On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems

Delano M. Beder [1], Brian Randell [2], Alexander Romanovsky [2] and Cecília M.F. Rubira [1]

[1] *University of Campinas (UNICAMP)*
P.O. Box 6176, 13083-970, Brazil

[2] *University of Newcastle upon Tyne*
NE1 7RU, United Kingdom

Abstract

Modern concurrent and distributed applications are becoming increasingly complex; so, in order to provide fault tolerance, special structuring mechanisms are required to help reduce this complexity. Unfortunately, such structuring techniques are mostly introduced as design and implementation features, which complicates their employment. The approach we are proposing in this paper relies on introducing the appropriate software structuring together with associated fault tolerance measures at the earlier phases of software development and on supporting it with special software architectures and design patterns.

0. Introduction

Modern computer system development is complicated by two main factors: their complexity and the need to provide their dependability. Complex systems are prone to errors of many kinds, and the most reasonable way of dealing with them is to accept that any complex system has faults and to employ appropriate features for tolerating them during run time. In this paper, we focus on fault tolerance as one of the chief means for guaranteeing system dependability. The most beneficial way of achieving fault tolerance in complex systems is to use system structuring which has fault tolerance measures associated with it. In this case, structuring units serve as natural areas of error containment and error recovery. To this end, a number of techniques have been proposed to help system developers achieve fault tolerance [11]. These techniques can be classified into masking, forward and backward error recovery features. Our focus is on exception handling, which has proved to be the most general way of providing both forward and backward error recovery.

Many modern complex systems are concurrent and distributed, which requires special structuring techniques and special ways of associating exception handling with structuring units. Atomicity of such units is vital for decreasing system complexity when the system exhibits both normal and, in particular, abnormal behavior. Recently a concept of Coordinated Atomic (CA) actions [24] has been developed to be used for the structured design of such systems and for providing fault tolerance using various techniques (including, exception handling, rollback and design diversity). The 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 object-oriented system. This paradigm provides a conceptual framework for dealing with cooperative and competitive concurrency and for achieving fault tolerance by extending and integrating two complementary concepts - atomic actions [6] and ACID (atomicity, consistency, isolation and durability) transactions [10]. CA actions have characteristics of both of them: atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain consistency of shared resources in the presence of failures and competitive concurrency. This allows tolerating faults of various types, as well multiple concurrent faults occurring in the different components involved in the CA action execution (using an extended resolution mechanism [22]).

Unfortunately there are several factors that complicate the use of modern structuring and fault tolerance techniques. Firstly, programming languages do not include features that support them directly. Secondly, these techniques are mainly developed for employment at the late design and implementation phases. Thirdly, it is not often easy to apply them correctly, as the developers have to take into account many details. The approach we are proposing in this paper relies on introducing the right software structuring with the associated fault tolerance measures starting from the earlier phases of the software development (that is, from architectural design, through detailed design to coding) and on supporting this by special architectural styles and design patterns.

The remainder of the paper is organized as follows. Section 1 presents a generic software architecture for developing dependable object-oriented systems, discusses a set of design patterns which refine the architectural elements of the proposed software architecture and shows the feasibility of our approach by means of a realistic case study. Section 2 proposes approaches to applying

Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)
0-7695-1089-2/01 $10.00 © 2001 IEEE

architectural styles and design patterns to allow system designers to address the typical problems in developing systems of systems. Finally, Section 3 summarizes the contributions of the paper.

1. Concurrent and Distributed Systems

1.1. Coordinated Atomic Actions

A Coordinated Atomic (CA) action is designed as a stylised multi-entry unit with action roles which are activated by action participants cooperating within the CA action. Logically, the action starts when all action 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 all action roles are involved in recovery. When several exceptions are concurrently raised in an action they are resolved using either resolution trees [6] or graphs [22], and a resolved exception is handled by all action participants. 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 a single atomic unit with a clearly defined interface. Systems can be designed recursively using action nesting. Fault tolerance features are associated with such units. 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 in the action execution in well-defined states, thus facilitating the recovery at the higher level. Significant experience has been gained in designing and implementing several applications using CA actions; in particular, a series of Production Cell case studies [25], including one in which faults of various production devices have to be tolerated [23] and one with real-time constraints [16]. In other experiments a distributed Internet Gamma computation [17] and an experimental Internet auction system [19] have been designed.

A number CA action schemes have been implemented based on Java and Ada, and used in the course of this research including ones with synchronous and asynchronous entry, intended for distributed and single-computer settings, relying on decentralised and centralised control, and with concurrency control at various different levels of sophistication (e.g. relying on CORBA [3] transaction service or a simple object locking). In some of these schemes nested actions were immediately aborted and participants interrupted when an exception is raised, in others the support waited until those are completed. Actions and participants were designed as classes, objects or tasks depending on the application requirements and the paradigm used.

