Exception Handling In Software Analysis

Muhammad Jamal Sheikh

Master of Science

Department of Computer Science

McGill University

Montréal, Québec

February 2008

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements of the degree of
Master of Science in Computer Science

ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this thesis possible.

First and foremost, i would like to thank my supervisor Jörg Kienzle, the director of the Software Engineering Lab at McGill University. He has been an inspiration for me for the last year. He has always been very encouraging and understanding. His guidance and teachings have helped me throughout.

I would also like to express my gratitude to Sadaf Mustafiz, Alexandre Denault and Riry Pheng for their contribution and suggestions.

I would like to thank my father, Muhammad Iqbal Sheikh and my sisters for their continuous support and love.

I also want to give special thanks to my wife Zoya Afzal whose love and care has helped me all the way.

ABSTRACT

With the advancement in technology, software systems are more and more in control of devices that we use in our daily lives. Complex computer systems are increasingly built for highly critical tasks. 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. 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 identified during requirements elicitation have to be systematically carried over to a subsequent analysis phase, and included in the system specification in order to ensure that the implementation of the system later on can deal with all relevant exceptional situations. This thesis advocates a more methodical approach to exception handling by extending the requirements elicitation and analysis phases of the Fondue development process to address exceptions. Exceptions are discovered at the requirements stage of software development using a use-case based approach, and then mapped to the Fondue specification models: the environment model, in which exceptional messages that signal exceptional situations to the system, the concept model, in which exceptional state is specified that keeps track of the failures of secondary actors, and the operation model, in which recovery functionality is specified. The proposed ideas are illustrated by a comprehensive case study, the 407 ETR Electronic Toll Collection system.

ABRÉGÉ

Avec les avancements d'aujourd'hui, les appareils que nous utilisons dans notre vie quotidienne sont de plus en plus dans contrôlés des systèmes informatiques. Les systèmes informatiques complexes prennent une nouvelle importance dans la réalisation des tâches hautement critiques. Un échec dans un tel système peut causer des pertes d'occasions d'affaires, des dommages physiques ou, dans le pire des cas, la perte de vies humaines. Les systèmes avec de telles responsabilités doivent être très fiables. La découverte et la documentation des situations potentiellement anormales, sans oubliér le comportement irrégulier d'utilisateur qui peut interrompre les interactions régulières du système, sont d'une grande importance dans le contexte de développement de systèmes fiables. Les exceptions qui sont identifiées pendant la phase de collecte des spécifications doivent être systématiquement utilisées à la phase d'analyse, et doivent être également intégrées dans les spécification systèmes pour assurer que l'implémentation du système peut réagir à toutes les situations exceptionnelles relatives. Cette thèse propose une approche plus méthodique pour la gestion des exception en prolongeant les phases de collection de spécifications et d'analyse du procéssus de développement Fondue pour y ajouter la gestion d'exceptions. Les exceptions sont découvertes dans la phase de spécification du cycle de développement en utilisant une approche cas d'utilisation "use case", pour ensuite les associer aux modèles de spécification Fondue : le modèle d'environnement où se trouve les messages exceptionnels qui sont envoyés au système dans le cas d'une situation exceptionnelle, le modèle conceptuel, où les états exceptionnels sont spécifiés un suivi des échecs des acteurs secondaires est effectué; et finalement le modèles opérationel, où les fonctionalités sont spécifiées. Les idées proposées sont illustrées par une étude de cas éllaboreé, le système 407 ETR Electronic Toll Collection.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 Motivation

Software engineering has been the backbone of commercial programming in all software development. Most successful development projects follow a rigorous development approach for structuring the activities of requirements elicitation, analysis and design. The ideal development approach depends on the project or product size and the application domain. However, most popular software development methods are composed of the five basic phases, i.e. requirements gathering, analysis, design, development and quality assurance.

Software systems are becoming more and more wide-spread. Some of them seamlessly become a part of human life. For example, software controls machines that we use on a daily basis, such as elevators or microwave ovens. The failure of these systems can cause critical damage in some cases.

Mission-critical systems or safety-critical systems need to be highly dependable. Dependability [LAK92] is that property of the system such that reliance can justifiably be placed on the service it delivers. Dependability involves satisfying several requirements like availability, reliability, safety, maintainability, confidentiality and integrity. In this thesis, we focus mainly on reliability and safety.

The reliability of a system measures its aptitude to provide service and remain operating as long as required [JG]. Reliability of a service is typically measured by the probability of success of the service or else by mean time to failure.

The safety of the system is determined by the lack of catastrophic failure it undergoes [JG]. The seriousness of the consequences of the failure on the environment can range from benign to catastrophic. Seriousness of a consequences can be measured by a safety index. According to DO-178B standard for civil aeronautics the safety index is defined as:

1. Without effects
2. Minor effects lead to upsetting the stakeholder or increase in system load.
3. Major effects lead to minor injuries of the users, or minor physical or monetary loss.
4. Dangerous effects lead to serious injuries of users, or serious physical or monetary loss.
5. Catastrophic effects lead to loss of human lives or destruction of the system. [RTC92]

Requirements elicitation, analysis, and design have been the focus of research to formulate new and better techniques and methodologies. Specifically, development has been the center of attention for more professional companies. However, many researchers have focused on the requirements, analysis and design, which is the foundation for any software.

Although requirements elicitation, analysis and design have been the focus of research for many years, current mainstream software development methods do not place enough emphasis on dependability. Safety and reliability are often considered secondary during software development, and only looked at too late in the software development life cycle. Any software, and especially mission-critical and safety-critical systems, require a software engineering approach that addresses safety and reliability issues from the beginning. Requirements elicitation should also consider desired system reliability and safety. If a subsequent analysis reveals undesired flaws or properties, the specification has to be refined [MK].

This thesis builds on the dependability-focussed requirements elicitation approach proposed in [MK]. It applies the proposed approach to a case study of considerable size, and proposes ways of carrying the dependability information from the requirements phase on to a subsequent analysis phase. The analysis models described in this thesis are a first step in mapping dependability-focussed requirements to a dependable software architecture and design.

1.2 Summary of Contributions

The main contributions of this thesis are the following:

- Application of the Exceptional Use Case Approach to a Case Study

  Standard use case models do not address exceptional situations in particular. The thesis applies the exceptional use case approach presented in [SMK06] and [MSKV06] to a case study, the 407 Electronic Toll Road Collection System (407 ETR). The approach suggests to identify exceptional situations within use cases, and handlers for exceptions in the use case model. This allows the system analysts to start thinking about exceptional situations already at the requirements phase, which is the start of the software life cycle. This thesis is the first effort to apply this approach to a case study of considerable size.

- Exceptions In Environment Model.

  The Fondue Environment model identifies all the inputs and outputs of a system, considering the system as a black box. The Fondue model does not pay particular attention to exceptional messages, nor correlate interactions with each other. In fact, it relies on other models to express each interaction, and the sequencing in which these interactions take place. This thesis extends the environment model, making it possible to model exceptional input and output messages. Furthermore, each of the exceptional interactions is linked to the original interaction which is interrupted. Exceptional messages are clearly identified, which makes it possible to apply high quality standards to that part of the system that handles reliability and safety concerns. The suggested approach is demonstrated by applying it to the 407 ETR case study.

- Exceptions in Concept Model.

  The Fondue Concept Model is a model that is utilized to model the conceptual state of the system. This model defines the system boundaries, describes the different concepts and explains the associations between concepts and between concepts and external actors. This thesis proposes to extend the concept model to also model exceptional state, i.e. state that is needed within the system to correctly detect and handle exceptional situations.

Exceptional state is clearly identified, which makes it possible to apply high quality standards to that part of the system that handles reliability and safety concerns. The suggested approach is demonstrated by applying it to the 407 ETR case study.

- Exception Handling in the Operation Model.

  The operation model specifies each system operation declaratively by defining its effects in terms of conceptual system state changes and messages output by the system by using the operation schemas. The schemas describe the initial state by the precondition and the change in the system state observed after the execution of the operation by the postcondition, both written in UML's OCL formalism [SS02]. The fondue operation model has been extended to include exceptions. These alterations ensure that the exceptional operations synchronize with the Exceptional Environment Model and hence caters all the exceptional input or output messages. Moreover, an important and desirable property of the exceptional part kept in mind is that it can also be easily distinguished and separated from the normal fondue operation model.

1.3 Thesis Road Map

The rest of the thesis is organized as follows:

Chapter 2 presents the some fundamental definitions and concepts for the understanding of the thesis.

Chapter 3 presents the requirements specification of the 407 ETR Case study.

Chapter 4 provides the complete use cases.

Chapter 5 provides the environment model extensions for exception handling.

Chapter 6 provides the new concept model with exceptions.

Chapter 7 provides the new operation model with exceptions.

Chapter 8 presents the related work.

Chapter 9 contains the conclusions of the thesis, as well as a future work section.

CHAPTER 2
Background

2.1 Requirements Elicitation, Analysis and Design

The requirements elicitation, analysis and design phases are the central part of software development activity. The requirement elicitation phase is composed of gathering and documenting the product requirements. The requirements are elicited by talking to all the stakeholders of the system.

During analysis, a complete and concise description of the system under development is elaborated. The specification is important for many reasons. Firstly, it defines unambiguously what functionality and qualities the system under development should have. Hence, the specification is the perfect documentation that developers can use when they have to elaborate a software design that fulfills the required functionality and quality. Secondly, a complete and concise specification allows to derive test cases against which the system, once implemented, can be tested.

During later steps, later design conforms to the stakeholder's thinking.

The analysis and design phases consists of elaborating the requirements in more technical documents. There are certain standards used for these phases that help the developer understand the needs of the system. This is necessary to form an indirect but reliable communication between the stakeholder and the developer.

2.2 Fusion and Fondue Model

The Fusion model is an analysis and design process for object-oriented software development. This process that was originally proposed by Derek Coleman of Hewlett Packard Labs in his book *Object Oriented Development - The Fusion Method* [CAB$^+$94] in 1994. The Fusion model is a systematic approach that combines the best practices from other objected oriented analysis and design methods.

The Fondue [SS99] process is based on the Fusion model. The Fondue process keeps the models and processes of Fusion, however, it uses the Unified Modeling Language (UML) as a notation. UML [OMG04] is the most commonly used design language and has become a standard for graphical models in software analysis and design. Fondue proposes the Use Case model for requirement elicitation. Moreover, it also defines a number of deliverables to lead the process of software development from initial requirements to implementation. The Environment model, Concept model and Protocol model are graphical models. Besides graphical notations, Fondue also prescribes the creation of Operation Schemas using the Object Constraint Language (OCL).

2.3 Exception Handling

Despite all the research conducted in the field of software engineering, there has been very little research regarding the handling of exceptions during the analysis process, and current software engineering practices often do not focus on the handling of exceptions as part of the development process. Current professional system analysts often only deal with exceptions at late design or development stage. Eventually, the developers handle the exceptions based on their individual preference. This causes problems like undefined standards, different exception handling techniques and in some cases major bugs in the produced software.

2.4 Exceptions

Before getting into more details regarding exception handling, we have to understand what exceptions are. Moreover, it is also important to understand the difference between exceptions and any alternate flow of events.

Exceptions are any events that may alter the state of the system or may threaten the user goal.

An exceptional situation or short exception describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. Hence, an exceptional occurence during a program execution is a situation in which the standard computation cannot pursue. For the program execution to continue, an atypical execution is necessary [Knu01].

2.5 Exception Handling System

A programming language or system with support for exception handling, subsequently called an exception handling system (EHS) [Don90], provides features and protocols that allow programmers to establish a communication between a piece of code which detects an exceptional situation while performing an operation (a signaller) and the entity or context that asked for this operation. An EHS allows users to signal exceptions and to define handlers. To signal an exception amounts to:

1. identify the exceptional situation,

2. to interrupt the usual processing sequence,

3. to look for a relevant handler and

4. to invoke it while passing it relevant information about the exception.

[SMK06]

2.6 Handlers

Handlers are defined on (or attached to, or associated with) 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 signalled during the execution or the use of the associated context or the nested context. To *handle* means to set the system back to the coherent state, and then:

1. to transfer control to the statement following the signalling one. (resumption model [Goo75])

2. to discard the context between the signalling statement and the one to which the handler is attached (termination model [Goo75])

3. to signal a new exception to the enclosing context. [SMK06]

In case of the occurence of an exceptional situation, the base behavior of the context is put on hold or abondoned and the interaction in the handler is initiated. A handler can temporarily take over the system interaction, perform some activity and switch the control back to the normal scenario. Such a handler is tagged by the ≪interrupt and continue≫ stereotype. However, some interactions, cannot be handled smoothly and cause the system to abort tagged by ≪interrupt and fail≫ stereotype. [SMK06]

2.7 Exception Types

Exceptions can be categorized as Safety or Reliability exceptions. The difference between the two is very clear from the name and has been further explained in section 1.1. The Safety exceptions threaten the safety of the system and thus the safety of the user in Safety critical systems. Whereas, the reliability exceptions threaten the reliability of the system and may cause problems like data errors or communication errors.

Although both categories are important for the proper working of the system and there is no particular way to handle one of them differently from the other. However, in some systems, human life may be at risk and hence safety exceptions are considered more important. So, safety exceptions are generally given a higher priority for solving critical issues.

CHAPTER 3
407 ETR Case Study Requirements

3.1 Problem Statement

The case study chosen for this analysis is rather simple but comprehensive. The problem presented has a fairly simple solution, however, it does cover most of the analysis and design issues specially regarding exception handling. This will be more evident when the the analysis models are discussed.

The 407 Express Toll Route 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 (ETC) system constructed by Raytheon. The ETR technology allows motorists to pass through toll routes without stopping or even opening a window.

To make this happen, each highway entry and exit point is equipped with a gantry (see Figure 3–1).

3.1.1 Processing Registered Vehicles

The most cost-efficient way to pay for highway use is to open an account with the 407 ETR system. Accounts can be personal or linked to a company. In either case, billing information (name and address) is saved with the account. Once an account is created, vehicles can be registered with it. Registered vehicles require a small electronic tag, called a transponder (see Figure 3–2), to be attached to the windshield behind the rear-view mirror. Transponders are leased for a

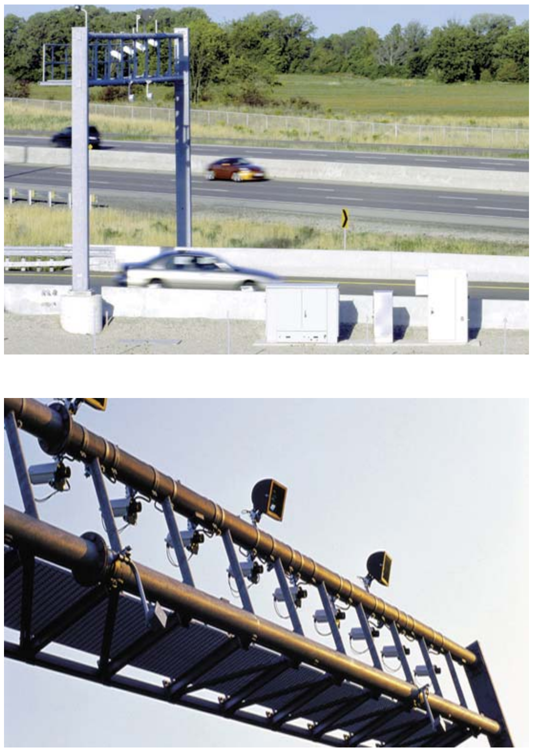


Figure 3–1: The 407 Entry/Exit Gantry

small monthly fee. The registration includes the vehicle details. The system automatically records the entry and exit of vehicles, and creates a transaction for each trip. This is done in the following way. When the vehicle enters the highway, it passes under the overhead gantry. The hardware devices of a gantry are shown in Figure 3–3.

