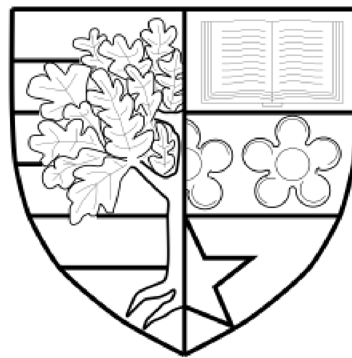
A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios

by Rajiv Murali



Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
February 2016

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

Abstract

Nearly all serious accidents, in the past twenty years, in which software has been involved can be traced to requirements flaws. Accidents related to or involving safety-critical systems often lead to significant damage to life, property, and environment in which the systems operate.

This thesis explores an extension to use case modelling that allows safety concerns to be modelled early in the systems development process. This motivation comes from interaction with systems and safety engineers who routinely rely upon use case modelling during the early stages of defining and analysing system behaviour.

The approach of embedded formal methods is adopted. That is, we use one discipline of use case modelling to guide the development of a formal model. This enables a greater precision and formal assurance when reasoning about concerns identified by system and safety engineers as well as the subsequent changes made at the level of use case modelling. The chosen formal method is Event-B, which is refinement based and has consequently enabled the approach to exploit a natural abstractions found within use case modelling. This abstraction of the problem found within use cases help introduce their behaviour into the Event-B model via step-wise refinement.

The central ideas underlying this thesis are implemented in, UC-B, a tool support for modelling use cases on the Rodin platform (an eclipse-based development environment for Event-B). UC-B allows the specification of the use cases to be detailed with both informal and formal notation, and supports the automatic generation of an Event-B model given a formally specified use case. Several case studies of use cases with accident cases are provided, with their formalisation in Event-B supported by UC-B tool. An examination of the translation from use cases to Event-B model is discussed, along with the subsequent verification provided by Event-B to the use case model.

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Andrew Ireland, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I would also like to thank my second supervisor, Gudmund Grov, for always being there to offer an extra point of view, insight and feedback.

The work which is presented in this thesis was funded by an Industrial CASE studentship funded by the EPSRC and BAE Systems (EP/J501992). I would like to express my special thanks of gratitude to Benjamin Gorry, Rod Buchanan and Paul Marsland from BAE Systems. I am very grateful to them for providing me with constructive feedback during my PhD and for giving me the opportunity to work with the Intelligent Systems team at BAE System, Warton.

I would also like to thank the following people: my internal and external examiners, Rob Pooley and Michael Poppleton, for the helpful feedback they gave me; members of Dependable Systems Group at MACS; members of the DReaM group at Edinburgh University; and the staff in MACS for always being helpful; and also thanks to Maria Teresa Llano for the guidance toward the final stages of my PhD.

I would like to express my deepest love and gratitude to my family. I want to thank my mother, Indulekha, and my father, Krishnapillai, for their support over the past twenty six years. Finally, special thanks goes to Peggy for her endless love and encouragement through this not-so-easy journey, thanks da.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.1.1 Industry Context . . . 6
1.2 Thesis Contribution . . . 7
1.3 Thesis Roadmap and Outline . . . 8

2 Background **10**
2.1 Requirements Engineering . . . 10
2.1.1 Problem Frames . . . 12
2.1.2 Goal-Oriented Requirements Engineering . . . 15
2.1.3 UML Use Cases . . . 17
2.2 Safety Engineering . . . 20
2.2.1 Accidents . . . 21
2.2.2 System Hazards . . . 22
2.2.3 Hazard Analysis . . . 23
2.3 Formal Methods . . . 27
2.3.1 Formal Specification . . . 28
2.3.2 Refinement . . . 28
2.4 Event-B . . . 31
2.4.1 Structure and Notation . . . 31
2.4.2 Refinement in Event-B . . . 33
2.4.3 Proof Obligations . . . 34
2.4.4 Rodin . . . 37
2.5 Summary & Discussion . . . 38

3 Accident Cases **39**
3.1 Introduction . . . 39

3.2 Accident Case . . . 40
3.2.1 Cause of an Accident . . . 42
3.2.2 Accident Scenarios . . . 43
3.2.3 Safety Guided Design . . . 44
3.3 Notation and Semantics . . . 47
3.3.1 Accident Case . . . 47
3.3.2 Deviate . . . 48
3.3.3 Prevent . . . 49
3.4 Related Work . . . 49
3.5 Summary & Discussion . . . 50

4 Formal Use Cases **52**
4.1 Introduction . . . 52
4.2 Use Case Model . . . 54
4.3 Agent . . . 55
4.3.1 Role . . . 57
4.4 Use Cases . . . 57
4.4.1 Contract . . . 57
4.4.2 Scenario . . . 59
4.4.3 Steps . . . 59
4.4.4 Accident Case . . . 62
4.4.5 Extension Use Case . . . 63
4.5 Abstract Syntax for Use Cases . . . 66
4.6 Related Work . . . 69
4.7 Summary & Discussion . . . 70

5 Encoding Use Cases in Event-B **71**
5.1 Introduction . . . 71
5.2 Use Case . . . 73
5.2.1 Encoding in Event-B . . . 74
5.2.2 Verification . . . 79
5.3 Accident Case . . . 80
5.3.1 Encoding in Event-B . . . 81
5.3.2 Verification . . . 82
5.4 Extension Use Case . . . 83
5.4.1 Encoding in Event-B . . . 83
5.4.2 Verification . . . 88
5.5 Encoding Branching in Event-B . . . 89

5.6 Translation Rules . . . 92
5.6.1 Rule Type: Project . . . 92
5.6.2 Rule Type: FlowType . . . 94
5.6.3 Rule Type: Static . . . 96
5.6.4 Rule Type: Contract . . . 97
5.6.5 Rule Type: Scenario . . . 99
5.6.6 Rule Type: Flow . . . 102
5.6.7 Rule Type: Extension . . . 103
5.7 Related Work . . . 105
5.8 Summary & Discussion . . . 106

6 UC-B: Tool Development 107
6.1 Introduction . . . 107
6.2 Goals . . . 109
6.3 Architecture & Technologies . . . 110
6.3.1 Eclipse . . . 110
6.3.2 The Eclipse Modelling Framework (EMF) . . . 110
6.3.3 UC-B Meta-model . . . 111
6.3.4 UC-B . . . 115
6.4 Summary & Discussion . . . 119

7 Case Studies & Evaluation 120
7.1 Introduction . . . 120
7.2 Case Study UC1: Water Tank System . . . 121
7.2.1 Use Case Model . . . 123
7.2.2 Event-B . . . 126
7.3 Case Study UC2: Train Door Control System . . . 131
7.3.1 Use Case Model . . . 132
7.3.2 Event-B . . . 135
7.4 Case Study UC3: Automated Teller Machine . . . 139
7.4.1 Use Case Model . . . 140
7.4.2 Event-B . . . 143
7.5 Case Study UC4: Sense and Avoid . . . 148
7.5.1 Use Case Model . . . 151
7.5.2 Event-B . . . 154
7.6 Summary & Discussion . . . 159

8 Future Work & Conclusion 160
8.1 Contributions revisited . . . 160
8.2 Limitations . . . 161
8.3 Future Work . . . 162
8.4 Concluding Remarks . . . 165

A Syntax of Event-B Mathematical Language 166
A.1 Predicate Language . . . 166
A.2 Expression Language . . . 167

B Case Studies: Event-B Model 168
B.1 UC1: Water Tank System . . . 168
B.2 UC2: Train Door Control System . . . 176
B.3 UC3: Automated Teller Machine . . . 184
B.4 UC4: Sense and Avoid . . . 193

C Case Studies: State Charts 204
C.1 UC1: Water Tank System . . . 204
C.2 UC2: Train Door Control System . . . 206
C.3 UC3: Automated Teller Machine . . . 209
C.4 UC4: Sense and Avoid . . . 210

Bibliography 212

Chapter 1

Introduction

1.1 Motivation

With the significant rise in computational power, safety-critical systems are increasingly software-intensive [72,104]. In the development of these systems, the requirements document forms the starting point and are an attempt to establish what the system should do. The conventional techniques for capturing requirements are typically informal [86], and do not lend itself to automatic analysis and checking. The implication of defects and limitations in the requirements can be significant, as much of the remainder of the development process is aimed at implementing the system described within the requirements document [73,77]. Errors introduced in the development of safety-critical systems are particularly costly, and can have significant impact on life, property, and the environment in which these systems operate.

Software-Related Accidents

> *"Most software-related accidents have been system accidents that stem from the operation of the software, not from its lack of operation and usually that operation is exactly what the software engineers intended."*
>
> – Nancy G. Leveson

Nearly all serious accidents in the past twenty years, in which software has been involved, can be traced to flaws introduced in the requirements [57, 72, 81]. In the loss of the Mars Polar Lander (MPL) [39], the software requirements did not include information to ignore the sensor readings from the landing-legs during the descent phase. The on-board software mistook a jolt, recorded by the sensor readings, during the deployment of the landing-legs for "ground contact", and shut down the descent engines, causing the MPL to fall from a presumed height of 40 meters (130 feet) on to

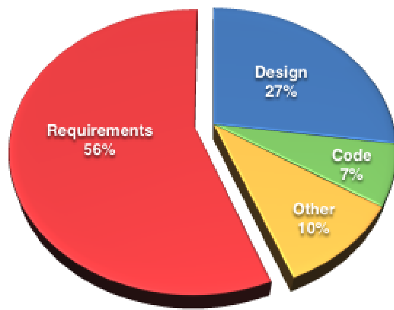
the surface of Mars. A post-mortem analysis determined that this scenario was most likely the cause of the accident that resulted in the MPL to strike the surface of Mars at a high velocity.

In the batch chemical reactor accident [64, 72], the computer was in charge of: controlling the flow of catalyst into the reactor, and also the flow of water into a condenser to cool off the reaction. The systems engineers were told that if a fault occurred in the plant, they were to leave all controlled variables as they were, and to sound an alarm. On one occasion, the computer received a signal indicating a low oil level in a gearbox (a fault). The computer reacted as its requirements specified: it sounded an alarm and left the controls as they were. By coincidence, a catalyst had been added to the reactor, but the computer had just started to increase the cooling-water flow to the reflux condenser; the flow was therefore kept at a low rate. This resulted in the reactor to overheat that caused the relief valve being lifted and the discharge of contents in the reactor into the atmosphere. This accident resulted from the systems engineers not being made aware of the safety requirement where the water valve was required to be opened before the catalyst valve, and therefore assumed the ordering was irrelevant.

The problem may also stem from unhandled control system states and environmental conditions. An F-18 was lost as a result of the aircraft getting into an altitude that the engineers had assumed was impossible and that the software was not programmed to handle [44]. Requirements are typically where the most errors are introduced in the development process and also the most expensive to fix [77, 81]. Figure 1.1, illustrates the software defects and their cost to fix ration with respect to development phases. There is hard data to support this premise that the majority of the software errors are often introduced in the requirements. Lutz [77] examined 387 software errors uncovered during integration and system testing of the Voyager and Galileo spacecraft. She concluded that a majority of software errors identified as potentially hazardous to the system were produced by: (1) discrepancies between the documented requirements specification and the requirements needed for the correct functioning of the system. (2) misunderstanding about the interface with the rest of the system.

Formal Methods for Requirements Analysis

When the requirements are defined informally throughout the early stages in the development, the process of verification relies heavily on engineering judgement at later stages. In requirements engineering, UML *use cases* [55] are an informal notation for modelling the required behaviour of a system with respect to its operational environment. They are widely used and highly accessible. Use cases provides a basis on



(a) *Distribution of software defects.*

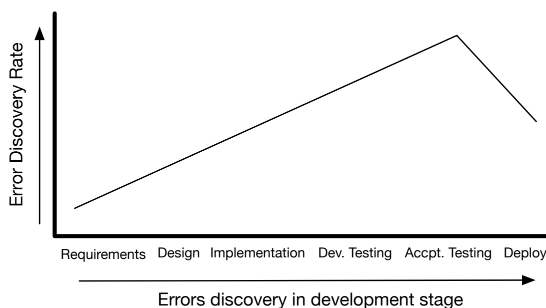
Phase	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Unit Testing	15-40
System Testing	30-70
Production	40-1000

(b) *Distribution of cost to fix defect.*

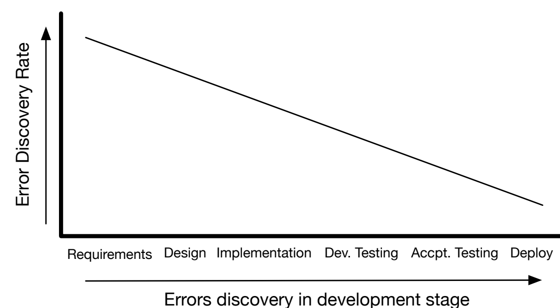
Figure 1.1: Software defects and their cost to fix ratio with respect to development phases [81].

which initial system behaviours can be defined and analysed. The lack of formality in specifying use cases means that the process of analysis is typically *review-based*, and thus lacks the rigour that comes from formal methods, i.e. systematic identification of ambiguities, inconsistencies and incompleteness.

Formal methods can be used at all stages in the development process, from requirements analysis to system acceptance testing [66]. It can provide *precision* which is key to eliminating errors in requirements, while *abstraction* is key to mastering requirements complexity. Without a rigorous approach to understanding requirements and constructing specifications, it can be very difficult to uncover such errors other than through testing after a lot of development has already been undertaken. Figure 1.2 illustrates error discovery rates at different stage in the development process, with and without the use of formal methods [66]. However, formal methods are not easily accessible due to the gap that exists between the informal and formal specifications.



(a) *Typical error discovery.*



(b) *Error discovery with formal methods.*

Figure 1.2: Error discovery rate at different stage in development [66].

This thesis presents an approach that adds rigour to use cases via the Event-B [1]

formal method. Event-B provides verification methods that support the discovery and elimination of inconsistencies in models. The formalism, Event-B, was selected because it promotes a layered style of formal modelling, where a design is developed as a series of abstract models – level by level concrete details are progressively introduced via provably correct refinement steps. Sometimes referred to as posit-and-prove, this style of modelling can increase the clarity of design decisions as well as simplifying the complexity of the verification task. Abstraction and refinement are key methods to manage system complexity for structuring formal modelling effort by supporting separation of concern and layered reasoning. This thesis aims to help bridge the gap between the informal (i.e. use cases) and the formal (i.e. Event-B) by leveraging the structure that is imposed by UML use cases, thus reducing the skills and experiences that the user will require in formal methods.

This thesis provides a novel approach in exploiting a natural abstraction found in use cases to aid the formal modelling effort with refinement. An encoding that exploits this mapping is presented in this thesis. That is, for a given use case it is possible to automatically generate an Event-B development that models the behaviour of the use case using step-wise refinement. The completion of the development relies upon the user formalizing the details of their use case, e.g. constants, variables, pre-, post-condition, invariants, assignments.