1.2. Architectural Description of Systems

CA actions (as well as numerous other fault tolerance techniques) are mostly introduced as design and implementation features., We believe, however, that it is important to have special architectural solutions which can be applied at the earlier phases of system development. Software architecture provides an abstract description of a system by focusing on its structure and abstracting from implementation details. Usually software architecture is decided on during the first design phase, when the basic approach to solving a specific problem is selected. The system is constrained to conform to this software architecture as more details are added.

In general, the description of software architecture consists of the following building blocks [18]: **components** that abstractly characterise units of computation and have interfacing points (component ports) with other architectural elements; **connectors** that abstractly characterise composition patterns among components and have interfacing points (connector roles) with other architectural elements (thus, a connector prescribes the interaction protocol that takes place among the components that are composed through it); and **configuration** that defines the structure of the system by composing a collection of component instances through bindings via connector instances. A software architecture is then defined as a configuration instantiating component and connector types. There is a lot of evidence to demonstrate that it is beneficial to build software architecture upon previous knowledge about related systems. This issue is tackled with the help of the notion of **architectural style**, which provides means for exploiting commonalities between systems. An architectural style defines a set of properties shared by the configurations that are members of the style.

1.2.1. The Idealised Fault-Tolerant Component Style. This style is a specialisation of the Layer architectural style [18] defining a layered system structure in which each layer provides services to the layer above and uses

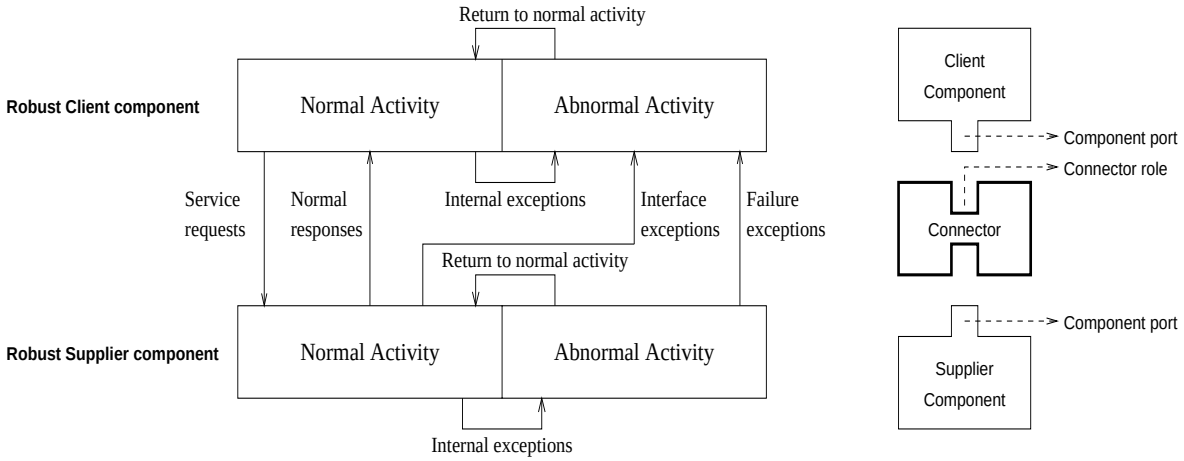

Figure 1 – The Idealised Fault-Tolerant Component Architectural Style

the layer below as a client. This style relies on a well-know concept of an *idealised fault-tolerant component* [11]. If such component cannot satisfy a request for a service, it returns an exception. At each system level this component either deals with exceptional responses from the components below or propagates an exception to the higher system level. There are only two forms of communication between components in this style: service requests and service responses. The components of this style are idealised fault-tolerant components that have two different ports (request or provide). The connector described by this style defines two different kinds of components. ***Client components*** request services of the supplier components (via a request port). The client component is responsible for providing means of handling normal and abnormal responses from the supplier components. ***Supplier components*** receive requests for services and produce responses (via a provide port). A component can serve as a supplier and a client at same time. A component can be a supplier to several clients if overall component consistency is guaranteed.

In an ideal situation, the client and the supplier components interact producing only normal responses. However, considering that it is likely that each system has to deal with errors of different types, exceptions may be produced as responses to the client requests that cannot be satisfied. An idealised fault-tolerant component is divided in two parts, the normal part that implements its normal activities and the abnormal part that implements the measures for tolerating faults that cause exceptional responses (Figure 1).

The abnormal responses (i.e. exceptions) that can be returned by a component are: the interface exceptions that are signalled in response to a request which does not conform to the component specified interface. For instance, a parameter value is not in a specified range. The failure exceptions are signalled if the component determines that for some reasons it cannot provides its specified service. Besides, there are internal exceptions that are raised by the component in order to invoke its own internal exception handlers. If these exceptions are handled successfully (that is, the component is able to mask the exception), the component returns to providing normal services. However, if the component does not succeed in dealing with such exceptions, it should signal a failure exception to a higher level of the system.

