DREP: A Requirements Engineering Process for Dependable Reactive Systems

Sadaf Mustafiz and Jörg Kienzle

School of Computer Science, McGill University
Montreal, Quebec, Canada
sadaf@cs.mcgill.ca, joerg.kienzle@mcgill.ca

Abstract. Discovering and documenting potential abnormal situations and irregular user behavior that can interrupt normal system interaction is of tremendous importance in the context of dependable systems development. Exceptions that are not identified during requirements elicitation might eventually lead to an incomplete system specification during analysis, and ultimately to an implementation that lacks certain functionality, or even behaves in an unreliable way. This paper presents a requirements engineering process, DREP, that systematically guides the developer to consider reliability and safety concerns of reactive systems. After the discovery of normal system behavior by means of use cases, the developer is lead to explore exceptional situations arising in the environment that change the context in which the system operates and service-related exceptional situations that threaten to fail user goals. The process requires the developer to specify means that detect such situations, and to define the recovery measures that attempt to put the system in a reliable and safe state. The process is iterative, and refinements are carried out, if necessary, to achieve desired quality levels. To conclude the requirements phase, an extended use case diagram summarizes the normal interactions, exceptions, handlers and their relationships. The proposed process is demonstrated with the 407 Express Toll Route System case study.

1 Introduction

Complex computer systems are increasingly built for highly critical tasks, from military and aerospace domains to industrial and commercials areas. Failures of such systems may have severe consequences ranging from loss of business opportunities, physical damage, to loss of human lives. Systems with such responsibilities should be highly *dependable*.

On the software developer's part, this involves acknowledging that many exceptional situations may arise during the execution of an application, and providing measures to handle such situations. When using a standard software development process to develop systems, there is no guarantee that such situations are considered during the development. Whether the system can handle these situations or not depends highly on the imagination and experience of the developers. In addition, even if the application can actually deal with these special situtations, the particular way that the developer chose to address that situation might not be the one that a typical user of the system would expect if it was not explicitly agreed upon and documented in the requirements.

As a result, the final application might not function correctly in all possible situations or react in unexpected ways. This can at best annoy or confuse the user, but can also have more severe repercussions.

When developing dependable systems, nothing should be left to chance. Following the idea of integrating exception handling into the software life cycle [1, 2], this paper describes an extension to standard *use case*-based requirements elicitation that leads the developers to consider dependabaility issues early on. Our approach focusses in particular on reliability and safety concerns. We believe that thinking about behaviour or events that affect the reliability or safety of the system has to start at the requirements phase, because it is up to the stakeholders of the system to decide how they expect the system to react to exceptional situations. Only with exhaustive and detailed user feedback is it possible to discover and then specify the complete system behavior in a subsequent analysis phase, and decide on the need for employing fault masking and fault tolerance techniques for achieving run-time dependability during design.

This paper describes *a use case-driven requirements engineering and analysis process*, DREP, that leads the developers to consider dependability issues early on during software development. Our approach focuses in particular on *reliability* and *safety* concerns. This paper focuses on the process itself, and hence complements the papers [3, 4], which describe the exceptional use case notation used in the process, and papers [5, 6], which describe our model-driven approach on mapping exceptional use cases to DA-Charts and Markov chains to perform dependability analysis.

The paper is structured as follows: Section 2 introduces the dependability attributes and gives a brief overview of exceptions, handlers, and use cases. Section 3 describes our proposed process, and the ideas are illustrated by means of the 407 highway toll route case study in Section 4. Section 5 presents DREP in the context of model-driven engineering. Section 6 describes an academic experiment we conducted with 40 software engineering graduate students to validate the applicability and effectiveness of our proposed process. Section 7 presents related work in this area and Section 8 discusses future work and draws some conclusions.

2 Background

2.1 Requirements Engineering

Requirements engineering can be categorized as *requirements development* and *requirements management*. Requirements development involves several activities: *discovery* and *elicitation* of the system functionality, properties and qualities, *definition* and *specification* of the requirements and precise definition of the system boundary, and *analysis* of the requirements to ensure that they are correct, complete and that they meet the stakeholders expectations. If the analysis reveals undesired properties or flaws, the specification has to be refined. Once the system is implemented, the running system can be validated against the requirements.

2.2 Use Cases

Use cases are a widely used formalism for discovering and recording behavioral requirements of software systems [7]. A use case describes, without revealing the details

of the system's internal workings, the system's responsibilities and its *interactions* with its environment as it performs work in serving one or more requests that, if successfully completed, satisfy a goal of a particular stakeholder. A use case can contain several scenarios including the main success scenario and alternate scenarios. The external entities in the environment that interact with the system are called *actors*.

Use cases are stories of actors using a system to *meet goals*. The actor that interacts with the system in the pursuit of a well defined goal is referred to as the *primary actor*. External entities that are required by the system in order to achieve its functionality are called *secondary actors*. Secondary actors include software or hardware that is out of our control. The system, on the other hand, is the software that we are developing and which is under our control.

2.3 Dependability

Systems are developed to satisfy a set of requirements that meet a need. A requirement that is important in mission- and safety-critical systems is that they be highly dependable. *Dependability* [8] is that property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability involves satisfying several requirements: availability, reliability, safety, maintainability, confidentiality, and integrity. The dependability requirement varies with the target application, since a constraint can be essential for one environment and not so much for others. In this paper, we focus on the *reliability* and *safety* attributes of dependability.

Reliability. The *reliability* of a system measures its aptitude to provide service and remain operating as long as required [9]. Reliability of a service is typically measured in *probability of success* of the service, once requested, or else in *mean time to failure*. If the average time to complete a service is known, it is possible to convert between the two values.

Safety. The *safety* of a system is determined by the lack of catastrophic failures it undergoes [9]. The seriousness of the consequences of the failure on the environment can range from benign to catastrophic. Seriousness of consequences can be measured with a safety index. For instance, the DO-178B standard for civil aeronautics defines safety index values from 0 to 4 with the following meaning:

0. *Without effects*;

1. *Minor effects* lead to upsetting the stakeholders or increasing the system workload;

2. *Major effects* lead to minor injuries of users, or minor physical damage or monetary loss;

3. *Dangerous effects* lead to serious injuries of users, or serious physical damage or monetary loss;

4. *Catastrophic effects* lead to loss of human lives, or destruction of the system.

Each application has different safety requirements. It is now up to the developer in consultation with all the stakeholders to define the number of safety levels to consider, and their exact definitions.

Fault tolerance is a means of achieving system dependability. As defined in [10], fault tolerance includes error detection and system recovery. Error detection involves identification of erroneous state in the system by means of acceptance tests or active

redundancy. Late or dead processes can be detected using timers that sound an alert when a deadline for a specific interaction or functionality expires. The error might lead to a permanent or transient failure. *Permanent failures* are failures that persist, and lead to a loss of service until appropriate recovery measures are taken. *Transient failures* are failures which disappear over time. System recovery involves correcting such problems to ensure that the system continues to deliver its services. Forward error recovery techniques restore the system to a new, possibly degraded, state. This approach requires knowledge of the errors and hence is application-specific, but is efficient and suitable in cases of anticipated faults and missed deadlines. A popular forward error recovery technique, exception handling, is discussed in Section 2.4.

At the use case level, error detection involves detection of exceptional situations by means of secondary actors such as sensors and time-outs. Recovery at the use case level involves describing the interactions with the environment that are needed to continue to deliver the current service, or to offer a degraded service, or to take actions that prevent a catastrophe. The former two recovery actions increase reliability, whereas the latter ensures safety.

2.4 Exceptions and Handlers

An exceptional situation, or short *exception* [1], describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. Hence, an *exception occurrence* during a program execution is a situation in which the standard computation cannot pursue. For the program execution to continue, an atypical computation is necessary [11].

A programming language or system with support for exception handling allows users to signal *exceptions* and to define *handlers* [12]. To *signal* an exception amounts to detecting the exceptional situation, interrupting the usual processing sequence, looking for a relevant handler, and then invoking it.

Handlers are defined on (or attached to) entities, such as data structures, or *contexts* for one or several exceptions. According to the language, a context may be a program, a process, a procedure, a statement, an expression, etc. Handlers are invoked when an exception is signaled during the execution or the use of the associated context or nested context. To *handle* means to put the system to a coherent state, i.e. to carry out forward error recovery, and then to take one of these steps: transfer control to the statement following the signaling one (*resumption model* [1]); or discard the context between the signaling statement and the one to which the handler is attached (*termination model* [1]); or signal a new exception to the enclosing context.

3 A Dependability-Focused Requirements Engineering Process

Our **D**ependability-focused **R**equirements **E**ngineering **P**rocess (DREP) targets the development of dependable reactive systems. It defines detailed steps or *tasks* that lead the

[1] It should be noted that the terms *exception* and *handler* are used in this paper at a higher level of abstraction and does not necessarily map to programming language exceptions.

developer to pay particular attention to system safety and reliability when performing requirements elicitation, specification and analysis. The following subsections describe the different activities in detail.

The basic tasks carried out as part of these activities in DREP are outlined in a hierarchical manner in Fig. 1. To clearly illustrate our extensions, the tasks that are part of standard use case analysis are shown in boxes with dashed line borders.

3.1 Requirements Elicitation and Discovery

Task 1: Discovering Actors, Goals, and Modes The first task can be divided into several sub-tasks.

