Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems

Gisele R. M. Ferreira
Institute of Computing
University of Campinas - Brazil
gisele.ferreira@ic.unicamp.br

Cecília M. F. Rubira
Institute of Computing
University of Campinas - Brazil
cmrubira@ic.unicamp.br

Rogério de Lemos
Computing Laboratory
University of Kent at Canterbury - UK
r.delemos@ukc.ac.uk

Abstract

Exception handling is a structuring technique that facilitates the design of systems by encapsulating the process of error recovery. In this paper, we present a systematic approach for incorporating exceptional behaviour in the development of component-based software. The premise of our approach is that components alone do not provide the appropriate means to deal with exceptional behaviour in an effective manner. Hence the need to consider the notion of collaborations for capturing the interactive behaviour between components, when error recovery involves more than one component. The feasibility of the approach is demonstrated in terms of the case study of the mining control system.

1. Introduction

In the engineering of high assurance systems from existing components, it is fundamental to consider the exceptional behaviour of the components, in addition to their normal behaviour, if trust has to be placed on the services delivered by the system. Otherwise, the system might fail in unexpected ways from the undesirable combination of exceptions that might be raised from the different components. However, if the description of exceptional behaviour is considered not in a structured way, this will unavoidably increase the complexity of the system, making its understanding more difficult and thus leading to the construction of an untrustworthy system. Hence the need for incorporating exceptional behaviour in a carefully structured and controlled way that would minimise its impact on the system complexity. One solution to this problem is to adopt a systematic approach for including exceptional behaviour in the development of component-based software.

Dealing with concurrent manifestations of several faults at different phases of system development has been recognised as a serious problem that has not received enough attention [1]. In the context of component-based software development, related work has been scarce, and most of it associates exception handling with objects, making no attempt in considering exception handling within the software lifecycle. Although exception handling and object-oriented languages raise some conflicts, these conflicts exist mainly during the late phases of the software development [10], and some of these can be solved if exception handling is considered right from the early phases of the software development. In a recent work, the integration of exceptional behaviour within the software lifecycle was supported by a co-operative object-oriented approach that allows representing collaborative behaviour between objects at different phases of the software development [3]. The approach being presented in this paper is distinct from the co-operative oriented approach in several aspects. The most significant aspect is the fact that our approach is considered in the context of an existing methodology, which allows to detail more concretely on how to incorporate exception handling in the development of dependable software. Another area in which new approaches have been recently developed for specifying and designing activities incorporating multiple parties (objects or processes) and co-operative handling of exceptional situations is multiparty interactions [12].

In this paper, we describe a systematic approach for explicitly representing exception handling at all phases of the software development. Exceptional behaviour is identified in the use cases of the problem definition, refined during specification and design, represented as idealised fault-tolerant components [8] in the architectural design of the software system, and finally implemented as a programming language by means of a meta-object protocol (MOP) [6]. Considering that, components alone are not able to provide an effective means to deal with exceptions, it is important to employ the notion of collaborations for capturing the interactive behaviour between components. In this paper, our approach is presented in the context of Catalysis [4], which is a methodology for developing object-oriented and component-based systems where collaborations are treated as first-class entities. Although Catalysis provides the means for representing exceptions, it does not consider exceptional behaviour within the software lifecycle. The

basis of our work is to incorporate in the Catalysis methodology the representation of exceptional behaviour in the process of developing component-based systems, thus increasing our confidence on the quality of services provided by a system. Although in this paper we have considered our exception handling strategy in the context of Catalysis, our approach could nevertheless be adapted to other object-oriented or component-based methodologies.

The remainder of this paper is organised as follows. Section 2 presents an overview of the Catalysis component-based methodology, which was adopted as a basis for our work. Section 3 defines a dependable software architecture based on the idealised fault-tolerant component. Section 4 presents how the exceptional behaviour can be integrated in the development of dependable systems. The feasibility of this work is demonstrated in Sections 5, 6, 7 and 8, in terms of the mining system. Finally, Section 9 summarises this work and provides some concluding remarks.

2. An Overview of the Catalysis Process

Catalysis is a methodology for the systematic development of object-oriented and component-based systems that has three primary modelling constructs:

- A set of related actions between objects are grouped into a *collaboration* to indicate that they serve a common purpose. Collaborations are often used to model how a group of objects jointly behaves when configured together in specific ways.
- A *type* defines a specification of the externally visible behaviour of an object, abstracting from details of its internal representation, algorithms, and data structures.
- *Refinement* supports multiple levels of description related to the same phenomenon.

Similarly to other object-oriented methodologies, Catalysis defines three major phases for developing software systems: requirement analysis, component specification and design, and implementation. What differs Catalysis from traditional methodologies are its collaborations, which are considered as first-class entities because Catalysis assumes that decisions about the interactions between objects are the key for good-coupled design. In the following, we present an overview of the Catalysis approach, in terms of its phases and objectives.

The *requirement analysis* aims to understand the problem at hand, and capture the user needs, i.e., find out what is to be built.

During *component specification and design* the aim is to provide a design for the software system in terms of collaborations. This phase supports three levels of description, or steps:

- *Type specification*: this level defines the external behaviour required from a system/component described as a *type* specification. Defining the *type* involves describing its internal components and its external behavior. For that, each action the system participates is specified as an operation of the type.
- *Architecture design*: this level enumerates the components and the ways they are integrated.
- *Collaboration design*: this level effectively *"opens the box"* for a given component, and describes how it will be designed internally to provide its externally specified behaviour. The design is described in terms of collaborations, where the use cases are implemented as a set of actions with their pre- and post-conditions.

This process is recursive as far as necessary or useful. We would typically stop once components can be made available; that is, when they can either be bought or built.

During *implementation* the purpose is to convert the design components into source code with an object-oriented programming language. Each file component (source code) may implement several design classes that correspond to a C++ or a Java class. A component is a replaceable part of a system that conforms to and provides the realisation of a set of interfaces.