The locator antennae determine if the vehicle is equipped with a transponder. Next, the read / write antennae read the account number from the transponder and the point of entry, time and date is recorded. In addition, as a vehicle passes under the gantries, the system uses laser scanners to determine the class of vehicle (e.g. light vehicle, heavy single unit vehicle, heavy multiple unit vehicle). It does this by measuring the height, width and depth of each approaching vehicle. A check is made to verify that the class of vehicle corresponds to the one registered for this particular transponder. 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. When the route is exited, the transponder gives a green signal followed by four short beeps to indicate a successful completion of the transaction.

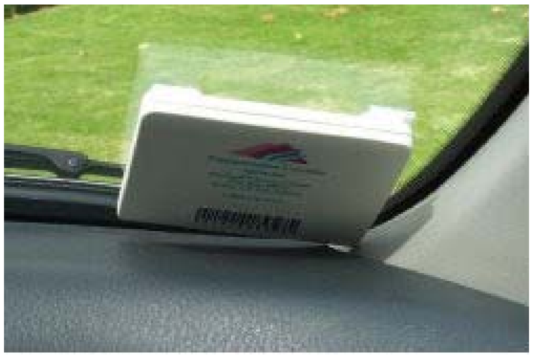


Figure 3–2: Transponder

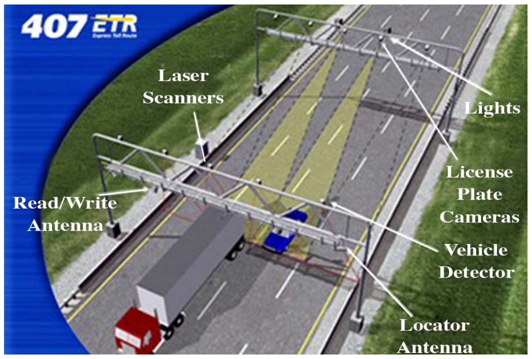


Figure 3–3: An Entry/Exit Gantry

3.1.2 Processing Unregistered Vehicles

Transponders are mandatory for heavy vehicles, i.e. vehicles with a gross weight of 5,000 kg. However, light vehicles can use the 407 ETR without registering. When a motorist without a transponder enters the highway and passes under the two tolling gantries, the system triggers a set of digital cameras to take pictures of the rear number plate of the vehicle from different angles. In order to get good images, a set of lights are turned on before the images are taken. The lights automatically adjust their intensity to ensure the best conditions for taking an image of the number plate. At the same time, the laser scanners are activated to classify the vehicle and tell the toll collection system whether to charge for a passenger or commercial vehicle. The owner of the vehicle is identified by electronic access to government records. The cameras and lights have been tested to ensure accuracy, even in blizzards and rainstorms. However, 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.

3.1.3 Payment

Registered car owners, registered companies, and motorist that used the highway, receive an invoice in the mail at the end of the month containing the trips of all of the vehicles registered with their account. The price of each trip is calculated based on the time of day, distance traveled and type of vehicle. If the entry or exit time falls within peak hours (6am - 10am, 3pm - 7pm), the toll rate is 16.25 cents/km for light vehicles, 32.5 cents/km for heavy vehicles, and 48.75 cents/km for heavy multiple unit vehicles. Otherwise, the toll rate is 15.5

cents/km for light vehicles, 31 cents/km for heavy vehicles and 46.5 cents/km for heavy multiple unit vehicles. If a light vehicle uses the highway without a transponder, an additional video toll charge of \$3.50 is applied per trip. Cheating motorists (for instance, motorists using a transponder with an unregistered vehicle, or heavy vehicles taking the highway without a transponder) are fined with \$50. Refusal to pay invoices for 3 months results in plate denial, meaning that a debtor cannot renew the license plate of his cars or obtain a new license plate from the government until all tolls and fees have been paid.

3.1.4 Hardware / Software Decisions

The software to be developed has to interface with the hardware devices of the gantry. The development of the software running on the transponders is assumed to be outsourced to a different company, and hence does not have to be considered for this project.

CHAPTER 4
407 Exceptional Use Cases

4.1 Use Case Model

The first step of the Fondue process consists in building the Use Case model. The use cases are based on the initial requirements elicitation phase and are supposed to be easily understandable by technical and non-technical stakeholders.

Use cases are a widely used formalism for discovering and recording behavorial requirements of software systems [Lar04]. A use case is a description of the possible sequence of interactions between the system under discussion and its external actors, related to a particular goal [Coc00].

A use case describes, without revealing the details of the internal working of the system, the system's responsibilites and its interactions with its environment as it serves requests that, if successfully completed, satisfy a goal of a particular stakeholder. The external entities in the environment that the system interacts with are called *actors.* The actors that need to achieve a particular goal are called the primary actors and entities that the system needs to fulfill the goals are called secondary actors. Alternatives or situations in which the goal is not achieved are usually described as extensions to the main use case scenario.

The use cases are basically text based, however, they can be scaled up or down in terms of formality and sophistication. Hence they can be very effectively used as a communication means between technical and non-technical stakeholders of the software [MSKV06].

The use cases can be divided according to their level in an inheritance tree. The top level are the summary level use cases that provide an overview of the whole system.

4.2 The 407 ETR Use Case Model

4.2.1 UseHighway Summary-Level Use Case

The Fondue version of the UseHighway use case is presented in Figure 4–1 to give an example of a summary level use case. It makes it easier to understand the basic interaction between users of the highway and the system.

Use Case: UseHighway
Scope: 407 ETR System
Level: Summary
Intention in Context: The intention of the Driver is to use the 407 ETR highway on a regular basis.
Multiplicity: One Driver can only drive one vehicle at a time on the highway. However, different Drivers can use the highway simultaneously.
Primary Actor: Driver
Secondary Actor: GovernmentComputer
Main Success Scenario:
1. Driver registers vehicle.
Steps 2-4 are repeated once a month as long as the vehicle is registered.
2. Driver takes highway.
Step 2 can be repeated any number of times per month.
3. At the end of the month, System generates a monthly bill and sends it to Driver.
4. Driver pays bill.
5. Driver cancels registration.
Extensions:
1a. Driver uses highway without registering vehicle. Use case continues at step 2.
4a. Driver does not pay bill for 3 consecutive months.
4a.1. System informs GovernmentComputer of refusal to pay the bill. Use case continues at step 2.

Figure 4–1: The UseHighway Use Case

4.2.2 407 User-Goal Use Case

Each user goal use case describes the provision of one service, or one functionality to a particular primary actor. Finally, sub-function level use cases are ”helper” use cases that can be useful to decompose complex user goal use cases.

In the use case model, the normal flow of interaction events between the environment and the system is separated from any irregular interactions, which are described in the *extensions* section. However, in standard use cases, the definition of what is to be considered an extension can vary: an extension can describe a minor change in the ideal flow of events, e.g. an alternative way of providing a requested service, but can also describe how the system should react in an exceptional situation where the safety of a user is in danger.

According to [dLR01], to handle exceptional situations properly in the analysis and design phase, exceptions have to be taken into account from the start of the software engineering process. The exceptions are also a form of irregular interraction and hence are placed in the ”extensions” section in the use case model. However, exceptions are different from extensions. The system is prone to unhandled exceptions and thus exceptions have to be distinguished from other extensions. Hence, to identify these exceptions easily and separate them from the regular flow of events, they have been tagged using the {Exception} tag.

Some of the exceptions have to be handled outside the software process. However, others require certain steps or necessary procedures. Yet some more may be a combination of the two. For exceptions that require any certain procedure, it is suggested to use handler use cases. These handler use cases are invoked whenever an exception tag is encountered in the use cases.

The sub-function level use cases are explicitly included in the user goal level use cases and are similar to a synchronous method call. However, unlike this relationship, the handlers are not explicitly called from the higher level use case. The handler use cases may effect the higher level use case either altering their further flow or aborting them in some cases. The handler use cases may or may not be synchronous to the calling use cases. In some cases, the calling use case waits for the handler for the exception handling. The handler might do the failed process in an alternate manner. In others, once invoked, they do not wait for the handlers to finish before proceeding with their normal flow. The raising of the exception might cause the use case to fail, however, this use case may be included in higher level use cases that may continue and completely or partially fulfill the user goal.The handler use case may continue in parallel to solve any abnormal issues.

The Exceptions and hence the handlers related to exceptions are also categorized as safety or reliability handlers. The case study presented mostly deals with reliability handlers only. However, this cannot be considered as a major problem as both exceptions are handled in the same way. The only difference is that the safety exceptions are sometimes given a higher priority due to their crucial nature in safety critical systems.

The RegisterVehicle is the basic use case for opening any account and verifying the information before the account opening. The system communicates with the Government Computer for the verification. There may be a communication error between the two. This error may be due to communication failure or the Government computer server may be down. The reason for the failure has no effect on the process or the execution of the handler. This is a ≪ interrupt and fail≫ exception and is shown in the RegisterVehicle Use Case extension 4a, in Figure 4–2. In extension 6a, the transponder failure is a ≪interrupt and continue≫ exception as the control is transferred back to the normal scenario. There is no particular handler use case required in these cases as these are single step processes.

The TakeHighway use case described in Figure 4–3 is the core of the system. It is executed each time a driver takes the highway. For the purpose of the study, it handles all major exceptional scenarios. These scenarios include communication error of the system with the Government computer and the Operator Terminal. These are both ≪interrupt and fail≫ exceptions as the system cannot progress with the proper communication in these cases.

LicensePlateRecognitionFailure raised in extension 3b is handled on the sub-extensions 3b.1 and 3b.2. If there is any hardware failure at the entry or exit gantry that causes a system failure, the Undetected Entrance or Undetected Exit exceptions are raised to handle such cases. These are the most typical examples of ≪interrupt and continue≫ exceptions where a handler use case is required (Figure 4–4 and Figure 4–5). Moreover, most of the sub-function level use cases are included in the TakeHighway use case.

The No Entrance Trip and No Exit Trip (Figure 4–4 and Figure 4–5) are perfect examples of handler use cases. These are synchronous handlers and are completely executed before switching the control back to the normal execution flow. These are major exceptions that can be handled easily providing a solution that guarantees at least partial success. The amounts will not be correct thus threatening the reliability of the system. However, considering that these exceptions are rare, the handlers ensure that the amount calculated by these handlers are minimum for the driver. Although, this does cause a loss to the Highway authorities but it can be catered for after identifying the reason for the exception.

The PassThroughGantry use case in Figure 4–6 makes sure that the record of the vehicle at both entry and exit points is stored properly. The PassThroughGantry and its subfunction level use cases handle all hardware failure or communication failures at each of the gantry and raises a corresponding exception for any hardware device or gantry problem. The exception DetectorFailure is a ≪interrupt and continue≫ exceptions which raises a flag for hardware failure, although the system keeps on working after the handling.

ProcessRegisteredVehicle (Figure 4–7), ProcessUnregisteredVehicle (Figure 4–8) and ClassifyVehicle (Figure 4–9) are using the hardware devices separately. Hence failure at any point can be used to clearly indicate the exception point and hence the reason for the failure. A very important point to note is that most hardware failures are ≪interrupt and fail≫ exceptions, the sub-function level use case fails as the use case was based on the hardware device. However, the enclosing use case, takes the failure of the use case as an ≪interrupt and continue≫ exception, handles the hardware failure and finds an alternative to keep the system running. This ensures at least partial success.

The RepairHardware use case in Figure 4–10 is handling all the hardware failures that we just mentioned. This is a really interesting example of asynchronous handlers. This means that whenever any hardware failure exception is encountered, this handler is invoked and instead of waiting for the handler the context use case continues its working. The context use case might fail, however, the higher level use case still continues and can provide partial or complete success. The RepairHardware alerts the repair team and it is executed in parallel till the problem is solved. It is assumed that backup hardware devices are available at each gantry. Moreover, after the repair, the system is notified to finish the handler.

The MonthlyBill use case in Figure 4–11 is a simple use case that is executed in the system, the only communication with external actors being the printer. It is a resource intensive use case in terms of accessing the database or bill calculation, however, it has no external dependencies. Hence, even in case of failure, it can be executed again to get the desired results.

The payment use cases (Figure 4–12 and Figure 4–13) are responsible for taking care of all sorts of payments. There may be exceptions like CreditCardCompanyUnavailable, InsufficientCredit or BouncedCheck, however, these can be handled easily. All these exceptions are handled by notifying the customer either through the Operator or by mail. Once again, the payment failure cannot be handled by the system and it is considered the customer's responsibility to go through the process again.

The CancelRegistration use case in Figure 4–14 again a basic user goal level use case. It has no impact on the trip itself and has no safety or reliability issues from an exceptions point of view.

The important point to note in all these use cases is the difference of the exceptional use case model with the normal use case model. This exceptional use case model was made based on the ten step process provided in [SMK06]. The process ensures the minimum probability of missing any exception due to its extensive nature. Moreover, due to the iterative analysis process, this process is considered quite reliable for handling exceptional use cases.

Some other important points to note about the particular case study is that there are no considerable safety exceptions. The only exceptions are reliability exceptions that do threaten the correctness of the system process. However, with the early study of these exceptions, it can be easily made sure that the reliability exceptions are handled so as to ensure at least partial success of the user goal.

4.2.3 RegisterVehicle Use Case

Use Case: RegisterVehicle
Scope: 407 ETR System
Level: User Goal
Intention in Context: The goal of the Driver is to register a vehicle with the system, which involves opening an account and linking a transponder to it.
Multiplicity: A driver registers his vehicles one at a time. However, the system should be able to handle multiple simultaneous registrations done by different drivers.
Primary Actor: Driver
Secondary Actor: OperatorTerminal, GovernmentComputer, PostalService
Main Success Scenario:
The Driver interacts with the System by calling an Operator.
1. Driver provides System with personal data and vehicle information.
2. System acknowledges opening of a new account for the Driver.
3. System sends vehicle information to GovernmentComputer for verification.
4. GovernmentComputer notifies System that vehicle information is correct.
5. System assigns a new transponder to the vehicle, and informs Postal Service to deliver the transponder to the Driver.
6. Driver installs and tests transponder.
7. Driver notifies the System of successful installation of the transponder.