1.1 Brainstorm services/goals and outcomes
1.2 Brainstorm actors
1.3 Classify services/goals and actors
1.4 Decompose services into subgoals
1.5 Brainstorm modes

The first activity in use case based requirements elicitation consists in establishing a list of actors and stakeholders, with a special emphasis on *primary actors*, i.e. external entities in the environment that interact with the system in the pursuit of a well defined goal. Secondary actors are also documented during this brainstorming activity, if their use is indeed part of the requirements and is not already part of the solution domain.

For each of the discovered goals, a *use case outline* is written. This outline consists in a textual summary of the goal, an explanation of the context in which the primary actor wants to achieve the goal, and a clear description of the value or service that the system has to provide to satisfy the primary actor. Complex goals can be split into several subgoals to form a hierarchy [13], in which case a use case outline is written for each of the subgoals. The goals are further be classified as normal services or other special services.

The brainstorming activity can also lead the developer to discover that a given service might have several acceptable *outcomes*, i.e., the system can satisfy the goal of the primary actor in multiple ways.

Finally, during this task, the developer should also consider possible modes of operation to be offered by the system. An operation mode is defined by the set of services that the system offers when operating in that mode[2]. During normal operation, a system should try and provide all of the services it is intended to provide at any given time. There is no need to artificially create different normal modes of operation. Some systems, however, need more than one normal mode of operation, and allow the user to switch between these modes by request or to accommodate changes in the environment. For example, a cell-phone can be put into a child-safe mode, in which the only service offered is to place local calls.

[2] For each service provided in a mode, reliability and safety levels have to be specified as explained in task 4.

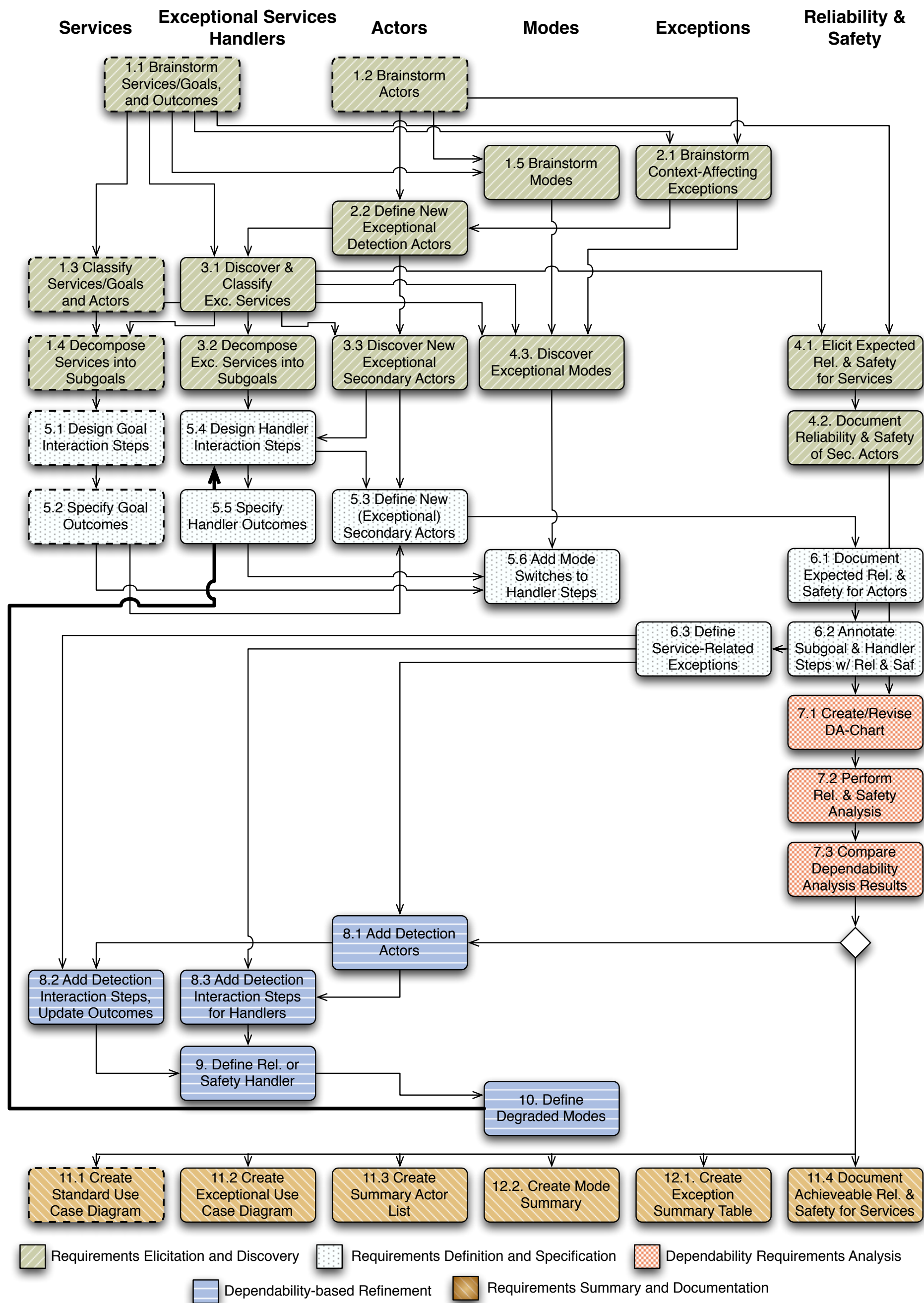

Fig. 1. Task Structure of DREP

Task 2: Discovering Context-Affecting Exceptions Task 2 involves carrying out the following two sub-tasks.

2.1 Brainstorm context-affecting exceptions
2.2 Define new exceptional detection actors

In this step, the developer has to focus on *context-affecting exceptional situations*, i.e., situations that change the context in which the system operates. Certain context changes might require a dependable system to adapt in order to continue to provide reliable and safe service. To help discover these situations, the following questions should be answered:

- What situations / conditions / changes in the environment make it impossible for the entire system to provide safe service? In such situations, should the system provide some other service?
- What situations / conditions / changes in the environment prevent the system from satisfying a primary actor's goal (or subgoal)? In such situations, can the system partially fulfill the service?
- What situations take priority over the primary actor's goal?
- What situations / conditions / changes in the environment could make the primary actor change his goal? In such situations, how can the primary actor inform the system of the goal change?

For each exceptional situation that is discovered, a *named exception* is defined, together with a small text that describes the situation in more detail. All discovered exceptions are documented in an exception table.

This activity typically leads to the discovery of new *exceptional goals*. Often, the occurrence of the situation cannot be detected by the system without help from the environment, which means that new *exceptional actors* have to be introduced. For example, in an elevator system where safety is the main concern, in case of a fire outbreak in the building, the elevator operator or a smoke detector, both exceptional actors, should activate the fire emergency mode of the elevator control software.

Task 3: Eliciting Handlers for Context-Affecting Exceptions This task can be split into the following sub-tasks.

3.1 Discover and classify exceptional services
3.2 Decompose exceptional services into subgoals
3.3 Discover new exceptional secondary actors

For each identified exception, a *handler use case* outline has to be established describing how the system is supposed to react or *recover* from that situation. A handler can be further classified as a *safety* or *reliability* handler depending on the concern it attempts to satisfy. A handler can also be linked to one or several contexts, i.e. use cases during which the exceptional situation can occur. Upon occurrence of the exception, the current interaction is interrupted and the exceptional interaction begins. In an elevator system, for example, in case of a fire outbreak signalled by a smoke detector, standard elevator operation is interrupted. To ensure safety, the elevators are brought to the ground floor.

Task 4: Eliciting Dependability Expectations and Discovering Exceptional Modes This task can be split into the following sub-tasks.

4.1 Eliciting dependability expectations for each service
4.2 Document provided reliability and safety of mandatory secondary actors
4.3 Discover exceptional modes of operation

For dependable systems, it is at this phase important to discover the requirements with respect to safety and reliability for each service that the system provides. Mission and safety-critical systems often have to comply with *safety standards*, but even if the requirements do not require compliance with a standard, stakeholders and primary actors explicitly or implicitly *expect a certain degree* of safety and reliablility from a dependable system. To document the desired dependability, reliability and safety annotations have to be added to the use case outlines that, for each goal, specify the desired probability of successful achievement of the goal, as well as the maximum tolerable probability of occurrence of a safety violation.

The desired safety and reliability values should be elicited not onlyfor the normal services of the system, but also for the new *exceptional goals* discovered in task 2.

It is important to note here that in the real world, 100 percent dependability is never achievable. If the specified safety and reliability are too high, then it might be impossible (or too expensive) to implement a system that fulfills the requirements. It is hence important that the stakeholders decide on acceptable risks at this point.

Next, for each service offered by a mandatory secondary actor, the developer has to document the service's reliability and safety properties. If the secondary actor is a piece of hardware, then the reliability can be found in the specification manual.

Whenever a dependable system has encountered difficulties performing a requested service due to some exceptional situation, the effect of the encountered problem on future service provision of the system has to be evaluated. If the reliability or safety of future service provision is threatened, then a *mode switch* is necessary. Switching to a different operation mode (an exceptional mode) allows the system to signal to the environment that the services offered by the system have changed, and reject any requests for services that cannot be performed with sufficient reliability or safety. We have addressed exceptional modes of operation in the behavioural models used in DREP, and details can be found in [14].

While normal modes of operation have been discovered in task 1.5, this task concentrates on the discovery of emergency and restricted modes. In an *emergency mode*, normal services are suspended and only emergency services, possibly initiated by a new exceptional actor, are available. The system is in a state in which it cannot provide any of its normal services anymore, not even in a degraded form. This is usually due to safety reasons. For example, in case of a fire alarm, an elevator system does not handle user requests anymore, but moves all the elevator cabins to the ground floor. In an *restricted mode*, a combination of emergency services and normal services are offered. The system is in an exceptional state in which only a subset of the normal services are available and the functions of particular emergency services are also required.