1.2.2. The Role-based Collaboration Architectural Style. In architectural descriptions which use collaboration-based designs as a basis, software systems are represented as a composition of collaborations. Collaboration consists of a group of components together with a group of activities that determine how components interact. The **component role**[1] is a part of a component which prescribes the activity of the component within a particular collaboration [20]. In the Role-based Collaboration architectural style, a component role represent the component port (i.e., the explicit interface of this component for a particular collaboration) and collaborations, the connectors among these components. The notion of collaboration is used here to represent collaborative work involving several interacting components. However, in order to develop complex dependable systems, it is also necessary to capture the notion of coordination supporting coordinated error recovery between interacting components. Thus, at least two different kinds of collaborations are defined by this style:

Simple collaboration incorporates a group of components cooperating to perform a task. Connectors of this kind are not concerned with the provision of dependability.

[1] Unfortunately, the term **role** is used in several areas. We always say **component role** when discussing the collaboration-based design and **connector role** when we address software architecture issues.

Dependable collaboration describes a collaboration that deals with both cooperative and competitive concurrency and with faults. It should implement coordinated error recovery and maintain the consistency of external resources in the presence of failures and concurrency among several collaborative activities competing for these resources.

1.2.3. A Dependable Software Architecture. In this section, we present a software architecture for developing dependable software that is based mainly on a combination of the two architectural styles: the Idealised Fault-Tolerant Component and the Role-based Collaboration styles. We assume a layered structure where each layer provides services to the layer above and is a client of the layer below. In addition, the idealised fault-tolerant components at same layer can perform collaborative work, each of them playing a specific component role in a collaboration. It is very important to guarantee the overall component consistency as each component can serve several clients and be involved in several cooperations at same time.

There should be some design rules of how to apply these styles in a consistent way because even though they provide two orthogonal views on system architecture, they cannot be applied orthogonally. This is a possible sequence of design steps that should be taken by system designers: the first step is to identify the idealised fault tolerant components and their abnormal responses; the second step it to identify the collaborations (derived from use-case scenarios in the requirement analysis) and component roles; and the last step is to refine the abnormal behaviour of each component in order to include the handlers for the exceptions which have to be handled cooperatively within a collaboration. It is worth mentioning that the exceptions which have to be handled cooperatively within a collaboration are interface and failure exceptions raised by individual idealised fault-tolerant components.

1.3. Design Patterns for Exception Handling and Dependable Collaboration

As the size and complexity of systems increase, software designers are learned to appreciate the importance of exploiting and reusing knowledge in the definition of their overall system architecture. Design patterns [8] have proved to be very useful in achieving this kind of reuse. They have a particular structure and format, and describe a problem that occurs over and over again in a specific domain and a solution to the problem. Design patterns are applied at the later design phases. Usually the choice of design patterns is influenced by the architectural styles previously chosen. In this section, we present a set of design patterns that refine the general architectural elements of the proposed dependable software architectures, bridging the transition from software architecture design to coding. The **Handler** and **Exception Handling Strategy** patterns [9] provide design solutions to implementing idealised fault-tolerant components (section 1.2.1). The **Reflective Role** pattern [5] delivers solutions to implementing component roles**.** Pattern **Competitive Collaboration** [5] provides design solutions to implementing the connector that prescribes the interaction protocol taking place among the components involved in a collaboration (section 1.2.2).

These patterns follow the overall structure defined by the computational reflection concept and, as a result allow a clear separation of concerns between the application functionality and dependable quality requirements. Computational reflection [12] is defined as the ability of observing and manipulating the computational behaviour of a system through a process called *reification.* This technique allows a system to maintain information about itself (meta-information) and use this information to change its own behaviour. It defines a **meta-level architecture** which is composed of at least two levels: a **base level** and a **meta level**. The base level encompasses components responsible for implementing the functionality of the application, whereas the meta level encompasses components dealing with the processing of self-representation and management of the application. The latter includes management activities for sequential and concurrent exception handling, coordinated error recovery and component roles. A **meta-object protocol** establishes an interface among base-level and meta-level components. We have defined a set of components [5,9] and a meta-object protocol called Guaraná [14] to support the implementation of these patterns in Java.

1.3.1. The Exception Handling Patterns. Ideally, local and cooperative exceptions should be defined uniformly and the effort spent on composing the resolution trees or graphs should be minimised. Local exceptions are handled internally by local handlers attached to the component that raises them, while cooperative exceptions are handled by all collaboration participants. The **Exception** pattern [9] allows application designers to deal with exceptions and their (concurrent) compositions in a uniform way. Software architecture should be able to address the complexity that comes from the following concerns: designers should define the exceptions handlers in a way that separates them from the system normal activity; exception handlers for local and cooperative exceptions should be defined in a uniform manner; components responsible for the deviation of the normal control flow and for the handler search should perform their management activities in a non-intrusive way.

The **Handler** and **Exception Handling Strategy** patterns [9] provide design solutions to implementing idealised fault-tolerant components. These patterns allow

designers to define exception classes implementing handlers for both local and cooperative exceptions. The methods of exception classes are the handlers for the exceptions raised during the execution of normal class methods. Normal classes implement the system normal activities. Metaobjects are responsible for transparent intercepting method results, changing the normal control flow to the exceptional one if exceptions are raised, searching the handler that should be executed and invoking it (with passing available meta-information useful for handling).