Extensions:
2a. Driver already has an account with the system. Use case continues at step 3.
4a. Exception{GovernmentComputerUnavailable}. Use case ends in failure.
6a. Transponder installation and testing fails. Exception{TransponderInstallationFailure}. Driver notifies System of the problem. Use case continues at step 5.
7a. Driver forgets to acknowledge installation and simply starts using the transponder on the highway. Use case ends in success.

Figure 4–2: The RegisterVehicle Use Case

4.2.4 TakeHighway Use Case

Use Case: TakeHighway
Scope: 407 ETR System
Level: User Goal
Intention in Context: The intention of the Driver is to drive a vehicle from one location to another by taking the 407 ETR highway.
Multiplicity: One Driver can only drive one vehicle at a time on the highway. However, different Drivers can take the highway simultaneously.
Primary Actor: Driver
Secondary Actor: RWAntenna, GovernmentComputer, OperatorTerminal
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 verifies vehicle classification and adherence to the highway rules.
5. System determines the amount owed based on the trip information and adds the transaction to the vehicle's records.
6. 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, vehicle classification and owner's address to System.
3a.2a. Exception{GovernmentComputerUnavailable}: use case ends in failure.
3a.3. System creates a new vehicle record. Use case continues at step 4.
3b. Vehicle is unregistered and licence plate is unrecognizeable by the system. Exception{LicensePlateRecongnitionFailure}.
3b.1. System displays pictures on OperatorTerminal.
3b.2. OperatorTerminal sends licence plate information to System. Use case continues at step 3.
3b.2a. Exception{OperatorTerminalFailure}: use case ends in failure.
5a. Exit unsuccessful.
5a.1a. If entry was successful, Exception {Undetected Exit}.
5a.1b. If entry was unsuccessful as well, use case ends in failure.
5b. Entry unsuccessful.
5b.1a. If exit was successful, Exception {Undetected Entrance}.
5c. The classification of unregistered vehicle does not match the Government Records. System informs the Government Computer. Use case continues at step 5.
6a. Vehicle is not registered. Use case ends in success.
6b. Exception{RWAntennaFailure}: use case ends in success.
*__Trip Information Details__: A complete trip information record includes entry and exit time and place, measured and obtained vehicle classification, transponder account or licence plate information or licence plate images.

Figure 4–3: The TakeHighway Use Case

4.2.5 NoEntranceTrip Use Case

Handler Use Case: NoEntranceTrip
Handler Class: Reliability
Context & Exception: TakeHighway {Undetected Entrance}
Primary Actor: N/A
Intention: System wants to calculate the amount owed by the Driver based on the exit point.
Level: Sub-function
Main Success Scenario:
1. The System calculates the amount by using the nearest entry point as the vehicle's entry.
2. The System records marks the trip as a no entrance trip.

Figure 4–4: The NoEntranceTrip Use Case

4.2.6 NoExitTrip Use Case

Handler Use Case: NoExitTrip
Handler Class: Reliability
Context & Exception: TakeHighway {Undetected Exit}
Primary Actor: N/A
Intention: System wants to calculate the amount owed by the Driver based on the entry point.
Level: Sub-function
Main Success Scenario:
1. The System times out waiting for an entered car to exit.
2. The System sets the exit point according to the nearest exit from the point of entry.
3. The System records marks the trip as a no exit trip.

Figure 4–5: The NoExitTrip Use Case

4.2.7 PassThroughGantry Use Case

Use Case: PassThroughGantry
Scope: 407 ETR System
Level: Sub-Function
Intention in Context: The Driver passes through a entry or exit gantry as part of his trip.
Multiplicity: One Driver can only drive one vehicle at a time through a gantry. However, different Drivers can pass through the same or different gantries simultaneously.
Primary Actor: Driver
Secondary Actor: VehicleDetector
Main Success Scenario:
1. VehicleDetector informs System that vehicle is approaching entry gantry.
Steps 2 and 3 are performed in any order or in parallel.
2. System processes registered vehicle or processes unregistered vehicle.
3. System classifies vehicle.
4. System records entry time and vehicle information for the trip.
Extensions:
1a. Exception{DetectorFailure}
1a.1. System processes registered vehicle. Use case continues at step 3.
1a.1a. Processing of registered vehicle was unsuccessful. Use case ends in failure.
3a. Processing of registered vehicle was unsuccessful. System processes Unregistered Vehicle.
3a.1. Use case continues at step 3.
3a.1a. Processing of unregistered vehicle was also unsuccessful. Use case ends in failure.
4a. Classification was unsuccessful. Use case ends in success.

Figure 4–6: The PassThroughGantry Use Case

4.2.8 ProcessRegisteredVehicle Use Case

Use Case: ProcessRegisteredVehicle
Scope: 407 ETR System
Level: Sub-Function
Intention in Context: The System communicates with the transponder to identify the approaching vehicle.
Multiplicity: The System must be able to process multiple registered vehicles simultaneously.
Primary Actor: N/A
Secondary Actor: LocatorAntenna, R/WAntenna
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.

Extensions:
1a. The approaching vehicle does not have a transponder. Use case ends in failure.
1b. Exception{LocatorAntennaFailure}: use case ends in failure.
(2-3)a. Exception{RWAntennaFailure}: use case ends in failure.
3a. R/WAntenna is unable to obtain account information. Use case ends in failure.

Figure 4–7: The ProcessRegisteredVehicle Use Case

4.2.9 ProcessUnregisteredVehicle Use Case

Use Case: ProcessUnregisteredVehicle
Scope: 407 ETR System
Level: Sub-Function
Intention in Context: The System wants to identify the approaching vehicle using the license plate information.
Multiplicity: The System must be able to process multiple unregistered vehicles simultaneously.
Primary Actor: N/A
Secondary Actor: Cameras, Lights
Main Success Scenario:
1. System turns on the Lights.
2. System triggers the Cameras.
3. Cameras send images to System.

Extensions:
2a. Exception{LightFailure} Use case continues at step 2.
3a. Exception{CameraFailure} Use case ends in failure.

Figure 4–8: The ProcessUnregisteredVehicle Use Case

4.2.10 ClassifyVehicle Use Case

Use Case: ClassifyVehicle
Scope: 407 ETR System
Level: Sub-Function
Intention in Context: The System wants to classify the approaching vehicle as light vehicle, heavy single unit vehicle, or heavy multiple unit vehicle.
Multiplicity: The System must be able to classify multiple vehicles simultaneously.
Primary Actor: N/A
Secondary Actor: LaserScanner
Main Success Scenario:
1. System activates LaserScanner.
2. LaserScanner sends vehicle dimensions to System.
3. System classifies vehicle and records classification in trip information.

Extensions:
2a. Exception{LaserScannerFailure} System records classification failure in trip information. Use case ends in failure.

Figure 4–9: The ClassifyVehicle Use Case

4.2.11 Repair Hardware Use Case

Handler Use Case: Repair Hardware
Handler Class: Reliability
Context & Exception: PassThroughtGantry Exception{DetectorFailure}, ProcessRegisteredVehicles Exception{LocatorAntennaFailure, RWAntennaFailure}, ProcessUnRegisteredVehicles Exception{LightFailure, CameraFailure}, ClassifyVehicle Exception{LaserScannerFailure}
Primary Actor: Repair Team
Intention: System wants to use backup hardware and repair the original one.
Level: Sub-function
Main Success Scenario:
1. The System uses the backup hardware failure.
2. The System sends a message to the Repair Team about the hardware failure.
3. The Repair Team informs the System when the hardware is repaired.
4. The System switches back to the original hardware.

Figure 4–10: The RepairHardware Use Case

4.2.12 Monthly Bill Use Case

Use Case: MonthlyBill
Scope: 407 ETR System
Level: User Goal
Intention in Context: The goal of the system is to generate and send monthly bills to the drivers.
Multiplicity: Bills is calculated for all the drivers that had a trip within the month. One bill is calculated at a time.
Primary Actor: N/A
Secondary Actor: OperatorTerminal, Printer
Main Success Scenario:
1. The system calculates the bill for each transponder in the record.
AND
The system calculates the bill for each driver who travelled through the highway in an unregistered vehicle.
2. A copy of each bill is printed to be sent to the driver.
Extensions:
2a. Exception{PrinterUnavailable}: System displays an error for the operator to print the bill later. Use case ends in success.

Figure 4–11: The MonthlyBill Use Case

4.2.13 PayByCreditCard Use Case

Use Case: PayByCreditCard
Scope: 407 ETR System
Level: User Goal
Intention in Context: The goal of the Driver is pay for his trip by credit card.
Multiplicity: Every driver pays for his trips once a month. The system must support concurrent payments of different drivers, be it by credit card or by cheque.
Primary Actor: Driver
Secondary Actor: OperatorTerminal, CreditCardCompany
Main Success Scenario:
Driver interacts with System by calling an Operator.
1. Operator provides System with credit card information.
2. System contacts CreditCardCompany to validate credit.
3. CreditCardCompany notifies System of successful validation.
4. System notifies Operator of success.

Extensions:
2a. Exception{CreditCardCompanyUnavailable}: System notifies Operator. Use case ends in failure.
3a. Exception{InsufficientCredit}: System notifies Operator. Use case ends in failure.

Figure 4–12: The PayByCreditCard Use Case

4.2.14 PayByCheck Use Case

Use Case: PayByCheck
Scope: 407 ETR System
Level: User Goal
Intention in Context: The goal of the Driver is pay for his trip by check.
Primary Actor: Driver
Secondary Actor: OperatorTerminal
Main Success Scenario:
Driver sends check to Operator.
1. Operator notifies System that check has been received.

Operator cashes check with Bank.
2. Bank notifies System that check has been cleared.

Extensions:
2a. Exception{BouncedCheck}: System notifies Driver. Use case ends in failure.

Figure 4–13: The PayByCheck Use Case

4.2.15 CancelRegistration Use Case

Use Case: CancelRegistration
Scope: 407 ETR System
Level: User Goal
Intention in Context: The goal of the Driver is to unregister a vehicle and potentially cancel his account with the 407 ETR system.
Multiplicity: A driver unregisters a vehicle one at a time. The system should be able to handle multiple concurrent unregistrations of different drivers.
Primary Actor: Driver
Secondary Actor: OperatorTerminal
Main Success Scenario:
Driver interacts with System by calling an Operator.
1. Operator notifies System that Driver wants to cancel his vehicle registration.
2. System marks vehicle registration as suspended and does not charge monthly fees anymore.
Driver sends transponder to Operator.
3. Operator notifies System that transponder has been received.
4. System cancels vehicle registration.
5. If Driver has no vehicles registered with the system, System cancels driver account.

Extensions:
3a. Exception{TransponderUsed}: System reactivates registration. Use case ends in failure.

Figure 4–14: The CancelRegistration Use Case

4.3 Use Case Diagram

Whereas individual use cases are text based, the UML use case diagram provides a concise high level view of all the use cases of the system. It allows the developer to graphically depict the use cases, the actors that interact with the system, and the relations between actors and use cases.

The use case model for the 407 ETR use cases has been provided in Figure 4–15.

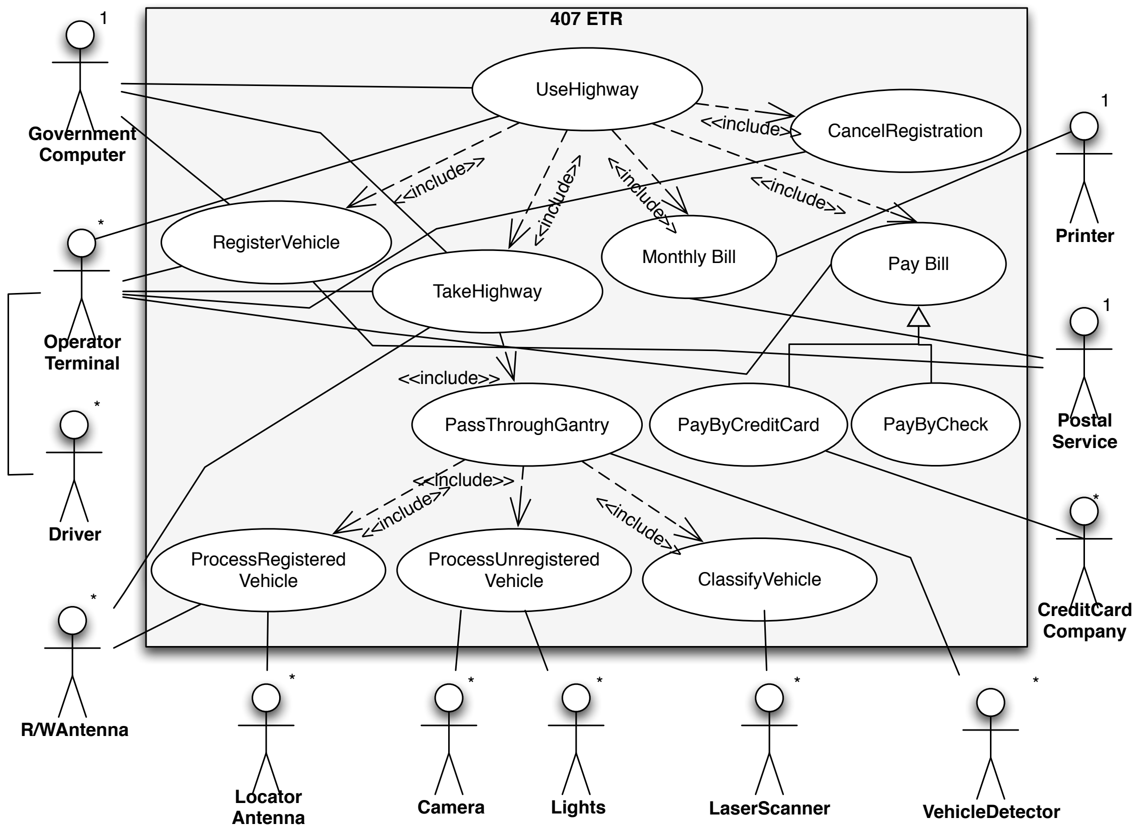


Figure 4–15: The 407 Use Case Model

4.4 Extended Use Case Diagram

The use case diagram has been extended to handle exception according to the model proposed in [SMK06]. In a standard use case diagram, use cases are shown as ellipses associated to the actors whose goals they describe. It has been proposed to identify handler use cases with a ≪handler≫ stereotype, which differentiates the handler use cases from normal use cases. A handler that is attached to a context use case is shown by a directed relationship (dotted arrow) in the diagram. This arrow specifies that the behavior of the context use case may be affected by the handler use case behavior in case an exception is encoutered [SMK06].

The exceptional use case model as designed according to the [SMK06] is shown in the Figure 4–16.