3. A Dependable Software Architecture

Following the terminology adopted by Lee and Anderson [8], a system consists of a set of components that interact under the control of a design. Software components receive service requests and produce responses when the service has been completed. The responses from a component can be separated into two distinct categories, namely *normal* or *abnormal* responses. *Normal responses* correspond to those situations where the component has provided its normal service satisfactorily. *Abnormal responses* or *exceptions* are usually signalled when there is a *fault* leading to an error in the system and this component cannot provide the requested service. As shown in Figure 1, the activity of the component must be separated into two parts: a *normal part*, which implements the component's normal service, and the *abnormal or exceptional handling part*, where the component's measures for fault tolerance are implemented.

Exceptions can be classified into two different categories: (i) *internal exceptions* that are raised by the component in order to invoke its own internal fault tolerance activity and (ii) *external exceptions* signalled if a component determines that for some reason it cannot provide its specified service. Thus, when an exception is internally signalled by the component, we assume that the exception handling part of the component is automatically invoked. If this exception is handled successfully, the component can return to providing its normal service. However, if the component does not succeed in dealing with such an exception, it should signal an external exception. In this sense, exceptions and exception

handling provide a suitable framework for structuring the fault tolerance activities incorporated in a system.

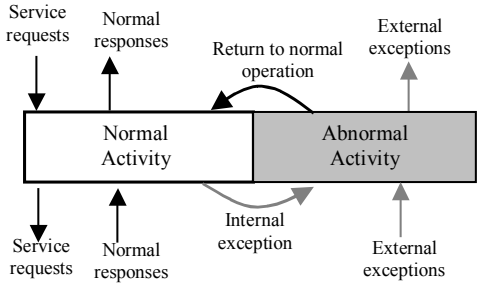

Figure 1 - Idealised fault-tolerant component

4. A Systematic Approach for Representing Exceptional Behaviour

Exception handling has been traditionally associated with the implementation phase of the software lifecycle, during which all the effort is made to protect the application software from faults that may be introduced during requirements analysis, design, and implementation, or can occur at the support level. The consequence of such approach is that the appropriate context in which errors should be detected and recovered is lost. Also it is lost the potential correlation that might exist between the error states of the different contexts and how these should be recovered in an optimised way [3]. Hence the need for each identified phase of software development, to define a class of exceptions depending on the abstraction level (or context) of the software system being modelled and analysed. As the software development progresses, new exceptions are identified and their respective handlers specified. However, the exceptions identified at the different phases can be causally and timely related, which might constraint the specification of their respective handlers. Moreover, it might be the case that the rationalisation of exceptions might enable the usage of a single handler for different classes of exceptions.

The purpose of this section is to provide an overview of the proposed systematic approach for incorporating exceptional behaviour in the process of software development. This approach was integrated into Catalysis (see Section 2), and based on the idealised fault-tolerant component architecture (see Section 3). The representation of exceptional behaviour begins during the requirement analysis and continues until implementation. The following features form the basis of our exception handling strategy:

- *Precise specification:* a precise specification of actions and collaborations turns natural language statements into a less ambiguous specialised notation, for that they are expressed in terms of pre- and post-conditions. The pre-condition specifies the condition that must be satisfied at the time that the action or collaboration is initiated. The post-condition similarly tells us what conditions must be satisfied for the action or collaboration to finish. The pre- and post-conditions specifications embody the principle of *design by contract* [9]. Although there is extra effort involved, it saves work later in the phases of development and also during maintaining by focusing attention on the important issues, earlier in the development phases, such as, gaps and inconsistencies. Furthermore, the principle of design by contract facilitates the identification of those exceptional situations that are violations of contracts.
- *Separation of normal and exceptional activities*: the proposed approach was primarily designed to facilitate the development of dependable software components. As discussed before, when building reliable components it is necessary to keep its complexity under control. Hence, it is important to separate exceptional behaviour from normal behaviour specifications. The idealised fault-tolerant component provides an architecture that separates the normal from the exceptional activities.

In the following, we proceed to describe the features that identify our approach when incorporating exception handling in the Catalysis methodology.

4.1. Requirement Analysis

In this phase the functionality of the system is described by use cases. A *use case* is a description of a set of sequence of actions, including variants that a system performs to yield an observable result of value to an actor [2]. An actor represents a coherent set of roles that users of use cases play when interacting with them. In the systematic approach proposed in this paper, the textual description of use cases is expanded. In addition to the description of normal behaviour, we have also included *the description of failure behaviours*. The description of normal behaviour is detailed by defining the use case in terms of its pre- and post-conditions, invariants that should hold, and the basic operations performed by the use case. The exceptional description for the use case is related to the violation of the contract defined by its pre- and post-conditions, and its invariant. Just as use cases are introduced in a narrative-style, the exceptional behaviour is also incorporated into a narrative form. The post-conditions for the various identified exceptional behaviours might not be identical because of the different degraded outcomes of a use case, as a result of an exception occurring. The template employed for the specification of use cases is described in the Figure 2

Use Case Name
Actors: List of components participating in the use case;
Description (Normal behaviour):
A brief description of the normal behaviour of the use case;
Pre-condition: The set of conditions that have to be valid for the use case to start;
Invariant: The conditions that should hold in the use case;
Operation: The activity that has to be performed by the use case;
Post-condition: The set of conditions that have to be valid for the use case to finish;

Description (Exceptional behaviour):
A brief description of the exceptional behaviour of the use case;
Signal: The error state or event that identifies an exception;
Handler: The activity for bringing the system to an error free state;
Post-condition: The set of conditions that have to be valid for the use case finish, after the occurrence of an exception;

.

Figure 2 - Descriptive template for exceptional behaviour of robust use cases

The *extend* relationship between use cases is employed to represent that the base use case implicitly incorporates the behaviour of another use case. The base use case may stand alone, but under certain conditions, its behaviour may be extended by behaviour of another use case. It is a way to separate subflows that are executed only under certain conditions.