Integrating Safety Analysis with Requirements Analysis

In the development of safety-critical systems, apart from capturing and analysing the functional requirements, safety analysis is performed to identify potential dangers, i.e. undesired or unplanned behaviours, that may occur in the operation of the system [73,104]. The identification of these undesired behaviours helped investigate statements that form the *safety requirements* of the system. The safety requirements defines what the system must and *must not do* in order to ensure safety, and place integrity constraints on existing core functions. In addition, new functional requirements may be needed to prevent or mitigate the effects of failures identified in the analysis.

Often the requirements and safety analysis processes are performed in an ad-hoc manner [72]. This results in that the safety requirements and the new additional behaviours appear in the requirements document, without due acknowledgement from their origins in safety. Most requirements engineering techniques focus on capturing only the *desired behaviours* of the system. For instance, UML use cases [19] do not provide any special mechanisms for representing *undesired behaviours* or *safety concerns* identified by the safety analysis. UML use cases only consider positive scenarios, also known as “sunny day” scenarios, where there are no failures considered in the

interaction between the system and entities in the environment, to achieve the desired functionality. This results in over-simplified assumptions about the problem domain and a tendency to go prematurely into design considerations.

This thesis aims to bridge the gap between safety analysis and the UML use cases via the notion of *accident cases*. UML use cases are extended to include a use case type *accident case* that would allow the requirements analysis to consider undesired or unplanned behaviour from the safety analysis. The purpose of this extension is to allow the desired behaviour to be specified in context of the undesired behaviours that may occur.

Lightweight Application of Formal Methods

> *"Industry will have no reason to adopt formal methods until the benefits of formalization can be obtained immediately, with an analysis that does not require further massive investment."*
>
> – Daniel Jackson

Historically, formal methods have been viewed as a pure alternative to traditional development methodologies, requiring massive investment in the development process, for industry to adopt [6, 59]. Recently, there has been a new trend of lightweight applications of formal methods, documented by Jones [59], Jackson, Wing [51] and by Easterbrook [33]. A lightweight approaches exhibit, partiality with respect to language, modelling, analysis and composition, and has focused area of application. These lightweight approaches are targeted primarily on the early stages of development and are focused towards defect detection through rigorous examination.

In order for the formalisation of use cases to be more accessible, the research focused on the development of a prototype plug-in UC-B for the Rodin [4] platform (development environment for Event-B). The purpose of UC-B is to enable use cases to be authored and managed in Rodin. The plug-in aims to maintain the familiarity of detailing a textual use case specification with informal notation, while also allowing a corresponding formal specification, written with Event-B's mathematical language, to co-exist side-by-side. The aim of this dual representation of the specification is to bridge the gap between the informal and formal specification.

Furthermore, given a formally specified use case, the purpose of the tool is to support the automatic generation of an Event-B model, using the natural abstraction found in the structure of use cases. This is aimed to reduce much of the modelling effort with Event-B, while allowing the use case modeller to focus on specifying the requirements. The generated Event-B model is immediately subjected to provers and

syntax checkers, provided by the Rodin platform, that allow defects to be identified. As a consequence, inconsistencies and defects identified by formal verification tools can be related back to the level of the use case specification.

The concept of an accident case, the formalisation of use cases in Event-B, and the initial tool development has been published in [82]. The research reported in this thesis was supported by an EPSRC Industrial Case grant EP/J501992, with BAE SYSTEMS[1] as the industrial project partner.

1.1.1 Industry Context

The work presented in this thesis is an on-going effort to help with the industrial adoption of formal methods at early stage during requirements analysis and of a more specific effort to consider safety concerns. This research has benefited from an industrial project partnership with BAE SYSTEMS, which has provided a practical perspective on the current challenges faced by engineers during the early stages in system development. It has helped shape the focus of this research towards tools and techniques that are actively used by, and familiar to, industry practitioners. The following are summations of what was learnt through this partnership.

There are many techniques for early stage analysis of requirements: goal-oriented [107], problem-oriented [53], and use case based [55] requirements engineering techniques. At the start of this research, the requirements analysis was performed using Problem Frames, a problem-oriented requirements analysis technique, as a means to capture and analyse system behaviour. This was due to popularity and interest of Problem Frames in the Event-B community, as part of the DEPLOY[2] project, as means to bridge the gap between informal and formal specification [95]. However, through the industry partnership, UML use cases was understood to be actively used at early stages in the systems development process for requirements analysis, and their notations to be more familiar to practitioners, in comparison to Problem Frames.

In the communication with systems and safety engineers, approaches towards the integration of the requirements and safety analysis processes were also found to be of interest. The requirements and safety analysis processes are often known to be performed in an ad-hoc manner. This results in safety requirements appearing in the requirements documentation without due acknowledgement to their origins from safety analysis. This guided the research towards an approach to consider safety in UML use cases, via the notion of the accident case. This extension aims to provide a platform for systems and safety engineers to communicate appropriate design recommendations via

[1]BAE SYSTEMS - `http://www.baesystems.com/`
[2]DEPLOY Project - `http://www.deploy-project.eu`

additional functionality to help prevent accidents as part of the development process.

1.2 Thesis Contribution

In summary the main contributions of this thesis are:

Accident Cases for UML Use Cases

UML use cases is extended with a use case type *accident case*. The purpose of this extension is to enable accidents identified in the safety analysis process to be considered along-side the use cases of a system. The accidents represent undesired or unplanned behaviour that are introduced as deviations from the expected core functions of the system, which are defined by regular use cases. The notations and semantics for the accident case are provided.

Formal Use Case Specifications

The textual use case specifications that are typically informal, are enhanced with Event-B's mathematical language that support the use cases to be detailed with precise semantics. The aim of this enhancement is to allow the textual specification to have both informal and formal descriptions of the use case specification that are allowed to co-exist, side-by-side. The purpose of this dual representation is to provide a step towards bridging the gap between informal and formal specifications. The abstract syntax for the use case specification is provided.

Encoding Use Cases in Event-B

An encoding of the use case specification in Event-B is provided. This encoding exploits a natural abstraction found in use cases to model its behaviour in an Event-B model via step-wise refinement. Gluing invariants are identified to help ensure that the abstract model is related to the concrete model. The encoding also provides the verification support provided in the generated Event-B model for a use case with its relation to defects in the use case specification. Translation rules from a generic use case to an Event-B model is provided.

Tool Development

The development of a prototype tool, UC-B, for the Rodin platform is provided. The tool aims to implement the above contributions. It enables the use cases to be authored and managed in Rodin. The specification of the use cases is required to contain both

formal and informal notation. Given a formally specified use case the tool supports automatic generation of a corresponding Event-B model. The mechanisation of this process decreases the formal modelling effort. The generated Event-B model is immediately subjected to the automatic verification tools that Rodin provides which helps identify defects in the use case specification.

Case Studies and Evaluation

A collection of case studies is used to describe concepts of the use case specification and the encoding in Event-B. The case studies aim to cover the use case types: use case, accident case, and extension use case. In addition, types of branching within the scenario of a use case are discussed. The verification provided by the Event-B model is discussed with relation to the use case specifications.

1.3 Thesis Roadmap and Outline

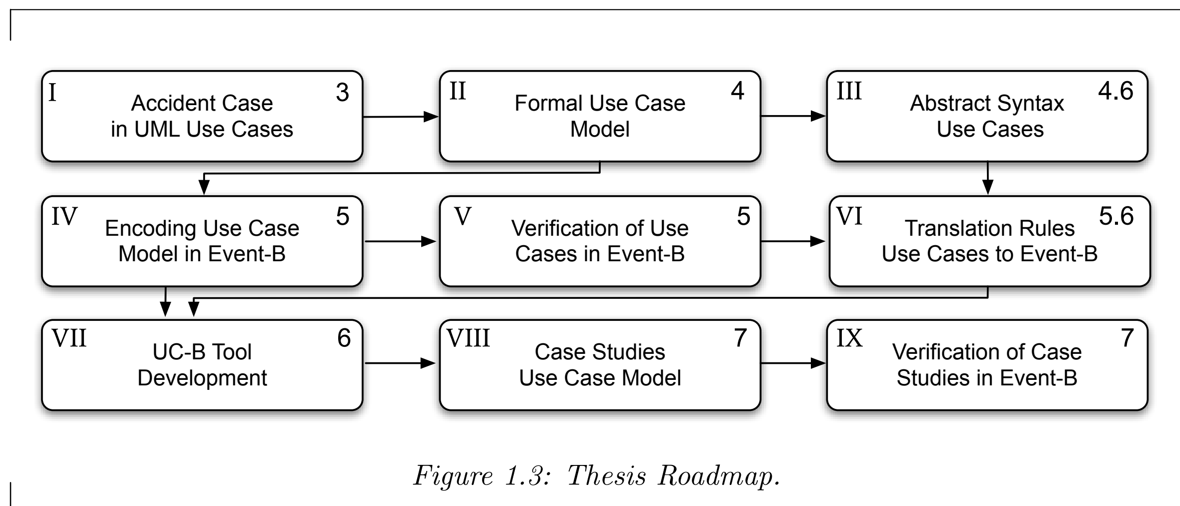


Figure 1.3: Thesis Roadmap.

Figure 1.3 shows the roadmap of the thesis. It has been divided into nine distinct parts. The arrows how dependency and are assumed to be "transitive". The thesis is organised as follows:

Chapter 2 contains relevant background information on requirements engineering, safety engineering and formal methods.

Chapter 3 extends UML use cases with accident cases.

Chapter 4 describes a formal underpinning of the use cases via the use case model. The use case model provides the structure for the use case specifications.

Chapter 5 describes the encoding of the use case model in Event-B.

Chapter 6 provides the tool development for UC-B for the Rodin platform.

Chapter 7 describes the case studies and their evaluation.

Chapter 8 and finally, Chapter 8 outlines future directions and concludes.

Chapter 2

Background

In this chapter, the necessary background on the field of requirements engineering, safety engineering, and formal methods is provided in Sections 2.1, 2.2 and 2.3, respectively. In particular, the background provides the concepts, notations and semantics for UML use cases and Event-B that form parts of the approach proposed in this thesis.

2.1 Requirements Engineering

> *"The single hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."*
>
> – Frederick P. Brooks, Jr

Requirements analysis was part of the formation of *Software Engineering* (SE), which was created as a result of the so-called "*software-crises*" [83] in the late 1960s. At this stage requirements analysis was perceived to be as potentially high leverage but neglected area of software development [41]. By the mid 1970s, a review by Bell and Thayer [16] had produced plenty of empirical data, conforming that the "*requirements problem is a reality*". The growing recognition of the critical nature of requirements in software engineering gradually established *Requirements Engineering* (RE) as an important sub-field of Software Engineering (SE) [46].

Brooks [43] highlighted the role of requirements engineering in his seminal paper, "*No Silver Bullet: Essence and Accidents of Software Engineering*". The paper suggested that the essential difficulties in requirements engineering are harder to solve due to the inherent properties of modern software-intensive systems. Difficulties arose

as a result of the software product being embedded in a cultural matrix of applications, users, laws, and other machine vehicles. These all change continually, and their change inexorably forces change upon the software product. Although much progress has been made since the 1960s, requirements deficiencies in many software development projects are still a main contributing factor for project failures and occurrences in software-related accidents [31, 73, 77]. Sommerville and Sawyer [102] observe that a large number of project cost overruns and late deliveries still exist because of poor requirements engineering processes. The following provides the definition of requirements engineering by Zave [117]:

Definition 2.1 (Requirements Engineering)**.** *Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families.*

In this section, an overview of the requirements engineering techniques Problem Frames, Goal-Oriented Requirements Engineering (KAOS and i* approach) and UML use cases are discussed. A running example of a *water tank system* is used to provide a viewpoint on how requirements are modelled by the requirements engineering techniques.

A Running Example: A Water Tank System

A simple case study of a *water tank system* is used as a means to describe how its core functionality can be captured and analysed by the different requirements engineering (RE) techniques. This case study is partly inspired by [21]. The aim of the water tank system is to maintain the *water level* between the high (H) and low (L) limits of the *water tank*, via the use of a *controller* (referred to as the water tank system), as seen in Figure 2.1. To achieve this intent, the controller interacts with two external components, namely the *sensor system* and *pump*. The sensor system monitors the water level in the tank with respect to the high threshold (HT) and low threshold (LT) sensor readings. Based on these readings, the controller either *activates* or *deactivates* the pump. When the pump is active, its motor is switched *on*, which subsequently *increases* the water level in the tank. On the other hand when the pump is deactivated, its motor is switched *off* which then allows the water level in the tank to gradually *decrease*.

In addition, the controller interacts with a *drain* component that is introduced as a safety control structure. In the event of a component failure, the controller may *activate* or *deactivate* the drain, which subsequently *opens* or *closes* an exit valve. This

exit valve is located at the base of the water tank, at the low limit (L). When the exit valve is open, the water level is reduced to the low limit.

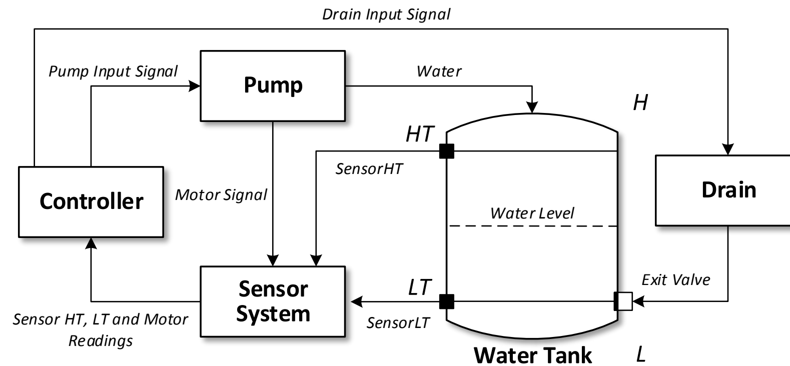


Figure 2.1: A description of the water tank system.

2.1.1 Problem Frames

The Problem Frames approach was introduced by Jackson [52] in 1995, and a fuller and more systematic representation of problem frames can be found in his later book "*Problem Frames: Analysing and Structuring Software Development Problems*" [53], in 2001. It provides a framework that allows a requirement to be viewed as a *problem* in a real-world context for which a *solution*, i.e. a software specification, is sought. The process of software development is then regarded as a *problem-solving* process that eventually leads to a solution that satisfies the requirement in its context [53].

This approach makes a clear distinction between the *solution* (the machine being built) and its *problem* (the requirement). The *world* between the machine and the requirement are represented by *domains*. These concepts are represented graphically in what is called a *problem diagram* [53]. Figure 2.2, provides a problem diagram for the water tank system, where the water tank system (controller) is represented as the *machine* to be built (rectangle with double lines) and a requirement to maintain the water level below the high (H) limit, is captured as a *problem* (dashed oval). The world between the machine and the requirement, i.e. sensor system, pump and water tank are introduced as *domains* that have annotated interfaces between each other and the machine. The annotated connections between domains indicate shared phenomena, including events, operations and state information. The requirement is introduced as a constraint on the water tank (domain) via the dashed arrow-headed line. Often, the requirements provided by the stakeholder specify constrains on the environment rather than on the machine.