To discover emergency and restricted modes, all context-affecting exceptions identified in task 2.1 need to be considered. In a new context, some of the services provided under normal circumstances might not be adequate anymore. Therefore, the developer

should reflect on the impact of the context change on each of the services provided by the system. If the safety of normal services is threatened by a situation, then an appropriate exceptional or emergency mode should be defined.

If during this task new modes of operation have been defined, then it is important to specify the expected reliability and safety of each service (see task 4) provided in each mode.

It is important to note that mode definitions are not based on the developer's creativity. Each mode has to be validated with the stakeholders to check if, according to them, the services provided in the mode form a coherent set, and that the provided levels of reliability and safety for each service are sufficient.

3.2 Requirements Definition and Specification

Now that the goals and subgoals have been identified, detailed use case descriptions have to be elaborated for each of them. We suggest to describe use cases with a predefined template as done by others [15], which forces the developer to explicitly document all relevant features.

Reactive systems only perform work or produce output after they receive an input event. Therefore, the main parts of our use case template consist of a numbered list of individual base interaction *steps*, each one describing either an *input interaction* – an external actor decides to send a message/data/event to the system – or an *output interaction* – the system sends a message/data/event to an external actor. If a use case is decomposed into subfunction-level use cases, a step can also be a reference to a lower level use case, which in turn describes the base interaction steps that leads to the completion of the subgoal. In any case however, a use case describing a user goal could be flattened into a sequence of base interaction steps, if needed.

Task 5: Designing Interactions Interaction design in DREP is comprised of several sub-tasks.

5.1 Design goal interaction steps
5.2 Specify goal outcomes
5.3 Define new (exceptional) secondary actors
5.4 Design handler interaction steps
5.5 Specify handler outcomes
5.6 Add mode switches to handler steps

The standard way of achieving a goal is described in the *main success scenario* part of the template. The ordering of the individual interaction steps are often dictated by logic, by required usage patterns, by user interfaces, or by protocols enforced by secondary actors interacting with the system. Where flexibility exists, the stakeholders should be consulted to choose the most adequate interaction pattern.

When designing the goal interaction steps, it is also necessary to define the service outcome. The main success scenario of a user goal can end in only one possible way, and the use case should clearly show this. From the users' perspective, the goal outcome

can be one of the following: $\ll \mathit{success} \gg$, $\ll \mathit{failure} \gg$, or $\ll \mathit{goal\ abandoned} \gg$[3]. If alternate scenarios are available, it is also necessary to specify the outcome of all such alternate paths, and to document them in the use case extension section.

When the requested service cannot be provided, a dependable system should strive to handle the current situation and attempt to provide partial service, if possible. Partial service, or *degraded service outcome* as we call it, happens when a service does not deliver what initially promised, but yet provides something that potentially satisfies the requester of the service. A degraded outcome is better than a complete failure to deliver the service.

Intuitively, a service provision can only result in a degraded outcome when an exceptional situation has occurred. Reacting to such an exceptional situation, and providing a well-defined outcome can only be done within a handler use case. Therefore, after the detailed interaction steps of a handler have been designed, the outcome of the handler should be clearly defined. Handlers can end in $\ll \mathit{success} \gg$, $\ll \mathit{degraded} \gg$ or $\ll \mathit{failure} \gg$.

Whenever an exception has lead to the definition of a new mode, then the steps of the handler that addresses the exception have to be updated to indicate a mode switch. In general, mode switches should be performed as soon as possible, i.e. as soon as it becomes apparent that the provision of the current services at the required reliability and safety level can not be sustained.

Task 6: Defining Service-Related Exceptions and Effects on System Reliability and Safety This task focuses on discovering service-related exceptions and documenting dependability values.

6.1 Document expected reliability and safety for actors
6.2 Annotate subgoal and handler steps with reliability and safety
6.3 Define service-related exceptions

The successful completion of a user goal may be threatened due to service-related exceptional situations. Service-related exceptions have many natures:

- The system state makes the provision of a service impossible[4],
- Failure of secondary actors that are necessary for the completion of the user goal,
- Failure of communication links between the system and important secondary actors,
- Actors violate the system interaction protocol, i.e. they invoke system services in the wrong order, or at the wrong time.

Possible service-related exceptions can be discovered most effectively following a bottom-up approach. DREP requires the developer to examine each individual base step of a use

[3] To correctly calculate reliability, it is important to separate the situations in which the user voluntarily abandons the goal from the situations in which the service fails. A service that is successfully cancelled upon user request represents a correct and reliable system behavior

[4] Addressing these situations is of course not new to our approach. Standard use case driven requirements engineering techniques usually specify the handling of such situations in an extension section of the use case.

case, sub-use case or handler, and reflect on the consequences that a failure of the step has on reliability, i.e. the achievement of the goal, and on system safety. The developer should answer the following questions:

- If this step is omitted, will the goal fail? If yes, the step should be annotated with a *reliability tag*, together with the probability of success of the step.
- If this step is omitted, is the safety of the system threatened? If yes, the step should be annotated with a *safety tag*, together with the corresponding safety level and the probability of success of the step.

A *named exception* should be defined for each service-related exceptional situation, together with a small text that describes the situation in more detail. For example, in an elevator system, a motor failure, i.e. the situation in which the motor does not react to commands anymore because of a hardware or communication failure, is a serious threat to safety and reliability. The identified exception is then added to the exception table of environmental exceptions for documentation reasons.

3.3 Dependability Requirements Analysis

Task 7: Assessing Safety and Reliability The sub-taks involved in the assessment phase are listed here.

7.1 Create/revise DA-Chart
7.2 Perform reliability and safety analysis
7.3 Compare dependability analysis results with expected dependability values

In [16], we proposed a model-based approach for analyzing the safety and reliability of our use cases. Since each interaction step in a use case is annotated with a probability reflecting its chances of success, and a safety tag if the failure of the step hampers the system safety, it is possible to map the use case to a formalism that is well-suited for dependability analysis. For this purpose, we developed the DA-Charts formalism [16] which is a probabilistic extension of part of the statecharts formalism. We have implemented our formalism in the AToM^3 tool [17] to provide support for automatic dependability analysis. The tool allow a developer to create a DA-Chart that corresponds to the use cases established in tasks 1 - 6. The tool also verifies the formalism constraints and ensures that the mapping rules are adhered to. Based on path analysis of the DA-Charts, the tool quantitatively determines probabilities of reaching safe or unsafe states, or achieving the goal, providing a degraded success, or failing. For details on the DA-Charts formalism and the dependability analysis see [18].

The dependability determined by the tool can now be compared with the dependability required by the stakeholders as determined in task 4. If the analysis reveals an acceptable level of reliability and safety, then the requirements engineering process is complete, and a summary specification can be established (see tasks 11 and 12). Otherwise, the requirements need to be refined with handler use cases that address the service-related exceptions, which is described in tasks 8, 9 and 10.

3.4 Dependability-based Refinement

Based on the output of our analysis tool, the service-related exceptions that have significant negative effect on the system's reliability and safety can be identified. To improve the situation, the following tasks should be performed for *each one of them*.

Task 8: Specifying Detection Mechanisms Once possible exceptional situations have been elicited, it is important to carry out the following tasks.

8.1 Add detection actors
8.2 Add detection interaction steps and revisit goal outcomes
8.3 Add detection interaction steps for handlers

Before any recovery actions can be taken by the system, the exceptional situation has to be *detected*. The developer should investigate if the current actors and their interactions make the detection possible, and if not, adapt the interaction pattern or even add secondary detection actors to the system's environment.

Detection is usually done differently for *input* and *output* interactions. Omission of input to the system can usually be detected using timeouts. Invalid input data can be detected with checksums, etc. In both cases, no additional detection actors have to be introduced. The use case has to be updated by adding the discovered exception to the extension section of the template as an alternative to the essential input step. If necessary, new use case outcomes might have to be defined.

Output failure is more difficult to handle. Whenever a system output triggers a critical action of an actor, then the system must make sure that it can detect eventual communication problems or failure of an actor to execute the requested action. This very often requires additional hardware, e.g. a sensor, to be added to the system. The job of this new actor is to inform the system that the essential actor successfully executed the system's request. This new acknowledgement step has to be added to the main success scenario after the essential output step in the use case or handler, and an exception representing the failure of the output, detected by a timeout while waiting for the acknowledgement, is added to the extension section as an alternative to the acknowledgement step. For example, an elevator control software might request the motor to stop, but a communication failure or a motor misbehaviour might keep the motor going. Additional hardware, for instance, a sensor that detects when the cabin stopped at a floor, might be necessary to ensure safety or reliability.

Task 9: Specifying Handler Use Cases If the exception puts the user in danger, then measures must be taken to put the system in a safe state. If the exception threatens the successful completion of the user goal, reliability is at stake. It should then be investigated if the system can recover and meet the user goal in an alternative way.

In any case, exceptional interaction steps with the environment are performed during recovery, and hence must be specified in a separate reliability or safety handler use case[5]. Very often, actors – especially humans – are "surprised" when they encounter an

[5] Separation of handlers also enables subsequent reuse of handlers. Just like a subfunction-level use case can encapsulate a subgoal that is part of several user goals, a handler use case can encapsulate a common way of handling exceptions that might occur while processing different user goals. Sometimes even, different exceptions can be handled in the same way.

exceptional situation, and are subsequently more likely to make mistakes when interacting with the system. Exceptional interactions must therefore be as intuitive as possible, and respect the actor's needs.