The description of exceptional behaviour in the specification of use cases allows at the initial phases of software development to establish the relationships between failures that might affect either the use cases or the actors. This allows impact analysis of failures to be performed at higher levels of abstraction, thus enabling, for example, to check the consequences on the rest of the system if a particular use case fails to behave as required. Once failure behaviours and their impacts are specified, this provides the basis for obtaining robust specifications, since protection mechanisms against failures and their relationships, can be incorporated from the initial phases of software development. The specifications should consider the propagation of exception between use cases until the appropriate context is found for handling an exception, if the use case in which the exception has originally occurred does not provide the necessary redundancies for handling it.

The outcome of this phase is a textual description of the use cases of the system in terms of their normal and abnormal behaviour following the use case template illustrated in Figure 2. The latter would include the specifications of exceptions that occur in the context of the use case, or those exceptions that have been propagated from other use cases. Also it should include the specification of the handlers if the context of the use case is appropriate to handle the exception, or provide a reference to another use case to which the exception should be propagated.

4.2. Component Specification and Design

The phase of component specification and design follows the three steps of Catalysis presented in Section 2: *type specification*, *architecture design* and *collaboration design*. Collaborations implement the uses cases previously specified and represent the use case exceptional behaviour in terms of *exceptions* and *handlers*, with the latter incorporating the exception treatment. Exceptional behaviour, in addition of being associated with the failure of uses cases, it should also be considered in the context of invalid interactions between two or more components of the system, or the combination of component failures. At this point, the black-box abstraction of the system is refined to show the internal software components that compose the system. The exceptional behaviour identified by *collaborations* should be reflected in system *type* and *architecture*. In the following, we describe how to incorporate the description of exceptional behaviour at each step of component specification and design.

4.2.1. Type Specification

A type specification involves the description of the systems internal components and external behaviour. Traditionally, a component is structured by a set of normal classes, which implement its normal activities. In the approach proposed in this paper, the component should be extended by an exceptional part, which implements its abnormal activities (Figure 1). While the normal classes implement the normal behaviour of the use cases, the exceptional classes should implement the exceptional behaviour. This structuring fosters traceability with the modelling of the use cases, and makes it easier to maintain the consistency of the system. In this approach, designers compose an exceptional class hierarchy that is orthogonal to the normal class hierarchy of the application. Exceptional class hierarchies allow exceptional subclasses to inherit the handlers form their superclasses, thus allowing exceptional code reuse.

During the collaboration design some exceptions are identified and their handlers are specified. Exceptions and handlers constitute the abnormal behaviour of the component that implements the collaboration. In this approach, after the collaboration design, the type specification must be refined to include the component abnormal behaviour. External collaboration action becomes public methods and each internal collaboration action is represented as private methods of the normal

class. Each method represents explicitly the exceptions that may be raised by showing send dependences between operations and their exceptions. Following the idealized fault-tolerant component model presented in Section 3, exceptions raised by the normal part of the component are internal exception that should be treated by the exceptional part. Handlers for internal exceptions are represented as methods of the exceptional class. If the abnormal part of the component cannot handle an exception, a failure exception is raised to the caller.

In Figure 3, the methods of ExceptionalSupClient are the handlers for the exceptions that are raised by the methods of the class SupClient, and its subclasses. The methods of ExceptionalClient are the handlers for exceptions raised by class Client. In this example, we consider that SupClient, or its subclass, requests a service from class Server. When the method m3() of the class Server is invoked, it can raise two exceptions, E2 and E3. The exception E3 is treated locally by the component Server, more precisely, by the method E3Handler of the class ExceptionalServer. However, exception E2 is propagated to the caller, which must have a handler to treat it. The E2Handler is defined as a method of the class ExceptionalSupClient, which is inherited by ExceptionalClient, thus allowing class Client to request a service from class Server without needing to redefine the E2Handler.

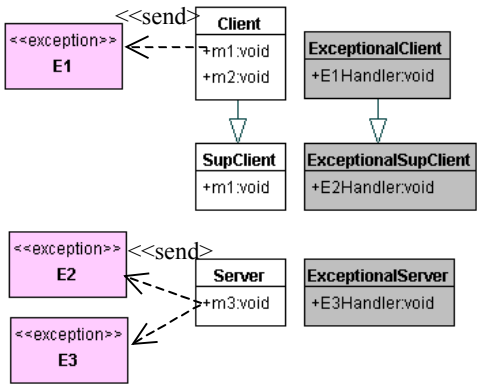

Figure 3 - Normal and exceptional class hierarchies

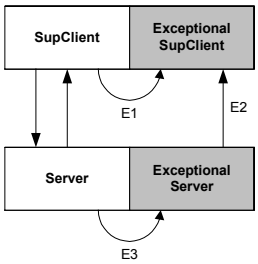

Figure 4 – Robust software architecture

In Figure 4, the component SupClient raises an internal exception E1 that is treated locally by the class ExceptionalSupClient. When SupClient requests a service from component Server, it can raise two exceptional situations. The normal activity of the component Server signals an internal exception E3 that is treated within the component, and its exceptional activity signals an external exception E2 that is propagated to the caller SupClient.

4.2.2. Architecture Design

The *architecture design* uses the model of idealised fault-tolerant component described in Section 3. In this architectural model, a component requests services to a low-level component that returns either normal or exceptional responses. This makes it easier to identify which handlers each component should incorporate. In our exception handling model, we adopted explicit propagation of exceptions between adjacent components. As a consequence, the client is aware of all exceptions it is responsible for handling. If a signalled exception cannot be handled at the caller, then this exception should be propagated to the components at higher level of abstraction.

4.2.3. Collaboration Design

In the *collaboration design*, use cases are implemented as collaborations, a collection of actions and component types. The normal behaviour of the use cases is represented by a set of actions (operations) that incorporate the use case pre-, post-conditions and invariant. The exceptional behaviour is the violation of the action assertion, i.e., pre-, post-conditions or invariant. The description of abnormal behaviour is in terms of *exceptions* that are signalled, and *handlers* that incorporate the exception treatment. The description template employed for the specification of collaborations is presented in Figure 5.