As part of the *problem-solving* process, Jackson [53], provides an outline for the following techniques:

Problem Patterns The complexity of a problem is reduced to fit *elementary problem frames* [53]. Jackson introduces the *Work Pieces*, *Required Behaviour* and *Information Display*, problem classes. For instance, Figure 2.2, is a required behaviour frame where the machine, i.e. the water tank system, is required to impose a particular behaviour on a controlled domain, the water tank. These elementary frames give rise to *frame concerns* that are associated with the different forms.

Frame Concerns This can be regarded as loosely analogous to the operational principle of a device class in normal design, i.e. how the characteristic parts of the device fulfil their special function in combining an overall operation which achieves the devices purpose. By addressing the frame concern [53] for each problem frame, the solution is likely to be acceptable.

Problem Progression This is part of the problem-solving process. The requirements are transformed to a specification via the problem progression technique. This technique results in sequence of problem frame descriptions that start with the full description (including the original requirement) and ends with a description containing only the machine and its specification.

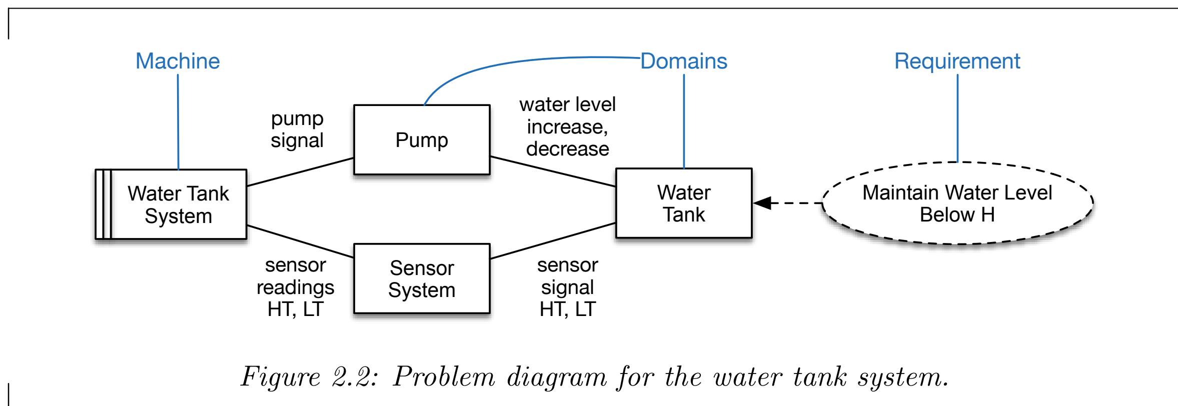


Figure 2.2: Problem diagram for the water tank system.

Formal Analysis with Problem Frames

Seater and Jackson [97] have presented a technique for obtaining a specification from a requirement through a series of incremental steps. This technique is similar to the progression technique presented by Jackson. However, as the requirement is moved towards the machine, a trail of "breadcrumbs" is left behind. These breadcrumbs are

partial domain descriptions representing assumptions about the behaviours of those domains. Each step is justified by a mechanically checkable implication, ensuring that, if the machine obeys the derived specification and the domain assumptions are valid. The technique is formalized in Alloy [50].

Nelson et al. [85] presents an approach where the descriptions of the problem domains, machine and requirements are written in the Alloy language. The approach enables automated formal analysis to reason about problem frame concerns. Their analysis provides an evaluation of results and counterexamples provided by a model finder that is aimed to help remove inconsistencies as well as composition errors.

Gmehlich et al. [45] provide a report on an industry experience in the use of Problem Frames to represent and trace informal requirements to a formal Event-B model. The article presents an experiment carried out at Bosch to develop a model of a cruise control system.

Representation of Failures in Problem Frames

As the causes of failures are typically rooted in the complex structures of software systems and their world contexts, the problem frames framework have been used to investigate areas in the system structures where failures are likely to occur [76, 105].

In [105], Tun et al. describes the use of problem frames as a means to investigate the role of software systems in the power blackout that affected parts of the United States and Canada on 14 August 2003. Their work identified safety-related *concerns* that were related to problem-patterns in problem frames. These concerns, *reminder concern*, *system precedence concern*, *outdated information concern*, *failure concern*, raise a number of specific issues that must be addressed if the solution is to be acceptable.

Lin et al [76] introduces *abuse frames* in problem frames to analyse security problem in order to determine security threats and vulnerabilities. They consider threats to a problem frame from the point of view of an *attacker*. Abuse frames can provide a means for bounding the scope of and reasoning about security problems in order to analyse security threats and identify vulnerabilities.

Limitations with Problem Frames

Problem frames provides an advantage by requiring all descriptions to be grounded in the real world, that is, be as faithful as possible to reality. The problem owners, i.e. stakeholders with requirements, usually do not have expertise in the computing machine but have experiences or expertise in the application domains. Problem frames allows the basis of communication with domain experts and users to be in a language that they can understand [93]. However, the notations for problem frames with comparison to

other requirements engineering techniques are less familiar to practitioners. In addition, there exists a gap between problem frames to other commonly used design languages such as UML. This introduces challenges in the use of problem frames in the systems development process.

2.1.2 Goal-Oriented Requirements Engineering

There are two approaches for Goal-Oriented Requirements Engineering (GORE), namely, the KAOS approach (Keep All Objects Satisfied) [109] and the i* appraoch [116]. Goal-oriented approaches have gained popularity in requirements engineering as they are useful in providing guidance in acquiring requirements, and relating requirements to organisational and business context. They also play a role in identifying and dealing with conflicts and driving design [115].

In goal-oriented approaches, requirements are expressed as *goals*, which may range from high-level goals (e.g., strategic concerns within an organisation) down to low-level operational goals (e.g., technical constraints on the software agent or particular concerns on the environment agent). Therefore, goal refinement can be seen as a form of requirement transformation. The definition of a goal is given by van Lamsweerde in [107], as: "*an objective the system under consideration should achieve. Goal formation thus refers to intended properties to be ensured, they are optative statements as opposed to indicative ones, and bounded by the subject matter*".

Software specifications are then derived from the subset of operational goals which are assigned as responsibilities to *agents*. These agents may for example represent humans, devices, or software.

The KAOS Approach

The KAOS [106] method is comprised of five core models: goal model, object model, agent model, behaviour model, and operation mode. These are used for modelling and structuring requirements. This background will only address the the goal model, which is the starting points for KAOS where *goals* are identified. The goal model has a two-level structure: the *outer graphical semantic layer* and the *inner formal layer*. The outer layer shows semi-formal relationships among goals. The inner layer formally defines goals and their relationships. The formal layer of KAOS is based on Linear Temporal Logic (LTL) [90].

The goal model of KAOS is in the shape of a tree. For instance, the goal model for the water tank system can be seen in Figure 2.3, where each goal can be viewed as high-level requirements. The tree consists of a refinement graph expressing how higher-level

goals are refined into lower-level ones and, conversely, how lower-level goals contribute to higher-level goals. A parallelogram denotes a goal. In the refinement graph, a node represents a goal which is either an *achieve goal* or a *maintain goal*. The maintain goals prescribe behaviours where some target properties must be permanently satisfied in every future state. The achieve goals prescribe system behaviours where some target properties must be eventually satisfied in the future.

An AND-refinement is used to link and relate parent goal to a set of sub-goals. A parent goal must be satisfied when all of its sub-goals are satisfied. The relationship between a parent goal and the set of its sub-goals is called *goal refinement*. The KAOS approach uses logic to support reasoning about goal refinement with some patterns and tool support, such as GRAIL [28]

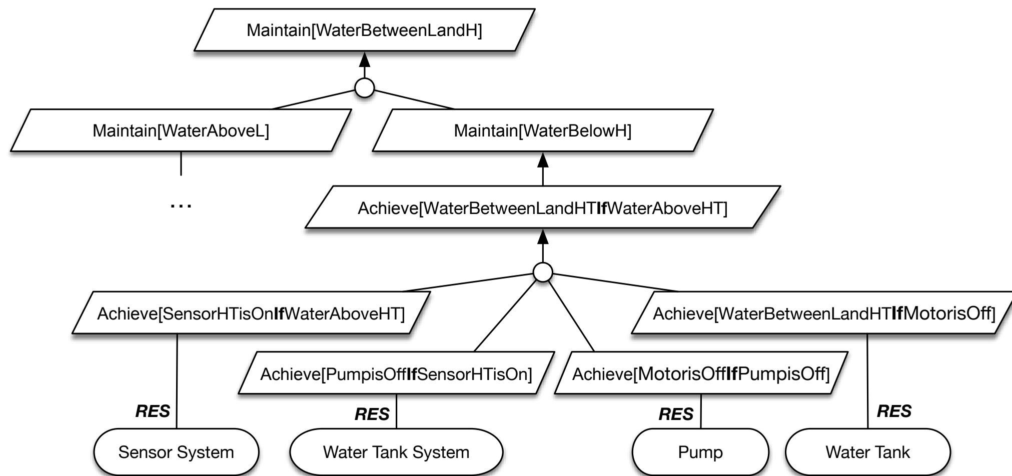


Figure 2.3: A goal model for the water tank system.

Obstacles

The goals, requirements and assumptions about agent behaviour are often too ideal, where some of them are likely not to be satisfied from time to time in the running system due to unexpected agent behaviour [106, 108]. The concept of *obstacle* is introduced in [108]. Obstacles are a dual notion to goals; while goals capture desired conditions, obstacles capture undesirable (but nevertheless possible) ones. An obstacle obstructs some goal, that is, when the obstacle becomes true then the goal may not be achieved.

The obstacles have been introduced to counter the lack of anticipation of exceptional behaviours that results in unrealistic, unachievable and/or incomplete requirements. This is aimed to prevent poor performance or failures in the software developed from those requirements that are considered both goals and obstacles.

The i* Approach

The i* framework has been developed for modelling and reasoning about organisational contexts and their information systems. It has two major modelling components: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. SD describes the dependency relationships among actors in an organisational environment; SR describes stakeholder interests, concerns, and how they may be addressed by various configurations of systems and environments [116]. The framework is used in contexts where there are multiple parties with strategic interests that may be reinforcing or conflicting each other.

The starting point of the i* approach is usually far away from the computing machine. Unlike KAOS, the primary focus of i* are soft goals [24], that is, the so-called non-functional requirements. Since this approach focuses on soft goals, some global non-functional property requirements such as security, usability, performance or flexibility can be expressed as goals for refinement.

2.1.3 UML Use Cases

lvar Jacobson [54] introduced the concept of use cases in the context of his work on large telecommunication systems. The behaviour of such systems is complicated and can be analysed at many levels. To manage the complexity, Jacobson had the idea of describing the desired behaviour of a system by telling a story from the point of view of a user [9]. Such a story called a scenario supported by subsidiary scenarios and associated information, he called a *use case*.

The use case concept was quickly understood to be useful, and was adopted freely, especially within object-oriented software engineering. Use cases now form one of many techniques included in the Unified Modelling Language (UML) [19]. Use cases is a widely used requirements analysis technique and is not restricted to only object-oriented software engineering. It has also been used for hardware-software systems, as indicated by Jacobson's original work [54].

Concepts

In requirements engineering, UML use cases [19] have been used as a means to capture the desired behaviour of systems, i.e., what systems are supposed to do. The key concepts for UML use cases [19] are *actors*, *use cases*, and *subject*. A subject represents a *system under consideration* to which the use case can be applied to. In the running example, the water tank system is considered as the subject in UML use cases. Users and any other systems that may interact with a subject are represented as *actors*,

i.e. the *Pump*, *Sensor System*, *Water Tank*, and *Drain*. Each use case specifies some *behaviour* that a subject can perform in collaboration with one or more actors. In the water tank system, the desired behaviour to maintain the water level below the H limit could be introduced as a use case, MaintainH. Each of these concepts are defined as follows [19]:

Subject A subject of a use case could be a system or any other element that may have behaviour.

Use Case Each use case specifies a unit of *useful* or *desired* functionality that the subject provides to its users (i.e., a specific way of interacting with the subject). This interaction must always be completed for the use case to be considered "complete".

Actor An actor models a type of role played by an entity that interacts with the subjects of its associated use cases (e.g., by exchanging signals and data). Actors may represent roles played by human users, external hardware, or other systems.

These behaviours, involving interactions between the actors and the subject, may result in changes to the state of the subject and communications with its environment.

Relationship: Extends

An *extend* [19] is a relationship from an extending use case (in this thesis we refer to this as an *extension use case*) to an extended use case (a regular use case or even an *extension use case*) that specifies how and when the behaviour defined in the extending extension use case can be inserted into the behaviour defined in the extended use case. The extension takes place at one or more specific *extension-points* defined in the extended use case.

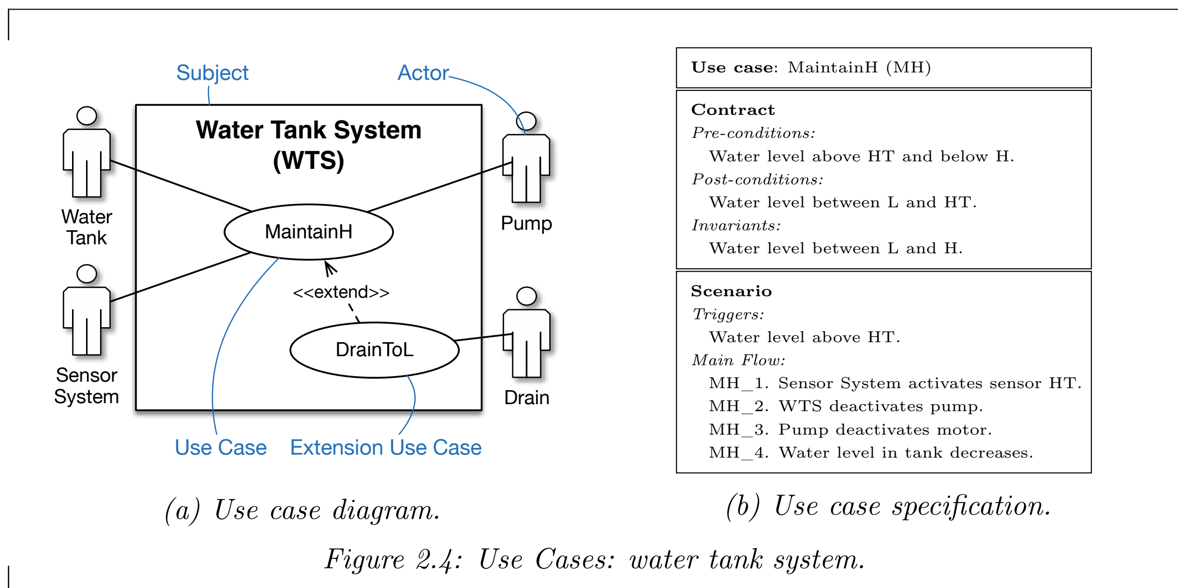
Extend is intended to be used when there is some additional behaviour that should be added, possibly conditionally, to the behaviour defined in one or more use cases. For the water tank system, the functionality to drain the water level to the low limit in the event of a component failure can be introduced as an extension use case, DrainToL, that extends the functionality of the use case MaintainH.