1.3.2. The Reflective Role Pattern. This pattern [5] provides design solution to implementing component roles. There is a separation between components and the role hierarchies that they may play. The pattern transparently adapts the component to different collaborations by attaching the role objects, each of which represents a role it is to play in each collaboration. Metaobjects manage these role sets dynamically. When component roles are represented as individual objects, different contexts are kept separate and system configuration is simplified. Maintaining constraints between component roles and preserving the overall component consistency become difficult since a component can play several roles which are mutually dependent. An approach based on lock/release component attributes is used by the pattern to preserve such consistency.

1.3.3. The Competitive Collaboration Pattern. This pattern [5] offers a design solution to applying and implementing the CA action concept, i.e. collaborations that implement coordinated error recovery and maintain consistency of external resources in the presence of failures and concurrency among several collaborative activities competing for these resources.

This pattern separates objects into two well-defined levels. The base level designers work with classes intended for creating collaborations and for defining nested collaborative activities (to allow better structuring of normal and error handling activities of the enclosing collaboration). The meta level implements a management mechanism based on reification of method invocations. This pattern introduces five classes: `Collaboration`, `Participant`, `MetaCollaboration`, `MetaParticipant` and `MetaAtomic` (Figure 3).

Designers extend the `Collaboration` and `Participant` classes by adding application-specific information. Instances of `Collaboration` have references to the collaboration participants, exceptions, the enclosing collaboration, nested collaborations and shared (local) objects used for inter-participant communication. Internal exceptions are the exceptions that should be handled within a collaboration by all collaboration participants, while failure exceptions are signalled to the enclosing collaboration. Instances of the `Participant` subclasses represent the collaboration participants; they hold references to the collaboration, to the component role (section 1.2.2) and to its method to be executed during the collaboration.

In our approach, instances of the `Collaboration` subclasses correspond to the connectors, component roles represent the component interfacing points (component ports) and instances of the `Participant` subclasses correspond to the connector roles of the Role-based Collaboration architectural style. External (transactional) resources are simple objects which are associated with instances of the `MetaAtomic` class: this is how their transactional semantics are guaranteed. Instances of the `MetaCollaboration` and `MetaParticipant` classes are responsible for synchronising the participants, for resolving concurrent exceptions and for invoking the handler of the resolved exception.

1.4. Real Time Issues

We believe that the architectural styles and design patterns described above are applicable in the context of the development of real time systems because they allow us to introduce exception handling starting from the earlier phases of system development. Using application specific exception handling is the only way of dealing with recovery in a timely fashion avoiding abortion or retry that are not suitable for real time systems.

Besides, our patterns follow the overall structure defined by the computational reflection concept and could be extended to follow the RTR model [7]. According to this model, timing and synchronisation constraints, exceptions and real-time scheduling algorithms can be developed at the meta level simplifying the development of real time systems.

CA actions have proven to be able to address the real time issues very effectively as they allow system designers to deal with real time exceptions [16]. In addition, they offer a flexible choice of the locking techniques used (full scale transactional vs. application specific) and of the way to present component concurrency (competitive vs. cooperative).

Although the design of real time systems is an established topic of research, to the best of our knowledge, architectural styles and design patterns supporting the whole life cycle have not been defined yet and further research is required.

1.5. Case Study: the Train Station Application

This section describes the Station Case Study and shows how the architectural styles and design patterns have been used in developing this system. More information about the case study and details of the implementation can be found in [4]. This case study

focuses on developing a subsystem of a 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 (with an obvious restriction that no more than one train can stop on each platform at a time). We assume that trains can execute some join cooperative activity when they stop at the some station together; for example, passengers can change trains during this stop to make their journey faster.

Basic requirements. A correct control system must satisfy certain requirements. System **Safety** means that collisions of trains at stations are prevented and sufficient distance between trains is maintained. Control **Fairness** guarantees that trains get access to stations fairly. **Liveness** is the property of the whole railway control system, which in our case mains that our station control systems should provide enough information for a high-level component that performs global scheduling that guarantees that the overall system is live and that the passengers eventually reach their destinations. The development of such component is not the purpose of the case study. We assume that the control system to be developed follows the schedule given by such component. We assume as well that trains at stations take all passengers who have to be in these trains and that all passengers who have to change within an action always do this.

Actuators. Trains are controlled using the following commands: set the direction of a railway switch, start/stop a train, decrease/increase the speed of a train and reverse the direction of the train movement.

Sensors. Railway tracks have sensors that report useful information to the control system. These sensors are the only means for determining the position of the trains. It is important that the integrity of the information provided by these sensors is verified, because it is used for both keeping track of the locality of each train and preventing disasters from occurring.

Our system should employ features for detecting errors caused by faults (for example, if a train cannot be stopped) and for tolerating them to restore normal computation. These are some additional failure assumptions: (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.