Abnormal behaviour is not restricted to the failure of a single component. Invariably, it is also associated with invalid interactions between two or more components and the combination of component failures. The description of abnormal behaviour in terms of collaborations allows identifying these potential failures. Moreover, collaboration diagrams are useful for representing diagrammatically multiple-failures in systems, thus providing support for understanding their abnormal behaviour.

The approach presented in this paper, in addition of identifying the exceptional behaviour through collaborations, also promotes the refinement of the system type and architecture to include the identified exceptions, and their respective handlers. In the refinement of the type specifications, exceptions raised by a component are hierarchically organised and associated with the type at same abstraction level. This approach allows the handling

of either the more general supertype exceptions, or the more specific subtype ones in a controlled manner.

Normal Behaviour	
Action:	The activity specified by the use case;
Pre-condition:	The set of conditions that have to be valid for action to start;
Invariant:	The conditions that should hold during the action;
Post-condition:	The set of conditions that have to be valid for the action to finish;
Exceptional Behaviour	
Signal:	The error state or event that identifies a failure;
Handler:	The action that should try to handle the failure;
Post-condition:	The set of conditions that have to be valid, after the exception has been handled;

Figure 5 – Template for the description of robust collaboration

Additionally, the handlers for the exceptions of a component are incorporated as methods of the abnormal class. Furthermore, to design robust components that will not fail in unanticipated ways, it is necessary to have a clear specification of internal exceptions. Based on the exceptions identified during the step of collaboration design, the architectural design should be refined to incorporate those exceptions that are propagated and caught by a component. This approach facilitates the identification of components that can be affected by an exception, and the location of the respective exception handlers.

In UML, exceptions are represented as stereotyped classes and arranged in a hierarchy [2]. In our model, exceptions are also hierarchically organised, and are related to collaborations that are at the same abstraction level as the exceptions themselves. A handler defined for an exception E is suitable for any exception subtype of E. Permitting several exceptions to be treated by the same handler avoids code replication when the exceptions are all handled in the same way. For facilitating the specification of exceptions, we define a notation to represent exception names that relate the collaboration name with the type of the exception (either internal or external). The template for the naming of exceptions is Type_ExceptionName_CollaborationName. Type can be {I, E}, where "I" represents internal exception, and "E" represents external exceptions.

The approach being proposed for specifying and designing components has several outcomes, including:

- The type specification of each component is structured in terms of two orthogonal class hierarchies: the normal and the abnormal. The former implements the normal services provided by the component. The latter implements the handlers for exceptions signalled internally in the component, or raised by another component whose service has been requested.
- The system architecture design is based on the model of idealised fault-tolerant component, which explicitly represents internal and external component exceptions. This architectural model makes it easier to identify the handlers each component must incorporate. Considering the explicit exception propagation, each exception signalled by a component must have a handler in its caller.
- Collaborations are implemented as a collection of actions and component types. The normal behaviour of collaborations is implemented by the actions with invariants, pre- and post-conditions. The exceptional behaviour is captured from the failures and invalid collaborations between two or more components.

4.3. Implementation

For implementing the components defined above, we need an exception handling mechanism to support the explicit separation of their normal and abnormal activity. Moreover, we advocate that the exceptions should be represented as full objects (rather than merely signals with little information content), organised hierarchically, and that the exception propagation to higher-level components should be performed explicitly. All these properties can be satisfied using the mechanism presented in [6], which is implemented using the Java programming language by means of a meta-object protocol (MOP) [7]. The components of the application will be implemented in the base level while the meta-objects implement the specific responsibilities of the exception mechanism. When a normal class of the component signals an exception, it is intercepted by the MOP and the meta-objects will find an adequate exception handler in the abnormal class of this component. The abnormal classes are hierarchically organised, allowing subclasses to inherit handlers from their superclasses and, consequently, permitting the reuse of abnormal code. The abnormal class hierarchy is orthogonal to the normal class hierarchy.

5. Case Study: Requirement Analysis

The example that has been chosen is a simplified version of the pump control system for the mining environment [11]. The extraction of minerals from a mine produces water and releases methane gas to the air. The mining control system is used to drain mine water from a sump to the surface, and to extract air from the mine when the methane level becomes high. A schematic representation of the mining system is given in Figure 6. The mining control system consists of three control stations: one that monitors the level of water in the sump, one that monitors the level of methane in the mine, and another that monitors the mineral extraction. When the water reaches a high level, the pump is turned on and the

sump is drained until the water reaches a low level. A water flow sensor is able to detect the flow of water in the pipes. However, the pump is situated underground, and for safety reasons it must not be started, or continue to run, when the amount of methane in the atmosphere exceeds a safety limit. For controlling the level of methane, there is an extractor control station that monitors the level of methane inside the mine, and when the level is high an extractor is switched on to remove air from the mine. The whole system is also controlled from the surface via an operator console that should handle any emergencies raised by the automatic system.

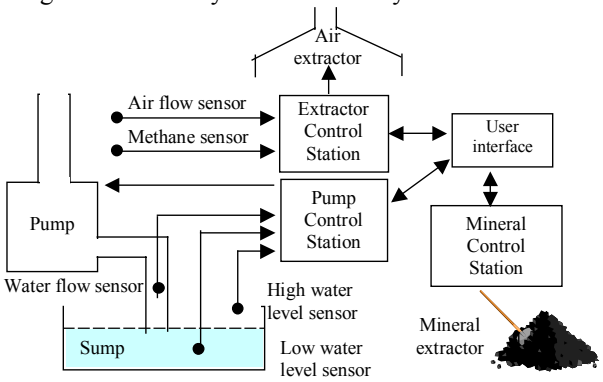

Figure 6 – Schematic diagram of the mining system

In this case study, we employ the three phases of our systematic approach described in Section 4. Firstly, the functional requirements are captured through the use case modelling (Section 5.1). Secondly, the specification and design of the mining system, and its components, are presented in Section 6 in terms of types, architecture and collaborations. The specification and design of the system is refined in Section 7. Finally, the issues related to the implementation of the normal and abnormal classes in terms of a meta-object protocol (MOP) are discussed in Section 8.