The extended use case is defined independently of the extending use case, and is meaningful independently of the extending use case [9]. On the other hand, the extending use case typically defines behaviour that may not necessarily be meaningful by itself.

Use Case Diagram

The concepts of a use case are illustrated via a *use case diagram* [19]. For the water tank system, its use case diagram can be seen in Figure 2.4a. The Water Tank System is introduced as the subject, i.e. the system under consideration, denoted by the *rectangular box*. The required behaviour for the subject which is to maintain the water level in the tank below the high limit, is introduced as a use case, MaintainH. This is represented as an *oval ellipse*. The external entities, the Pump, Water Tank, Sensor System, and Drain, which interact with the subject to achieve the desired behaviour, are introduced as actors. They are represented as a *stick man* icon. Each actor that plays a role in the use case has a *line* to indicate an association.

An extension use case DraintToL is introduced to MaintainH via the *extends* relationship. An *extends* relationship between use cases in the use case diagram, is shown by a dashed arrow with an open arrowhead pointing from the extending extension use case towards the extended use case. The arrow is labelled with the keyword ⟨extend⟩.



(a) Use case diagram.

Use case: MaintainH (MH)

Contract
Pre-conditions:
Water level above HT and below H.
Post-conditions:
Water level between L and HT.
Invariants:
Water level between L and H.

Scenario
Triggers:
Water level above HT.
Main Flow:
MH_1. Sensor System activates sensor HT.
MH_2. WTS deactivates pump.
MH_3. Pump deactivates motor.
MH_4. Water level in tank decreases.

(b) Use case specification.

Figure 2.4: Use Cases: water tank system.

Use Case Specification

The behaviour of a use case can be further detailed in a *use case specification* [9, 27]. There is no UML standard for a use case specification, but there are proposed templates for documenting use cases [9]. The template used in this thesis takes into account constraints, exceptions and scenarios, as seen in Figure 2.4b. In essence the specification is composed of two main components, *contract* and *scenarios*. The contract specifies the pre-condition, post-condition and invariant properties, as described below:

Pre-condition The writer of the functional requirements and the implementation team can rely upon the preconditions to be established prior to the initiation of the use case, i.e. they are conditions that must be true before the use case executes.

Post-condition The post-condition is used to document conditions that must be *true* after the execution of the use case.

Invariant An invariant condition specifies the conditions that are true throughout the execution of the use case.

A scenario is defined as a sequence of interactions happening under certain conditions, in order to achieve an external actor goal, and having a particular result with respect to that goal (contract) [26]. Scenarios have been a focus in requirements engineering research and practice because they can offer narratives to bridge the communication gap among various stakeholders in a development project.

In the use case specification, the scenario is captured as a sequence of *steps*. It specifies a *trigger condition*, which causes the use case to execute. The difference between the trigger and the pre-condition is that, there is no promise that the trigger will occur, only an indication that these conditions will start the execution of the use case. The use case provides a *main flow* that captures the expected sequence of steps in the interaction between the actor and subject to achieve the goal (contract) of the use case. In the specification of MaintainH use case, the trigger condition to initiate its main flow is for the water level to be above the high threshold limit. The main flow captures a sequence of steps that describes the interactions of the actors and subject to achieve the overall goal.

The difference between pre-condition and trigger is that a precondition is a promise, contract or guarantee while the trigger is the initiator of a use case. The writer of the functional requirements and the implementation team can rely upon the preconditions to be established prior to the initiation of the use case. Trigger is what causes the use case to start (there is no promise that this trigger happens).

2.2 Safety Engineering

Safety engineering is a discipline which assures that engineered systems provide acceptable levels of *safety* [74, 104]. It is strongly related to systems engineering, industrial engineering and the subset system safety engineering. Safety engineering assures that a safety-critical system behaves as needed, even when components fail. A sufficient definition for *safety* for the needs of this thesis, is as follows:

Definition 2.2 (Safety [30]). *An overall mission and program condition that provides sufficient assurance that accidents will not result from the mission execution or program implementation, or, if they occur, their consequences will be mitigated.*

This section provides the background on the early stage analysis of safety, with respect to the identification of accidents, the identification of system-level hazards associated to an accident, along with hazard analysis techniques to determine the cause of a hazard.

2.2.1 Accidents

Accidents or *losses*, are considered early in the development of safety-critical systems [73]. Their identification is part of the safety analysis process and is the first step in any safety effort. The definition of what constitutes an accident varies greatly among industries and engineering disciplines. The one used in this thesis follows the definition provided by Leveson [73], where an accident is defined as:

Definition 2.3 (Accident [73]). *An undesired or unplanned event that results in a loss, including loss of human life or human injury, property damage, environmental pollution, mission loss, etc.*

An accident does not necessarily involve loss of life, but it does result in some form of loss that is unacceptable to the stakeholder. As an example, in the water tank system a potential accident (labelled ExceedH) is as follows:

Water level exceeds high (H) limit in water tank (damage to water tank). (ExceedH)

This accident does not involve loss of life (at least not directly), but there is potential for the water tank to be damaged as a result of this accident. The criterion for specifying accidents is that the losses are so important that they need to play a central role in the design of the system. This accident represent a *safety concern* in the operation of the water tank system. The focus of this thesis, is towards investigating how accidents identified in the safety analysis process can be introduced as constraints on system goals, i.e. the required behaviour, within the requirements analysis process. This is aimed at enabling the safety analysis to *guide* and *limit* the effort of the requirements analysis process.

Once the accidents have been identified, priorities and evaluation criteria may be assigned to the accidents to indicate conflicts between system goals (required behaviour) and safety goals. However, identifying the priorities for accidents and their relation to

conflicts with system goals are outside the scope of this research.

2.2.2 System Hazards

Once the accidents have been defined, a set of high-level *system hazards* can be identified as part of the safety analysis process. In this thesis, the definition of a hazard follows that of System Safety [73], where they are defined as within the system being designed and its relation with the environment. This definition of a hazard is as follows:

Definition 2.4 (Hazard [73])**.** *A system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to an accident (loss).*

The hazard may be defined in terms of conditions or actions. There have been many arguments about whether hazards are conditions or actions, and this distinction is said to be irrelevant as long as one of these choices are used consistently [73]. In system safety, these hazards are defined as something that can lead or result in an accident. Leveson [73] describes the cause of an accident as a combination of a hazardous action performed together with a set a worst-case environmental conditions, as follows:

$$\text{Hazard (Action)} + \text{Environmental Condition (State)} \Rightarrow \text{Accident (Event)}$$

What constitutes a hazard depends on where the boundaries of the system are drawn. A system is an abstraction, and the boundaries of the system can be drawn anywhere the person defining the system wants. Where the boundaries of the system are drawn will determine what actions are considered part of the hazard and the conditions for the environment. For the water tank system, the designer has control over the action to either *increase* or *decrease* the water level in the tank (albeit not directly), via the interaction with the Pump component. Furthermore, it monitors the water level at the high and low thresholds of the water tank via the readings from the Sensor System. For the ExceedH accident, a hazardous action would be for the system to increase the water level in the tank even after the water level has exceeded the high threshold (HT) limit. The cause of the accident for ExceedH can be written as follows:

$$\text{Water level in the tank increases} + \text{Water level above HT} \Rightarrow \text{ExceedH}$$

There are no tools for identifying hazards [104]. It takes domain expertise and depends on the subjective evaluation by those constructing the system. Moreover, there are no right or wrong set of hazards, only a set that the system stakeholders agree is important to avoid [73].

These system hazards that have been identified in the safety analysis, are accom-

panied with safety requirements and constraints necessary to prevent the hazard from occurring. These constraints are often used to guide and limit the system design. For the water tank system, the safety constraint for the identified hazardous action to increase the water level, is constrained by the safety requirements for the water level to always be maintained between the low and high limits of the water tank. That is, the system action to increase the water level in the tank must be shown to be within the constraint of this safety requirement.

It is not sufficient to only identify hazards and safety requirements (constraints), it is also necessary to identify how these hazardous control action may be performed that violate the safety constraints. These hazards at the system level can be related to component failures via hazard analysis techniques.

2.2.3 Hazard Analysis

Hazard analysis can be performed to identify the *cause* of a hazard [73]. In this thesis, the focus is placed on scenario-based modelling of hazards [30]. These establishes a linkage between hazards and adverse consequences (accidents) of interest. This is illustrated in Figure 2.5, where a scenario begins with the identification of an *initiating event* for each hazard along with the necessary *enabling events* that result in undesired consequences. The enabling events often involve the *failure* of or *lack of protective barriers* or safety subsystems (controls). The resulting accident scenario is the *sequence of events* that is comprised of the initiating event and enabling events that lead to the adverse consequences.

Analysing hazards in relation to the enabling conditions, supports activities that involve:

- *Prevention* of adverse accident scenarios ones with undesired consequences, e.g. the water level exceeding the high limit (damage to water tank).
- The promotion of favourable scenarios that may *mitigate* or limit the severity of such consequences.

Fault Tree Analysis (FTA), Event Tree Analysis (ETA) and System-Theoretic Process Analysis (STPA) are hazard analysis techniques that can be used for identifying scenarios for accidents [73, 74, 104]. These techniques are often used to provide a linkage between hazardous control action and the accident. However, FTA and STPA is generally not recommended for scenario-based modelling of hazards, as seen in Figure 2.5, where an accident scenario involves a chronological sequence of events [30]. ETA is more suitable scenario-based modelling of hazards as it is inductive and determines

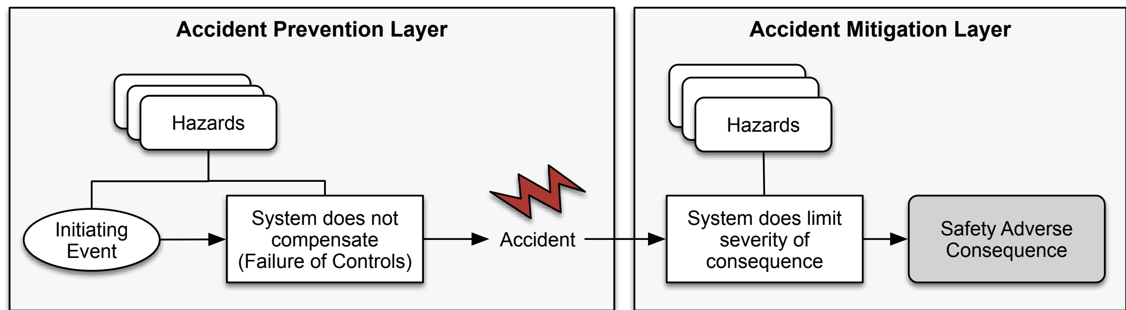


Figure 2.5: Scenario-based hazard modelling framework [30].

an event sequences and its resulting consequences. The following provides an overview of each these hazard analysis techniques applied to the hazard (water level increased while above high threshold) identified for the water tank system.

Fault Tree Analysis

Fault Tree Analysis (FTA) [38, 74, 104] is a top down, deductive failure analysis in which an undesired state of a system is analysed using Boolean logic to combine a series of lower-level events. An event known as the *top event*, is first defined for which *causes* are to be resolved. For the water tank system, the increase of water level above the high limit of the tank is taken as the undesired top event in the fault tree, as seen in Figure 2.6.

This event is resolved into its immediate and necessary sufficient causal events using Boolean logic. This stepwise resolution of events into immediate causal events proceeds until *basic events* (often component failures) are identified. Starting with the undesired (top) event the possible causes of that event are identified at the next lower level. If each of those contributors could produce the top event alone an OR gate is used; if all the contributors must act to result in the top event an AND gate is used. The fault tree explicitly shows all the different relationships that are necessary to result in the top event. The fault tree analysis allows for exhaustive identification in causes of a failure, identify weaknesses in a system, assess a proposed design for its reliability or safety, and more. There are several advantages to FTA: exhaustively identify the causes of a failure, identify weaknesses in a system, assess a proposed design for its reliability or safety, and quantify the failure probability and contributors.

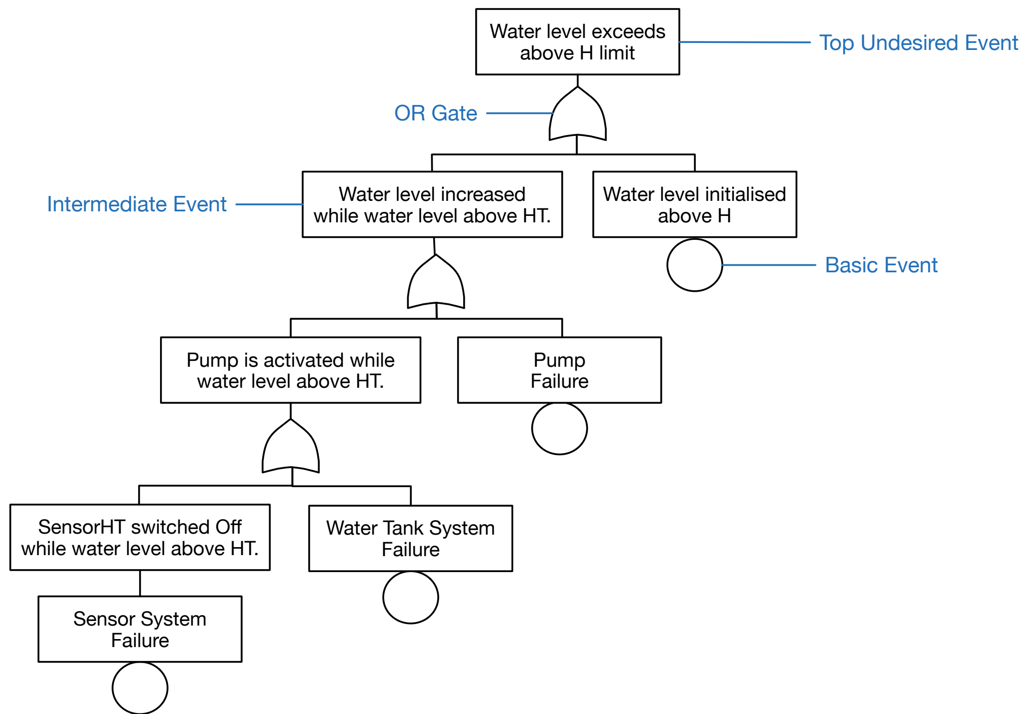


Figure 2.6: Fault Tree Analysis (FTA) of water tank system.

Event Tree Analysis

Event Tree Analysis (ETA) [37, 74, 104] is an inductive, or *forward logic*, technique which examines all possible responses to the *initiating event*, then progressing left to right, identifying the consequences that can result following an initiating event. The potential hazardous trigger event is known as the *initiating event*. The branch points on the tree structure usually represent the *success* or *failure* of different systems and subsystems which can respond to the initiating event.

Figure 2.7 shows a very simple event tree structure for the water tank system. The initiating event is the increase of the water level over the high threshold limit. The branch points then consider the *success* and *failure* of the components in the system, namely, the sensor system, water tank system, pump and finally the water tank. The outcomes determined by the end point of each event tree branch identifies a different consequence following the initiating event.