Time-related failures. The control system provides normal service if the following timing constraint is met: a train neither arrives at the station after t_{Entry} nor leaves it before t_{Exit}. If a train fails to arrive before t_{Entry}, then the other trains continue their activities but some corrective actions must be taken, for example, some passengers can be left 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.

Actuator failures. This covers two failures, because it relates to the situation where a train or switch fails. The fault may be permanent or intermittent, and re-trying the operation may solve the problem. 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 that all other trains be stopped if there is any danger of the collision.

Sensor failures. The failures 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).

1.5.1. Architectural Design. The aim of this section is show the feasibility of applying the software architecture elements in the architectural design of this case study. The two styles introduced in section 1.2 have been applied in combination. We assume a layered structure (section 1.2.1) of the control system and that trains execute some join cooperative activity (section 1.2.2) when they stop at some station together.

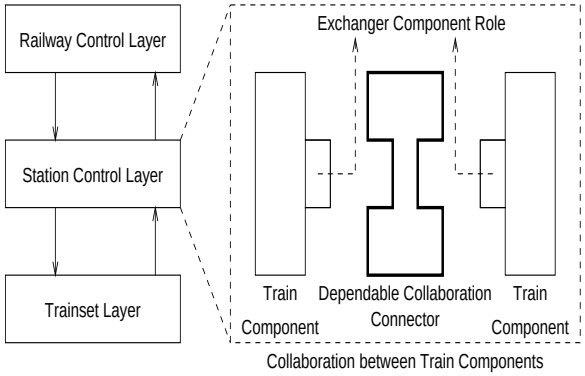

Figure 2 - Station Case Study: Architectural Design

Our design separates the safety and functionality requirements between a set of collaborations that occur during system execution and a set of train controllers that determine the train routes and hence the order in which the collaborations are executed. We assume that the safety requirement is guaranteed by the underlying layer (in our experiments this is done by the trainset layer) on the top of which we build the train controllers and the cooperative activity among trains (see Figure 2).

We also assume that each passenger has a ticket that describes her/his journey. If failures affect this journey, the scheduling component can be used to recalculate it.

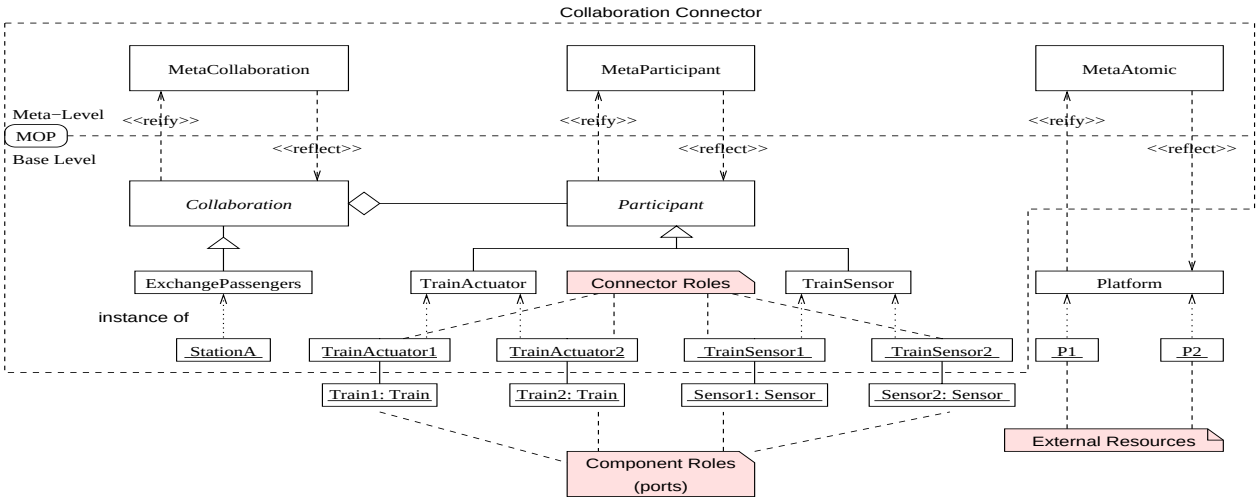

Figure 3 - Case Study Configuration

This component is part of the railway control layer and is used for providing the liveness property but we do not develop it as we have said before. Station platforms are modelled as external resources (objects) and have waiting rooms for passengers. The collaboration should implement coordinated error recovery and maintain the consistency of these external resources in the presence of failures and concurrency among different collaborative activities competing for these resources.

1.5.2. Detailed Design. The aim of this section is to show how the patterns presented in section 1.3 can be applied for providing detailed design solutions to problems arising in developing this case study.