5.1. Use Cases

The use cases of the system are shown in Figure 7. The main use case is related to the extraction of minerals (ExtractMineral use case) from the mining, however this operation produces water and releases methane gas. Hence, it is necessary to stop the mineral extraction for draining water from a sump (DrainSump use case) or extracting air from the mine (ExtractAir use case).

In order to simplify the presentation of the case study, we consider that the sensors AirFlow, WaterFlow, MethaneHigh, WaterHigh and WaterLow will never fail, while the actuators Pump and AirExtractor are prone to failure. The exceptional behaviour of the MiningSystem is related to failures that can affect one of the three major activities of the system that are, the extraction of mineral, the extraction of air and the drainage of water, which can be summarised as follows:

- If the methane sensor detects high level of methane, the collaboration ExtractMinerals will be interrupted, and the collaboration ExtractAir will be initiated to reduce the methane level.
- If the water sensor detects high level of water in the sump, the collaboration ExtractMinerals will be interrupted, and the collaboration DrainSump will be initiated to drain the sump to reduce the level of water.
- If both high level of methane and water flow are detected concurrently, the collaboration ExtractMinerals will be interrupted, the collaboration ExtractAir will be initiated, and the pump will be switched off.

The relationship between the base use case ControlMiningSystem and the other three use cases ExtractMinerals, ExtractAir and DrainSump is that of extending the base use case by incorporating the behaviour of the other three use cases. For illustration purposes, in the following we briefly present the textual description only for the use case ExtractAir.

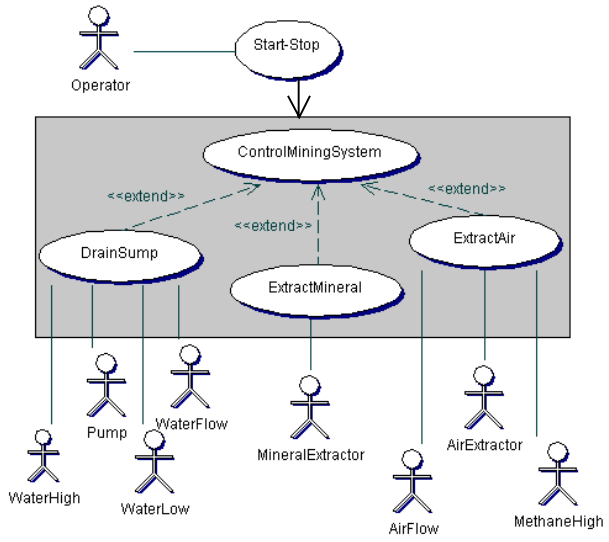

Figure 7 - Use case diagram for the MiningSystem

.

Use Case ExtractAir

The textual description of ExtractAir is described in the Figure 8. The normal behaviour of this use case is associated with the extraction of air from the mine. It describes the operation of AirExtractor: when the methane levels are high the AirExtractor is switched on, and when they drop to acceptable levels the AirExtractor is switched off. Its invariant states that the Pump should remain switched off while the AirExtractor is on. The post-condition for the normal behaviour establishes that the level of methane should be normal and the AirExtractor should be off. The identified exceptional behaviour is when no flow of air is detected, though the air extractor is switched on. This situation is identified as

a failure in the AirExtractor. The handling of this exception is to switch off the AirExtractor, and raise an exception.

Use Case ExtractAir
Actors: AirExtractor, MethaneHigh and AirFlow ;
Description (Normal): When the operator starts the mining process, air is extracted from the mine when the methane level is high;
Pre-condition: The operator has started the mining process, the level of methane is high, the air extractor and pump are off, and there is no flow of air or water;
Invariant: The pump is off and there is no flow of water;
Operation: If the level of methane is high, the air extractor is switch off, and there is no airflow then the air extractor is switched on. The air extractor is switched off when the methane level is normal;
Post-condition: The level of methane is normal, the air extractor is off, and there is no flow of air;

Description (Exceptional): When the methane level becomes high, air is extracted from the mine by switching on the extractor, but there is no flow of air (air extractor fails);
Signal: The air extractor is on and there is no flow of air;
Handler: It switches off the extractor and raises an exception;
Post-condition: The level of methane is high, the air extractor is off, and there is no flow of air;

Figure 8 – Use Case ExtractAir

The specification of the other three uses cases, ControlMiningSystem, DrainSump and ExtractMineral, follows the same process of the specified above. In addition to the normal behaviour, we also specify the abnormal behaviour of the use cases, which includes the definition of exception situations that can affect their services, and how they are handled by the use case, or propagated.

6. Case Study: Specification and Design

In this section, we proceed to specify and design the mining system following the approach described in Section 4.2, which follows the three steps of Catalysis with additional activities for representing exceptional behaviour.

The three steps were applied recursively to the mining system and its components. Initially, we have to apply them to the MiningSystem itself. The MiningSystem is composed by the user interface, sensors, actuators, and a control component. The control component, called ControlStation, is also considered a system by itself and should be specified and designed recursively by applying the definitions of type, architecture and collaborations according to Catalysis (Section 2). However, for the sake of brevity, the presentation will not detail the complete specification and design process, but instead we provide an overview of the approach being proposed by applying it to a partial key aspects of the mining system problem.

In this paper, we presented the type specification of the ControlStation in Section 6.1. In Section 6.2 we present the specification and design of the AirExtractorControl component, which is a component of ControlStation. In section 7 the type specification of the AirExtractorControl component is presented including the architecture considered in the context of the MiningSystem, which is shown in Figure 13.