System-Theoretic Process Analysis

System-Theoretic Process Analysis (STPA) [73] is a relatively new hazard analysis technique that is based on the STAMP causality model. STPA was developed to include new causal factors identified in STAMP that are not handled by older techniques.

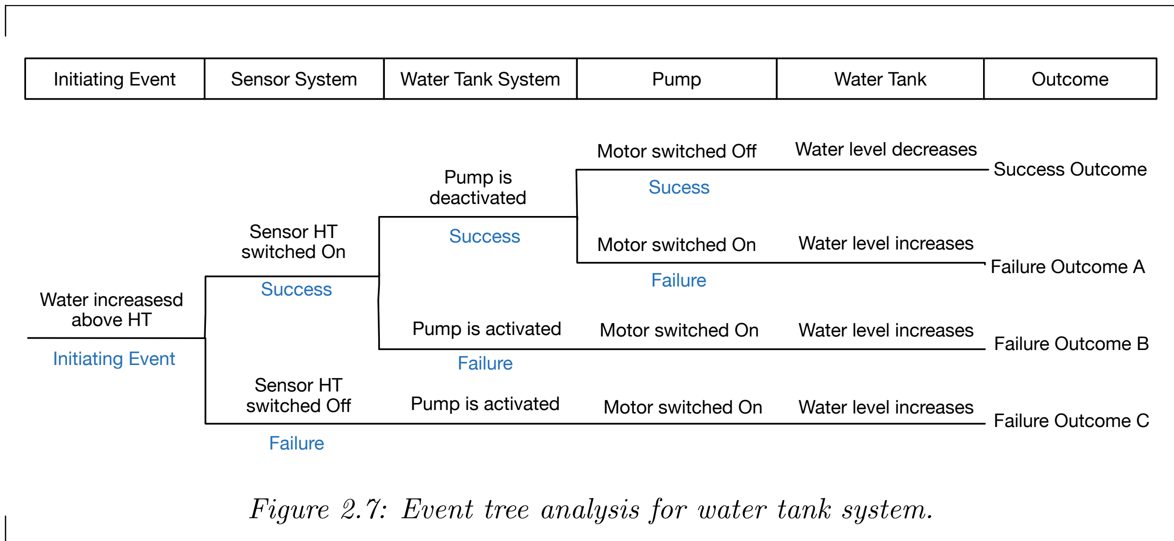


Figure 2.7: Event tree analysis for water tank system.

The STAMP accident model takes into account complex human interactions, software behaviour, design errors and flawed requirements. STPA can be used to identify accident scenarios that describe how a hazardous control action could happen. These accident scenarios are aimed to encompass the entire accident process, not just the electromechanical components. To gather information about how the hazard could occur, the parts of the control loop based on the STAMP causality mode is examined to determine if they could cause or contribute to it.

Figure 2.8 shows the results of the causal analysis of a hazard for the water tank system in a graphical form. The hazard in Figure 2.8, is the increase in water level while the water level is above the high threshold. Looking first at the controller itself, the hazard could occur if the requirement is not passed to the developers of the controller, the requirement is not implemented correctly, or the process model incorrectly shows the water level is below the high threshold when that is not true. Working around the loop, the causal factors for each of the loop components are similarly identified using the general causal factors shown in Figure 2.8.

These causes include: that the command is sent but not received by the actuator; the actuator delays in implementing the command; the commands are received or executed in the wrong order; the increase of the water level above the high threshold (HT) is not detected by the sensor system; there is an unacceptable delay in detecting it; the sensor fails or provides spurious feedback; and the feedback about the state of the water level is not received by the controller.

Once the causal analysis is completed, each of the causes that cannot be shown to be physically impossible must be checked to determine whether they are adequately handled in the design, or design features are added to control them if the design is being developed with support from the analysis. This allows the engineers to design controls

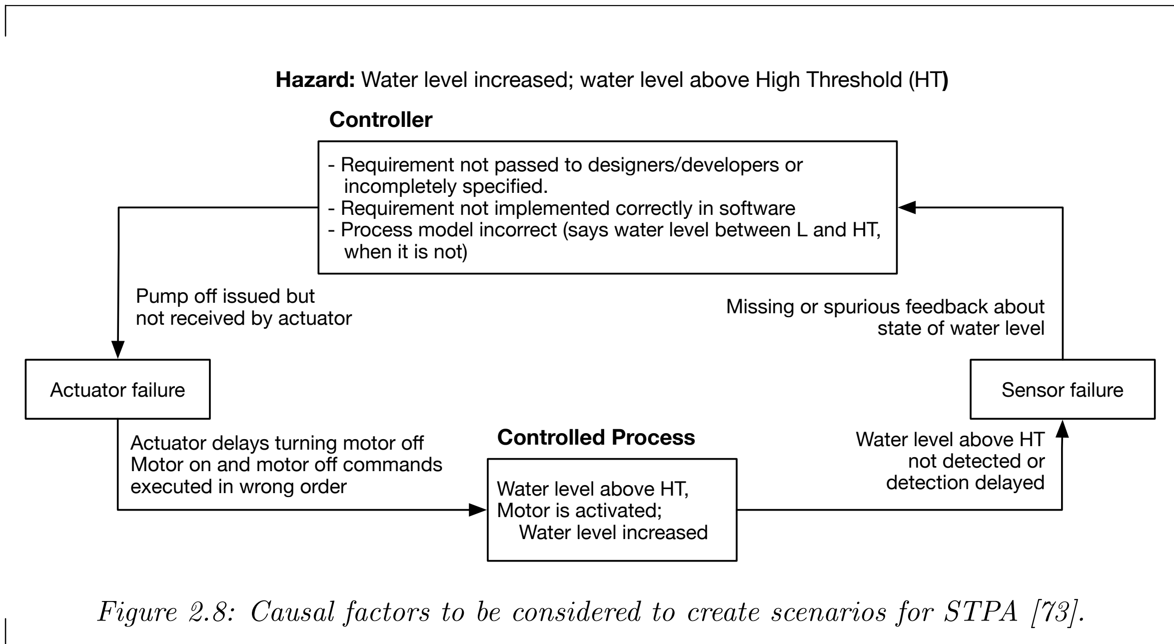


Figure 2.8: Causal factors to be considered to create scenarios for STPA [73].

and mitigation measures if they do not already exist, or evaluate existing measures if the analysis is performed on an existing designs.

2.3 Formal Methods

> *"A formal method is a set of tools and notations (with a formal semantics) used to specify unambiguously the requirements of a computer system that supports the proof of properties of that specification and proofs of correctness of an eventual implementation with respect to that specification."*
>
> – Michael G. Hinchey and Jonathan P. Bowen

The failure of software systems to perform as expected can produce high losses for companies. In safety critical systems this can even result in loss of human lives. Past experiences have shown evidence of the need for high-quality software. For example, the Ariane 5 launcher flight [17] self-destructed after 40 seconds of its launch due to an overflow error when trying to convert 64-bits of data into 16-bits. An investment of over 850 million dollars was lost as a result of this accident. Another case was the Therac-25 [75], a computer-controlled radiation therapy system that overdosed six people resulting in the death of two. The new design of the Therac-25, the successor of the Therac-20, contained errors which caused a failure in the interlocking system and lead to the overdoses.

Formal methods are mathematical rigorous techniques used for the development and verification of software and hardware systems. They complement traditional de-

velopment techniques, increasing confidence about the correctness and reliability of systems. The use of formal methods offers a solution, as it may be used at all stages in the development process from requirements analysis to system acceptance testing. One of the main benefits of using formal methods as a step in the development process of a system is minimising failure risks and costs in the testing phase.

2.3.1 Formal Specification

Formal specifications are mathematical descriptions of systems whose semantics are well defined and that can be subject to formal analysis, i.e. it is possible to reason about their correctness. Abstraction is a key aspect in formal specifications. It is a modelling process that focuses on describing the intrinsic requirements of systems while hiding away implementation details. In other words, a formal specification describes what the system does rather than how it does it. Different types of systems can be described through formal specification; for instance, process algebras like CSP [48] and CCS [80] are used to model concurrent systems and to reason about them via the use of algebraic laws; the Z [113], VDM [61], B [3] and Event-B [1] formalisms are used to specify state-based aspects of systems.

However, the development of high quality and correct models has been identified as a difficult task. In the formal methods survey presented in [114], formal specification was estimated to be the phase with the higher increase in the development time, while in [101] it was reported that choosing the right set of abstractions was the main barrier when writing formal models. As described in [62], techniques such as decomposition and refinement have been developed in order to aid formal modelling. Decomposition allows the verification of a system through the individual verification of its sub-components while refinement enables the gradual verification of systems through the use of incremental steps. The focus of this thesis is on refinement.

2.3.2 Refinement

Refinement is a technique used to model systems at different levels of abstraction. Its main purpose is to handle the complexity of large systems through the gradual introduction of steps that are verified by proof. Starting from an abstract representation of a system, details are added incrementally in the search for a more concrete representation which is closer to implementation. Roever and Engelhardt [29] provide an analogy for refinement as looking through a microscope. The microscope does not change anything, only that some previously invisible parts of the reality are now revealed by the microscope.

Refinement allows us to tackle system complexity. Additionally, refinement can be achieved via two main approaches. Firstly, the rule based approach, which uses predefined rules whose correctness has been previously verified. The most notable example is the technique proposed by Carroll Morgan [99] where a set of basic refinement transformation rules are introduced. Secondly, the posit-and-prove approach, which allows users to explore their own refinements but a formal proof is then required in order to determine the correctness of the steps. Formalisms such as VDM [76], B [3] and Event-B [1] implement this style of refinement. The techniques developed in this thesis are tailored for the posit-and-prove approach. The techniques developed in this thesis are focused on the refinement-based formalism, Event-B. A brief description is given next about some relevant formalisms that are based on refinement.

VDM

VDM (Vienna Development Method) [61] is one of the longest-established formal methods for the development of computer-based systems, introduced by a research group in the IBM laboratory in Vienna in the 1970s. It has grown to include a group of techniques and tools based on a formal specification language - the VDM Specification Language (VDM-SL) [60]. Use of VDM starts with a very abstract model and is developed into an implementation. Each step involves data reification [63], then operation decomposition. Data reification develops the abstract data types into more concrete data structures, while operation decomposition develops the (abstract) implicit specification of operations and functions into algorithms that can be directly implemented in a computer language of choice.

VDM has been extended to VDM++ [32], which supports the modelling of object-oriented and concurrent systems. VDM has been widely used in the industry; one of its most recognised applications is the development of compilers, in particular the first European Ada compiler [23]. Overture [69] is a tool that support developing and analysing VDM models.

Z

In 1977, Abrial proposed Z [113] with the help of Schuman and Meyer, it was developed at Oxford University. The Z notation is based on mathematical constructs used in set theory and first order predicate logic. The state of a system in a Z specification is represented by global variables; predicates are used to express the types of variables as well as invariants, and operations are structured through schemas. Refinement is possible in Z via ZRC [22]. Z has also been extended to allow the specification of complex systems by introducing object-oriented constructs and notions such as classes,

inheritance and polymorphism [67]. Tool support is also available for the development of Z specifications; this includes test case generation tools, model checking, animation and type-checkers, among others.

B-Method

The B-Method (also known as classical B), was originally developed by Abrial [3] in the mid 1980s. The B-Method is a model-based method for formal development of computer software systems. A B specification is composed of variables, which describe the state of the system, invariants, which describe properties of the variables that must always hold, and a set of operations, which define changes in the state. B specifications are built by means of refinement of abstract machines. An abstract machine specifies the basic requirements of the system and is subsequently refined all the way to implementation via refined machines, which refine an abstract or a refined machine; and an implementation machine, which represents the last model from which code can be automatically generated. The verification of B developments is achieved through the generation of proof obligations, which are used to check the correctness of the model against the invariants and the consistency between different levels of refinement.

Compared to Z, B is more focused on refinement rather than just formal specification. In particular, there is better tool support such as Atelier-B [70]. These tools support two main proof activities: (1) consistency checking, shows that invariants are preserved by machine operations, and (2) refinement checking, which proves the validity of each refined machine. The B method has been successfully applied to industrial projects, one of the most successful applications is its use in the development of Line 14 of the Paris metro [2].

Event-B

Event-B [3] is a formalism used for the modelling of discrete event systems. An Event-B development is structured into models and contexts. A context describes the static part of a system, i.e. constants and their axioms, while a model describes the dynamic part; i.e. variables, invariants and events. Event-B promotes refinement- based formal modelling, where each step of a development is underpinned by formal reasoning. That is, each refinement step generates proof obligations that must be discharged in order to prove the correctness of the step.

Event-B is an evolution of the B-method [3], and it builds upon the Action System formalism [13]. It has a same structure as an action system which describes the behaviour of a reactive system in terms of the guarded actions that can take place during its execution. Event-B is different from B-Method in some aspects. The B-Method is

organized in a way that is suitable for the development of non-concurrent programs, whereas Event-B is geared towards the development of systems including reactive and concurrent systems. A detailed description of Event-B is provided in Section 2.4.

2.4 Event-B

2.4.1 Structure and Notation

Event-B is a formalism that is used for the modelling of discrete event systems [1]. As seen in Figure 2.9, an Event-B development is composed of a collection of *context* and *machine* components. The context component models the *static* aspects of a system while the machine models the *dynamic* aspects. Contexts provide a means to state static properties of an Event-B model, whereas machines provide behavioural properties of an Event-B model. Items of machines and contexts are called modelling elements, and are presented in this section. There are various relationships between contexts and machines. A context can be *extended* by other contexts and *seen* by machines. A machine can be *refined* by other machines and can see contexts as its static part.

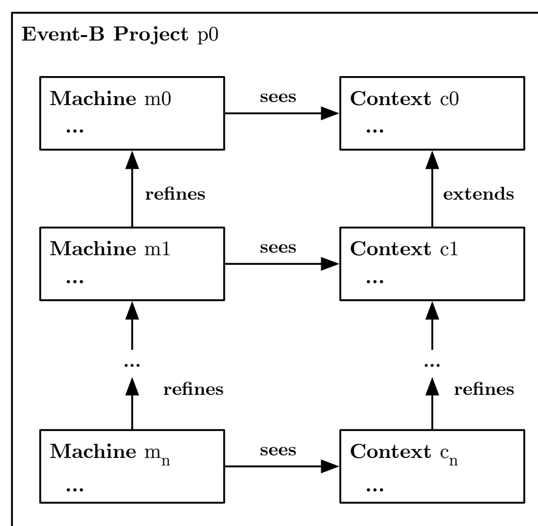


Figure 2.9: Event-B project component structure and relationships.

Context

The modelling elements of a context are from four types: *sets*, *constants*, *axioms*, and *theorems*. It is illustrated in Figure 2.10a. Axioms are predicates that describe

the properties of sets, constants and theorems. A context can extend more than one context, and can also be seen by several machines in a direct or indirect way. By indirect, we mean that a context may be referenced by a machine whose abstract machines sees that context. Theorems list the various *theorems* which have to be proved within the context.