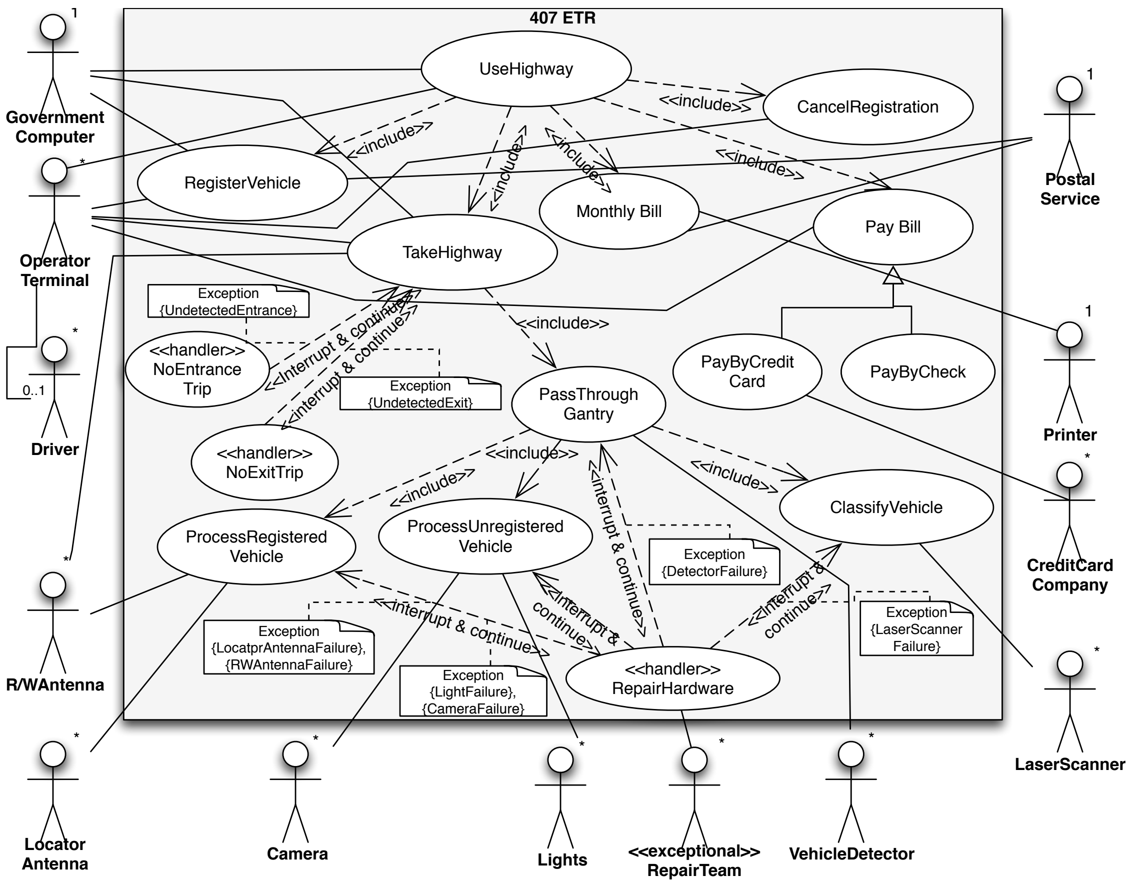


Figure 4–16: The 407 Exceptional Use Case Model

CHAPTER 5
Exceptional Environment Model

5.1 Fondue Environment Model

The Fondue development process is targeted at the development of reactive systems. In reactive systems, the system only executes some functionality when it is stimulated by a trigger. This trigger can be a user input or any other kind of message sent to the system.

The analysis phase of software development in the Fondue process prescribes the creation of 3 models of the system under development: the Environment model, the Concept model and the Operation model. This chapter focusses on the environment model.

Since it is the interaction based view, a very crucial part of the environment model is the communicating party in other words the actor. The system may be contacted by different actors which may be users or automated machines programmed to communicate with the system. Similarly, the system provides responses to different or similar actors.

The Fondue environment model is a UML 2.0 communication diagram. It focusses on the interactions and communication between the system and the actors in it's environment, which may be users or automated machines programmed to communicate with the system, or other existing software systems that the system under development has to interact with.

This model is based on interactions and not processes, so each of the interactions is considered separate. Hence, all the interactions are shown by an asynchronous message. Though, these interactions may be linked in other parts of analysis and design.

In the environment model, the system is typically shown as a black box. No details are given on how the system provides, executes or implements any particular functionality. In fact, the system is considered one entity which provides external actors some functionality when triggered. The emphasis in the environment is on each of the interaction steps that are needed for the system to provide the desired functionality to the actors. All interactions are shown using asynchronous messages.

The Fondue environment model for our ETR 407 case study is shown in Figure 5–1(version 1.0)

5.2 Environment Model With Exceptions

The standard Fondue environment model does not highlight interactions with the system that are due to exceptional situations. In particular, it does not address (or distinguish) exceptions or handler messages. However, for the development of dependable systems it is of great value to specify which interactions are safety-critical or needed in order to provide reliable service. For instance, a specification that highlights safety-critical messages could be used in a later design phase to decide on the use of a separate, reliable communication channel for sending these messages.

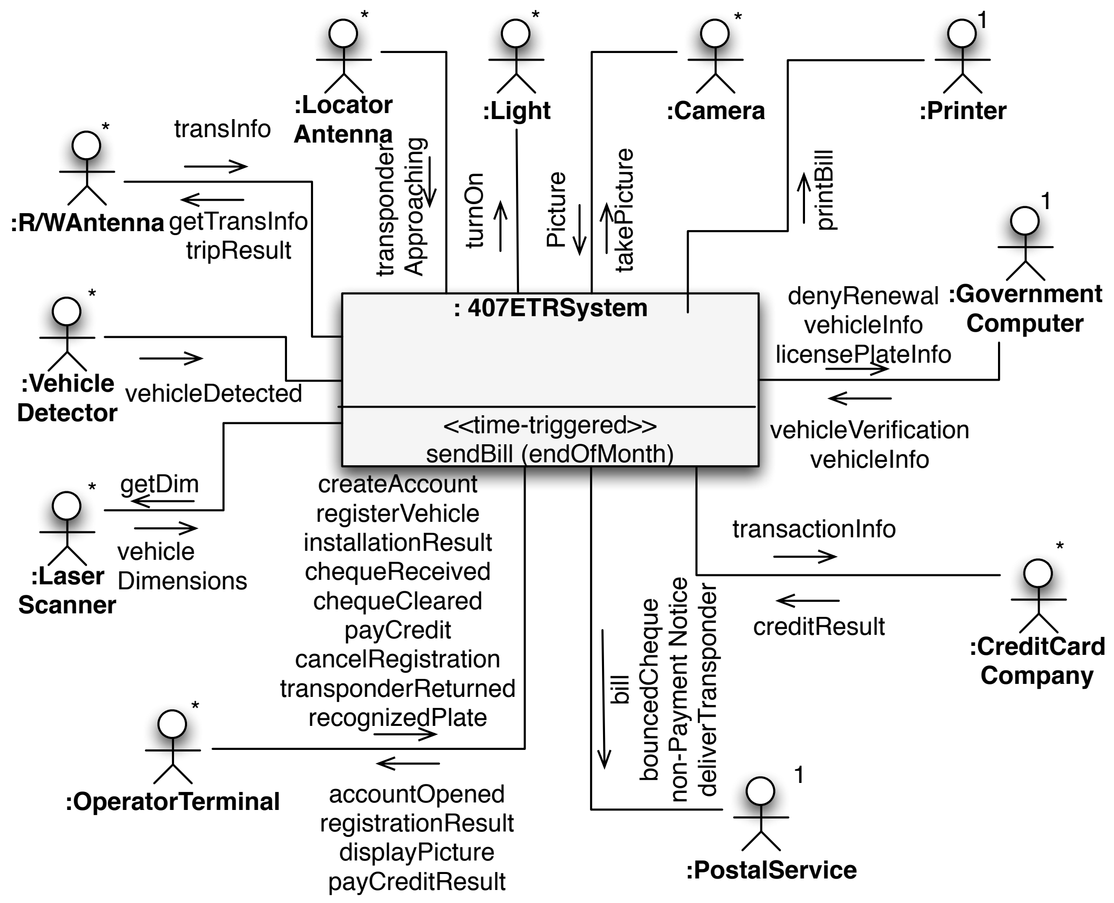


Figure 5–1: The 407 Environment Model Version 1.0

Another problem with the standard Fondue environment model is its scalability. When considering all possible points of failures in the environment, many additional exceptional and handler messages are usually added to the system. This can result in an environment model with too many messages, which makes it hard to recognize messages that are semantically related.

5.2.1 Boolean Value Interactions

The first problem seen in these diagrams is when we encounter an interaction with boolean possibilities. These cases were quite common in case of a boolean input or output such as success or failure of a certain process.

One option is to send these boolean values in parameters with a name used to identify the interaction. However, this is contrary to the idea of environment model, since the given name is not actually a message.

Another solution is to separate these two messages as success or failure. In this case, another problem arises as we cannot separate the success/failure of one process from another. If we look at the environment model, messages such as success, failure for checkClearing or installationResult cannot be identified. Infact, in some cases it becomes impossible to identify different messages.

We solved this problem by just giving them names like checkClearingSuccess, checkClearingFailure, installationResultSuccess and installationResultFailure. A similar trend is seen in cases where the result may be approved or rejected.

5.2.2 Exceptional Messages Interface

The second important question is whether we actually need to separate the exceptional messages? Also, if we need to separate them, how do we distinguish exceptional messages from normal messages. We definitely need to separate them for the convenience and understandability of any user. This is quite important while designing or implementing the system.

One other reason is to provide the users including the owner of the diagram with different views of the environment model; i.e. one with the exceptions and one without the exceptions. This will certainly be very helpful for analysis and design.

Exceptional Messages Notation

As far as the second question is concerned, there were certain ideas presented for the denotation of exceptional messages. Keeping the current standards and notations in mind, it was decided to just denote the exceptional messages with an _e at the end.

Handler Messages

This discussion lead to two main problems. The first issue was to distinguish the messages including normal and exceptional messages that were triggered following an exceptional behavior. We call these messages handler messages as these are involved in the handlers required to execute whenever an exception is encountered. There were various suggestions regarding this issue. A few of them were to denote these messages with a _h or maybe _e'. Another one was to exclude these handler messages from the diagram. It was temporarily decided to denote these with an _e'.

Exception Type

Another concern at this time was to identify at this point whether the exceptions were reliability or safety exceptions. At this point it was decided that the role of the environment model is to serve the purpose of an interaction diagram. Hence, the exception details such as safety/reliability and its priority should be handled in other models.

5.2.3 Timeouts & Hardware Exceptions

The second issue was the inclusion of hardware and timeout exceptions. The main problem with these exceptions was that these are not actually interactions. Neither are these time-triggered. The timeout exceptions can be classified as time triggered based on certain events. But the timeouts are not a trigger that would occur every 'x' seconds, minutes or days. However, we decided that these exceptions were detected by the system and handled by the system. Consequently, at this point it was decided to place this exceptions in the system, alongwith the time triggered messages.

This discussions lead to an entirely changed environment model. The raw environment model is shown in Figure 5–2 (version 2.0)

5.3 Exception Interaction Relations

Even after tagging the exceptional messages, the model still seemed quite confusing and complicated with a lot of messages without any chronological order or link. Moreoever, some exceptional messages could have been triggered by different events and probably even at different actors. The solution to all of these complications was to relate the exceptional messages to the regular messages.

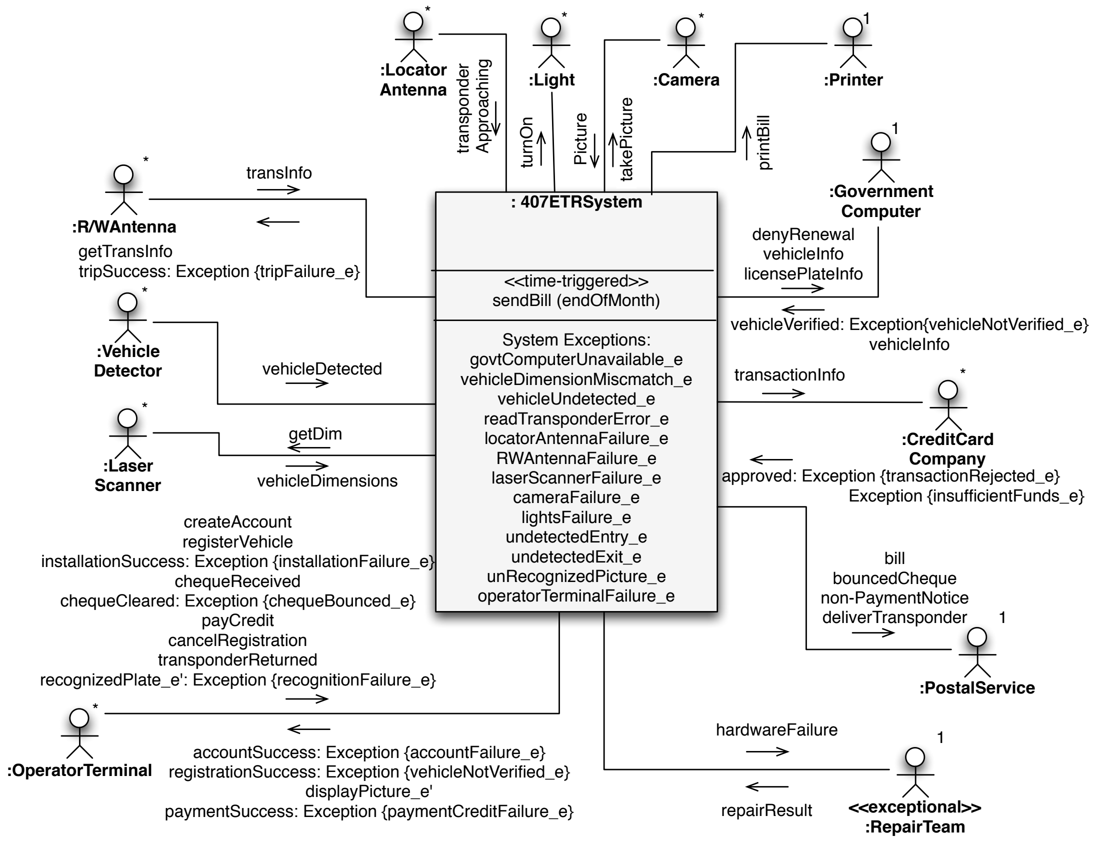


Figure 5–2: The 407 Environment Model Version 2.0

The need was to devise a standard notation to relate the normal and exceptional interactions.

There were several suggestions. Some of them were to actually show the messages together or maybe mark them with numbers related to each other.

The first solution changed the structure of the environment model a bit as each interaction showed much more than one message. Also, in some cases, the interactions were no longer asynchronous. However, the solution provided much better user understandibility and simplicity.

The second solution did not reduce the number of interactions. This solution definitely kept all the environment model characteristics alongwith creating a link between different messages. However, keeping a track of these numbers especially when making a change in the model is very complicated. In bigger software solutions, if the numbering has to be taken care of after every change, it would almost be an impossible task.

Hence, considering both the pros and cons of the solutions, we decided that the first one is the better of the two and we linked each of the messages with related exceptions.