The **Competitive Collaboration** pattern is used to design the collaborations discussed above. The `ExchangePassengers` class extends `Collaboration`, so that its instances represent the collaboration that coordinates the execution of an activity corresponding to cooperation of two (or more) trains calling at a particular station at the same time. The `TrainSensor` and `TrainActuator` classes extend the `Participant` class; instances of these classes represent the collaboration participants (i.e. connector roles). Component roles are activated by these participants. Instances of `TrainActuator` affect the execution of the `Train` instances (playing the component role **Exchanger)** by sending commands to stop at stations and leave stations after the cooperative activity has been finished. Instances of `TrainSensor` are associated with instances of the `Sensor` class that check if trains fail to respond to a request. Figure 3 shows the design of the cooperative activity between train1 and train2 calling at stationA (train1 stops at platform P1 and train2 stops at platform P2). Platforms P1 and P2 are modelled as objects associated with instances of MetaAtomic to guarantee their transactional semantics.

In addition, we use the **Handler**, the **Exception Handling Strategy** and the **Reflective Role** patterns to design the train component (Figure 4). The use of the **Handler** and **Exception Handling Strategy** patterns allows us to separate the normal and abnormal activity of this component and make it transparent for each other. These exceptional classes implement the handlers for local and cooperating exceptions. Besides, the Reflective Role pattern allows us to represent the hierarchy of roles that this component can play in different collaborations.

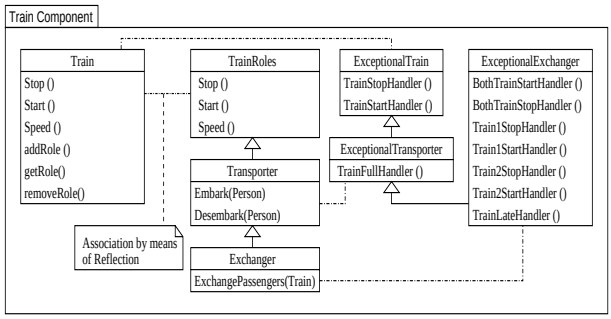

Figure 4 - Refinement of Train Component

Due to space limitation, the meta-level configuration associated with this component is not shown in Figure 4.

And finally, we have used the **Exception** pattern to define local and action-level (including concurrent) exceptions in a uniform way (Figure 5).

Implementation issues. The first version of the control system has been implemented in Java using a trainset Java API available at University of Newcastle and a meta-object protocol called Guaraná [14]. We believe that the results of this experimental research are

promising. CA actions offer a very powerful framework for designing complex systems in a structured way; they facilitate developing systems which meet high dependability requirements; the resulting systems have a clear structure which makes reasoning about the system (including reasoning about dependability properties) simpler. Two architectural styles proposed back the architectural development of the system, clearly facilitate it and make it more disciplined. Design patterns bridge the transition from describing the system architecture to implementing the system making these steps easier.

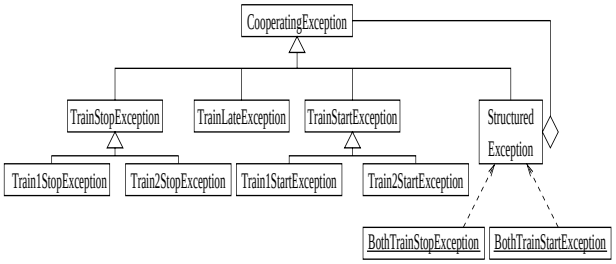

Figure 5 - Exception Tree Definition

2. Systems of Systems

2.1. Characteristics of Systems of Systems

Development of systems consisting of existing systems is a new emerging area of research [13]. Designers of such complex systems face new challenging problems, which are distinctly different from the problems addressed by developers of component-based or object-oriented systems. The main focus of our research is on developing fault tolerance architectures and techniques for such systems. Systems of systems (SoSs) are complex systems built out of ready made existing (and, often, running) component systems, which are quite autonomous and should continue to provide their individual services even if the whole SoS is not able to do this (when, for example, it or its component fails). It is usually not possible to develop a total control of such SoSs. After incorporation of such component systems into a SoS these systems should be able to act as individual systems and as parts of a bigger system (or, even, several of them). There is clearly a need in a special support for adaptivity and reconfiguration (including degradation) in such complex systems. This is a heterogeneous system as the component systems meet different standards, exhibit different time and fault behaviour, are developed with different fault assumptions in mind. Such complex systems report exceptions in different ways and should be able to handle complex situations when several component systems manifest abnormal behaviour. It is usually not possible to apply here traditional fault tolerance features which rely on abort (because, for example, some activities are long-lived), so the importance of applying adequate exception handling models (that is, forward error recovery) is very high and demanding in this problem domain.

All these characteristics require developing new fault tolerance techniques which are more flexible than the conventional ones, relying on less restrictive concurrency control and looser synchronisation. Besides, these techniques should allow for handling exceptions at the level of SoSs as the main recovery features which can be used for programming both cooperative exception handling involving several component systems and disciplined compensation activities. It is very important that such fault tolerance techniques should be always associated with structuring techniques used for developing (mainly composing) the SoS. These techniques are vitally important in the context of SoSs as they allow us to guarantee the consistency of individual component system states and to contain erroneous information in order to facilitate the recovery of the system as a whole.