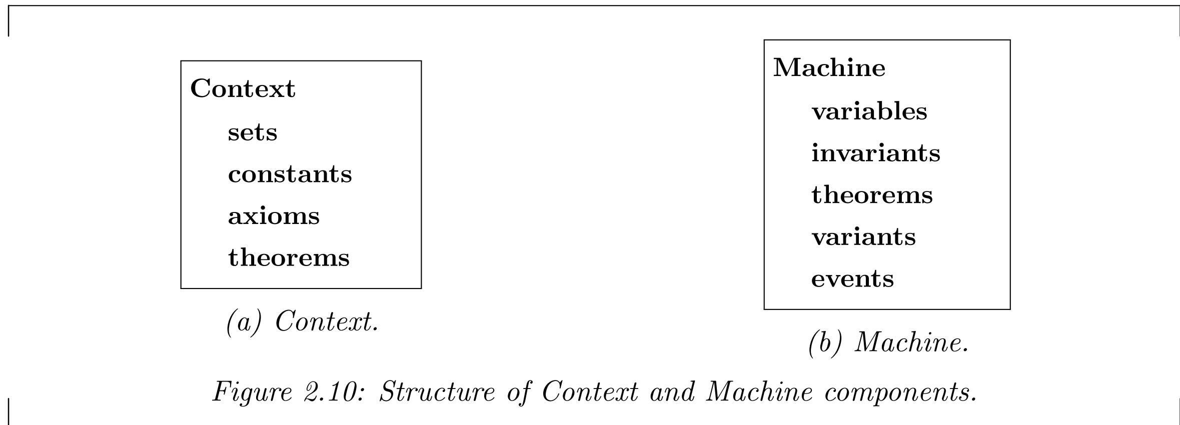


Figure 2.10: Structure of Context and Machine components.

Machine

A machine consists of variables, invariants, events, theorems and variants, illustrated in Figure 2.10b. Variables v define the state of a model. Invariants, $I(v)$, constrain variables, and are supposed to hold whenever variables are changed by an event. In Event-B the state of a model is changed by means of an *event* execution. An event, in its simplest form, is composed of a name, a list of named predicates, the guards, collectively denoted by $G(v)$, and a generalized substitution denoted by $S(v)$. All events are atomic and can be executed only when their guards hold. When the guards of several events hold at the same time, then only one of those events is chosen non-deterministically to be executed. An event E with guards $G(v)$ and generalized substitution $S(v)$ can be given the syntactic form:

$$\mathsf{E} \;\widehat{=}\; \textbf{\textit{when}}\; G(v)\; \textbf{\textit{then}}\; S(v)\; \textbf{\textit{end}}$$

As seen Figure 2.11, there are three kinds of generalized substitutions for expressing the transition associated with an event: (1) the deterministic substitution, (2) the empty substitution, and (3) the non-deterministic substitution.

In the deterministic and non-deterministic cases, x denotes a list of variables of v which are all distinct. In the deterministic case, $E(v)$ denotes a number of set-theoretic expressions corresponding to each of the variables in x. In the non-deterministic cases, there are two constructs by the means of the operator $:\mid$ and $:\in$. The first one is to be

Kind	Generalised Substitution
Deterministic	$x := E(v)$
Empty	skip
Non-deterministic	$x :\mid P(x', x, y)$
Non-deterministic	$x :\in S(v)$

Figure 2.11: Kinds of generalised substitutions.

read, x becomes such that the before-after predicate $P(x', x, y)$ holds, where x denotes some distinct variables of v, y denotes those variables of v that are distinct from x, and x' denotes the values of the variables x after the substitution is applied. The second one is to be read as x becomes a member of the set $S(v)$.

2.4.2 Refinement in Event-B

In an Event-B development, rather than having a single large model, it is encouraged to construct the system in a series of successive layers, starting with an abstract representation of the system. The abstract model should provide a simple view of the system, focusing on the main purpose and key features of *what* the system achieves. The details of *how* the purpose is achieved are added gradually via step-wise refinement. This process is called refinement. Each step creates a more concrete model, which is a refinement of the previous one and must be verified through the use of proof. The semantic of some refinement proof obligations are described in Section 2.4.3.

Types of Refinement

Refinement is the process of enriching or modifying the abstract model in order to introduce new functionality or add details of current functionality. Refinement in Event-B has different views or classifications. From the Event-B notation point of view, refinement of a machine can be classified into the following types:

1. **Refining existing events**:

    (a) Add new guards and actions to the existing abstract event. In this case, the resulting *concrete event* is labelled as *extended*. In an *extended* event, the existing guards and actions can not be modified.

    (b) Modifying guards and actions of the existing abstract event: in this case the resulting concrete event is labelled as *not extended*. Adding new guards and actions are allowed too.

In both these types, the guards of the concrete event must be proved to be stronger than its abstraction.

2. **Adding new events**: The new event refines an event in the abstraction which does nothing (*skip*).

3. **Adding new variables and invariants**:

   - *New Variables*: introducing new variables usually results in 2 or 1 types of refinement. Sometimes abstract variables can be replaced by new concrete variables. In this case the refinement can result in (1.b). Sometimes variable replacement results in redundant variables.
   - *Gluing Invariants*: a gluing invariant relate the states of the abstract variable to the concrete variables. The invariant of the concrete model including gluing invariants should be preserved by the concrete events.

Each abstract event should be refined by at least one concrete event. One abstract event can be refined by more than one concrete event. This is called *event splitting*. Furthermore, one concrete event can refine more than one abstract event. This is called *event merging*. Another view of classifying refinement is as follows:

- **Vertical Refinement** known also as *data refinement*, makes reference to the refinement of data types, i.e. the transition from abstract data types to concrete data structures. The rationale for the transition is usually specified through gluing invariants as in the Event-B formalism, or retrieve functions as in VDM. The consistency of the transformation is verified by proving that the concrete operations preserve that rationale.
- **Horizontal Refinement** refers to refinement steps in which new requirements or more detailed functionality are introduced into the model. The correctness of each step is verified by proving that the behaviour at the concrete level is consistent with the behaviour at the abstract level.

2.4.3 Proof Obligations

Event-B developments are verified through the use of Proof Obligations (POs). A PO is a sequent of the form:

$H \vdash G$

where H represents the set of hypotheses and G represents the goal to be proved. There are different proof obligations which are generated by the Event-B tool Rodin [4] during the development of a system using Event-B. The Rodin tool is discussed in Section 2.4.4. In Figure 2.12, an overview of the types of proof obligations is provided.

As an example, considering Figure 2.13, a machine m1 refines machine m0. Both machines *see* the context *c0*. m1 contains two events, evt3 as a new event and evt2 that is introduced as a refining event. This machine contains some gluing invariants glue_inv. The following describes some of the proof obligations generated for this Event-B model:

Proof Obligation	**Label**	**Description**
Well-definedness	x/WD	x is the name of axiom, theorem, invariant, guard, or action.
Invariant Preservation	evt/inv/INV	evt is the event name, inv is the invariant name
Feasibility of a non-deterministic event action	evt/act/FIS	evt is the event name, act is the action name
Guard Strengthening	evt/grd/GRD	evt is the concrete event name, grd is the abstract guard name
Action Simulation	evt/act/SIM	evt is the concrete event name, act is the abstract action name
Natural number for a numeric varaint	evt/NAT	evt is the new event name.
Decreasing of Variant	*evt*/VAR	*evt* is the new event name.

Figure 2.12: Proof Obligations: Name, label and description.

Well-definedness (WD) Ensures that a axiom, theorem, invariant, guard, action, variant is indeed well defined. For instance, to compute cardinality of a set, $card(S)$, it has to be proved that the set S is finite.

Invariant Preservation (INV) Ensures that each invariant is preserved by each event. For instance in Figure 2.13, one of generated proof obligation in the abstract machine m0 is evt1/inv_m0/INV. This ensures that inv_m0 is preserved by the state transition of event, evt1.

Feasibility (FIS) Ensures that each non-deterministic action is feasible. In Figure 2.13, for event evt1 in machine m0, the proof obligation evt1/act_evt1/FIS is generated. It means there should exist values for variables v_m0 such that the assignment act_evt1 is feasible.

Guard Strengthening (GRD) Ensures that each abstract guard is no stronger than the concrete ones in the refining event. As a result, when a concrete event is enabled, the corresponding abstract event is also enabled. For instance, in the model evt2/grd_evt1/GRD ensures that abstract guard grd_evt1 is weaker than the guards of the concrete event evt2.

Simulation (SIM) Ensures that each action in a concrete event simulates the corresponding abstract action. When a concrete event executes, the corresponding abstract event is not contradicted. In Figure 2.13, the simulation proof is evt2/act_evt1/SIM.

Numeric Variant (NAT) Ensures that under the guards of each convergent event, a proposed numeric variant is indeed a natural number. The PO evt3/NAT is the proof obligation generated for the machine m1 in Figure 2.13.

Decreasing of Variant (VAR) Ensures that each convergent event decreases the proposed numeric variant. As a consequence, the new event does not take control forever. evt3/VAR in Figure 2.13, ensures that event evt3 does not take control forever.

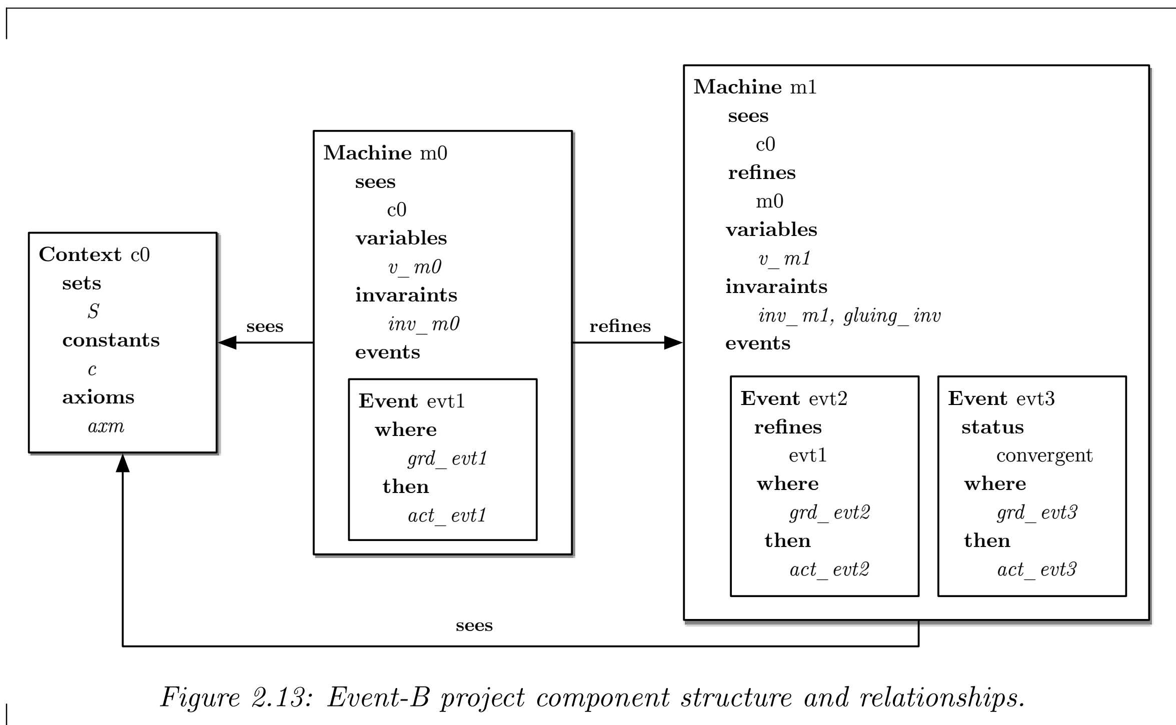


Figure 2.13: Event-B project component structure and relationships.

2.4.4 Rodin

Rodin [4] is a platform implemented on top of the Eclipse environment for the development and verification of Event-B specifications. Rodin allows a developer to reason about a model by giving instant feedback about its correctness. This is achieved by automatically generating and discharging POs, which allows the integration of reasoning as part of the modelling task during the development of Event-B models. Furthermore, discharging POs may not always be automatic; depending on the model, user interaction may be needed to discharge a PO. The close interplay between modelling and reasoning provided by the Rodin toolset facilitates the identification of problems when a PO fails to verify. It is important to stress that Rodin is not used to run programs but to reason about models at the design stage. The Rodin tool chain is composed of three main components:

A Static checker (SC) That analyses a model developed in Event-B in order to find syntax and type errors.

A Proof obligation generator (POG) That automatically generates the POs that must be verified for a given Event-B model. The different type of POs associated with Event-B were described in Section 2.4.3. The POG does not perform proofs, it only carries out simple rewritings within a PO sequent.

A Proof obligation manager (POM) That handles the POs' status as well as the associated proof tree for each PO. It works automatically alongside the automatic Rodin provers, or interactively with the user and external provers. As all POs are represented as sequent in predicate calculus, different external provers for predicate calculus can be used within Rodin.

Rodin provides similar functionalities to those provided by tools used for programming, in which tasks are performed automatically in the background. This facilitates and improves the modelling experience for Rodin users. Among the characteristics provided by Rodin are: (1) instant feedback when a change has been made to the model, i.e. syntax errors, inconsistent types, etc.; (2) automatic generation and verification of POs when a model is saved to the disk (no need of compilation processes); (3) error traces; (4) management of a schema of colours for reserved words (which make the models more readable); (5) templates for the creation of Event-B basic elements; i.e. events, variables, etc.

As Rodin is built on the Eclipse platform, new functionalities can be provided through the addition of plug-ins. This flexible architecture contributes to the improvement and extensibility of the tool as well as to the formation of a bigger community

working around Event-B. We mention some of the plug-ins available in Rodin which illustrate different aspects of the tool-set that have been extended:

UML-B [100] Is a graphical front-end for the modelling of Event-B systems as UML-like diagrams. Currently, it contains support for modelling and refinement of systems with class and state machine diagrams.

ProB [71] provides animation and model checking capabilities for Event-B models.

ProR [58] provides requirement traceability between an Event-B model and the natural language requirements associated to the model.

Currently the development of new plug-ins for the Rodin platform is growing. As part of this research, the development of a plug-in, UC-B, in the Rodin platform is provided in Chapter 6.

2.5 Summary & Discussion

This chapter has introduced the key concepts and the preliminary is for this thesis. An overview of the popular requirements engineering techniques have been discussed along with their comparison. The use case modelling is discussed with the water tank example, which is used in later section to describe the formalisation of use cases. The required background on safety engineering was provided, that described the early stage analysis in the identification of accidents, its relation to hazard and hazard analysis techniques. An overview of the different formal methods and their comparison to Event-B is provided. The target formal method Event-B; the verification support provided by Event-B to achieve reasoning of the behaviour specified by the use cases. The focus in the background provides some of the limitations of use case modelling. The next chapter shows how potentially bad behaviour is taken into consideration by extended the use case model.

