

Each computer contains a purpose-built device for controlling its part of the layout, and this appears to the controlling computer as a standard input/output device. The view of the operating system at this level is that there is a separate output device for each train and each switch, and a single input device representing all of the sensors. Within each of the controlling computers, a server program runs, presenting to its clients an object-oriented view of the basic facilities of the trainset.

The primary object is the trainset object, which offers methods for creating train objects, switch objects, and a single object representing all of the sensors. The precise details of the mechanisms by which clients communicate with the server are not relevant to the discussion, and are therefore be omitted. Full details are available in [12].

At this level, the system has no knowledge of the inter-connection of these objects. Methods are provided for changing the setting of a switch, and for reading the identities of the triggered sensors. Train objects have methods to set the train speed, including stopping and reversing the train. This abstraction leaves the client program free to decide the constraints on the movement of the trains (e.g. routing, collision avoidance, etc.)

These servers allow the trainset objects to be accessed remotely, using Inter Language Unification (ILU) [7], so that client programs can call the methods of these objects from remote locations, including, for example, a Web server.

As mentioned earlier, the client programs are free to create higher-level abstractions which may be used to allow trains to traverse arbitrary routes within the layout, to avoid collisions between trains moving around the layout, and to provide other similar facilities. Cooperation between trains, such as the station activity described in this paper, is another example of the use of such a higher-level abstraction.

Rubira [11] has suggested that such an abstraction might consist of stations (which are in one-to-one correspondence with a subset of the sensors) and connectors. Unfortunately, the use of the term “station” in this context conflicts with its use in other parts of our discussion, and hence we shall use the term location to refer to Rubira’s “stations”.

A connector is either an endpoint, a crossing or a switch. An endpoint is defined as a buffer (a position on the layout where a train may not make progress without reversing) or a border point, i.e. a position at the edge of a board where a train may pass to an adjacent board. Locations and connectors are joined together by edges. It is at this stage that the actual topology of the layout is used to determine the relationship between the edges and the locations/connections. Finally, the layout can be regarded as a collection of sections, which may loosely be described as being defined by an ordered pair of “adjacent” locations. A section object will contain information regarding the switches (if any) lying within the section, and their settings. Because the locations defining a section are adjacent, no other location may lie within a section. Other sensors may appear within a section, but these are sensors which do not map onto a location. For the purposes of this case study, we have designed more two types of objects: platform that consists of two consecutive section objects and station that is a set of neighbouring platform objects.

The higher level of abstraction described in the previous paragraph clearly requires information regarding the interconnection of the various components of the layout, and this information is provided in the form of a board definition file. Each board has a distinct board definition, although it is clear that each border point on a board must contain information indicating the border point on the adjacent board to which it is joined. Board data is clearly static, describing the physical relationships between the hardware entities in one part of the layout.

In describing the dynamic behaviour of the trainset, an abstraction defining a train is required. Other possible abstractions which might be considered are routes, defined as sequences of locations visited by a train, and journeys. A journey can be defined as the route taken by an object (which might be a passenger, a piece of freight, or similar). A journey will involve utilising part or all of a number of routes taken by trains. The dynamic nature of trains also implies that as a train traverses its assigned route, it will occupy the sections comprising the route. It is this notion of
occupancy, and indeed a corresponding notion of claiming a section (that is, indicating an intention to occupy the
section) which is at the heart of collision avoidance. There are clearly, therefore, certain dynamic attributes of the
trainset which are needed to control the safe and error-free movement of trains around the layout.

Figure 2 - Trainset software system organisation

Figure 2 shows the hierarchy of abstractions involved in this description, and we observe that there is a separate
hierarchy for each board. We have already noted that there is a (static) connection between the board definitions
(shown by the dotted lines in figure 2), in that a border point on one board is required to correspond with a border
point on the adjacent board. However, there is also a dynamic link in that a train whose route requires it to transfer
from one board to another must request permission to do so from the receiving board. If this permission is granted,
then there must also be a handshake exchange signifying that responsibility for the train has passed from the donat-
ing board to the receiving board.