2.2. CA Actions for Systems of Systems

Our analysis shows that CA actions serve as a solid basis for developing structuring and fault tolerance techniques suitable for SoSs [15]. They allow system developers to design, structure and provide fault tolerance of complex concurrent systems in which components cooperate and compete. They also provide support for exception handling, which is vital for components that are not capable of rolling back as well as for protecting long-lived activities using compensation activity which is structured as action handlers attached to CA actions. Moreover, CA actions are structuring units which always contain erroneous information; this makes system recovery simpler, faster and cheaper without any needs for tracing dependencies (the approach which many workflow systems use).

CA actions incorporate a rich concept of multiple outcomes to allow for informing the containing context about different types of abnormal behaviour which an action can exhibit, as well as, for delivering partial results and for introducing system degradation at the level of actions (to allows system designers to take into account component system autonomy). This supports developing complex systems in which structuring units provide a set of well-defined outcomes associated with a rigorously defined set of post-conditions in which the action and the participating components are left.

The concept of CA action offers a full support for maintaining system consistency and achieving fault tolerance using design diversity, backward and forward error recovery as they allow tolerating environmental faults, software design faults and crashes of nodes with transactional objects. CA actions are built on a very rich concept of atomicity and allow for a general way of

achieving fault tolerance. In addition, CA actions incorporate support for resolving concurrent exceptions which are likely to happen in complex systems of systems.

2.3. Architectural Design of Systems of Systems

By the very nature of SoSs, these systems are developed by integrating existing component systems. Such systems are always very intricate and heterogeneous (due to the complexity of the component systems and of the new services to be delivered), and their integration has to be based on a well-structured system architecture.

We believe that the dependable architectural design described in section 1.2 can be used to provide an abstract description of SoSs. First of all, the notion of collaborations (section 1.2.2) makes it possible to explicitly represent complex interaction among existing component systems, which in turn improves readability and maintainability and, consequently, helps integrate component systems. Component roles provide an explicit interface for the new functionalities to be delivered by each component system in the context of a SoS. Secondly, the abstraction of idealised fault-tolerant components (section 1.2.1) is recursive; that is, each idealised fault-tolerant component can be an (autonomous) system itself. The main issue is that the exception handling interface should be materialised (that is, explicitly represented at the architectural level) as well, to make it possible for each component system to implement its own exception handlers and recovery policies.

2.4. Design Patterns for System of Systems

Our preliminary analysis shows that the patterns described in section 1.3 are applicable in the context of SoSs because they allow us to introduce new interfaces for exception handling and collaboration (we need those because SoSs deliver new services) while addressing the following characteristics of SoSs (discussed in section 2.1): (i) disciplined compensation activities (cooperative exception handling involving several component systems); (ii) component autonomy: component roles can be used to specify the new services in the context of SoSs so that components can keep providing their usual individual services if different contexts are kept separate and the overall component consistency is preserved. Note that this approach makes it possible for the SoS designers to introduce new services even when there is a number of the running system components. However, these patterns do not address other characteristics of SoSs, such as autonomous recovery of component system in case of failures. Thus, further development of design patterns which are able to deal with specific characteristics of SoSs is required.

During the architectural and detailed design phase, the logical SoS components with explicit interfaces for collaboration, exception handling, error recovery and so on are specified. Afterwards the existing component systems have to be adapted to match the logical SoS components. Developing systems by component system integration can be complicated by the following factors: introducing new or extended functional and non-functional requirements (e.g. adding functionality, improving dependability), using components in a different (wider or narrower) context and heterogeneity of components. It is very unlikely that component systems will exactly match each other, so techniques for adopting component systems will be used. Component wrapping is an example of such techniques (a useful discussion on different adaptation techniques can be found in [21]) that can be used for both adaptation and reconfiguration of existing component systems. We believe that is it important to be able to apply some form of computational reflection for implementing component system wrapping because it promotes a transparent and non-intrusive adaptation while allowing us to design meta-level components which address the complexity caused by the characteristics of SoSs.

2.5. Discussion and Future Work

Implementation of a system of systems is to be supported by features providing component system wrapping and CA actions. In our opinion, this implementation should be oriented on modern component-based technologies (such as CORBA [3], DCOM [1] and EJB [2]) which can serve as unifying platforms for such heterogeneous systems. First of all, because they have useful supports for developing wrappers (e.g. by call intercepting, by incorporating each component into a container, etc.). Secondly, because all these technologies provide transactional services which can serve as a sound basis for developing CA action schemes [23,24,25]. And thirdly, because features are being developed to make it possible for CORBA, EJB and DCOM components to call each other or to incorporate a foreign component into a system.

Application of the architectural styles and design patterns to develop realistic case studies is an important area of our future work. Our analysis shows the case study reported in section 1.4 can be extended by introducing autonomous component systems 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 is to develop a control system supporting complex functionalities of such systems of systems.