If the goal of the primary actor cannot be achieved, then it is of paramount importance to inform him of the situation by an appropriate output interaction. In some cases, it might not be possible to satisfy a user's goal completely, but a dependable system can instead offer a degraded form of service. For example, a user might order a product online and request for delivery before a certain date. If the system is unable to satisfy this request, the user might be offered the option to pick-up the order at the store instead. The handler use case should then define a new *degraded outcome* for this situation.

If omission of input from an actor can cause the goal to fail, then, once the omission has been detected, different options of handling the situation have to be considered. For instance, prompting the actor for the input again after a given time has elapsed, or using default input are possible options. Safety considerations might make it even necessary to temporarily shutdown the system in case of missing input.

Invalid input data is another example of input problem that might cause the goal to fail. Since most of the time the actors are aware of the importance of their input, a reliable system should also acknowledge input from an actor, so that the actor realizes that he is making progress in achieving his goal.

Task 10: Defining Degraded Modes For each of the service-related exceptions identified in task 6.3 and handled in task 9, the developer should evaluate the effects that the service-related exception has on future requests for the same service or other services that could be affected by the exception. In the case where these effects lower the reliability and safety of the service below the required level specified by the current mode, then a degraded mode should be defined.

A *degraded mode* of operation (of a normal mode) offers only limited services. Some services of the normal mode are still provided as is. Some services are provided, but with a lower degree of reliability and safety. In this case, the service is said to be offered with *degraded quality of service* (QoS). For example, a web browser running low on memory might switch into a mode where only textual elements from webpages are displayed and graphical elements and other media are suppressed to save memory.

Iteration To complete this iteration, every interaction step of the newly defined handler of task 9 must again be elaborated (task 5.4), the outcomes must be specified (task 5.5), and the essential steps tagged with reliability probabilities and safety information (task 6.2). Finally, the developer can re-analyze the updated use cases (task 7) to determine if the required safety and reliability requirements can now be satisfied.

3.5 Requirements Summary and Documentation

To begin with the requirements documentation, the following tasks are suggested.

Task 11: Use Case Summary

11.1 Create standard use case diagram

11.2 Create exceptional use case diagram
11.3 Create summary actor list
11.4 Document achievable reliability and safety for services

Whereas individual use cases are text-based, the UML use case diagram provides a concise high level view of the use cases of a system. It allows developers to graphically depict the use cases, the actors that interact with the system, and the relationships between actors and use cases. To begin with in this phase, a standard use case diagram based on the actors and goals defined earlier should be created.

In a use case diagram, standard use cases appear as ellipses, associated to the actors whose goals they describe. In [19] we extended use case diagrams and proposed to identify handler use cases with a <<handler>> stereotype or even a different graphical symbol in order to differentiate them from standard use cases. We also suggest classifying the handlers as a <<safety handler>> or a <<reliability handler>>. Our notation for handlers is illustrated in Fig. 8. Having different type of handlers enables quick identification of functionality that affects safety or reliability of a system, as well as identification of safety-critical parts of the system. It allows the developer in collaboration with the stakeholders to decide, for instance, how much resources should be allocated to the development of the functionality defined in the handler use cases, or to prioritize between safety and reliability in case of conflict.

Handler use cases are associated to a base use case, which may be any standard use case or other handler use case. We suggest to depict this association in the use case diagram by a directed relationship (dotted arrow) linking the handler use case to its base use case. This relationship is very similar to the standard UML <<extends>> relationship. It specifies that the behavior of the base use case may be affected by the behavior of the handler use case in case an exception is encountered.

In case of an occurrence of an exceptional situation, the base behavior is put on hold or terminated, and the interaction specified in the handler is started. A handler can temporarily take over the system interaction, for instance to perform some compensation activity, and then switch back to the normal interaction scenario. In this case, the relationship is tagged with a <<interrupt& continue>> stereotype. Some exceptional situations, however, cannot be handled smoothly, and cause the current goal to fail. Such dependencies are tagged with <<interrupt & fail>>. The exceptions that activate the handler use case are added to the interrupt relationship in a UML comment, similar to what is done for extension points.

In addition, a list of all primary and secondary actors both normal and exceptional should be developed. For each service to be provided by the system, it is necessary to document the reliability and safety that can be achieved.

Task 12: Summary Tables

12.1 Exception summary table
12.2 Mode summary table

For traceability and documentation reasons, all discovered environmental and service-related exceptions are recorded in a table during tasks 2 and 6. As a summary, the entries in this table already contains a small textual description of the exceptional situation

should be complemented with the exception contexts in which the exception can occur, the associated handler(s), and the mechanism for detecting the exception.

Finally, the **mode table** is created to summarize all modes of the system. The mode table can of course also be created earlier and updated iteratively whenever a new mode is defined. For each mode the table includes a *mode name*, a *description* of the mode, followed by a list of services that are provided in the mode. For each service, the *service name*, the *expected minimal reliability*, and the *expected minimal safety* are given.

4 Case Study: 407 Express Toll Route System

We illustrate the process described in Section 3 with the 407 ETR (Express Toll Route) System. The 407 ETR is a highway that runs east-west just North of Toronto, and was one of the largest road construction projects in the history of Canada. The road uses a highly modern Electronic Toll Collection system that allows motorists to pass through toll routes without stopping or even opening a window. The ETR system is a hard real-time application requiring high levels of dependability.

Vehicles can be registered with the 407 ETR system, in which case the driver is issued a small electronic tag, called a transponder, to be attached to the windshield. When the vehicle enters the highway, it passes under the overhead gantry. The hardware devices of a gantry include a vehicle detector, a locator antenna, a read/write antenna, cameras, lights, and a laser scanner. The locator antenna determines if the vehicle is equipped with a transponder. Next, the read/write antenna reads the account number from the transponder and the point of entry, time and date is recorded. In addition, the system uses laser scanners to determine the class of vehicle. The same process occurs when the vehicle exits the highway. The entry and exit data are then matched and the transponder account holder is debited.

Unregistered vehicles are identified by their license plate number. The system triggers cameras and lights to take pictures of the rear number plate. At the same time, the laser scanners are activated to classify the vehicle in order to determine the trip charge. The owner of the vehicle is identified by electronic access to government records. If the video correlation and image processing fails to determine the license plate with sufficient probability, a human operator has to look at the pictures to make the call.

4.1 Elicitation and Discovery

Task 1: Discovering actors, goals, and modes. In the ETR system there is initially only one primary actor, the *Driver*. The summary-level use case *UseHighway* is shown in Fig. 2. When interacting with the system, the driver has the goal of registering (*RegisterVehicle*), taking the highway (*TakeHighway*), payment of bills (*PayBill*), and cancelling the registration (*CancelRegistration)*. These goals can be further split into subgoals as summarized in Fig. 2.

The 407 ETR system only has one normal mode of operation since there is only one primary goal, using the highway, that needs to be satisfied at all times.

Task 2: Discovering Context-Affecting Exceptions. In the ETR system, an accident on the highway or extreme weather leading to critical road conditions, would re-

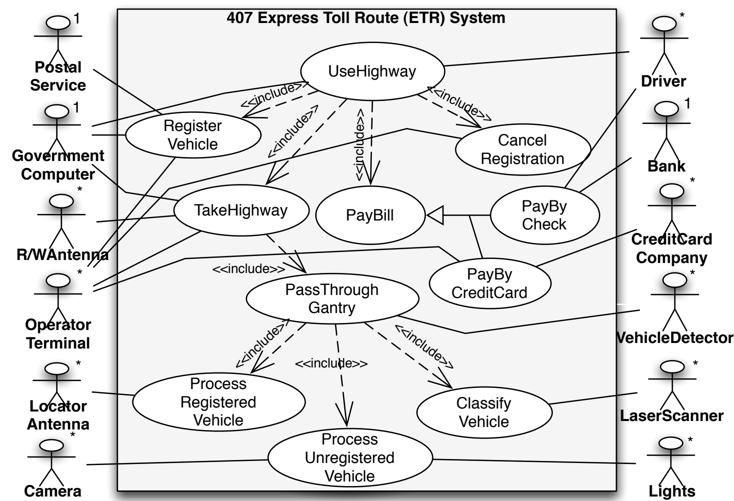

Fig. 2. 407 Standard Use Case Diagram

quire the highway operator, an exceptional actor, to temporarily close parts of the highway. Activating the emergency behavior is an *exceptional goal* for the operator, since this happens only in rare occasions. In this task, we identified an environmental exception: *HighwayUnavailable* which is signaled by the *HighwayOperator* exceptional actor when he receives the request from the Roadside Motorist Assistant Patrol.

Task 3: Eliciting Handlers for Context-Affecting Exceptions. After a discussion with the stakeholders it has been decided that closing the highway in case of emergencies is done by activating barriers that prevent new vehicles from entering the highway at the closed sections. The vehicles on the highway are to be informed of the situation by displaying messages on message boards. We therefore identified new secondary actors, the *Barrier*, and the *Message Board*.

Task 4. Eliciting Dependability Expectations. We define 3 safety levels for the 407 ETR: level 0 - without effects, level 1 - cars end up in a traffic jam / increased load on operators, level 2 - cars are damaged / system components are damaged. The reliability of *TakeHighway* and *PayBill* should be high, e.g. 0.999 (one of 10000 cars can fail to pay for a trip). Other services, e.g. registering and cancelling, require a reliability of 0.995. During an emergency, the chances of a level 2 safety violation should be very small, e.g. 0.00001, and a level 1 safety violation should be rare, 0.001. The required reliability of the emergency behavior should be very high, e.g. 0.99999.