Chapter 3

Accident Cases

"Safety is a system property, not a component property, and must be controlled at the system level, not the component level."

– Nancy G. Leveson

3.1 Introduction

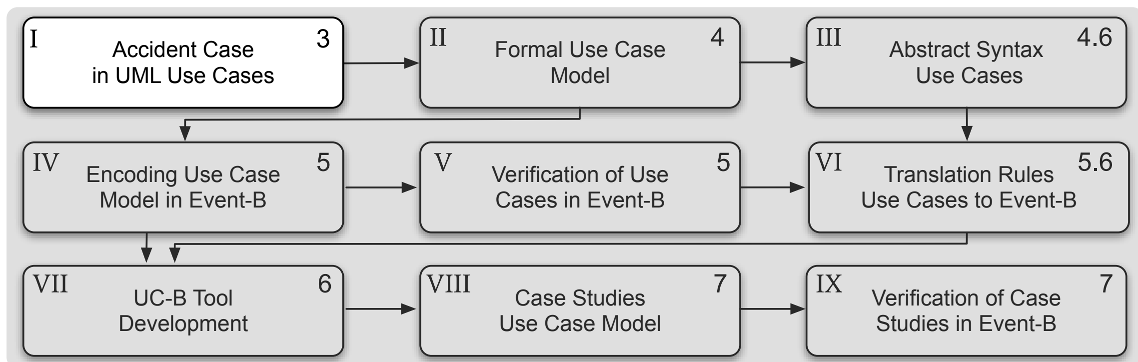


Figure 3.1: Thesis Roadmap for Chapter 3.

Figure 3.1 highlights which part of the roadmap this chapter implements. This chapter introduces an extension to UML use cases that allow *safety concerns* to be taken into account during requirements analysis. As discussed in Chapter 1, a majority of software-related accidents occur due to requirements flaws [36, 77], particularly due to *incompleteness*. Despite its importance, there is no consensus as to what precisely constitutes *completeness* in a requirements specification, nor how to go about achieving it. Many discussion [88, 92, 94], essentially state that the *"requirements specification is complete if some relevant aspect has not been left out"*. The most appropriate definition

in the context of this thesis is provided by Jaff [57]: *"software requirements specification are complete if they are sufficient to distinguish the desired behaviour from that of any undesired program that might be designed"*. The conclusions presented by Jaff [56] on producing a complete requirements specification, is that it is large, tedious, and that it may be unnecessary, as well. He states that it may be more feasible to perform safety analysis to determine what actions of the software are critical and to use this analysis to guide and limit the requirements specification.

Use cases have proven successful for the elicitation, communication and documentation of requirements. However, there are also problems with use case based approaches to requirements engineering. Typical problems are over-simplified assumptions about the problem domain and a tendency to go prematurely into design considerations [5,26]. As discussed in Chapter 2, a use case typically describes some function that the system should be able to perform. Hence, use cases are good for working with functional requirements, but not necessarily with those that are related to *safety*.

In the development of a safety-critical system, safety requirements are often stated directly by the *safety engineers*, who rather have concerns about what should *not* happen in the system. Use cases, by their nature, concentrate on what the system should do, and have less to offer when describing the *undesired* behaviour. This system behaviour that is undesired is still a behaviour, which could potentially be investigated through use cases. This motivated the extension of use cases with the use case type *accident case*.

The definition of the accident case is based on the concepts that belong to safety analysis, namely: *accidents*, *hazards* and *accident scenarios*.

3.2 Accident Case

As discussed in Section 2.2.1, *accidents* or *losses*, are considered early in the development of safety-critical systems [73]. Their identification is part of the safety analysis process and is the first step in any safety effort. The definition of an accident is provided in Definition 2.3, where it is described as an event that results in some form of loss that is unacceptable to the stakeholder. For example, in the water tank system a potential accident (labelled ExceedH) was identified, as follows:

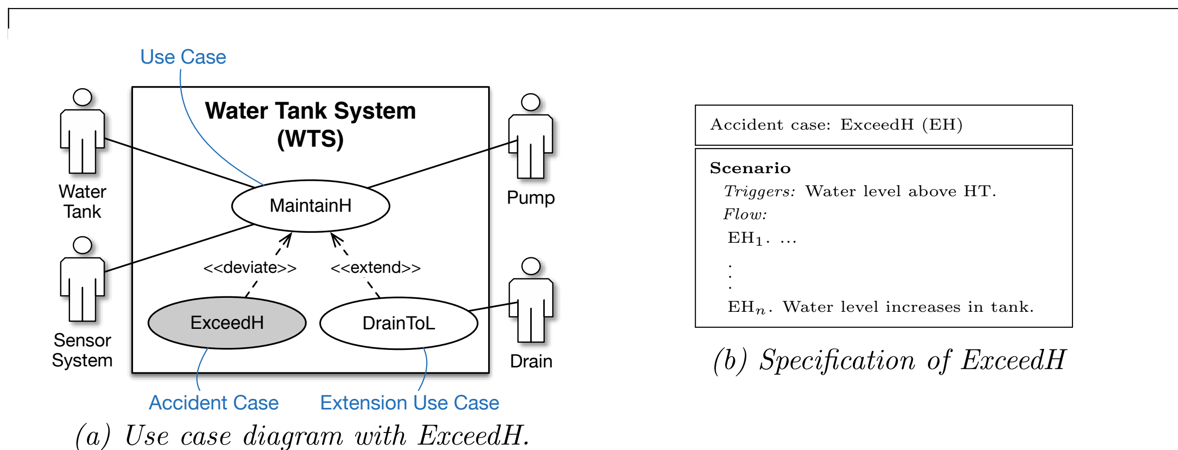
Water level exceeds the high (H) limit in water tank. (ExceedH)

The occurrence of this accident represents a potential for the water tank to be damaged. This can be considered a loss to a stakeholder of the water tank system. UML use cases is extended with the use case type, *accident case*, that allows an accident

identified from the safety analysis to be introduced as an *accident case* along side the existing (regular) use cases. The accident case is defined as follows:

Definition 3.1 (Accident Case)**.** *An accident case is a sequence of actions that a system or other entity can perform that result in an accident or loss to some stakeholder if the sequence is allowed to complete.*

For instance, this identified accident is introduced as an accident case, ExceedH, as seen in the use case diagram of the water tank system, Figure 3.2a. The accident case is denoted by a *shaded* or *grey ellipse*, and is placed within the subject (system under consideration). The name of the accident, or its label, is displayed within the ellipse. The purpose of the accident case is to allow the requirement analysis to take into consideration undesired behaviours identified from the safety analysis that may affect core functionalities of the system. The undesired behaviour of an accident case is introduced as a *deviation* from that of a (regular) use case. That is, during the execution of a use case, the behaviour of the accident case can be introduced as an alternate sequence of steps that represent *undesired* or *unplanned* behaviour. If the sequence of steps in the accident case is allowed to complete, it will result in a state that can be considered as some form of loss to the stakeholder. This deviation from the accident case to the use case is denoted by the directed relationship ⟨⟨deviate⟩⟩ in the use case diagram.



(a) Use case diagram with ExceedH.

Accident case: ExceedH (EH)

Scenario
Triggers: Water level above HT.
Flow:
EH_1. ...
⋮
EH_n. Water level increases in tank.

(b) Specification of ExceedH

Figure 3.2: Use case model for Water Tank System updated with accident case, ExceedH.

What is achieved by the accident case is expected to *violate* what is required to be achieved by the use case it deviates. For example, in the water tank system the complete execution ExceedH will result in the water level exceeding the high limit. This violates what is required by the contract of the MaintainH use case, where the invariant and post-condition that are required to maintain the water level below the high (H) limit, are violated.

So far, the accident case only describes *what* the accident is about without specifying the details of *how* the accident may occur, i.e. its sequence of steps. The definition of a hazard (Definition 2.4) is examined to determine the cause of an accident in Section 3.2.1. The semantic and notation for the accident case and deviate relationship is discussed in Section 3.3.1 and 3.3.2.

3.2.1 Cause of an Accident

As discussed in Section 2.2.2, Leveson [72] describes the cause of an accident with respect to a system-level hazard and worst-case environmental conditions as follows:

$$\text{Hazard (Action)} + \text{Environmental Condition (State)} \Rightarrow \text{Accident (Event)}$$

In this thesis, a hazard is described as an action that, together with a particular set of worst-case *environmental conditions*, results in an accident. What constitutes a hazard depends on where the boundaries of the system are drawn. Use cases establish the actors and system boundary (subject) in the use case diagram which determines what the system has control over. If one expects the systems engineer or designer to create systems that eliminate or control hazards, then those hazards must be in their design space. For the water tank system, the designer has control over the action to either *increase* or *decrease* the water level in the tank (albeit not directly). A hazardous action would be for the water level to be increased in the tank even after the water level has exceeded the high threshold (HT) limit. The cause of the accident for ExceedH can be written as follows:

$$\text{Water level in the tank increases} + \text{Water level above HT} \Rightarrow \text{ExceedH}$$

This hazardous control action and the environmental condition is introduced in the textual specification of the accident case. The hazard is introduced as the *final step* in the scenario of the accident case while the environmental condition is captured as the *trigger condition*. The scenario of the accident case is allowed to execute when the environmental condition is *true*. The execution of the final step in the scenario of the accident case will result in a state that is considered to be an accident. For now, the steps that lead to the final hazardous system action is not known. It is the role of the safety engineer to apply *hazard analysis* in order to determine the *accident scenarios* that may lead to the hazardous control action, in order to result in the accident. This is discussed in Section 3.2.2

For the water tank system, the cause for ExceedH is introduced in the specification

of the accident case, as seen in Figure 3.2b. The environmental condition where the water level is above the high threshold is captured by the trigger condition, while the final step EH_n, captures the hazardous control action where the water level increases in the tank. The steps from EH_1 to EH_{n-1}, that lead to the hazardous action can be identified using scenario-based hazard analysis techniques.

3.2.2 Accident Scenarios

The role of hazard analysis is to identify the *cause* of a hazard [73]. As discussed in 2.2.3, scenario-based hazard analysis techniques [18] provide a means to establish a *linkage* between hazards and adverse consequences (accidents) via scenarios. In the scenario-based hazard modelling framework, an accident scenario is the sequence of events that is comprised of the initiating event, and enabling events that lead to the adverse consequences. Analysing hazards in relation to the above sequence of events, support activities that involve the *prevention* of adverse accident scenarios, ones with undesired consequences.

The background provides an overview of the hazard analysis techniques FTA, ETA and STPA applied on the water tank system. However, it is out of the scope of this thesis to employ the hazard analysis techniques for systematically deriving accident scenarios in the use of the accident case. This is addressed as part of the future work as discussed in Section 8. In the water tank system, a potential accident scenario that may lead to a hazardous action of the water level increasing in the tank, is seen in Figure 3.3. In this scenario, the initiating event is the water level rising above the high threshold (HT). This results in the next two successful events, where the sensor system deactivates the sensor HT, and the controller deactivates the pump. However, in the next event, a failure in the pump component results in the motor remaining switched on, which then subsequently increases the water level in the tank. The complete execution of this scenario is expected to result in the accident ExceedH, where the water level exceeds the high limit in the tank.

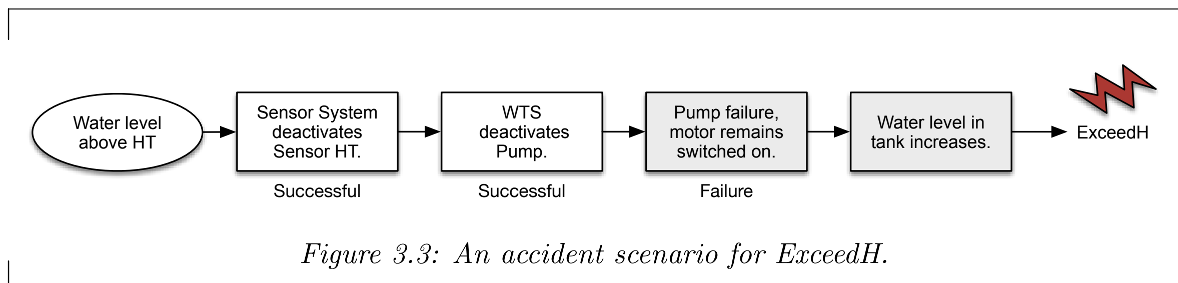
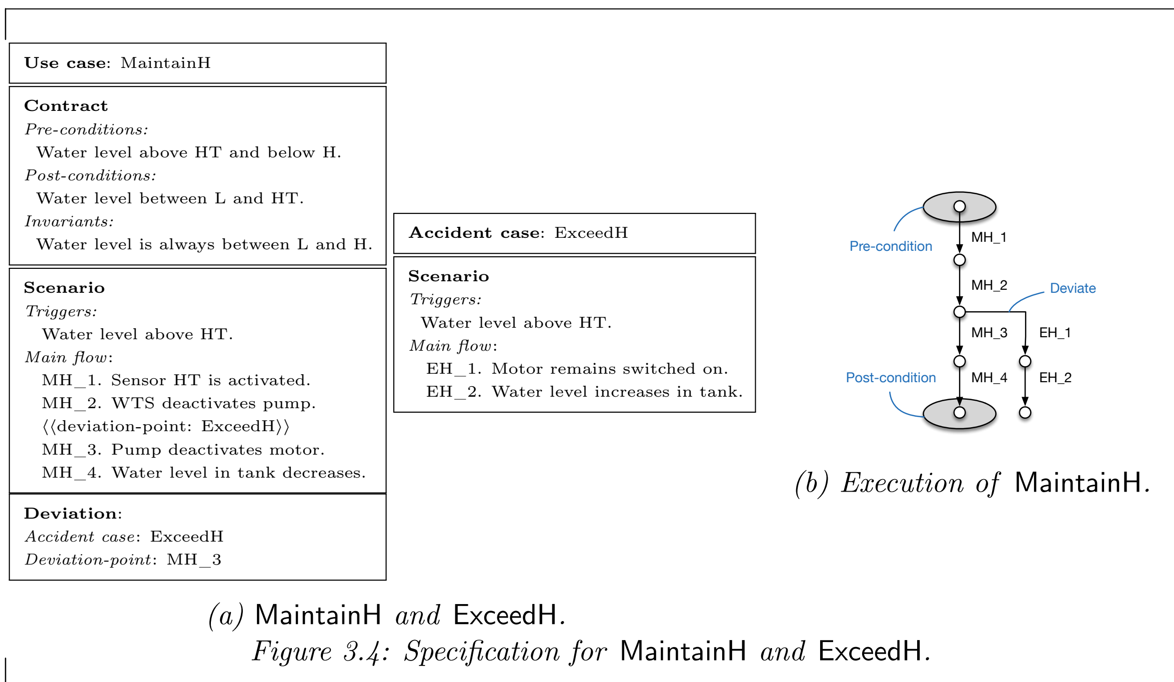


Figure 3.3: An accident scenario for ExceedH.