6.1. The ControlStation Component Specification and Design

The ControlStation component of the MiningSystem is responsible for guaranteeing the suitable environment conditions for mineral extraction. In adverse situations, the mineral extraction should be interrupted to initiate the extraction of air or the removal of water for maintaining the safety conditions of the mine. Therefore, in order to obtain components with specific responsibilities, the ControlStation should be decomposed into three small components: PumpControl component which is responsible for extraction of water, AirExtractorControl component which is responsible for extraction of methane, and MineralExtractorControl component which is responsible for extraction of mineral.

Figure 9 shows the type specification for the ControlStation and the operations associated with it

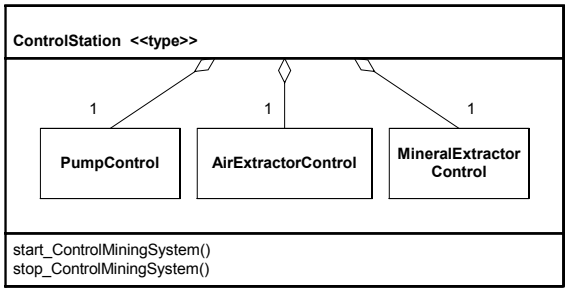

Figure 9 - Type model for the ControlStation

6.2. The AirExtractorControl Component Specification and Design

In this section, the AirExtractorControl component of the ControlStation is specified and designed, following the approach previously introduced. For illustration purposes, the specification and design will be restricted to the collaboration. Also in this section, we illustrate how the refinement of exceptional behaviour should be performed in the context of structural decomposition, and how the UML collaboration diagram can be employed in

the understanding of a system/component exceptional behaviour.

6.2.1. The AirExtractorControl Collaboration

When the methane sensor detects high level of methane in the mine, the collaboration ExtractAir is initiated to keep the methane level under control. The collaboration associated with the extraction of air from the mine is represented in Figure 10. For starting the air extraction, it is necessary to switch on the AirExtractor, which should be detected by the AirFlow sensor. When the methane level becomes normal, the collaboration ExtractAir has to finish by switching off the AirExtractor, which is also reflected by the AirFlow sensor.

An exceptional behaviour associated with this collaboration is the failure of the extractor that is detected by the AirFlow sensor. This can be described in Figure 11.

7. Case Study: Refinement of Exceptional Behaviour

In this section, in order to obtain a type specification and an architectural design that considers exceptional behaviour of the component, we refine the exceptional behaviour of the mining system, following the approach described in Section 4.2. The component type specification will include an abnormal class hierarchy that implements the exceptional behaviour of its associated normal class, which was identified during the design of the collaborations. On the other hand, the component architecture will be represented by idealised fault-tolerant components that explicitly represent internal and external components exceptions. In Section 7.1, the AirExtractor type is refined to incorporate exceptions and their handlers, identified for the collaboration ExtractAir, and in Section 7.2 the exception flow in the MiningSystem architecture is represented in terms of idealised fault-tolerant components.

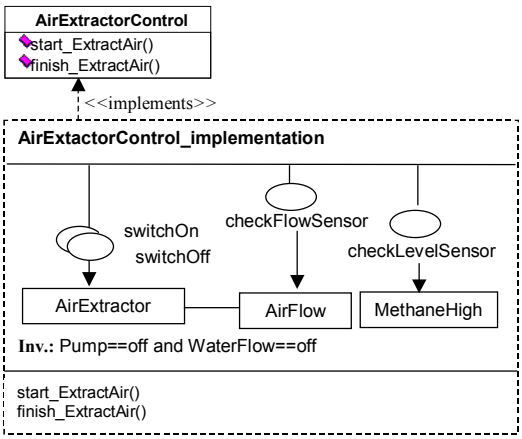

Figure 10 - Collaboration AirExtractorControl

Normal Behaviour	Exceptional Behaviour
Action(AirExtractor)::switchOn **Pre:** the level of methane is high, the air extractor is switched off, and there is no airflow; MethaneHigh==on and AirExtractor ==off and AirFlow==off **Post:** the air extractor is on, and there is airflow; AirExtractor ==on and AirFlow==on	**Signal**: the air extractor is on, but no airflow is detected; (AirExtractor ==on and AirFlow ==off) implies l_excSwitchAirExtractorOn_EA! **Handler:** switches off the air extractor and raises an exception; AirExtractor:=off and E_excAirExtractorFailure_EA! **Post**: the air extractor is switched off, and there is no airflow; AirExtractor ==off and AirFlow==off
Action(AirExtractor)::switchOff **Pre:** the air extractor is on, and there is airflow; AirExtractor==on and AirFlow==on **Post:** the air extractor is switched off, and there is no airflow; AirExtractor ==off and AirFlow==off	**Signal**: the air extractor is off, but the sensor detects airflow; (AirExtractor ==off and AirFlow==on) implies l_excSwitchAirExtractorOff_EA! **Handler:** propagates the exception; E_ excAirExtractorFailure _EA! **Post**: the air extractor is switched off, and there is airflow; AirExtractor ==off and AirFlow==on
Action(AirFlow)::checkFlowSensor **Post:** AirFlow==AirFlow@pre	**Signal**: the extractor is on and the sensor does not detect airflow (AirExtractor ==on and AirFlow==off) implies l_excAirExtractorFailure_EA! **Handler:** switches off the extractor and raises an exception AirExtractor := off and E_excAirExtractorFailure_EA! **Post**: the air extractor is switched off, and there is airflow; AirExtractor ==off and AirFlow==off
Action(MethaneHigh)::checkLevelSensor **Post**: MethaneHigh == MethaneHigh@pre	

Figure 11 – Normal and exceptional behaviour of collaboration AirExtractorControl

7.1. Identified Exceptions for AirExtractor Type

Accordingly with Section 4, software designers should structure their application by creating a set of *normal classes* that implement the normal activities of the software component, and *abnormal classes* which implement the abnormal activities (see Section 4.2.2). In Figure 12, the AirExtractorControl type is composed by its normal activity implemented by class AirExtractorControl, and its abnormal activity implemented by class ExceptionalAirExtractorControl.