Task 5: Discovering and Classifying Normal and Exceptional Modes.

As mentioned earlier, the system only offers one normal mode which includes all services associated to taking the highway, i.e. *TakeHighway*, *RegisterVehicle*, *PayBill*, and *CancelRegistration*.

In task 3, the context-affecting exception *HighwayUnavailable* was identified since critical road conditions might require the system to restrict vehicles from entering the highway. Emergency services such as activating the road barriers are carried out. However, the exit service needs to be active to allow the vehicles already on the highway

Use Case: TakeHighway

Level: User Goal

Primary Actor: Driver

Main Success Scenario:

1. Driver enters highway, passing through gantry.
2. Driver exits highway, passing through gantry.
3. System retrieves the driver's vehicle record based on trip information*.
4. System determines the amount owed based on the trip information and adds the transaction to the vehicle's records.
5. System informs Driver by sending a signal to the RWAntenna of successful completion of transaction.

Extensions:

- 3a. Vehicle is unregistered and does not have a record yet.
  - 3a.1. System sends licence plate information to GovernmentComputer.
  - 3a.2. GovernmentComputer sends vehicle information and owner's address to System.
  - 3a.3. System creates a new vehicle record. Use case continues at step 4.
- 3b. Vehicle is unregistered and licence plate is unrecognizable.
  - 3b.1. System displays pictures on OperatorTerminal.
  - 3b.2. OperatorTerminal sends licence plate information to System. Use case continues at step 3.
- 5a. Vehicle is not registered. Use case ends in ≪ *success* ≫.

Fig. 3. *TakeHighway* Use Case

to continue. Therefore, when such an exceptional situation arises the system needs to switch to a *restricted mode ExitOnly*.

4.2 Requirements Definition and Specification

Due to space constraints, we only discuss the user-goal level *TakeHighway*, and the subgoals *PassThroughGantry* and *ProcessRegisteredVehicles*.

Task 6: Designing Interactions.

The interaction steps required for *TakeHighway* are detailed in Fig. 3. To take the highway, the *Driver* enters the highway, and then exits it by passing through gantries. If the vehicle has a transponder, then the device beeps and blinks green after the driver exits the highway. The *ProcessRegisteredVehicle* use case, shown in Fig. 4, describes how the system communicates with the transponder, and verifies the class of the vehicle. The sub-functional level uses cases that describe the processing of unregistered vehicles and the classification of vehicles are not shown here for space reasons.

The extension section of *TakeHighway* on purpose describes only alternative ways to achieve the goal, since exceptional interaction will be shown later.

Fig. 5 shows the handler *ActivateBarrier* that handles the exception *HighwayUnavailable*. Handler use cases have an additional field in the use case template named *Contexts & Exceptions* that is used to document by which exception and in what context the handler is triggered. In our case, a *HighwayUnavailable* exception occurring puts the system in a restricted mode, at which time the system only allows exits but does not allow new goals to start. As a first handling step, the emergency road barriers are activated. Then the message boards are updated with a warning message. Subsequently when the road conditions improve, the *Operator* can deactivate the barriers and grant access to the highway once again. Since the handler attempts to satisfy user safety, we

Use Case: ProcessRegisteredVehicle

Level: Sub-Function

Primary Actor: N/A

Main Success Scenario:
1. LocatorAntenna notifies System that it detected an approaching vehicle with transponder.
2. System asks R/WAntenna to obtain account information from transponder.
3. RWAntenna informs System of account information.
4. System records account information for the trip.
5. System turns on the Lights.
6. System triggers the Cameras.
7. Cameras send images to System.
8. System determines licence plate information based on images.

Extensions:

1a. The approaching vehicle does not have a transponder. Use case ends in $\ll failure \gg$.

Fig. 4. *ProcessRegisteredVehicle* Use Case

Handler Use Case: ActivateBarrier

Handler Class: Safety

Context & Exception: TakeHighway{HighwayUnavailable}

Level: Usergoal

Primary Actor: Operator

Main Success Scenario:

System switches into restricted mode ExitOnly.
1. System activates barriers at entry gantries.
2. System displays "Highway Unavailable" at message boards on highway.
3. Operator informs System that highway is accessible again.
4. System deactivates barriers.
5. System clears message boards.

System switches back to normal mode.

Fig. 5. *ActivateBarrier* Handler Use Case

label the handler as a *safety handler*. This is shown in the *Handler Class* field in the template.

Task 7: Defining Service-Related Exceptions and Effects on System Reliability and Safety. We begin in a bottom-up way by examining each step in the *ProcessRegisteredVehicle* use case to determine how essential it's contribution is in order to achieve the goal. For example, step 1 involves the locator antenna notifying the system that a vehicle with a transponder is passing by. This is an input interaction, and its omission leads to an exceptional situation. An antenna defect would cause vehicles to be incorrectly identified as unregistered vehicles. Therefore, the step is annotated with a *reliability* tag together with the failure probability. Next, if the read/write antenna malfunctions, the registration information associated with the transponder would be inaccessible. Malfunctioning lights or cameras also hinder the success of the goal, so they are tagged as well. The exceptions that arise are defined as *LocatorAntennaFailure*, *RWAntennaFailure*, *LightFailure*, and *CameraFailure*.

In *TakeHighway*, the government computer might fail to send the requested information back to the system. The operator might fail to respond when a picture is sent to him. The transponder might not react to the acknowledgement signal sent by the read/write antenna. The service-related exceptions identified in this task that occur in the *TakeHighway* context are named as *GovernmentComputerUnavailable*, *Operator-*

Failure, and *TransponderUnreachable*. *Reliability* tags are attached to each of these steps. None of the steps is safety-critical.

We also need to consider the possibility of the handlers failing and the consequences of such failures. While handling the *HighwayUnavailable* exception, the barrier might fail to get activated resulting in a highly unsafe condition. Step 3 in Fig. 5 is therefore annotated with a *safety* tag and corresponding probability, and the extensions section is appended with a *BarrierFailure* exception.

4.3 Requirements Analysis

Task 8: Assessing Safety and Reliability. The probabilistic analysis of the system is not elaborated here for space reasons. The interested reader is referred to [16] for details. It reveals that system safety and reliability cannot be met with the current interaction: *BarrierFailure* has to be handled in order to improve safety, *GovernmentComputerUnavailable*, *OperatorFailure* and *TransponderUnreachable* have to be addressed in order to improve reliability.

4.4 Dependability-based Refinement and Iteration

We use the *TakeHighway* use case and the exceptions occurring in it to illustrate the tasks in this section.

Task 9: Specifying Detection Mechanisms. We first address the reliability issues in the *TakeHighway* use case. To begin with, detecting unavailability of the government records (exception *GovernmentComputerUnavailable*) can be done by using a timeout (the lack of reception of a message), as discussed in Section 3.4. The *OperatorFailure* exception can also be detected in a similar manner. To detect the exception *TransponderUnreachable*, we need to know whether the read/write antenna was able to reach the transponder. Hence, an additional acknowledgement step is needed. Detecting a failed entry is done when an exit is detected. Detecting a failed exit is done using a timeout. The updated use case is shown in Fig. 6.

To increase safety, we need to find a mechanism to detect the failure of the barrier. To this intent, we introduced an additional sensor which detects when a barrier is closed. A malfunctioning barrier can therefore be detected by the absence of the acknowledgement. The *ActivateBarrier* handler use case is updated with the detection and acknowledgement step (not shown for space reasons).

Task 10: Specifying Handler Use Cases.

In the case where an exit or entry of a vehicle is not detected, it is impossible to determine the length of the vehicle's trip. Therefore, the driver is billed for a minimal charge, shown in the *TakeHighway* use case in Fig. 6 with the degraded outcome *MinimalTrip* (steps 4a.1a and 4b.1a).

If the transponder is unreachable, the driver can not be notified of the success of the transaction. Therefore the use case ends in the degraded outcome *DriverNotNotified* (step 6a).

We know that the government computer is highly reliable and available, and therefore failures reaching the government computer are probably of temporary nature. There-

Use Case: TakeHighway
Level: User Goal
Primary Actor: Driver
Main Success Scenario:

1. Driver enters highway, passing through gantry.
2. Driver exits highway, passing through gantry.
3. System retrieves the driver's vehicle record based on trip information*.
4. System determines the amount owed based on the trip information and adds the transaction to the vehicle's records.
5. System informs Driver by sending a signal to the RWAntenna of successful completion of transaction. *reliability*
6. System receives confirmation from RWAntenna that the driver was notified.

Extensions:

- 3a. Vehicle is unregistered and does not have a record yet.
  - 3a.1. System sends license plate information to GovernmentComputer.
  - 3a.2. GovernmentComputer sends vehicle information and owner's address to System. *reliability*
    - 3a.2a. Exception{GovernmentComputerUnavailable}: use case ends in $\ll failure \gg$.
  - 3a.3. System creates a new vehicle record. Use case continues at step 4.
- 3b. Vehicle is unregistered and license plate is unrecognizable.
  - 3b.1. System displays pictures on OperatorTerminal.
  - 3b.2. OperatorTerminal sends license plate information to System. Use case continues at step 3. *reliability*
    - 3b.2a. Exception{OperatorFailure}: use case ends in $\ll failure \gg$.
- 4a. Exit unsuccessful.
  - 4a.1a. If entry was successful, minimum trip charge is added to vehicle's records. Use case ends in $\ll$ *degraded* $\gg$ *MinimalTrip*.
  - 4a.1b. If entry was unsuccessful as well, use case ends in $\ll failure \gg$.
- 4b. Entry unsuccessful.
  - 4b.1a. If exit was successful, minimum trip charge is added to vehicle's records. Use case continues in $\ll$ *degraded* $\gg$ *MinimalTrip* at step 4.
- 5a. Vehicle is not registered. Use case ends in $\ll$ *success* $\gg$.
- 6a. Exception{TransponderUnreachable}: use case ends in $\ll$ *degraded* $\gg$ *DriverNotNotified*.