Other areas of our future work are as follows: further development of the architectural styles and design patterns which are able to deal with specific characteristics of SoSs; implementation of the CA action schemes within existing component-based technologies; development of structural techniques for wrapping; implementation of re-usable wrappers providing functionalities necessary for wrapping component systems to allow their integration into SoSs.

3. Conclusions

This paper makes the following contributions:

- it presents a generic software architecture for introducing atomicity, exception handling, and coordinated error recovery into dependable object-oriented systems at the earlier phases of system development;
- it discusses a set of design patterns which refine the architectural elements of the proposed software architecture and provide a clear and transparent separation of concerns between the application functionality and the functionality related to providing system dependability;
- it shows the feasibility of the approach proposed by means of a realistic case study;
- it proposes approaches to applying these ideas in the context of developing complex systems of systems which require extensions and adjustments of the Coordinated Atomic action concept as well as of the architectural styles, design patterns and the implementation support to allow designers to address the typical problems of such system development.

Acknowledgements. This research is partially supported by the European IST DSoS project (IST-1999-11585) and CNPq/Brazil under grants 141425/97-0 for Delano Beder and 351592/97-0 for Cecília Rubira. Cecília Rubira and Delano Beder are also supported by the "Advanced Information Systems" Project (PRONEX-SAI-7697102200).

References

[1] *DCOM: A Technical Overview*, Microsoft Corporation, 1998.

[2] *Enterprise Java Beans Technology*, Sun Microsystems, 1998.

[3] *The Common Object Request Broke: Architecture and Specification Revision 2.0*, OMG, 1995.

[4] **D. Beder, A. Romanovsky, B. Randell, C. Snow & R.J. Stroud**, *An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling*, ACM Operating System Review, 34(4):21-31, 2000.

[5] **D. Beder & C. Rubira**, *A Meta-Level Software Architecture based on Patterns for Developing Dependable Collaboration-based Designs*, 2nd Brazilian Workshop on Fault Tolerance (WTF'00), 34-39, 2000.

[6] **R. Campbell & B. Randell**, *Error Recovery in Asynchronous Systems*, IEEE TSE-12(8):811-826, 1986.

[7] **J. Fraga, J.-M. Farines & O. Furtado**, *RTR Model: An Approach for Dealing with Real-Time Programming in Open Distributed Systems*, 3rd IEEE Workshop on Object-Oriented Real-time Dependable Systems (WORDS'97), 1997.

[8] **E. Gamma, R. Helm, R. Johnsonn & J. Vlissides**, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] **A. Garcia, D. Beder & C. Rubira**, *An Exception Handling Software Architecture for Developing Fault-Tolerant Software*, 5th International Symposium on High Assurance Systems Engineering (HASE'00), 311-320, 2000.

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

[11] **P. Lee & T. Anderson**, *Fault Tolerance: Principles and Practice*, Springer-Verlag, 2nd edition, 1990.

[12] **P. Maes,** *Concepts and Experiments in Computational Reflection*, ACM SIGPLAN Notices 22(12):147-155, 1987.

[13] **M. Maier**, *Architecting Principles for Systems-of-Systems*, Systems Engineering, 1(4):267-284, 1998.

[14] **A. Oliva & L. Buzato**, *The Design and Implementation of Guaraná*, 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), 86-90, 1999.

[15] **A. Romanovsky**, *Coordinated Atomic Actions: How to Remain ACID in the Modern World*. ACM SEN, 26(1), 2001.

[16] **A. Romanovsky, J. Xu & B. Randell**, *Exception Handling in Object-Oriented Real-Time Distributed Systems*, 1st IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'98), 1998.

[17] **A. Romanovsky & A. Zorzo**, *Coordinated Atomic Actions as a Technique for Implementing Distributed Gamma Computation*, Journal of Systems Architecture, 45(15):1357-1374, 1999.

[18] **M. Shaw & D. Garlan**, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[19] **J. Vachon, N. Guelfi & A. Romanovsky**. *Using COALA to Develop a Distributed Object-Based Application*, 2nd International Symposium on Distributed Objects and Applications (DAO'00), 195-208, 2000.

[20] **M. VanHilst & D. Notkin,** *Using Role Components to Implement Collaboration-based Designs*, 11th ACM Confererence on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA´96), 359-369, 1996.

[21] **G. Weiss & C. Rubira,** *A Comparative Study of Software Component Adaptation Techniques*, Technical Report, Institute of Computing, University of Campinas (in preparation).

[22] **J. Xu, A. Romanovsky & B. Randell.** *Concurrent Exception Handling and Resolution in Distributed Object Systems*, IEEE TPDS-11(10), 2000.

[23] **J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, E. Canver & F. von Henke**, *Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions*, 29th International Symposium on Fault-Tolerant Computing, 1999.

[24] **J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud & Z. Wu**, *Fault Tolerance in Concurrent Object-Oriented Software through Co-ordinated Error Recovery*, 25th International Symposium on Fault-Tolerant Computing, 1995.

[25] **A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud & I. Welch**, *Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study*, Software: Practice & Experience, 29(7):1-21, 1999.