The AirExtractorControl component implements the ExtractAir collaboration (Section 6.2). The external behaviour of this collaboration is implemented by the actions start_ExtractAir and finish_ExtractAir. These external actions become public methods of the normal class of the AirExtractorControl component. The internal collaboration actions (swithcOn, switchOff, checkFlowSensor, checkLevelSensor) are represented as private methods of the normal class. The exceptions identified during the collaboration ExtractAir are hierarchically organised and each operation represents explicitly the exceptions that may be raised by showing *send* dependencies between operations and their exceptions. The method switchOn raises the exception l_excSwitchAirExtractorOn_EA, the method switchOff raises l_excSwitchAirExtractorOff_EA, checkFlowSensor raises l_excAirExtractorFailure_EA and checkLevelSensor does not raise exceptions.

The class l_excExtractAir allows the definition of internal exceptions raised by the component AirExtractor. Handlers for these exceptions are represented as methods of the class ExceptionalAirExtractorControl. The method H_AirExtractorFailure, H_SwithcAirExtractorOn, H_SwitchAirExtractorOff of the ExceptionalAirExtractorControl are responsible for handling the exceptions l_excSwitchAirExtractorOn_EA, l_excSwitchAirExtractorOff_EA and l_excAirExtractorFailure_EA, respectively.

If an exception cannot be handled internally by the component, a failure exception is raised to the caller. The class E_excExtractAir allows the definition of failures exceptions that are raised when the exceptional class is not able to recover the state of the component to an error free state.

7.2. Exceptional Flow in the MiningSystem Architecture

As discussed in Section 3, a dependable system may be composed by a set of idealised fault-tolerant components. In the mining system case study, the components are hierarchically organised as shown in Figure 13. In their hierarchical structure, the high-level components encapsulate lower level ones, i.e., high-level components request services from lower level ones, from which they can receive either normal or exceptional responses. Exceptional responses are represented by external exceptions that a component raises. Both internal and external exceptions raised by a component are explicitly represented in the specification of its type, similarly to what it was

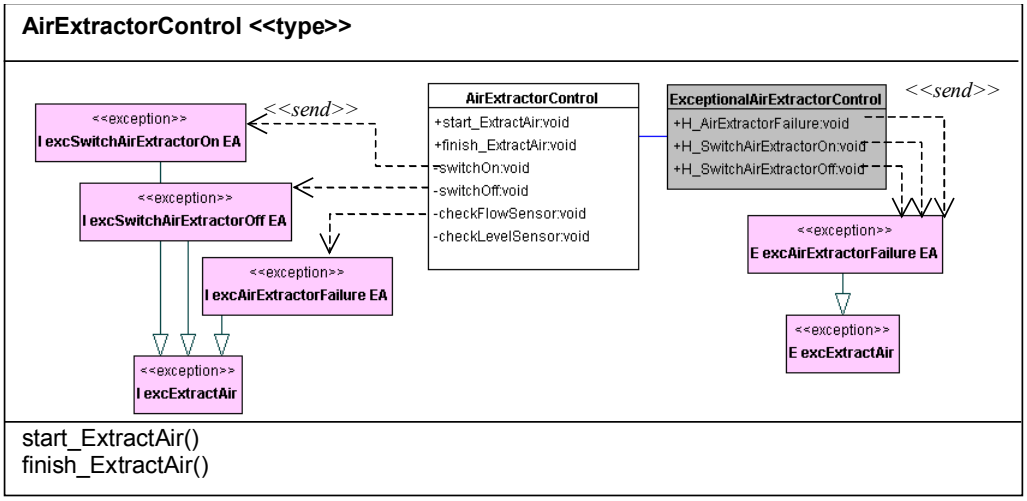

Figure 12 - Component AirExtractorControl

done for the AirExtractorControl component in Section 7.1.

Based on exceptions represented by the type specification model, the system architecture was refined to represent the internal component exceptions (I_excAirExtractorFailure_EA, I_excSwitchAirExtractorOn_EA, and I_excSwitchAirExtractorOff_EA) and external component exception propagation (E_excAirExtractorFailure_EA). Also in this diagram, we represent the exceptions associated with the component ControlStation and PumpControl identified with the same process performed for the AirExtractorControl component.

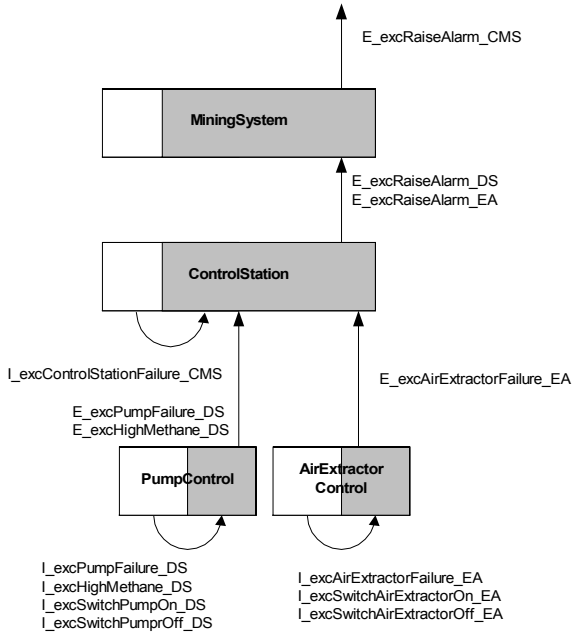

Figure 13 – Revised software architecture with exception flow

8. Case Study: Implementation Issues

The explicit separation of normal and exceptional behaviour of a component should be maintained when the designed components are translated into source code. On the contrary, these components will be difficult to understand, to change and to maintain in the presence of faults. Components designed in Section 6 must keep the separation of normal and abnormal activity. This can be achieved by using the exception handling mechanism presented in [5], which supports this separation in a structured and transparent manner.