5.2 Fault Hierarchy

Within our discussion we are considering faults which are probably caused by hardware failure, although faulty
software may manifest the same symptoms. At the lowest level of the hierarchy there are no detectable faults. This is
because commands are sent to trains and switches, but there is no mechanism (at this level) for discovering whether
the command has been obeyed. Similarly, the lowest level can only return the values of the sensors which have been
triggered. It has no notion of which sensors should be triggered, and therefore no knowledge that a sensor has either
been triggered in error, or has not been triggered but should have been.

The second level of the hierarchy is also unable to contribute to fault detection, since this level is merely adding
static information to define interrelationships between layout entities.

It is only when considering the movement of trains around the layout that errors can be detected. Commands are
propagated down the hierarchy, and are transformed into calls on the methods on the trainset objects. These methods
are unable to provide feedback, and consequently, the clients cannot know whether the commands have actually
been carried out correctly. The only way that the client level can detect the occurrence of an error is by observing
the triggering of an unexpected sensor.

6 Case Study Design and Implementation

Our design for this case study separates the safety and functionality requirements between a set of CA actions that
will occur during system execution and a set of sensor/actuator controllers that will determine the order in which the
CA actions will be executed. We assume a layered structure of the control system and that the safety requirement
(section 4.1) is guaranteed by the underlying layer (the trainset API) on the top of which we build the CA actions
and controllers. Also, we assume each passenger has a ticket that describes her/his journey. If some failures affect
this journey, the component of the railway control system that does the scheduling can be used to recalculate the
journey. This scheduler component is used for providing the liveness property (see section 4.1).
6.1 The Station Action Design

We will now show how CA actions can deal with various types of failures. Figure 3 illustrates the interactions (themselves involving nested CA actions) between the participating threads within the Station action. This action has four roles: Train1, Sensor1, Train2, and Sensor2, and represents the cooperation that allows trains to exchange passengers.

Figure 3 – The Station action

The Station action coordinates the execution of a complex activity concerned with cooperation between several trains calling at a particular station. Trains execute some activities when they stop at one station platform: passengers can change, embark or disembark from trains. Sensor roles check whether trains stopped at this station, and whether they leave the station after the cooperative activity has finished. Station platforms are external (transactional) resources and they have waiting rooms where passengers wait for their trains. Station actions know the number of train participants and hence the number of platforms they need.

```
CAA Station;
Interface
Use Platform;
Roles
Train1: platform1;
Sensor1;
Train2: platform2;
Sensor2;
Exceptions ;;exceptions to signal
FirstStopException, SecondStopException, BothStopException,
FirstStartException, SecondStartException, BothStartException;
Body
Use CAA ;;specify nested actions
Train1Entry, Train2Entry, ExchangePassengers, Train1Leave, Train2Leave;
Objects ;;shared local objects
PlatformChannel: Channel;
Exceptions ;;internal exceptions
TrainLateException;
Train1StopException, Train1StartException,
Train2StopException, Train2StartException;
BothStopException, BothStartException;
Handlers
Train1StopHandler, Train2StopHandler, ...;
Resolution ;;exception resolution graph
BothStopException -> Train1StopException, Train2StopException;
BothStartException -> Train1StartException, Train2StartException;
...
End Station;
```

Figure 4 – The Station action specification

The Station action is described using the COALA notation (see Figure 4), which was developed for the formal specification of CA actions [13]. The COALA definition of a CA action (keyword: CAA) is made of two parts: an Interface and a Body section. In the Interface part we declare: the list of classes that represent the external
objects used (Use); the list of the CA action roles (Roles) and the list of the interface exceptions which the CA action can signal to an enclosing action (Exceptions). Each role is parameterised with references on the external objects that it uses. The roles of an action can signal an exception directly but must guarantee that the exception that is signalled has been agreed by all the roles of that action. In the Body part we declare: the list of nested CA action names (CAA); the list of the CA action internal objects (Objects); the list of internal exceptions that roles can raise within a CA action (Exceptions); a resolution graph of the CA action which lists the combinations of internal exceptions which can be raised concurrently, together with the resolved exception which must be activated in each case (Resolution). After a resolving exception is identified, the corresponding handler declared in the Handlers part will be invoked.