5.3.1 Multiple Exception

The point to consider is that for each message there may be multiple exceptions. At this point, it was decided to devise a certain format for denoting each message linked with multiple exceptional messages.

TriggerMessage: output1, output2, ...,

Exception: Exception1, Exception2, ...

5.3.2 Timeouts and Hardware Failures

It was also decided to remove the timeouts and hardware exceptions from the system black box and put them as interactions. This was based on the detection of the timeouts and exceptions. Timeouts and hardware failures are detected by the interaction with different actors, hence they were shown as exceptions for the originating message.

TriggerMessage: output1, output2, ...,

Exception: Exception1, Exception2, ...

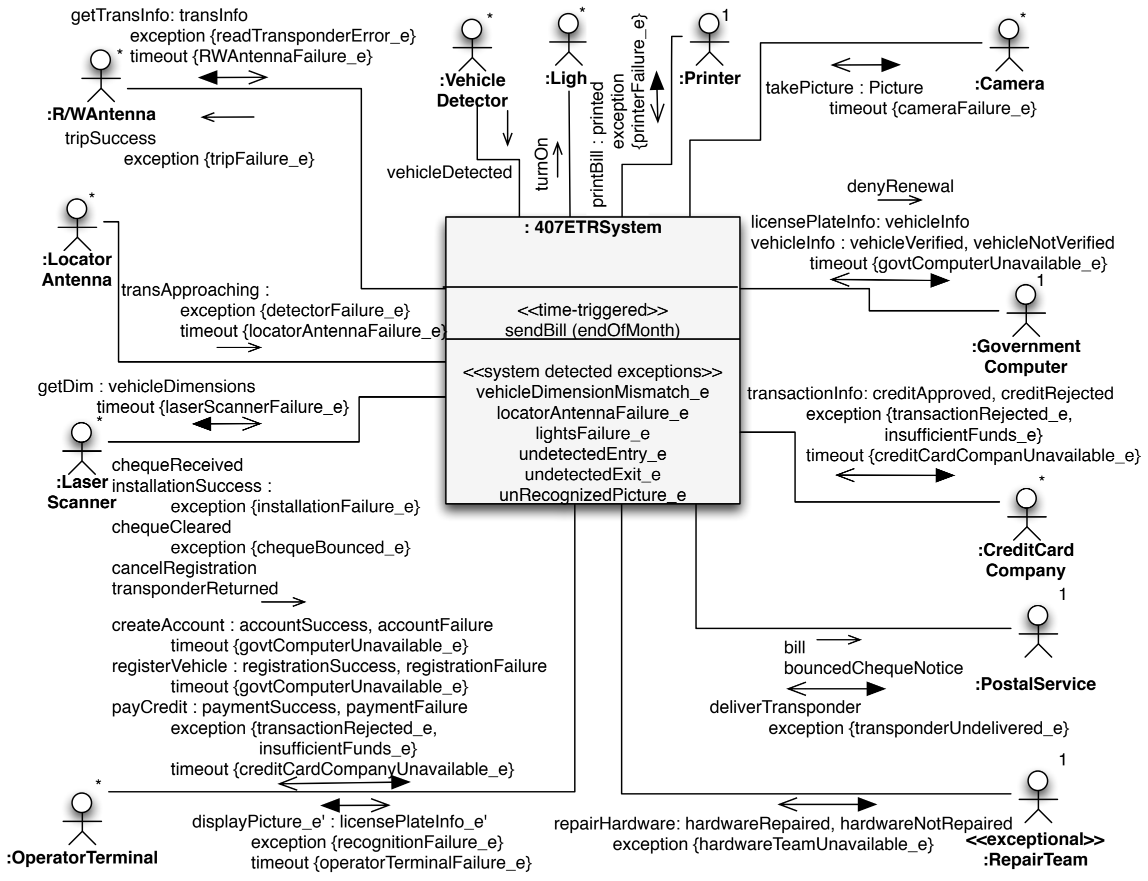
Timeouts: Timeout1, Timeout2, ...

5.3.3 Interaction Arrows

The linking of the normal messages, the responses and the exceptional messages caused some changes in the environment model. One of the major changes was that all the messages could not be considered asynchronous. Infact, most of the messages had to be synchronized.

A single arrow for all inputs from one actor or for all outputs to one actor could not be used anymore. Hence it was decided to bunch different asynchronous messages and denote them using one arrow. Similarly, another group was created for all synchronous messages.

After careful consideration of these issues, we came up with a much more refined version of the environment model. Although, still not upto our satisfaction, this version is shown in Figure 5–3 (version 3.0)



getTransInfo: transInfo
exception {readTransponderError_e}
timeout {RWAntennaFailure_e}
:R/WAntenna
tripSuccess
exception {tripFailure_e}
:Vehicle Detector
vehicleDetected
:Ligh
turnOn
printBill : printed
exception {printerFailure_e}
:Printer
takePicture : Picture
timeout {cameraFailure_e}
:Camera
:Locator Antenna
transApproaching :
exception {detectorFailure_e}
timeout {locatorAntennaFailure_e}
: 407ETRSystem
<<time-triggered>>
sendBill (endOfMonth)
<<system detected exceptions>>
vehicleDimensionMismatch_e
locatorAntennaFailure_e
lightsFailure_e
undetectedEntry_e
undetectedExit_e
unRecognizedPicture_e
denyRenewal
licensePlateInfo: vehicleInfo
vehicleInfo : vehicleVerified, vehicleNotVerified
timeout {govtComputerUnavailable_e}
:Government Computer
getDim : vehicleDimensions
timeout {laserScannerFailure_e}
:Laser Scanner
transactionInfo: creditApproved, creditRejected
exception {transactionRejected_e, insufficientFunds_e}
timeout {creditCardCompanUnavailable_e}
:CreditCard Company
chequeReceived
installationSuccess :
exception {installationFailure_e}
chequeCleared
exception {chequeBounced_e}
cancelRegistration
transponderReturned
createAccount : accountSuccess, accountFailure
timeout {govtComputerUnavailable_e}
registerVehicle : registrationSuccess, registrationFailure
timeout {govtComputerUnavailable_e}
payCredit : paymentSuccess, paymentFailure
exception {transactionRejected_e, insufficientFunds_e}
timeout {creditCardCompanyUnavailable_e}
bill
bouncedChequeNotice
deliverTransponder
exception {transponderUndelivered_e}
:PostalService
:OperatorTerminal
displayPicture_e' : licensePlateInfo_e'
exception {recognitionFailure_e}
timeout {operatorTerminalFailure_e}
repairHardware: hardwareRepaired, hardwareNotRepaired
exception {hardwareTeamUnavailable_e}
<<exceptional>>
:RepairTeam

Figure 5–3: The 407 Environment Model Version 3.0

5.4 Final Environment Model with Exceptions

However, during the modelling of this version, it was realized that there are many small issues that were left unsolved and thus ignored till this point. It was also seen that some of these issues may become bigger problems in different scenarios. Yet some others may be the reason for the confusion of a lot of users. Hence it was critical to solve these issues before finalizing a new refined environment model including exceptions.

Some of the issues are briefly discussed below:

5.4.1 Exceptions Raised At Different Situations

It was seen that in some cases, when a synchronous message was sent, the response received was either a message that showed success of the process or a failure of the process. The negative response was mostly an exception. However, in other cases, there may be exceptions raised with a positive or negative success message.

For example, when a TransponderDeliveryInfo message is sent to the post office, the answer may be success or it may be an exception such as exceptionTransponderUndelivered_e. Now, in another case, where the locator antenna sends a message to the system VehicleWithTransponder denoting that a transponder was detected. However, at this time if the Vehicle detector did not send any message, an ExceptionVehicleDetectorFailure_e must be raised. This is not a response but a message sent parallel to the actual message. Another example is when the LicensePlatePicture is sent to the Operator Terminal. The operator responds with a license plate number as well as an exception is raised based on the reason for the recognition failure.

Handling these problems separately would have complicated the environment model much more. Moreover, there was no particular need to distinguish these exceptional messages from each other in terms of interactions. This issue was brought to the attention and is mentioned here just to avoid confusion and questions from end users.

5.4.2 Linking of Interactions between different Actors

At this point, most of the messages sent to and from actors were synchronized. Hence, it was important to make sure there were no broken links. In some situations, an interesting observation was that the original message was sent to one actor. However, the response and the exceptional message was recieved from another actor.

The example of such a case is the message TransponderDeliveryInfo to the Postal Service. The response for the message is TransponderInstallationSuccess or the ExceptionTransponderInstallationFailure_e which are received from the operator terminal as asynchronous messages. Such interactions were consequently shown using another notation which links the original interaction with the response. Hence, the message from the Operator Terminal is shown as:

:PostalService :: TransponderDeliveryInfo: TransponderInstallationSuccess
Exception TransponderInstallationFailure_e

:Actor :: Original message : Normal Response
Exceptional Response

5.4.3 Handler Messages

Although handler messages were considered earlier, it was decided to denote the handler messages with an _h instead of the earlier suggested _e'. This was done to avoid confusion with exceptional messages.

5.4.4 Miscellaneous Issues

There were a lot of smaller issues with easier solutions and ignorable consequences. It is worth mentioning these issues such as the raising of the same timeout exception with different messages. Another question was to consider user goal threatening messages as normal messages or exceptions. It was also seen at times that a particular exception can be the reason to raise different kind of exceptions based on the handler execution.

These smaller problems were considered for a refined solution. However, it was decided that the solution to these would be different considering different situations. Hence, it was much better to leave these decisions for the analysis team related to the domain as per their convenience.

Hence, a final refined environment model was drawn including the exceptional messages. This model is shown in Figure 5–4 (version 4.0)

The interactions in the environment model are explained to provide all the missing and formal information for the consequent models.

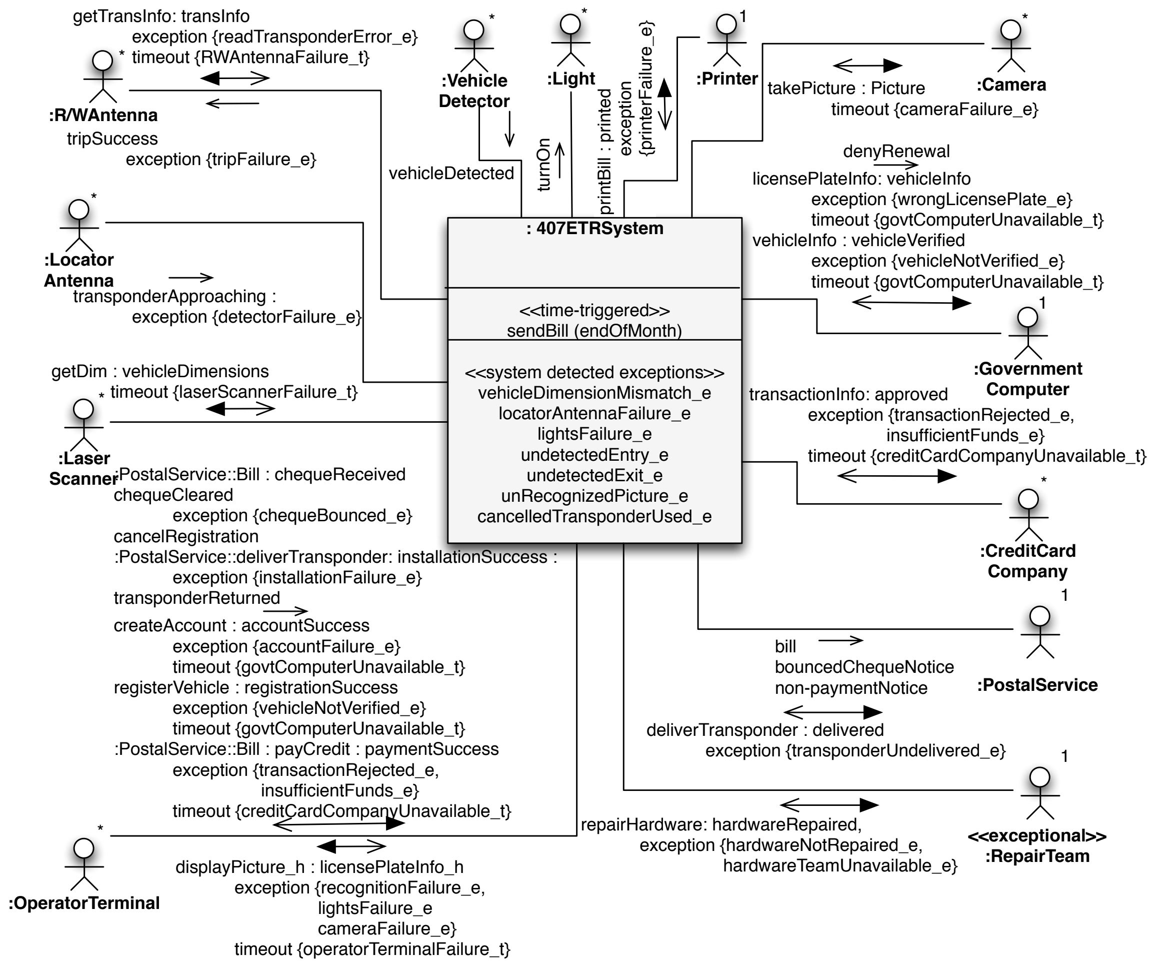

getTransInfo: transInfo
exception {readTransponderError_e}
timeout {RWAntennaFailure_t}
:R/WAntenna
tripSuccess
exception {tripFailure_e}
:Vehicle Detector
vehicleDetected
:Light
turnOn
printBill : printed
exception {printerFailure_e}
:Printer
takePicture : Picture
timeout {cameraFailure_e}
:Camera
denyRenewal
licensePlateInfo: vehicleInfo
exception {wrongLicensePlate_e}
timeout {govtComputerUnavailable_t}
vehicleInfo : vehicleVerified
exception {vehicleNotVerified_e}
timeout {govtComputerUnavailable_t}
:Government Computer
:Locator Antenna
transponderApproaching :
exception {detectorFailure_e}
: 407ETRSystem
<<time-triggered>>
sendBill (endOfMonth)
<<system detected exceptions>>
vehicleDimensionMismatch_e
locatorAntennaFailure_e
lightsFailure_e
undetectedEntry_e
undetectedExit_e
unRecognizedPicture_e
cancelledTransponderUsed_e
getDim : vehicleDimensions
timeout {laserScannerFailure_t}
:Laser Scanner
transactionInfo: approved
exception {transactionRejected_e,
insufficientFunds_e}
timeout {creditCardCompanyUnavailable_t}
:CreditCard Company
:PostalService::Bill : chequeReceived
chequeCleared
exception {chequeBounced_e}
cancelRegistration
:PostalService::deliverTransponder: installationSuccess :
exception {installationFailure_e}
transponderReturned
createAccount : accountSuccess
exception {accountFailure_e}
timeout {govtComputerUnavailable_t}
registerVehicle : registrationSuccess
exception {vehicleNotVerified_e}
timeout {govtComputerUnavailable_t}
:PostalService::Bill : payCredit : paymentSuccess
exception {transactionRejected_e,
insufficientFunds_e}
timeout {creditCardCompanyUnavailable_t}
bill
bouncedChequeNotice
non-paymentNotice
:PostalService
deliverTransponder : delivered
exception {transponderUndelivered_e}
repairHardware: hardwareRepaired,
exception {hardwareNotRepaired_e,
hardwareTeamUnavailable_e}
<<exceptional>> :RepairTeam
:OperatorTerminal
displayPicture_h : licensePlateInfo_h
exception {recognitionFailure_e,
lightsFailure_e
cameraFailure_e}
timeout {operatorTerminalFailure_t}
*
1

Figure 5–4: The 407 Environment Model Version 4.0

5.4.5 Type Definitions

The environment model, concept model and the operation model assume the existence of the following types:

- Picture: A type that encodes the pictures taken by the camera. It also stores the time of the image and the gantry where the camera exists.
- CreditCardNumber: A type to encode the credit card number required to track CreditCardKind.
- LicensePlate: A type that encodes the license plate number.
- Time: A type encoding the time.
- type GantryKind is enum {entry, exit}
- type VehicleClass is enum {light, heavy_single, heavy_multiple}

5.4.6 Input Messages

registerVehicle (String licensePlateNo, String owner): registrationSuccess ()

Exception vehicleNotVerified_e

Timeout govtComputerUnavailable_t

The driver registers his vehicle with the system using the operator terminal. The system verifies the vehicle with the GovernmentComputer and provides the result to the operator.

createAccount (String name, String address): accountSuccess ()

Exception {accountFailure_e}

Timeout GovtComputerUnavailable_t

The operator tries to create an account of the driver with the system. The system responds with the result.