Fig. 6. Updated *TakeHighway* Use Case

fore the service-related exception *GovernmentComputerUnavailable* is handled by resending the request. The handler defined for this task is shown in Fig. 7. The *OperatorFailure* exception can be handled in a similar manner, resending the request to the operator again or by trying another operator terminal.

In case of a malfunctioning transponder, the client is notified of the problem. He is given a grace period within which to service the transponder, and during which time he will not be charged a video toll charge. The handler defined for this purpose, *WarnClients*, is shown in Fig. 7. The handlers defined in this task are accordingly labelled as *reliability handlers*.

In case of the *BarrierFailure* exception, the system should immediately notify an operator. The operator can then evaluate the situation and, if necessary, call a service person and inform the patrol officers. This functionality is described in the safety handler use case *CallHighwayPatrol* (not shown here for space reasons).

Task 11: Defining Degraded Modes.

Handler Use Case: RetryGovtComp
Handler Class: Reliability
Context & Exception: TakeHighway{GovernmentComputerUnavailable}
Primary Actor: N/A
Secondary Actor:
Main Success Scenario:
1. System resends license plate information to the government computer.
   Step 1 is repeated 2 times.
2. Government computer sends vehicle information.

Handler Use Case: WarnClients
Handler Class: Reliability
Context & Exception: TakeHighway{TransponderUnreachable}
Primary Actor: N/A
Main Success Scenario:
1. System ascertains that the transponder is out of order.
2. System notifies operator that the transponder is not responding.
3. System flags transponder account as temporarily unavailable and cancels the video toll charge.
4. Operator issues a warning letter to the vehicle owner.
5. Owner brings transponder to the office for service.
6. Operator changes status of the account after transponder is serviced.

Extensions:
5a.1 System warns that grace period is over.
5a.2 System cancels discount of video toll charge.

Fig. 7. *RetryGovtComp* and *WarnClients*

In the 407 ETR system, even if the hardware of some entry or exit gantries are malfunctioning, it was decided that the highway should continue to operate (and charge minimal trips for vehicles that enter or exit through malfunctioning gantries).

Bad weather conditions might prevent a video camera from capturing clear pictures, or transmission problems might prevent the captured images from reaching the central computer. In this case, the detection and recognition services of a video surveillance system might temporarily be less reliable. In such a situation, the system would switch to a *degraded mode DegradedReliability*.

4.5 Requirements Summary

Task 12: Use Case Summary. Fig. 8 shows the use cases, exceptions and handlers related to *TakeHighway* by means of an extended use case diagram. All exceptional interactions are tagged with the <<handler>> stereotype along with the handler class *safety* or *reliability*, and all exceptional situations that trigger these interactions are documented using notes attached to the <<interrupt>> relationships. For space reasons, the secondary actors have been omitted from the diagram.

Task 13: Exception Summary Table. The exception table is very straightforward to create (as described in Section 3.5) and is presented in Table 1.

Task 14: Mode Summary Table. The mode summary table only contains three modes: the normal operation mode, the restricted mode *ExitOnly*, and the degraded mode *DegradedReliability*.

Fig. 8. Reliable and Safe 407 ETR Use Case Diagram

5 DREP and Model-Driven Engineering

The left hand side of Fig. 9 shows a summary of the tasks of our dependability-aware requirements engineering process.

The arrows illustrate that the developer is expected to go through several *iterations* of the process, refining the use cases and handlers if the analysis task reveals that the required system reliability or safety cannot be achieved within the current environment with the chosen interactions.

The iteration proceeds as follows. The analysis tool determines how severely each service-related exception affects system safety and service reliability. Among the service-related exceptions, the developer should start by addressing an exception that can be detected (with reasonable effort / costs), and for which a handling strategy that ensures safety or reliability can be envisioned. The exception to begin with might be one which requires immediate attention, or is safety-critical (in cases where safety is a priority). Then tasks 8 and 9 are executed, adding detection capabilities and handlers to the system. To complete this iteration, every interaction step of the newly defined handler of task 9 must again be elaborated (task 5.4), the outcomes must be specified (task 5.5), and the essential steps tagged with reliability probabilities and safety information (task 6.2)[6]. Finally, the developer can re-analyze the updated use cases (task 7) to determine if the required safety and reliability requirements can now be satisfied.

It should be noted here that the calculated dependability numbers do not represent the final system safety and reliability. They have to be interpreted as the *maximal dependability of the system if it were to be implemented without any flaws* (see section 5.2).

[6] If new exceptional goals have been discovered, the developer might even be required to go back to task 1 to brainstorm new secondary actors that are needed to achieve the new exceptional goals.

Table 1. 407 ETR System: Exception Table

Exception	Description	Context	Handler	Detection
Government Computer Unavailable	System unable to access vehicle record	TakeHighway	Retry Govt Comp	Timeout on government computer
Operator Terminal Failure	Operator unable to communicate with government computer	TakeHighway	Retry Operator	Timeout on operator
Transponder Unreachable	System cannot communicate with the transponder	TakeHighway	WarnClient	Lack of acknowledgement / Timeout
Detector Failure	System does not get info on incoming vehicles from the vehicle detector	Process Unregistered Vehicle	Use Redundant Detector	Locator antenna detects vehicles
Locator Antenna Failure	System receives no message from locator antenna; unable to identify transponders	ProcessRegistered Vehicle	Use RW Antenna	Timeout on antenna
RW Antenna Failure	System does not receive transponder account information from the RW antenna	ProcessRegistered Vehicle	SwitchTo Spare-Antenna	Timeout on RW antenna
Light Failure	Lights failed to turn on when taking images	ProcessRegistered Vehicle, Process UnregisteredVehicle	Call Service Person	Bad images
Camera Failure	System does not receive images from the camera	Process Registered Vehicle, Process Unregistered Vehicle	Call Service Person	Timeout on camera device
Laser Scanner Failure	System unable to classify due to lack of message from scanner	Classify Vehicle	SwitchTo SpareScanner	Timeout on scanner
Highway Unavailable	System attempts to block access to parts of the highway	TakeHighway	ActivateBarriers	Operator request
Barrier Failure	The barrier fails to get activated	ActivateBarriers	Call Highway Patrol	Timeout on barrier

Hence the calculated numbers should be *higher* than the required ones specified for each service in task 4.

The difference between the calculated and the required values determines how much effort has to be put into the design and implementation phases. If the difference is small, then stringent quality assurance, such as formal methods and proofs, extensive testing, or fault tolerance techniques, has to be employed by the implementors in order to assure that the internal flaws of the system are minimal. Refinement, i.e. defining new detectors and handlers, therefore has to continue until the calculated dependability numbers are sufficiently higher than the required ones.

5.1 Tool Support

In order to use our dependability-aware requirements engineering process efficiently, tool support is necessary. This is especially true for the probabilistic analysis of system reliability and safety. DREP relies heavily on the idea of model-driven engineering (as defined by OMG [20]), in which models of the system under development are built, and then incrementally modified and transformed as the development progresses from

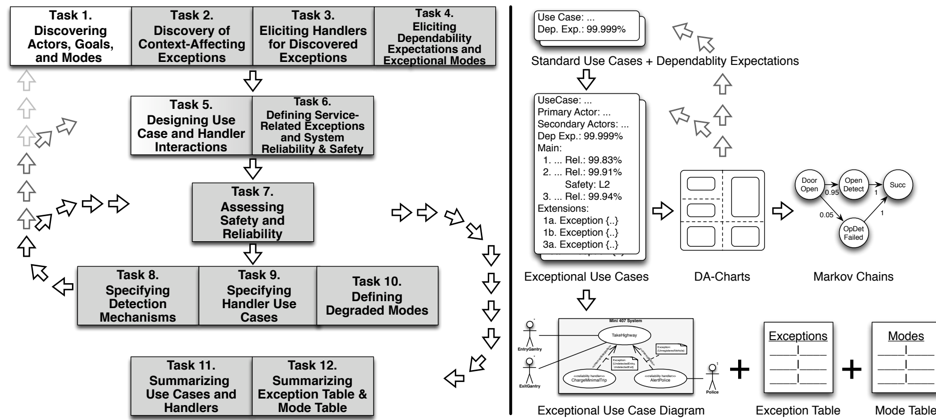

Fig. 9. Dependable Requirement Engineering Process Summary and Used Modelling Formalisms

requirements elicitation to analysis, design and implementation. At each phase, our process uses the modelling formalisms and notations that are most appropriate to express the concern at hand. The different modelling formalisms used in our requirements engineering process are shown in the right hand side of Fig. 9. The arrows depict the model transformations that occur when moving from one phase of the process to the next.

Currently, our tool [6] supports the creation of DA-Charts. The mapping to Markov chains and the dependability analysis is automated. We are working on the automated mapping of use cases to DA-Charts, and are even planning on providing automated support to map DA-Charts back to use cases. This would allow developers who are used to the DA-Chart formalism to apply reliability and safety increasing modification directly to the DA-Charts.