6.1.1 Dealing with Single Failures

During the execution of a CA action, if a failure (of a component involved in this CA action) occurs and is detected, a corresponding exception will be raised within the action by one of its roles. The exception is propagated immediately to other roles of the action and all roles then transfer control to their exception handlers for this exception so that they can attempt to perform appropriate error recovery.

By way of example, we outline the basic requirements for the handlers of three different exceptions:

- **Handler for the train late failure.** If a train fails to arrive before \( t_{\text{entry}} \), then the \( \text{TrainLateException} \) is raised. The \( \text{ExchangePassengers} \) action performs error recovery to this particular exception by embarking all passengers who have to be in the non-late train and leave some passengers at station waiting for the late train. When the late train arrives at a station, some of its passengers may have lost their connections; hence passenger rescheduling must be done.

- **Handler for the train1 stop failure.** The \( \text{Train1Entry} \) action performs error recovery by repeating the stop request and by backing up the train to previous section. If the failure persists, the \( \text{Train1Entry} \) action will produce an exceptional outcome (Train1StopException). The \( \text{Station} \) action performs error recovery to this particular exception by embarking all passengers who have to be in the non-fault train, rescheduling the passengers who should embark at the fault train and producing an exceptional outcome (FirstStopException).

- **Handler for the train2 start failure.** The \( \text{Train2Leave} \) action performs error recovery by repeating the start request. If the failure persists, the \( \text{Train2Leave} \) action will produce an exceptional outcome (Train2StartException). The \( \text{Station} \) action performs error recovery to this particular exception by try to replace the failed train. If a spare train is not available at station, the \( \text{Station} \) action will reschedule all passengers of the faulty train and produce an exceptional outcome (SecondStartException).

6.1.2 Dealing with Concurrent Failures

Now let us address the problem of possible concurrent failures. In the interests of simplicity, we assume that only one failure can happen on each track or on the train using this track during the interval of interest.

For a \( \text{Station} \) action, various exceptions are defined based on failure analysis and an exception graph for resolving concurrent exceptions is defined (see Resolution part in Figure 4). By way of an example, we outline the basic requirements for the handlers of two resolved exceptions:

- **Handler for both trains stop failure.** The \( \text{Station} \) action is unable to perform any error recovery to this particular exception. The only thing the \( \text{Station} \) action can do is produce an exceptional outcome (BothStopException).

- **Handler for both trains start failure.** The \( \text{Station} \) action performs error recovery by trying replace both trains. If spare trains are not available, the \( \text{Station} \) action will reschedule all passengers of these faulty trains and produce an exceptional outcome (BothStartException).

6.2 Controllers

Given a set of \( \text{Station} \) actions to control the interactions of trains, controllers are used to determine the stations at which trains stop and hence the order in which the \( \text{Station} \) actions are executed. Train controllers determine the train routes and trains with their schedules are embedded into such controllers. Sensor controllers provide train controllers with information (triggering of sensors) about the positioning of the trains. In our design train controllers...
activate Train roles of a Station action while sensor controllers activate Sensor roles. The `trainController1` and `trainController2` are shown below as a simple example.

```plaintext
TrainController1 {
    StationA.join("train1"); // stop at Station A
    StationB.join("train1"); // move train from Station A to Station B
    StationC.join("train1"); // stop at Station C
}
```

```plaintext
TrainController2 {
    StationD.join("train2"); // stop at Station D
    StationE.join("train2"); // move train from Station D to Station E
    StationF.join("train2"); // stop at Station F
    StationG.join("train2"); // move train from Station F to Station G
    StationH.join("train2"); // stop at Station H
    StationI.join("train2"); // move train from Station H to Station I
    StationA.join("train2"); // stop at Station A
    StationC.join("train2"); // stop at Station C
    StationD.join("train2"); // stop at Station D
}
```

As illustrated in this example, trains can also stop alone at one station (for example, `train1` at `StationB`). We have also designed the StationAlone action that arranges for trains to stop alone at stations. Due to space limitation, we will not show the design of this action. However, the StationAlone action design is very similar to the Station action design and the most important design aspects discussed in section 6.1 are still valid.