:PostalService::deliverTransponder (String name, String address) : installationSuccess ()

Exception installationFailure_e

This is in response to the transponderDeliveryInfo message to the post office. The driver informs the system of the successful or unsuccessful installation of the transponder.

vehicleDetected ()

The vehicle detector at each gantry informs the system of an approaching vehicle that is going to enter or exit the highway.

transponderApproaching ()

Exception detectorFailure_e

The locator antenna informs the system that there is a transponder approaching the gantry. The system relates this message with the vehicleDetected message from the vehicle detector. Hence, if the transponderApproaching is received, whereas the vehicleDetectged message is not received, it denotes the failure of the vehicle detector to detect the vehicle.

tripSuccess ()

Exception tripFailure_e

The system sends a message to the RWAntenna if the whole trip transaction was a success or failure. The RWAntenna sends a message to the transponder accordingly.

cancelRegistration (String licensePlateNo, String owner)

The driver may cancel his registration using the operator. This is a single step process.

:PostalService::bill : chequeReceived (String chequeNo, String bank)

The driver may send a cheque for his bill payment. The operator notifies the system about the cheque receipt.

:PostalService::bill : payCredit (Account account, CreditCard creditCard, Integer amount) : paymentSuccess

Exception transactionRejected_e, insufficientFunds_e

Timeout creditCardCompanyUnavailable_t

The driver may pay the bill using his credit card. This is in response to the sendBill message sent by the system. The transaction result is sent back by the system. This is based on the communication with the credit card company.

chequeCleared ()

Exception chequeBounced_e

The operator uses the cheque received from the driver for the bill payment. If the bill is paid, he notifies the system about the success. If for some reason the cheque bounces, an exception is thrown.

transponderReturned ()

The operator sends a message to the system when it receives the transponder back from the driver in case of an account cancellation or a damaged transponder.

5.4.7 Output Messages

vehicleInfo (String licensePlateNo, String owner) : vehicleVerified ()

Exception vehicleNotVerified_e

Timeout GovtComputerUnavailable_t

The system sends the vehicle information to the GovernmentComputer to check if the vehicle is registered with the Government and is clear for registration. The GovernmentComputer responds according to the government vehicle database.

deliverTransponder (String name, String address) : delivered ()

Exception transponderUnDelivered_e:

On account creation, the system sends the transponder to the driver. The post office informs the system of successful delivery of the transponder or failure.

getTransInfo (): transInfo (Vehicle vehicle)

Exception readTransponderError_e

Timeout rWAntennaFailure_t

The system sends a message to the RWAntenna to get the account information using the transponder. The RWAntenna reads the transponder and sends the record information to the system.

turnOn ()

If the vehicle did not have a transponder or the RegisteredProcessing failed due to some reason, the system tries to recognize the vehicle using the camera and the lights. Hence, the lights are asked to be turned on for the pictures.

takePicture (): picture

timeout cameraFailure_t

The cameras are used in the same case as the lights. Once the lights are on, the camera is asked to take several pictures of the vehicle license plate.

displayPicture_h () : licensePlateInfo

Exception recognitionFailure_e, lightsFailure_e, cameraFailure_e

Timeout operatorTerminalFailure_t

The system uses the internal image recognition system to identify the license plate number. However, if the image recognition does not work for some reason, the picture is sent to the operator for manual recognition. The operator identifies the license number. He could also mention the reason of the failure looking at the picture, lke lights or camera failure. Moreover, he might not be able to recognize the license plate even manually that is covered by recognitionFailure_e.

licensePlateInfo (Integer license, Picture image) : vehicleInfo

Exception wrongLicensePlate_e

Timeout GovtComputerUnavailable_t

Once the license number is identified, it is sent to the Government computer to obtain the vehicle record. The government computer provides the vehicle record and tells the computer if there is some problem with the license number.

getDim () : vehicleDimensions

Timeout laserScannerFailure_t

To verify and match all the vehicles, the system also activates the laser scanner. The laser scanner uses laser to get the dimensions of the vehicle. These dimensions are sent to the system to get the type and size of the vehicle.

printBill () : printed

Exception printerFailure_e

The system prints the bill using the printer. The printer either prints the bill or an exception printerFailure_e exception is thrown.

bill ()

The system sends the bill to the Postal Service and receives a confirmation. A timeout is used to identify non-payment.

non-PaymentNotice ()

The system sends a non-payment notice for each bill that has not been paid

transactionInfo (String creditCardNumber, Date expiryDate, Integer amount) : approved

Exception transactionRejected_e, insufficientFunds_e

Timeout creditCardCompanyUnavailable_t

The credit card information provided by the driver is used for the transaction with the credit card company. The transaction result is returned to the operator and hence the driver.

bouncedChequeNotice ()

If the payment for a particular bill was not successful, the driver is notified with a bouncedChequeNotice.

denyLicensePlateRenewal (String licensePlateNumber, String owner, String reason)

This is an asynchronous message sent to the Government computer for black listed drivers only. A driver is black listed if he does not pay the bill for 3 consecutive months or the system identifies illegal use of the license plate.

repairHardware (Integer hardwareId, integer gantryId): hardwareRepaired

Exception hardwareNotRepaired_e, hardwareTeamUnavailable_e

The repairHardware message is sent to the RepairTeam whenever a hardware failure is detected.

CHAPTER 6
Exceptional Concept Model

6.1 Fondue Concept Model

The concept model contains all the information required for the purpose of fulfilling the system's responsibility over time, i.e. the necessary information to process an input message or to send out a notification. [Kie16]

The concept model is a special kind of class diagram model that is used to describe all the concepts and relationship parts of the system and all the actors present in the environment. The classes and associations in a concept model are used to model the problem domain. The object and the associations are used to hold the system state [SBS04]. Hence, the model is basically a static information structure of the system.

The concept model is built by delimiting the domain model and defining system boundaries. Any class that has to be included in the system is separated from objects and classes that belong to the environment. The system is shown explicitly as a composite class with a single instance that includes all the entities of the system. Hence, all the classes in the system get a multiplicity with respect to the system. Each of the actors are modelled as classes in the environment as viewed by the system. There are associations between the system and the actors depicting the flow of messages.

6.2 407 ETR Concept Model

The concept model for the 407 ETR case study has been shown in Figure 6–1

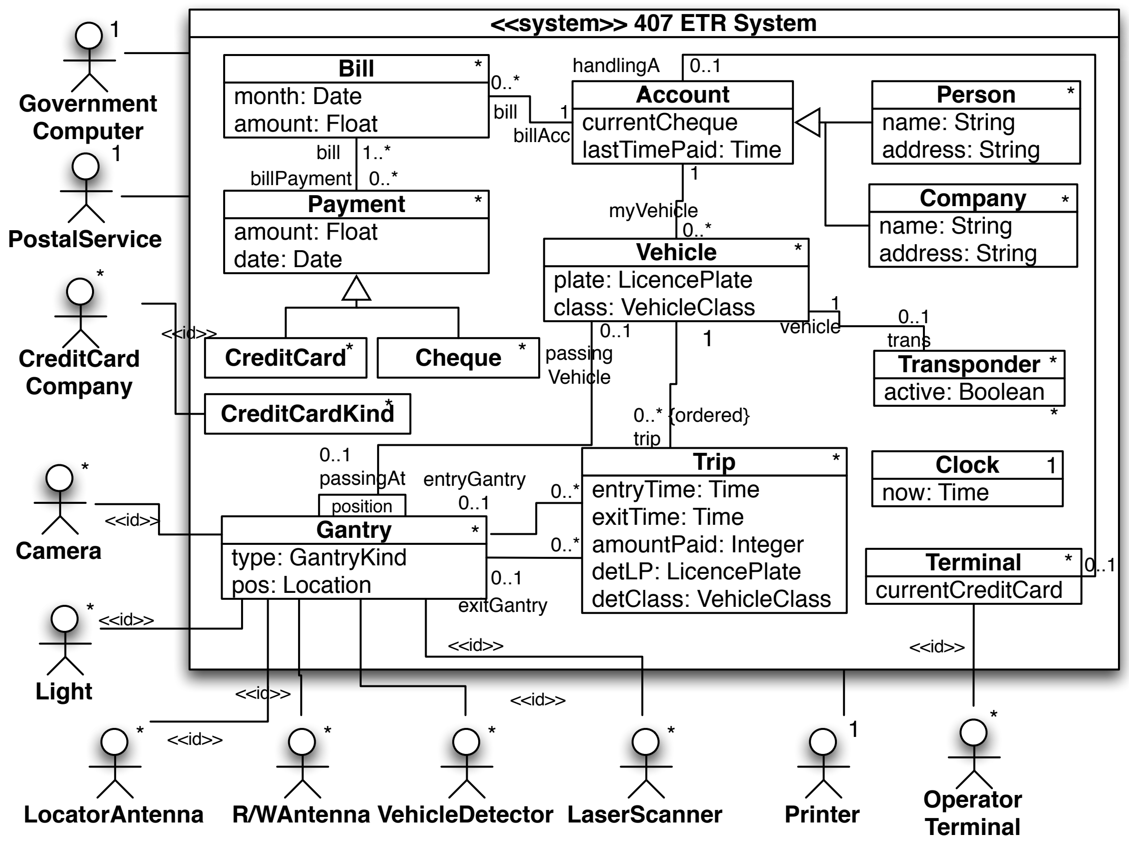


Figure 6–1: ETR 407 Concept Model

There were just a few major concerns during the design of the concept model. The relation between the account, vehicle and the transponder is the central part of the system. According to the functionality, each account may have multiple or no transponders associated. In that case, it was necessary to have two sub-classes of vehicles i.e. Registered and Unregistered vehicles. It was debatable even to have classes for the owner of the vehicle thus introducing Registered and Unregistered owner corresponding to the vehicle. Consequently, the solution drafted was to relate multiple vehicles with an account. Furthermore, associate the transponder class with the vehicle denoting whether the vehicle is registered or not.

Another important point was to keep track of trip information and how to get all the information from different gantries. This was handled by relating the trip to a vehicle. There were at most two gantries used in a trip. The gantry class was used to contain the temporary information related to the entry or exit of a vehicle.

6.3 Exceptional Concept Model

The exceptional concept model is provided in the Figure 6–2 and Figure 6–3.

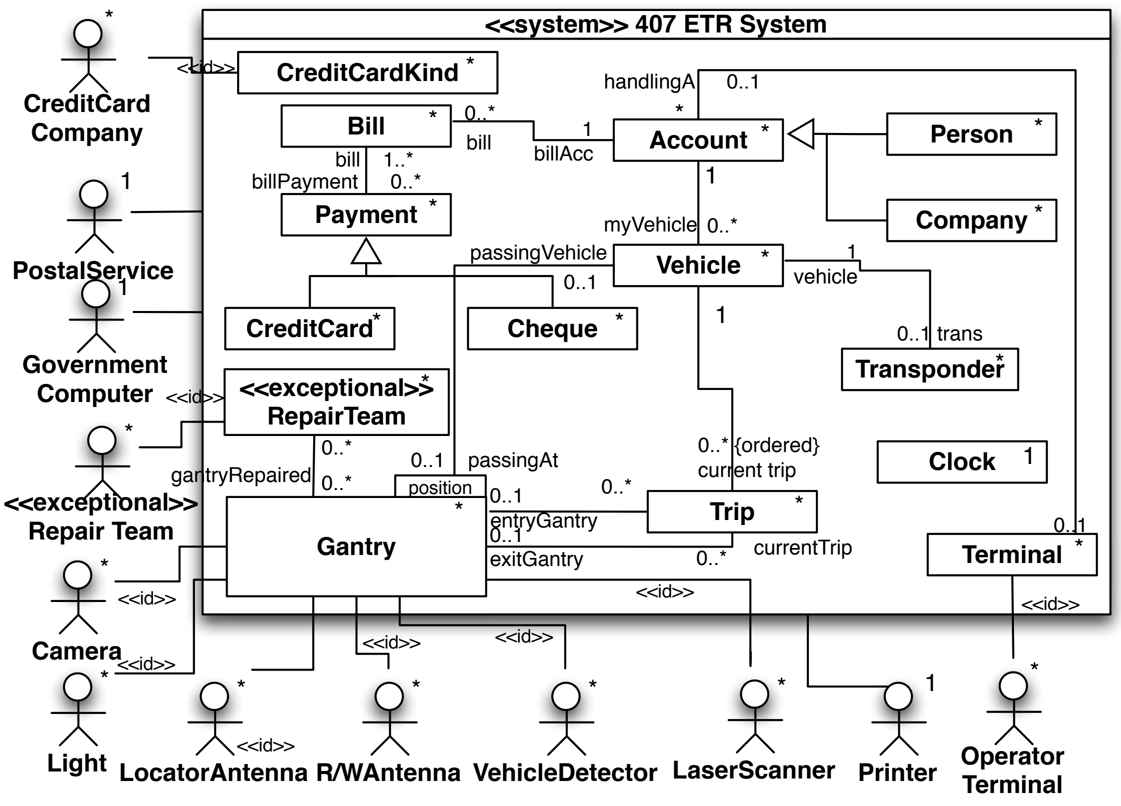


Figure 6–2: ETR 407 Exceptional Concept Model

There were very few but very important changes to cater for exceptional behavior. The first concern is whether to include any exceptional or handler classes in the concept model. Specifically for the ETR 407 case study, there are no such classes. There would probably be a need to introduce exceptional or handler classes in rare situation. For example, when a record of exceptions is to be maintained that affect the future system fuctionality. In fact, such classes would be a rarity in the concept model.