5.2 Discussion and Limitations

Our process helps the developer to discover potential exceptional situations that the system under development might be exposed to, and then guides the developer to investigate together with the stakeholder how the system should react in order to provide its services in the most reliable and safe way. Our approach is based on use cases that even non-technical people can read and understand, which makes getting feedback from all concerned stakeholders very easy.

As a result, however, our approach is also limited by the expressiveness of use cases. Use cases focus strictly on the interactions between the system and the environment. Hence, our dependability analysis only takes into account how the failures of actors and communication links affect the reliability and safety of the system under development. It does *not* consider failures internal to the system. The calculated dependability numbers represent the *best achievable safety and reliability of the system if it were implemented without any flaws*. If the numbers are too low, the developer should refine the interactions between the system and the actors, or even add new actors to the environment, to increase the achievable dependability of the system under development.

At this level of abstraction our approach cannot address internal flaws of the system. Use cases treat the system under development as a black box, and therefore no internal details are defined yet. Hence it is impossible to reason about conceptual system state, and even less to define invariants or pre- and postconditions on that state for system services.

To do this, a domain model describing conceptual system state must be created, and the use case models produced using our approach have to be mapped to operations that work with that system state. Popular development processes, e.g. the Unified Process [21], suggest to use graphical modelling formalisms such as activity diagrams or sequence diagrams for this purpose, together with OCL [22] constraints to express invariants, pre- and postconditions. If formal techniques are to be used to analyze system properties, a detailed system specification should be derived from our models using an appropriate formalism, e.g. B [23].

6 Validation

We conducted an empirical study in an academic environment using the 407 ETR case study to evaluate the applicability and effectiveness of our proposed process. We ran two separate experiments, both as part of an assignment in an undergraduate/graduate object-oriented software development course.

In the first experiment, a 3 hour lesson introduced a group of 20 students to use cases, after which they were asked to write use cases for the 407 ETR system using the standard use case template described in [15]. This first set of use cases is labelled *standard* in the following evaluation. Following the completion of this task, the students were presented with our exceptional use case notation as proposed in [19] in a one hour lecture. As a second part of the assignment, the students had to develop and extend their use cases by analyzing the cases for exceptional situations. They were required to document their results in an exception table, listing all exceptions discovered, their contexts, possible detection mechanisms, and handlers. This second set of use cases is labelled *EUC Notation* in the following evaluation.

The second experiment was carried out using the same case study, but with a different group of undergraduate/graduate software engineering students. In this case, the group was again introduced to use cases first (3 hours), but then presented with our dependability-driven requirements engineering process described in this paper (1 hour). Subsequently the students were asked to apply our task-based process to the 407 ETR system and elaborate use cases, handlers and a summarizing exception table following the guidelines outlined in Section 3.

As illustrated in the previous section, our specification of the 407 ETR included 10 exceptional scenarios. We went over the students submissions and analyzed the exceptions discovered by the members of each group. The results of the study are shown in Fig. 10 and 11.

Fig. 10 illustrates the measured improvements when using our proposed process as opposed to the standard use cases approach. In the first experiment using the standard use cases, about 67% of the students did not identify any exceptional situations. None of the students were able to identify context-affecting exceptions. Using DREP, students

of the second experiment were able to discover an additional 64.45% exceptions on average, and 96% discovered *HighwayUnavailable*. It is not surprising that none of the students using standard use cases discovered the *BarrierFailure* exception, since this exception is revealed only after revisiting and analyzing a handler use case.

Fig. 11 shows that students who applied our proposed process achieved even better results than those who just used the exceptional use case notation. Using DREP, the students of the second experiment were able to discover an additional 21.3% of the exceptions compared to the students in the first experiment using the exceptional use case notation only. It is to be noted that, using the exceptional use case notation, the designer is not lead to carry out the refinement and iteration tasks, and hence the results obtained by students of the first experiment were based on one iteration only. Iterations (as suggested in task 8 and 9) would probably have improved the use cases further.

There was one surprise in the experiment results: using our process, fewer number of students were able to discover the exceptional situation *TransponderUnreachable*. We believe that this is due to the fact that in our process exceptions are discovered by analyzing the use case interactions step by step. If the standard use case is incomplete to begin with, i.e. if the analyst forgets a normal interaction, this omission propagates further on and associated exceptions do not appear. A careful inspection of main success scenario of the *TakeHighway* use case of the students of the second experiment revealed that indeed many students had forgotten to include the acknowledgment step where the system informs the driver of the successful completion of the transaction.

The successful classroom study can be taken as a good indication that our process has potential. It would of course be useful to conduct a similar experiment in an industrial setting. Organizations are often willing to test new ideas in small, low risk projects. But in our case we propose a process to aid in the development of dependable systems, or in other words high risk projects, and it seems to be more difficult to convince companies to try new development techniques.

7 Related Work

We have carried out an extensive literature overview of specialized software development methods, domain-specific frameworks and general-purpose middleware that address dependability, timeliness, adaptability, or other QoS requirements [24]. To the best of our knowledge, mainstream development methods currently address such concerns only at the late design and implementation phases. However, several specialized approaches have been proposed that consider such issues at the early phases.

The frameworks TIRAN and DepAuDE [24] are two significant contributions to the development of dependable systems, but cater to a specific domain. The TARDIS project [24] provides a general framework that addresses various non-functional requirements, but does not define a step-by-step development process.

Some approaches have also been proposed that consider exceptions or non-functional requirements during requirements elicitation, and they are briefly discussed here.

De Lemos et al. [2] emphasize the separation of the treatment of requirements-related, design-related, and implementation-related exceptions during the software life-cycle by specifying the exceptions and their handlers in the context where faults are

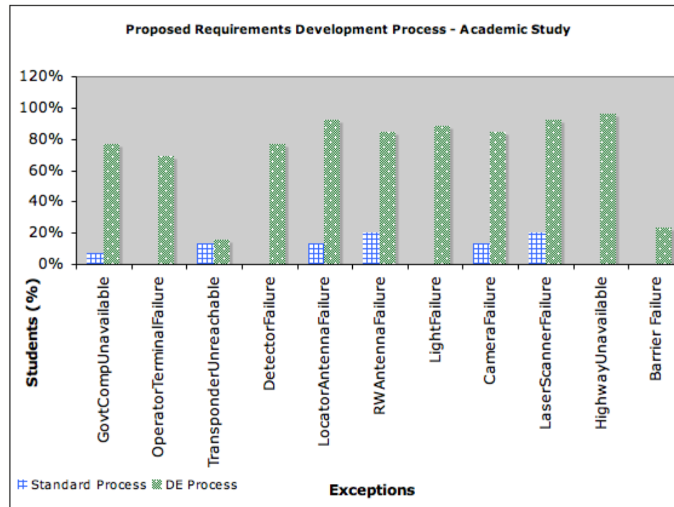

Fig. 10. Validation Results: Standard Use Cases versus RE Process

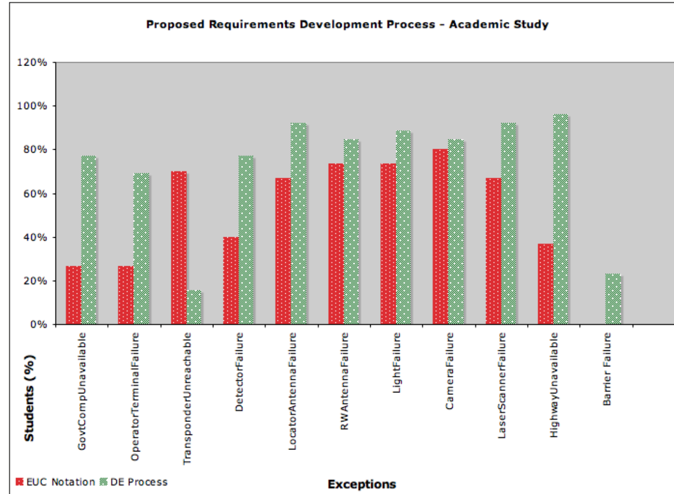

Fig. 11. Validation Results: Exceptional Use Cases versus RE Process

identified. The description of exceptional behavior is supported by a cooperative object-oriented approach that allows the representation of collaborative behavior between objects at different phases of the software development.

Alexander [25] proposes using *misuse cases* to document and analyze negative scenarios, for example scenarios that threaten the security or safety of the system. The paper describes concepts and modelling constructs along with tool support that can be used for this purpose. However, the support for eliciting and analyzing exceptional situations is quite minimal, and requires much imagination and experience. Sindre et al. [26] define *misuse cases*, and provide methodological support for eliciting security requirements. Ebnenasir et al. [27] propose an approach based on *misuse cases* for modelling of failsafe fault tolerance. The approach introduces faults in a model and defines *unsafe* and *at risk* use cases to allow analysis.

Lamsweerde [28] proposes the KAOS method, which is a goal-oriented approach for requirements modelling, specification, and analysis. It addresses quality-of-service issues, and present a high-level approach for specifying requirements and deriving the design based on refinements. Exceptional behaviour, defined as *obstacles*, is also addressed during requirements engineering [29]. To begin with goals are elaborated using goal graphs, from which the functional requirements are derived. Obstacles are generated from the goal specifications. The obstacles are then analyzed and refined if needed. Strategies for resolving the obstacles are then defined, and the goal structure is updated with the newly introduced goals. Goals and obstacles are expressed in a formal temporal language, and thus it requires time and expertise to develop correct and complete specifications. [29] briefly discusses informal obstacle identification but detailed guidelines are not provided, and informal or semi-formal techniques for resolution and elimination of obstacles are not offered.