### 6.3 Design Patterns and Implementation Issues

The aim of this section is to show how the patterns presented at section 3 can be applied to the design of this case study. We have used the Competitive Collaboration pattern to design our actions. The Station, TrainEntry, ExchangePassengers and TrainLeave classes extend Action (Figure 1) and instances of these classes represent the Station action (and its nested actions) that coordinates the execution of an activity concerned with cooperation between two trains calling at a particular station. The TrainSensor and TrainActuator classes extend Participant and instances of these classes represent the Station action participants. Roles are activated by these participants. TrainActuator instances influence Train instances by sending commands to stop at stations and leave stations after the cooperative activity has been finished. TrainSensor instances check whether trains failed to respond a request.

![Diagram](image.png)

**Figure 5 – The Station action design**

Figure 5 shows the design of the cooperative activity between train1 and train2 calling at stationA (train1 stop at platform P1 and train2 at platform P2). Platforms P1 and P2 are objects associated with MetaAtomic instances (section 3) that guarantee the transactional semantics on these objects.
We use the Reflective Contract pattern (Figure 6) to structure the TrainContract class that detects errors. This class detects:

- **train stop failure (post-condition check of Stop method):** if the next sensor of the train route is triggered, then the train failed to respond a stop request.
- **train start failure (post-condition check of Start method):** if the next sensor of the train route is not triggered (detected by time-out), then the train failed to respond a start request.
- **train late failure (pre-condition check of ExchangePassengers method):** both trains must arrive before $t_{\text{ENTRY}}$.

In addition, we use the Handler pattern to structure the ExceptionalTrain class, which implements handlers for the exceptions that can be raised by Station action participants, and the Exception pattern to define simple and concurrent (structured) exceptions uniformly.

The sequence of commands necessary to create an instance of Station action can be as follows (Figure 7). Line 12 initialises an array with exceptions that may be raised and must be treated cooperatively by Station action participants. Line 13 initialises an array with exceptions that may be signalled by Station action itself. Lines 01-11 create instances of such exceptions. Line 18 initialises an array with the Station action participants. Lines 14-17 create instances of such participants. Line 19 initialises an array with the external objects accessed by this action. At last, line 20 creates one instance of Station action.

The first version of the control system has been implemented using a meta-level software architecture [1] that provides support for implementing general dependable collaboration-based designs and a trainset Java API [12] available at University of Newcastle that provides necessary tools for executing generic trainset operations. The meta-level software architecture utilised was implemented using the Java programming language without any changes to the language itself by means of a meta-object protocol called Guaraná [8].
7 Conclusions

We believe that the first results of this experimental research are promising although more effort is needed to develop a realistic full-scale control system and to draw the final conclusions. This case study clearly suits our purposes: it relies on real life complex applications but, if necessary, can be made simpler for the research purposes; it allows us to add real life complexity gradually; we can use the existing trainset and the simulator for experiments. CA actions offer a very powerful framework for designing complex systems in a structured way; they facilitate developing systems meeting high dependability requirements; the resulting systems have a clear structure which makes reasoning about the system (including reasoning about dependability properties) simpler. Backing the CA action design with design patterns clearly facilities system development and makes it more disciplined and less error-prone without having to use special language constructs.

Acknowledgements. This research is partially supported by the European IST DSoS project (IST-1999-11585). Delano Beder is supported by CAPES (Brazil) under grant number BEX0553/99-9.

References