The reason is that in most cases, exceptions are not separate concepts or relations that define system state. In essence, they do modify the state of some concepts according to the situation. It has been made sure that, any change in the system state due to exceptional circumstances are handled in the Exceptional Concept Model.

This state modification due to exceptional situation can be catered for by keeping two points in mind:

1. To maintain the state at all times to detect the exceptional behavior in anticipation of the exceptional situation.

2. To capture the state after the exception has been raised to handle the exceptional scenario

In some cases, where the number of exceptions may be used to detect exceptional situation, they can be noted using counters. Similarly, to denote exceptional states, we can use flags. For the ETR 407 case study, this problem can be handled using the gantry class. All the hardware equipment is connected to one gantry,

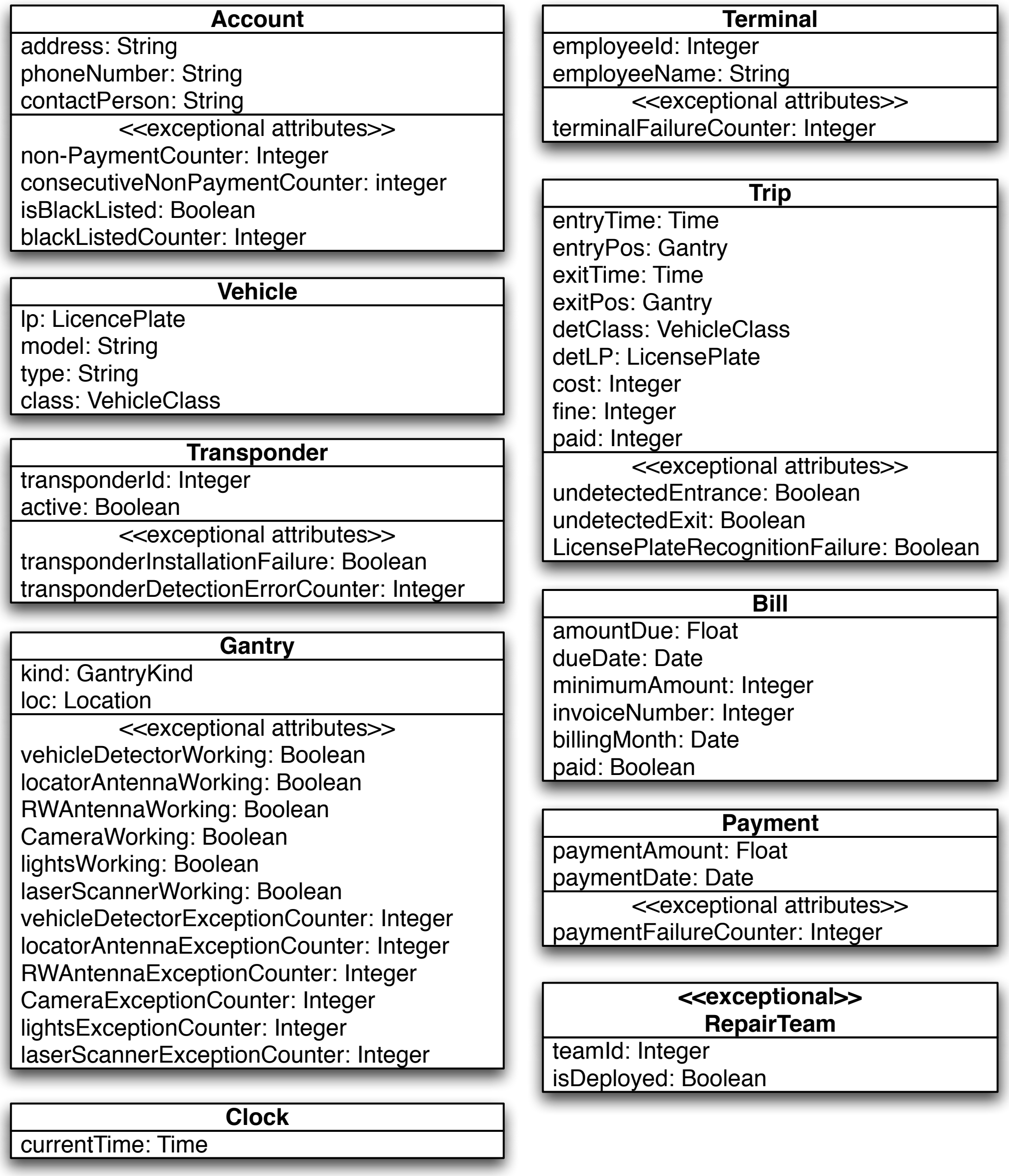

Account
address: String
phoneNumber: String
contactPerson: String
<<exceptional attributes>>
non-PaymentCounter: Integer
consecutiveNonPaymentCounter: integer
isBlackListed: Boolean
blackListedCounter: Integer
Vehicle
lp: LicencePlate
model: String
type: String
class: VehicleClass
Transponder
transponderId: Integer
active: Boolean
<<exceptional attributes>>
transponderInstallationFailure: Boolean
transponderDetectionErrorCounter: Integer
Gantry
kind: GantryKind
loc: Location
<<exceptional attributes>>
vehicleDetectorWorking: Boolean
locatorAntennaWorking: Boolean
RWAntennaWorking: Boolean
CameraWorking: Boolean
lightsWorking: Boolean
laserScannerWorking: Boolean
vehicleDetectorExceptionCounter: Integer
locatorAntennaExceptionCounter: Integer
RWAntennaExceptionCounter: Integer
CameraExceptionCounter: Integer
lightsExceptionCounter: Integer
laserScannerExceptionCounter: Integer
Clock
currentTime: Time
Terminal
employeeId: Integer
employeeName: String
<<exceptional attributes>>
terminalFailureCounter: Integer
Trip
entryTime: Time
entryPos: Gantry
exitTime: Time
exitPos: Gantry
detClass: VehicleClass
detLP: LicensePlate
cost: Integer
fine: Integer
paid: Integer
<<exceptional attributes>>
undetectedEntrance: Boolean
undetectedExit: Boolean
LicensePlateRecognitionFailure: Boolean
Bill
amountDue: Float
dueDate: Date
minimumAmount: Integer
invoiceNumber: Integer
billingMonth: Date
paid: Boolean
Payment
paymentAmount: Float
paymentDate: Date
<<exceptional attributes>>
paymentFailureCounter: Integer
<<exceptional>>
RepairTeam
teamId: Integer
isDeployed: Boolean

Figure 6–3: ETR 407 Exceptional Concept Model

hence, the flags or counters for each of the hardware device can be maintained in the Gantry class.

Record of the trip related exceptions can be stored in the Trip class and similarly for account and billing related exceptions, the Account and Monthly Bill class are utilized respectively.

The general concept to record exceptions is to maintain a flag for hardware equipment for its proper working while a counter is maintained for timeouts make sure that minute communication disruptions do not cause a major issue and repair team is deployed only if the counter exceeds a certain limit. Hence, the flags capture the state that is required for handling exceptional situations. Whereas, the counters maintain the state that help detecting exceptional situations.

CHAPTER 7
Exceptional Operation Model

7.1 Fondue Operation Model

The operation model describes the functionality that each system operation has to provide. Since Fondue focusses on the development of reactive systems, each system operation is triggered by a corresponding input message. The effects of each input message are described in an operation schema [SS00].

7.1.1 Operation Schema

An operation schema describes the effect of the operation on an abstract state representation of the system (i.e. the concepts represented in the Concept Model) and any events sent to the outside world [SS00]. The operation schema expressions are written using the Object Constraint Language (OCL), which is part of the Unified Modeling Language (UML).

7.1.2 System Operation

Each operation schema precisely describes a particular system action which executes atomically and is called a System Operation [SS00]. A system operation is considered to be a black box since there is no information given about the intermediate states when it is performed. The actual composition of the system state at any moment depends on the system operations invoked.

The operations are specified by logical predicates, i.e. Preconditions and Postconditions.

7.1.3 Preconditions and Postconditions

The precondition characterizes the valid initial states of the system when the operation is invoked.

The result of the system operation is expressed as a postcondition. The postcondition describes the changes made to the state of the system and what messages have been sent to actors. It must determine the behavior for all valid initial states.

The pre and post conditions assertions constitute the contract model of the operation. If the precondition is true, the operation terminates and the postcondition is true after execution of the operation. But if the precondition is not met, then nothing is guaranteed i.e. the effect of the operation is undefined. See Figure 7–1

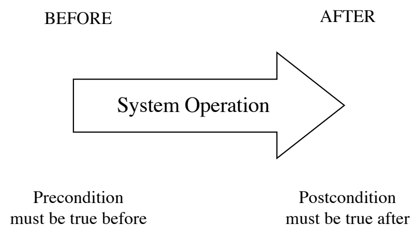

BEFORE
AFTER
System Operation
Precondition must be true before
Postcondition must be true after

Figure 7–1: Pre- and Postconditions

The post condition can assert the result of a system operation which may:

1. Create new object instances
2. Remove an object from the system state
3. Change the value of an attribute of an existing object
4. Add a link to an association or remove one

5. Send a message to an actor

7.1.4 Object Constraint Language (OCL)

The Object Constraint Language was first developed at IBM in 1995. Eventually it was standardized by the Object Management Group in 1997 as part of the UML standard 1.1. Since then, OCL has become the formal language for all UML users.

It is a declarative language that describes constraints on UML models. OCL expressions have no side-effects, i.e. an OCL expression constrains the system by observation rather than simulation. An OCL expression is evaluated on a consistent system state. Hence, no system changes are possible during or due to the evaluation of a constraint [SS00].

7.2 Adding Exceptions to the Operation Model

In the context of the operation schema, exceptions can be distinguished according to how they are detected. Depending on the situation, there are three different ways an exception can be detected:

- A standard incoming message triggers a system operation, but the current system state and the input parameters are such that the occurrence of the exception can be detected.
- An exception is detected by an actor (hence outside of the system), and the actor sends an input message containing the exception information to the system. The system receives this exceptional message in place of a normal message that it was scheduled to receive.

- The absence of an expected input message from an actor is detected by the system using a timeout.

The proposed exceptional operation model has to handle all these three kinds of exceptions. In addition, just like in the previous models, an important decision was to make sure that the specification of exceptional functionality is separated from the specification of normal system functionality. Thus minimum exception related expressions have been introduced in the normal schema, and new handler schemas have been introduced to specify handler functionality.

The first kind of exceptions are impossible to separate completely from the normal operation schema. They are detected when a normal interaction step happens based on the current system state and the value of input parameters. The thesis proposes to modify the template of the normal operation schema by adding an "Exceptional Conditions" field. In this field, the developer declares a named exception, together with a boolean expression and an exception handler predicate. By convention, all exception handler predicates should be tagged with the exception marker '_h'. If the system operation is executed, and the boolean expression evaluates to true, then the exception is said to have occurred, the postcondition of the operation schema is ignored and instead the exception handler predicate is true [1]

[1] The exception handling could have been done using a simple 'if' condition. However, the proposed approach allows a flexible solution where the exception handling part can be easily separated from the normal operation model.

The second kind of exceptions are not detected by the system and hence appear as exceptional input messages from the system's viewpoint. These messages can be treated as normal input messages as far as the operation model is concerned. Hence, a operation schema can be used to express such exceptions. Although, such an operation schema has been tagged with a marker denoting the exceptional nature of the schema. This tag is kept the same as for denoting handlers i.e. '_h'.

The third kind of exceptions are detected by the system. However these are timeouts and occur only when an input message is expected but does not arrive within a specified amount of time. Hence, they do not interfere with any other incoming messages. Therefore, these kind of exceptions can also be treated like exceptional input messages, and are therefore handled in the same way as the second kind of exceptions described above.

7.3 407 ETR Exceptional Operation Model

The 407 ETR operation model becomes quite big when elaborated. Therefore only a partial operation schema is discussed in this chapter. It is made sure that these operation schemas cover all possible kinds of exceptional scenarios in this case study.

The transponderApproaching message in Figure 7–2 is sent by the locator antenna. The system relates this message with the vehicleDetected message from the vehicle detector. Hence, if the transponderApproaching is received, whereas the vehicleDetectged message is not received, it denotes the failure of the vehicle detector to detect the vehicle.

This is an example of an exception that is detected along with the delivery of the original message. In the operation schema, we have shown such an exception by defining the Exceptional Condition. The boolean expression currentGantry.vehicleApproaching = false is checked, and if it is true, the exception detectorFailure_e is said to have occurred. The OCL predicate that describe the handling of the exception is entirely independent of the transponderApproaching operation schema and hence is shown in the form of a separate OCL predicate dispatchRepairTeam (Figure 7–3).

Another similar situation is shown in Figure 7–4, which shows the operation schema that records the time and location of entry and exit of vehicles. In this case, if transponder information is received from an exit gantry, and the system currently does not have a corresponding entry for the same vehicle, then the undetectedEntry_e exception has occurred, and the handler chargeMinimalTrip_h is taking effect (see Figure 7–5). This might have been caused by malfunctioning of the system during the entry of the particular vehicle.

Figure 7–6 illustrates an exception of the second kind, which is an entirely independent incoming message for the system. Here, the RWAntenna failed and sends a failure message ReadTransponderError_e instead of the normal transInfo message. Such an exception is handled in the operation schema as a exceptional input message and is expressed in an independent handler operation schema ReadTransponderError_h.

Figure 7–7 is also an example of the first kind of exception. If the image is not recognized by the system, the unrecognizedPicture_e exception is raised. This is also expressed as an exception function 7–8.

The handler message displayPicture_h is sent to the operator, so that he / she can handle this exception. In response the licensePlateInfo_h handler message is received that is shown in Figure 7–10. For the purpose of the schema, the handler message is treated as a input message.

Figure 7–11 is the same as the case described in Figure 7–4.