Exceptions should be implemented as data objects, and different types of exceptions should be organised hierarchically as classes, being the class Exception the root of this hierarchy. Handlers may be associated either with classes or objects. In the first case, an exceptional class should be created, and in the second case, object handlers may be also defined. In Figure 12, the methods of ExceptionalAirExtractorControl are class handlers for the exceptions that should be treaded within AirExtractorControl´s methods. To implement handlers associated to individual objects, a new exceptional class must be created. This new class contains methods that implement the object handlers for the exceptions that should be treated in any method of the object. For instance, the object Pump, instance of the class Actuator, may be associated to the handlers that are distinct from the handlers associated to the object MineralExtractor that is also an instance of the class Actuator (Figure 14).

In this example, the association between the actuator and their exceptional classes (ExceptionalPump, ExceptionalAirExtractor or ExceptionalMineralExtractor) is automatically performed by the exception handling mechanism.

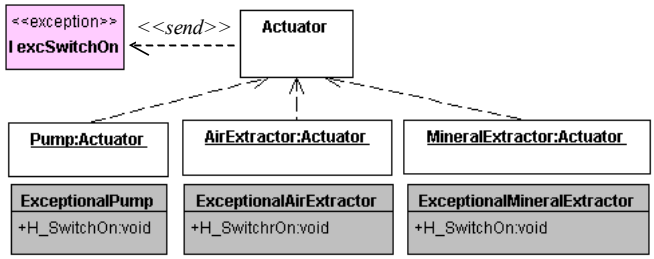

Figure 14 - Objects and their exceptional classes

9. Conclusions

In this paper we present a systematic approach for incorporating exceptional behaviour in the development of component-based systems, based on architectures that are composed by idealised fault-tolerant components. Such an innovative way of detecting and handling exceptions in the context of faults occurrence is particularly relevant to those systems in which high assurance is required from the services they provide.

The proposed approach relies on the premise that components alone are not able to provide an effective means to deal with exceptional behaviour when error recovery requires several components to interact. Hence the need for considering the notion of collaborations that would allow capturing the interactive behaviour between components. When collaborations (or contracts) are considered as first class design entities, there is also the need to employ development methodologies that incorporate these entities throughout the system development, such as Catalysis [4], the method on which our approach is based. However, Catalysis, as well as other similar approaches, lacks a coherent way of explicitly representing exceptional behaviour during the development of component-based software. For example,

although Catalysis defines alternative ways to express exceptions in collaborations, almost nothing is mentioned how these exceptions are captured during the requirements specification, transformed into a design, and finally implemented as software components.

In the approach presented in this paper, exceptional behaviour is explicitly captured in the use cases of the problem definition, in the architectural representation of the software system in terms of idealised fault-tolerant components, and in the final implementation. In the use cases, the exceptional behaviour is defined informally, as it is done for the normal behaviour. The architecture of the system is then presented using the model of idealised fault tolerant components, in which the propagation of exceptions between system components is explicitly represented. In the design representation of the system in terms of collaborations, the use cases are implemented as actions, which are defined in terms of pre- and post-conditions. The exceptional behaviour specified in the use cases is refined in the context of the actions that define the identified collaborations. Also at this stage, we define a template for representing the exceptional scenarios, and we present the collaboration diagrams for the normal and exceptional behaviour of the collaborations. And finally, we discuss how the resulting design can be implemented as software components.

While developing this work, we have come across several deficiencies regarding the modelling, analysis and design of exceptional behaviour in the software development. One of the problems that we have faced was the explicit representation of exception handling in the use cases, which were initially conceived for succinctly describing the problem at hand. However, if exceptional behaviour is to be considered at the requirements level, then exceptions have to be represented in the use cases, which tend to increase the complexity of their description. Hence the need to find alternative ways for describing use cases with different levels of detail, and to make sure that these descriptions are consistent and accurate.

Acknowledgements. Gisele R. M. Ferreira is supported by FAPESP grant number 00/03700-3, and Rogério de Lemos would like to acknowledge the financial support from The Nuffield Foundation.

References

[1] A. Avizienis. "Toward Systematic Design of Fault-Tolerant Systems". *Computer 30(4)*. April 1997. pp 51-58.

[2] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. Reading, MA. 1999.

[3] R. de Lemos, and A. Romanovsky. "Exception Handling in the Software Lifecycle*". Int. Journal of Computer Systems Science and Engineering 16(2)*. March 2001. pp.167-181.

[4] D. D'Souza, and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA. 1998.

[5] A.Garcia, D. Beder, and C. Rubira "An Exception Handling Mechanism for Developing Dependable Object-Oriented Software based on Meta-level Approach". *Proceedings of 10th IEEE Symposium on Software Reliability Engineering.*1999.

[6] A.F. Garcia, D.M. Beder, and C.M.F. Rubira. "An Exception Handling Software Architecture for Development Fault-Tolerant Software". *Proc. of the 5th IEEE International High-Assurance Systems Engineering Symposium (HASE 2000)*. Albuquerque, NM. November 2000. pp. 311-320.

[7] M.-O., and J.-C. Fabre. "Implementing a Reflective Fault-Tolerant CORBA System". Proceeding of the 19th Symposium on Reliable Distributed Systems (SRDS'2000). Nurnberg, Germany. 2000. pp. 154-163.

[8] P. Lee, and T. Anderson. *Fault-Tolerance: Principles and Practice*. 2nd Edition. Springer-Verlag. Berlin, Germany. 1990.

[9] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall. 1988.

[10] R. Miller, and A. Tripathi. "Issues with Exception Handling in Object-Oriented Systems". *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97).* Jyväskylä, Finland. Lecture Notes in Computer Science 1241. M. Aksit, S. Matsuoka (Eds.). Springer-Verlag. Berlin, Germany. 1997. pp. 85-103.

[11] M. Sloman, and J. Kramer. *Distributed Systems and Computer Networks.* Prentice Hall. 1987.

[12] A. F. Zorzo, and R. J. Stroud. "A Distributed Object-Oriented Framework for Dependable Mulitparty Interactions". *Proceedings of the Conference on Object-Oriented ProSgramming, Systems, Languages and Applications (OOPSLA'99)*. Denver, USA. November 1999. pp. 435-446.