Laibinis et al. [30] uses redundancy patterns as support for integrating fault tolerance into use cases. They suggest refining use case diagrams with recovery measures using the standard UML notation. They only focus on error recovery mechanisms, and do not discuss detection techniques or requirements elicitation methods that need to be used.

Rubira et al. [31] present an approach that incorporates exceptional behavior in component-based software development by extending the Catalysis method. The requirements phase of Catalysis is also based on use cases, and the extension augments them with exception handling ideas.

Whittle et al. [32] focuses on representing cross cutting concerns (including non-functional concerns) during requirements development. The issues of aspect modelling in scenario-based requirements elicitation are addressed.

Leveson [33] presents the hazard analysis approach which is used as part of the safety-life-cycle process. The risks are realized by considering different failures classes, and then discovering the failures in the context of the system under development based on experience and domain knowledge. The causes of hazards are identified and analyzed by using techniques such as fault trees. Each hazard is then assigned a criticality level and a probability to enable risk assessment. In comparison to hazard analysis, we believe that our approach leads to more complete specifications with respect to safety and reliability concerns.

Fault trees are also part of the safety-life-cycle process which comprises of several phases starting from specification of safety requirements, to design and implementation of safety concerns of critical systems. The initial phase, hazard analysis and risk assessment [33], has goals similar to our exceptional use cases method. Hazard analysis is carried out to identify the risks, to determine the causes, and then to assess and mitigate the risks. The analysis is based on fault trees. The risks are realized by considering different failures classes, and then discovering the failures in the context of the system under development. The hazards are then associated with a criticality level and with the likelihood of occurrence. Such a technique requires much experience and expertise on the developers part. In comparison, our use-case based approach is intuitive and provides a systematic process that allows developers to identify exceptional situations by analysing the set of interactions between the actors and the system.

Our approach is different from the above for several reasons. Firstly, we help the requirements engineers to elicit, specify, analyze, and refine dependability issues, exceptions and handlers with a well-defined *process* that they can follow. DREP focuses on reliability and safety concerns specifically, and guides analysts to develop a requirements specification document that exhaustively addresses dependability expectations of the stakeholders. Without a process, the only way a developer can discover exceptions and define recovery measures is based on only his imagination and experience. Secondly, our process increases dependability by helping the developers detect the need for adding "feedback" and "acknowledgement" interaction steps to counter communication problems. Additionally, the process recommends adding hardware to monitor request execution of secondary actors when necessary. Our handler use cases are stand-alone, clearly separate exceptional behavior from standard behavior, and can be associated with multiple exceptions and multiple contexts. DREP also gives support for automatic dependability analysis with tool support. The process is based on semi-formal constructs, and developers do not require expertise in formal specification languages to define or determine the quality of their requirements. In addition, communicating with end-users is simpler with use cases.

8 Conclusion

In most software systems today, it is crucial to guarantee that dependability requirements are successfully achieved. The discovery of all reliability and safety concerns is essential for the development of dependable systems. Exceptional situations are less common and the required behavior of the system in such situations is less obvious, and hence detailed user feedback on expected system behavior in such situations is very important. Also, users are more likely to make mistakes when exposed to exceptional situations, and therefore system interaction during handling of an exceptional situation is to be designed with great care. Early discovery of dependability concerns allow developers to discover and then document how the users of the system expect the system to react in every situation, which ultimately results in a more dependable system and saves considerable development costs. To this aim, we propose an approach that extends use case-based requirements elicitation, focussing on system reliability and safety.

Our task-based process begins with eliciting user goals and dependability expectations, and then discovering context-affecting exceptions. Recovery goals that dictate the system behaviour in such situations are defined in *handler use cases*. The process then goes on to give guidance on designing interactions to satisfy each of the discovered user and handler goals. The defined use cases are then examined step-by-step for reliability and safety related issues.The identified issues are labelled and documented as exceptions. As the next phase, the process suggests to analyze the use cases by using a probabilistic dependability analysis technique. The assessment results might show the need for increasing the dependability, and hence lead to further refinement. The refinement tasks require revisiting the use cases and integrating appropriate exception detection and recovery means. In the final task, the use cases, exceptions, and handlers are summarized in graphical and textual forms.

Based on our dependability focused use cases, a specification that considers all exceptional situations and user expectations can be elaborated during a subsequent analysis phase. This specification can then be used to decide on the need for employing fault masking and fault tolerance techniques when designing the software architecture and during detailed design of the system.

For future work, we intend to extend DREP to address other dependability constraints like availability and timeliness. We also plan to continue the development of our tool analysis tool to support DREP by providing a visual modelling environment for our dependability-focused use cases, and allow automatic mapping to analysis models for dependability assessment.

References

1. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM **18**(12) (December 1975) 683 – 696
2. de Lemos, R., Romanovsky, A.: Exception handling in the software lifecycle. IJCSSE **16**(2) (March 2001) 167 – 181
3. Shui, A., Mustafiz, S., Kienzle, J.: Exceptional Use Cases. In: 8th International Conference on Model Driven Engineering Languages and Systems - MoDELS 2005, Montego Bay, Jamaica, Oct. 2-7, 2005. Number 3713 in Lecture Notes in Computer Science, Montego Bay, Jamaica, Springer Verlag (October 2005) 568 – 583
4. Shui, A., Mustafiz, S., Kienzle, J.: Exception-Aware Requirements Elicitation with Use Cases. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advanced Topics in Exception Handling Techniques. Number 4119 in Lecture Notes in Computer Science, Springer Verlag (2006) 221 – 242
5. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-Driven Assessment of Use Cases for Dependable Systems. In: 9th International Conference on Model Driven Engineering Languages and Systems - MoDELS 2006, Genova, Italy, Oct. 1-6, 2006. Number 4199 in Lecture Notes in Computer Science, Springer Verlag (October 2006) 558 – 573
6. Zia, M., Mustafiz, S., Vangheluwe, H., Kienzle, J.: A Modelling and Simulation Based Process for Dependable Systems Design. Software and Systems Modeling (April 2007) 437 – 451
7. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2nd edn. Prentice Hall (2002)
8. Laprie, J.C., Avizienis, A., Kopetz, H., eds.: Dependability: Basic Concepts and Terminology. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1992)

9. Geffroy, J.C., Motet, G.: Design of Dependable Computing Systems. Kluwer Academic Publishers (2002)
10. Avizienis, A., Laprie, J., Randell, B.: Fundamental concepts of dependability (2001)
11. Knudsen, J.L.: Better exception-handling in block-structured systems. IEEE Software **4**(3) (May 1987) 40 – 49
12. Dony, C.: Exception handling and object-oriented programming: Towards a synthesis. In Meyrowitz, N., ed.: 4th ECOOP '90. Volume 25 of ACM SIGPLAN Notices., ACM Press (1990) 322 – 330
13. Cockburn, A.: Writing Effective Use Cases. Addison–Wesley (2000)
14. Mustafiz, S., Kienzle, J., Berlizev, A.: Addressing degraded service outcomes and exceptional modes of operation in behavioural models. In: Proceedings of the International Workshop on Software Engineering for Resilient Systems (SERENE 2008), ACM (2008)
15. Sendall, S., Strohmeier, A.: Uml-based fusion analysis. In: UML'99, Fort Collins, CO, USA, October 28-30, 1999. Volume 1723 of LNCS., Springer Verlag (1999) 278–291
16. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of use cases for dependable systems. In: MoDELS. Volume 4199 of LNCS., Springer (2006) 558–573
17. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: ETAPS, FASE. LNCS 2306, Springer (April 2002) 174 – 188
18. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of system dependability. Software and Systems Modeling (SoSym) (March 2007)
19. Shui, A., Mustafiz, S., Kienzle, J., Dony, C.: Exceptional use cases. In: MoDELS. Volume 3713 of LNCS., Springer (2005) 568–583
20. Mukerji, J., Miller, J.: Mda guide v1.0.1 (2003)
21. Jacobson, I., Rumbaugh, J., Booch, G.: The Unified Software Development Process. Object Technology Series. Addison–Wesley, Reading, Massachusetts, USA (1999)
22. Warmer, J., Kleppe, A.: The Object Constraint Language. 2nd edn. Object Technology Series. Addison–Wesley, Reading, MA, USA (2003)
23. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)
24. Mustafiz, S., Kienzle, J.: A survey of software development approaches addressing dependability. In Guelfi, N., Reggio, G., Romanovsky, A.B., eds.: FIDJI. Volume 3409 of Lecture Notes in Computer Science., Springer (2004) 78–90
25. Alexander, I.F.: Misuse cases: Use cases with hostile intent. IEEE Software **20**(1) (2003) 58–66
26. Sindre, G., Opdahl, A.L.: Eliciting security requirements with misuse cases. Requir. Eng. **10**(1) (2005) 34–44
27. Ebnenasir, A., Cheng, B.H.C., Konrad, S.: Use case-based modeling and analysis of failsafe fault-tolerance. In: RE, IEEE Computer Society (2006) 336–337
28. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: RE, IEEE Computer Society (2001) 249
29. van Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Software Eng **26**(10) (2000) 978–1005
30. Laibinis, L., Troubitsyna, E.: Fault tolerance in use-case modeling. In: Proceedings of RHAS 2005. (Sep 2005)
31. Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Fliho, F.C.: Exception handling in the development of dependable component-based systems. Software — Practice & Experience **35**(3) (December 2004) 195 – 236
32. Whittle, J., Araújo, J.: Scenario modelling with aspects. IEE Proceedings - Software **151**(4) (2004) 157–172
33. Leveson, N.G.: SAFEWARE: System Safety and Computers. Addison-Wesley Publishing Company (1995)