Figure 7–9 is an Exception of the third kind. This is detected in response to a message sent to an external actor. This timeout can be considered as a time triggered input message generated by the system and is handled in an independent operation schema. The timeout exceptions cover most of the hardware failures in the case study.

transponderApproaching Operation Schema
Operation: 407ETRSystem::transponderApproaching ()
Description: The locator antenna informs the system that there is a transponder approaching the gantry.
Use Cases: PassThroughGantry
Scope: Gantry;
Messages: R/WAntenna::{getTransInfo},
        LaserScanner::{getDim};
Alias: currentGantry : Gantry = sender.gantry,
        rw : R/WAntenna = currentGantry.R/WAntenna,
        ls : LaserScanner = currentGantry.LaserScanner;
Exceptional Conditions:
        detectorFailure_e: currentGantry.vehicleApproaching = false implies dispatchRepairTeam_h;
Pre: true;
Post:

```
    currentGantry.vehicleWithTransponder = true &
    currentGantry.rw^getTransInfo() &
    currentGantry.ls^getDim();
```

Figure 7–2: TransponderApproaching Operation Schema

dispatchRepairTeam_h Handler
Context : g : Gantry:
Messages: RepairTeam::{repairHardware};
Alias: repairTeam : RepairTeam = any(r:sender.gantry.repairTeam |
        r.isDeployed = false);
Post:

```
    g.vehicleDetectorExceptionCounter = g.vehicleDetectorExceptionCounter@pre+1&
    if g.vehicleDetectorWorking = true && g.vehicleDetectorExceptionCounter ≥ 3
        g.vehicleDetectorWorking = false &
        repairTeam^repairHardware (g.VehicleDetector.id, g.id) &
        repairTeam.isDeployed = true
    end if;
```

Figure 7–3: dispatchRepairTeam_h Predicate

transInfo Operation Schema
Operation: 407ETRSystem::(getTransInfo()):transInfo(veh : Vehicle)
Description: This message is in response to the system query to the R/WAntenna to send it the account information.
Scope: Vehicle, Trip, Gantry, Clock;
Use Cases: ProcessRegisteredVehicle;
New: newTrip : Trip;
Alias: g = sender.Gantry,
currTime = Clock.currentTime;
Exceptional Conditions: undetectedEntrance_e : g.type = GantryKind::exit and veh.currentTrip→isEmpty() implies chargeMinimalTrip_h
Pre: true;
Post:

```
if g.type = GantryKind:entry then
    newTrip.oclIsNew() &
    newTrip.entryTime = currTime &
    newTrip.entryGantry = g &
    newTrip.vehicle = veh
else
    let currTrip = veh.currentTrip in
        currTrip.exitTime = currTime &
        currTrip.exitGantry = g
    end let
end if
```

Figure 7–4: transInfo Operation Schema

chargeMinimalTrip_h Handler
Context : g : Gantry:
New: newTrip : Trip;
Post:
    newTrip.oclIsNew() &
    newTrip.vehicle = veh &
    newTrip.undetectedEntrance = true &
    newTrip.entryTime = calculateSmallestTime (currTime) &
    newTrip.entryGantry = calculateClosestGantry (self) &
    currTrip = newTrip

Note: It is assumed that calculateSmallestTime () and calculateClosestGantry are OCL functions that find the closest gantry from the exit gantry and estimate the smallest time interval between the gantries.

Figure 7–5: undetectedEntrance_e Function

readTransponderError_h Operation Schema
Handler Operation: 407ETRSystem::readTransponderError_h ()
Description: This message is in response to the system query to the R/WAntenna to send it the account information. The registered vehicle procedure has failed so the system has to follow the process for unregistered vehicle.
Scope: Gantry
Use Cases: ProcessUnregisteredVehicle;
Messages: Light::{turnOn},
        Camera::{takePicture};
Alias: g = sender.Gantry,
Pre: true;
Post:
    g.Light^turnOn() &
    g.Camera^takePicture();

Figure 7–6: readTransponderError_h Handler Operation Schema

picture Operation Schema
Operation: 407ETRSystem::(takePicture()):picture (image : Picture)
Description: The camera takes a picture of the passing vehicle and sends it to the system.
Scope: Vehicle, Trip, Gantry, Clock;
Use Cases: TakeHighway;
Messages: GovernmentComputer::{licensePlateInfo};
Alias: licenseNum = self.imageRecognized (image);
Exceptional Condition: unrecognizedPicture_e : licenseNum = 0 implies unrecognizedPicture_h;
Pre: true;
Post:
GovernmentComputer^licensePlateInfo (licenseNum, image)

Figure 7–7: picture Operation Schema

unrecognizedPicture_h Handler
Messages: Terminal::{displayPicture_h};
alias: myTerminal = any (Terminal);
Post: myTerminal^displayPicture_h (image)

Figure 7–8: unrecognizedPicture_h Handler

cameraFailure_e Operation Schema

Operation: 407ETRSystem::(takePicture()): cameraFailure_e ()
Description: This message is generated if the camera fails and does not provide the picture of the vehicle to the system.
Scope: Gantry
Use Cases: ProcessUnregisteredVehicle;
Messages: RepairTeam::{repairHardware};
Alias: g = sender.Gantry,

```
        repairTeam : RepairTeam = any(r:sender.gantry.repairTeam |
        r.isDeployed = false);
```

def: cameraFailure_e () =

```
    g.cameraExceptionCounter = g.cameraDetectorExceptionCounter@pre+1&
    if g.cameraWorking = true && g.cameraExceptionCounter $\geq$ 3
        g.cameraWorking = false &
        repairTeam^repairHardware (g.Camera.id, g.id) &
        repairTeam.isDeployed = true
    end if;
```

Note: Please note that all the hardware failure (i.e. locatorAntennaFailure_e, rWAntennaFailure_e, lightFailure_e, laserScannerFailure_e) as in the environment model will be handled the same way as the cameraFailure_e. Hence they have not been repeated in the operation model.

Figure 7–9: cameraFailure_e Operation Schema

licensePlateInfo_h Operation Schema

Operation: 407ETRSystem::(displayPicture_h()):licensePlateInfo_h(licenseNo : Integer, image : Picture)
Description: The operator reads the license number from the image and sends it to the system.
Scope: Vehicle, Trip, Gantry, Clock;
Use Cases: TakeHighway;
Messages: GovernmentComputer::{licensePlateInfo};
Pre: true;
Post:

```
    GovernmentComputer^licensePlateInfo (licenseNum, image);
```

Figure 7–10: licensePlateInfo_h Operation Schema

vehicleInfo Operation Schema
Operation: 407ETRSystem::(licensePlateInfo()):vehicleInfo (veh : Vehicle, image : Picture)
Description: The Government computer gets the vehicle data from the Government database on the basis of the license number.
Scope: Vehicle, Trip, Gantry, Clock;
Use Cases: TakeHighway;
Alias: g = image.Gantry,
    currTime = image.time;
Exceptional Conditions: undetectedEntrance_e : g.type = Gantry::exit and veh.currentTrip→isEmpty() implies undetectedEntrance_h
New: newTrip : Trip;
Pre: true;
Post:

```
  if g.type = GantryKind:entry then
      newTrip.oclIsNew() &
      newTrip.entryTime = currTime &
          newTrip.entryGantry = g &
      newTrip.vehicle = veh
  else
      let currTrip = veh.currentTrip in
          currTrip.exitTime = currTime &
          currTrip.exitGantry = g
      end let
  end if
```

Figure 7–11: vehicleInfo Operation Schema

CHAPTER 8
Related Work

8.1 Requirements Elicitation and Analysis

There have been a lot of researchers formulating standards and devising new techniques for each part of the software life cycle. Although, the focus of this thesis has been requirements elicitation and analysis.

8.1.1 Requirements Elicitation

The most industry wide approach for describing the system specification is done with Use cases. Alistair Cockburn provided a really good explanation of how use cases fit into the rest of the software engineering process in his book, *Writing Effective Use Cases* [Coc00]. He also provides a lot of tips to write the use cases effectively.

Use cases have now become an excellent tool for capturing behavorial requirements of the system that is targetted towards non-technical stakeholders comprehension as well. Writing better use cases for a system understanding has been been the focus of different researchers that have done vast improvements in the model.

8.1.2 Analysis

Most of the focus of the software life cycle besides development has been on analysis and design. Proper analysis and design techniques and models have been proposed and researched for the last two decades.

The importance of analysis and design has resulted in the formulation of the *Unified Modelling Language (UML)*[OMG04]. The contribution of James Rumbaugh, Ivar Jacobsen and Grady Booch in the maturity of the UML is immense. UML has been the standard for notations used during analysis and design in the software industry.

The proposal of the Fusion model [CAB$^+$94] was one big step in the analysis models for object oriented software development. Eventually, the Fondue model [SS99] was evolved based on the Fusion model using the UML with defined deliverables for analysis. Besides notations and other deliverable a huge role of the Fondue model was to introduce the concept of Object Constraint Language (OCL).

8.2 Exception Handling

Despite all the researches in the requirements and analysis phases of the software life cycle, there has not been much work done on handling exceptions. The exception handling procedures have been defined and incorporated in development softwares. They are commonly used in the entire software industry. However, they have been ignored in terms of incorporating exceptions and their handling during the earlier phases of software engineering.

Although the earlier stages of software engineering have been ignored, however, some work has been done on exception handling during requirement elicitation.

The use case based requirements elicitation process was extended to systematically investigate all possible exceptional situations that the system may be exposed to. The paper *Exceptional Use Cases* [SMKD05] defines means to detect exceptional occurences and the exceptional interaction between the actors and the system necessary to recover from such situation as described in the handler use cases.

The study was further refined to extend the use case based requirements elicitation process. The motivation of the study is not to rely on the developer's imagination and ideas for discover exceptions and handlers. The process leads the developer to systematically discover exceptions and handlers that the system may be exposed to. The idea is based on analysis of the system using a top down approach followed by a bottom up approach to make sure no possibilities are missed. Moreover, the UML use case diagram is extended to show exceptional behaviour [SMK06]. This process was further optimized to get better results and was elaborated with a simple case study [MSKV06].

Exception handling is supposed to make the system more reliable. Although, according to Miller and Tripathi in [MT97], exceptional handling when integrated with object oriented systems caused some major issues.

The concept of Co-ordinted atomic (CA) actions was introduced as a unified approach for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object oriented system [XRR98].

De Lamos et al. [dLR01] suggested the separation of exception during the different phases of the software life cycle. Instead of restricting discovering of exceptions and handling them to one particular phase, the idea is to provide a systematic and effective approach for dealing with exceptions at all phases of the software life cycle. This is done by defining a step wise method to identify exceptions and their contexts where the faults are identified. This study tries to address most of the issues presented by Miller et al. [MT97] and also extends the approach of the Co-ordinated atomic actions.

CHAPTER 9
Conclusion

9.1 Conclusion

In today's world, where computer software is taking over every day life, the importance of the reliability of software is undeniable. Exception handling is the backbone of dependability in all software systems. Moreover, a proper mechanism for the process of the detection and handling of exceptions is key for effective exception handling.

The idea of taking the exceptions into account early in the software development phase seems very advantageous. In fact, detecting exceptions at an early stage is comparatively easier and more cost effective. It allows more useful input from the stakeholders regarding critical exceptions which facilitates handling these exceptions. [MK] have presented an exceptions detection process that ensures minimum probability of ignoring exceptions at the requirements phase. In their approach, the exceptions are documented using {exception} tags and handler use cases.

This thesis is focused on defining the exception handling process in the analysis phase of the software life cycle. It has proposed changes to the environment model, the concept model and the operation model of the Fondue development process to incorporate exceptions.

In the environment model, the exceptions resulting from an interaction between the system and the actors are associated with the interaction. This simplifies the model by reducing the large number of messages that would appear if exceptions were shown separately. For understandibility, synchronous and asynchronous messages are differentiated using identifiable message arrows. The exceptional messages and handler messages are tagged to distinguish them from normal messages. Hardware failures, timeouts and system detected exceptions are visualized differently in this model.

The concept model does not require major modifications for the assimilation of exception handling. The crucial part is to allow the addition of any exceptional or handler classes in the model and relating them to the normal classes. The classes are defined such that they contain all the exceptional state needed to perform adequate recovery.

In the operation model, exceptions can be categorized into three kinds: the ones that are detected during normal interaction, the ones that are detected externally and directly communicated to the system and the ones that are detected by the system itself using timeouts. The first kind is handled by adding an *Exceptional Condition* section to the operation schema and including exceptional functions. The other two are essentially treated just as standard incoming messages, but their effects are described in handler operation schemas.

In this thesis, a methodological exception handling procedure has been devised that can be used through the entire software analysis process. The combination of the 3 analysis models, the environment model, the concept model and the operation model, ensures that a complete and coherent specification including all exceptional functionality needed for recovery is created.

It would be really interesting to advance the exception handling process through the design phase and into the development phase for commercial softwares. This would definitely ensure better quality softwares and fewer system failures.

REFERENCES

[CAB+94] D. Coleman, P. Arnold, S. Bdoff, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-oriented Development: the Fusion method.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Professional, January 2000.

[dLR01] R. de Lemos and A. Romanovsky. Exception Handling in the Software Lifecycle. *International Journal of Computer Systems Science and Engineering*, 16(2):167–181, March 2001.

[Don90] C. Dony. Exception handling and object-oriented programming: Towards a synthesis. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25, pages 322–330, New York, NY, 1990. ACM Press.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[JG] J.C.Geffroy and G.Motet. *Design of Dependable Computing Systems.* Springer, 1st edition, Englewood Cliffs, NJ.

[Kie16] Jörg Kienzle. On atomicity and software development. vol 11(5):687–702, 2016.

[Knu01] Jörgen Lindskov Knudsen. Fault tolerance and exception handling in BETA. pages 1–17, 2001.

[LAK92] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[MK] Sadaf Mustafiz and Jrg Kienzle. A dependability-focused requirements engineering process.

[MSKV06] Sadaf Mustafiz, Ximeng Sun, Jörg Kienzle, and Hans Vangheluwe. Model-driven assessment of use cases for dependable systems. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 558–573. Springer, 2006.

[MT97] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. *Lecture Notes in Computer Science*, 1241, 1997.

[OMG04] OMG. Uml 2 superstructure, final adopted specification, 2004.

[RTC92] RTCA. Do-178b, software considerations in airborne systems and equipment certification, 1992.

[SBS04] Alfred Strohmeier, Thomas Baar, and Shane Sendall. Applying Fondue to Specify a Drink Vending Machine. *Electronic Notes in Theoretical Computer Science, Proceedings of OCL 2.0 Workshop at UML'03*, 102:155–173, 2004.

[SMK06] Aaron Shui, Sadaf Mustafiz, and Jörg Kienzle. Exception-aware requirements elicitation with use cases. In *Advanced Topics in Exception Handling Techniques*, pages 221–242, New York, NY, USA, 2006. Springer Berlin / Heidelberg.

[SMKD05] Aaron Shui, Sadaf Mustafiz, Jörg Kienzle, and Christophe Dony. Exceptional use cases. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 568–583. Springer, 2005.

[SS99] Shane Sendall and Alfred Strohmeier. UML-based Fusion Analysis. In Robert France and Bernard Rumpe, editors, *UML'99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999*, volume

1723 of *LNCS (Lecture Notes in Computer Science)*, pages 278–291, 1999. extended version also available as Technical Report EPFL-DI No 99/319.

[SS00] Shane Sendall and Alfred Strohmeier. From Use Cases to System Operation Specifications. In Stuart Kent and Andy Evans, editors, *UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, volume 1939 of *LNCS (Lecture Notes in Computer Science)*, pages 1–15, 2000. Also available as Technical Report EPFL-DI No 00/333.

[SS02] Shane Sendall and Alfred Strohmeier. Using OCL and UML to specify system behavior. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 250–280, London, UK, 2002. Springer-Verlag.

[XRR98] Jie Xu, Alexander B. Romanovsky, and Brian Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *International Conference on Distributed Computing Systems*, pages 12–21, 1998.