The deviate relationship from an accident case to an use case, allows the textual specification of the use case to provide a *deviation-point*, i.e. between the steps in the

scenario of the use case, where the accident case can be introduced as an alternative (undesired) route. This allows the accident scenario to only specify the steps that are related to the failure of the system and not repeat the same steps that already exist in the use case it deviates. For example, the specification of the use case MaintainH, provides a *deviation-point* for the accident case ExceedH, between step MH_2 and step MH_3, as seen in Figure 3.4a. The scenario of the accident case is updated with the steps that describe the failure of the pump component, where the motor remains switched on (EH_1) and the water level increases in the tank (EH_2). The deviation-point, allows the execution of the use case scenario to deviate to that of the accident case.

Figure 3.4b provides an informal description in the execution of MaintainH. It shows the execution may deviate to the accident scenario after step MH_2. The execution of the accident scenario for ExceedH will result in the water level exceeding the high threshold, which would not achieve the post-condition and also violate the invariant of the use case MaintainH. The accident case provides a platform for communication undesired behaviours and identify appropriate safety recommendations that could control potential accidents at an early stage in the development process.



(b) Execution of MaintainH*.*

(a) MaintainH *and* ExceedH*.*

Figure 3.4: Specification for MaintainH *and* ExceedH*.*

3.2.3 Safety Guided Design

The accident case derived through safety analysis will place integrity constraints on existing core system function defined by use cases that they deviate. New functional

requirements may be introduced to *prevent* the effects of the accidents identified by the safety analysis. The accident case allows the requirements analysis to consider the desired behaviour of the system with respect to the potential undesired behaviour suggested by the safety analysis. It is aimed to provide a platform for systems and safety engineers to communicate appropriate *design recommendations* that may guide the development of the system with safety as an early consideration.

Allowing UML use cases to analyse accident scenario support activities to prevent adverse accident scenarios. This extension of the accident case, introduces the relationship ⟨⟨prevent⟩⟩. The prevent relationship allows new additional behaviour to be introduced in to a deviated use case that that prevents the deviating accident case from achieving its undesired outcome. The notations and semantics for the prevent relationship is provided in Section 3.3.3.

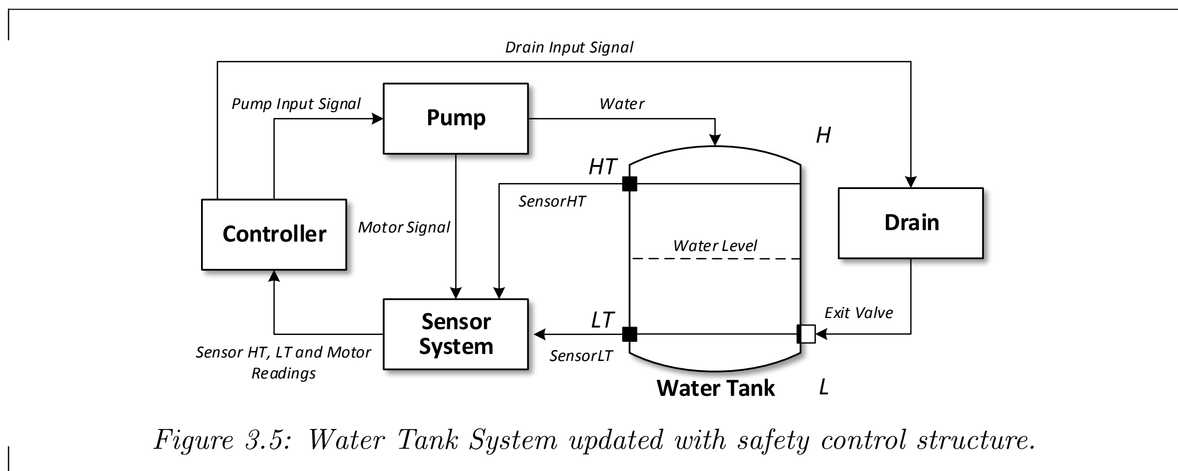


Figure 3.5: Water Tank System updated with safety control structure.

Prevent

Prevent is a directed relationship from a use case to an accident case. It allows the behaviour of the use case to be introduced as additional behaviour in the target accident case. The behaviour introduced by the preventing use case is aimed to limit the severity that results from the execution of the accident case. In the water tank system, in order to strengthen the overall safety of the system against the accidents such as, ExceedH, an additional safety control structure, a drain component, was introduced as a design recommendation, as seen in Figure 3.6.

The water tank system may activate the drain if it detects a failure in the pump component where its motor remains switched on (motor reading from the sensor system) even after the pump has been deactivated. When the drain is activated, it opens an *exit valve* that is located on the low limit (L) of the tank, which subsequently drains the water level to the low limit. The combination of the water level being drained by

the exit value and the undesired increase in water level by the failure of the pump component, is expected to mitigate the accident where the water level does not exceed the high limit (H).

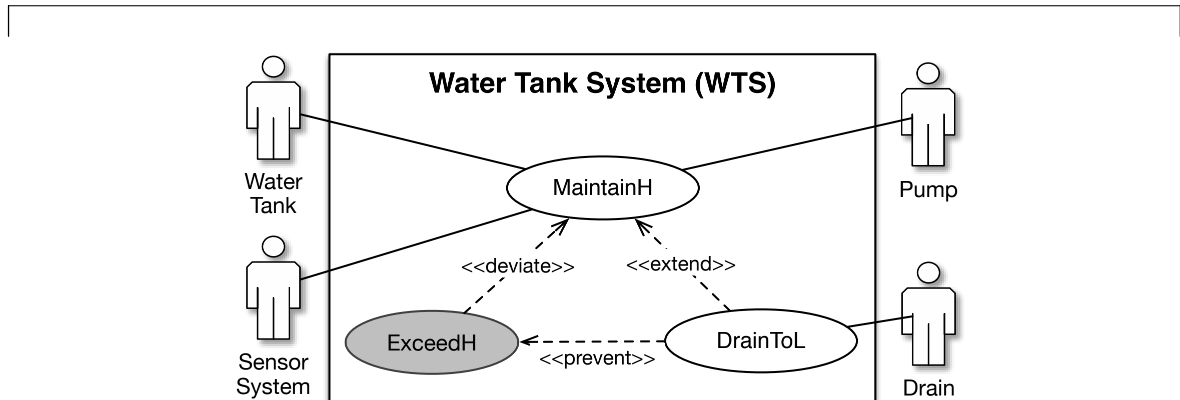


Figure 3.6: Use case development of water tank system with accident case.

In UML use cases, these types of additional or exceptional behaviour are often introduced to the system via an *extension use case*, as discussed in Chapter 2. Extension use cases can be used to describe how a system can respond to when things do not go as expected. An extension use case, DrainToL, is introduced in the use case diagram of the water tank system, as seen in Figure 3.4, that introduces the functionality of the additional safety control structure, drain.

The drain has been introduced as an actor that is associated with the extension use case, DrainToL. This extension use case introduces an *extends* relationship to the use case MaintainH, and now a *prevent* relationship to the accident case ExceedH. The prevent relationship requires the extension-point to be specified between the step in the accident scenario, instead of the use case. The *extension-point* is specified between the step EH_1 and step EH_2, in the scenario of the accident case ExceedH. The behaviour of the extension use case is expected to prevent the severity of the accident case, i.e. water level exceeding the high limit, whenever it deviates the use case MaintainH.

The specification for MaintainH, ExceedH, and DrainToL, are seen in Figure 3.7. The specification for DrainToL describes the behaviour introduced by the drain competent to reduce the water level to the low limit (L). Once the execution of the extension use case is complete, the execution returns to the step, EH_2, of the accident scenario. The behaviour of the extension use case DrainToL prevents the accident scenario from exceeding the water level above the high limit, as the additional behaviour has reduced the water level to the low limit.

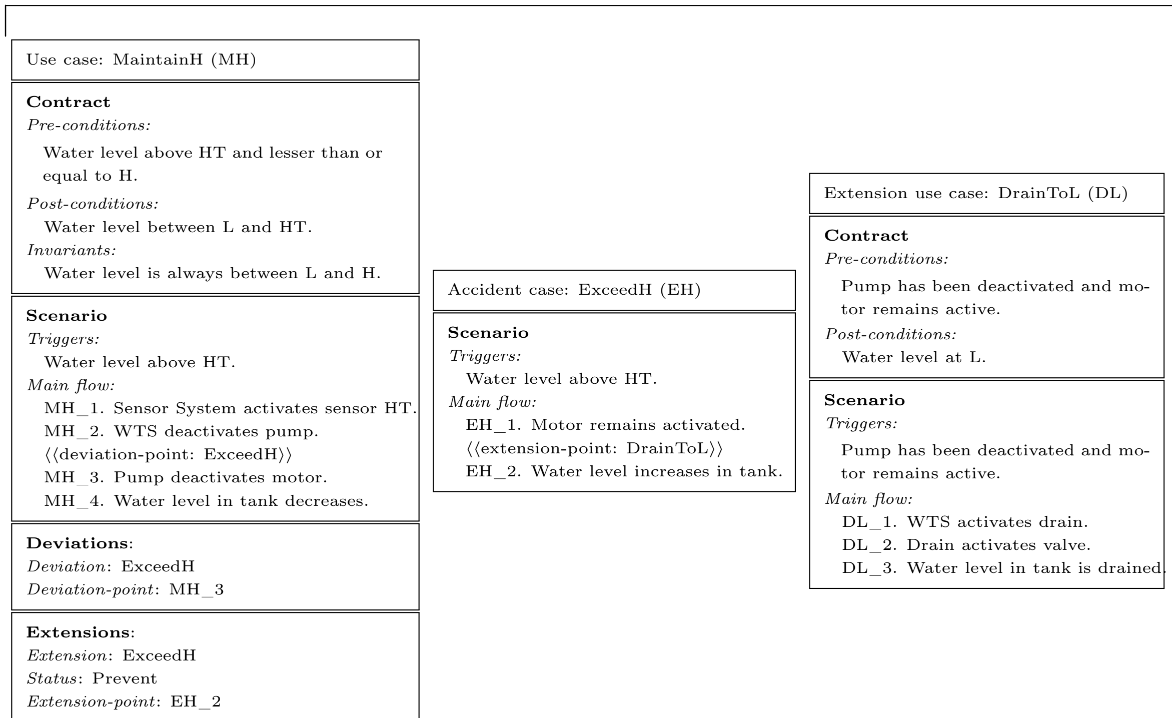


Figure 3.7: Updated specificatin for MaintainH*,* MonitorPump *and* ExceedH*.*

3.3 Notation and Semantics

This section specifies the notations and semantics for the accident case (Section 3.3.1) and the relationships *deviate* (Section 3.3.2), and *prevent* (Section 3.3.3).

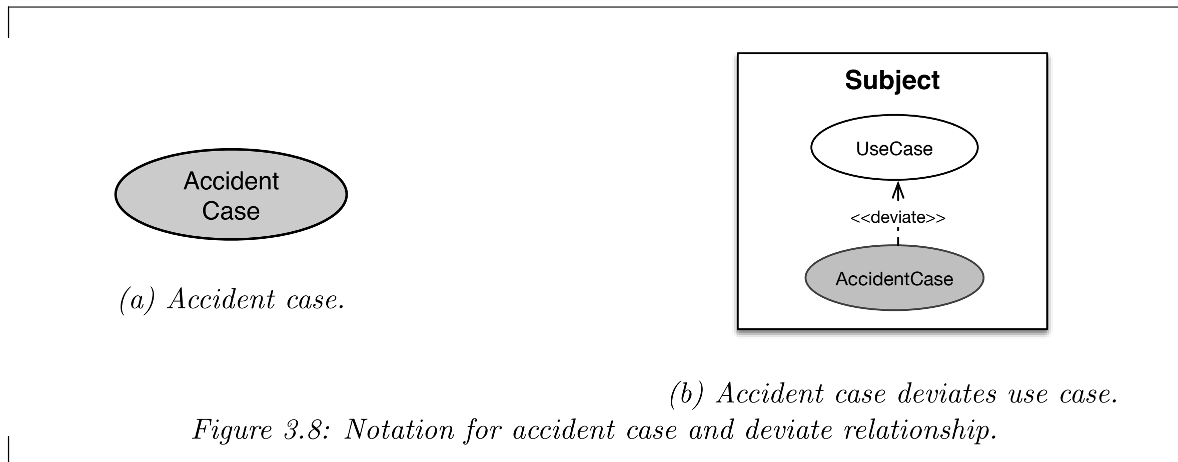
3.3.1 Accident Case

The accident case is a means of specifying *undesired* or *unplanned* usages of a system. It is introduced as a deviation of a use case via the *deviate* relationship. The execution of the accident case is dependent of the use case it deviates. Its behaviour may be performed only during the execution of the deviated use case. The behaviour of the deviated use case is defined independently of the deviating accident case, and is meaningful independently. On the other hand, the deviating accident case typically defines behaviour that is not necessarily meaningful by itself.

The specification of the accident case provides a set of actions performed by the system under consideration, which yields an observable result that is, typically, some form of loss to one or more actors or other stakeholders of the system if allowed to complete, i.e. execution of the final action. The accident case may be prevented or mitigated by a use case using the relationships *prevent* and *mitigate*, respectively. An accident case may deviate one or more use cases and is defined within the subject (system under consideration).

Notation

An accident case is shown as an ellipse, either containing the name of the accident, as seen in Figure 3.8. If a subject is displayed, the accident case ellipse is visually located inside the system boundary rectangle. This does not necessarily mean that the subject owns the contained accident case, but merely that the accident is applicable to the system under consideration.



(a) Accident case.

(b) Accident case deviates use case.

Figure 3.8: Notation for accident case and deviate relationship.

3.3.2 Deviate

The deviate relationship is a directed relationship where the source is the deviating accident case and the target (or destination) is the deviated use case. This relationship specifies how and when the undesired behaviour defined in the accident case can be introduced as an alternate set of actions to the desired behaviour defined in the target use case. The deviate relationship allows the target use case to specify one or more *deviation-points* which specifies a location in the use case where the behaviour of the accident case is introduced.

The execution of the use case may change to that of the accident case if the trigger condition of the accident case is *true* at that point. The scenario of the accident case leads to the *end* of the deviated use case. However, if the trigger condition for the accident case is *false* then the deviation does not occur.

Notation

A deviate relationship between an accident case and use case is shown by a dashed arrow with an open arrow head from the accident case providing the deviation to the base use case. The arrow is labelled with the ⟨⟨deviate⟩⟩ keyword, as seen in Figure 3.8b.

3.3.3 Prevent

Prevent is a directed relationship form a source use case to a target accident case, where the behaviour of the use case augments the undesired behaviour of the accident case by preventing the accident from taking place. Extension use cases are often used to introduced the additional behaviour into the accident case. Figure 3.9, describes an accident case AC that deviates a use case UC. The extension use case EC is introduced that *extends* the functionality of the UC by preventing any occurrence of the accident case from resulting in a loss to the stakeholder.

Notation

A prevent relationship between an accident case and use case is shown by a dashed arrow with an open arrow head from the accident case providing the deviation to the base use case. The arrow is labelled with the ⟨⟨prevent⟩⟩ keyword, as seen in Figure 3.9.

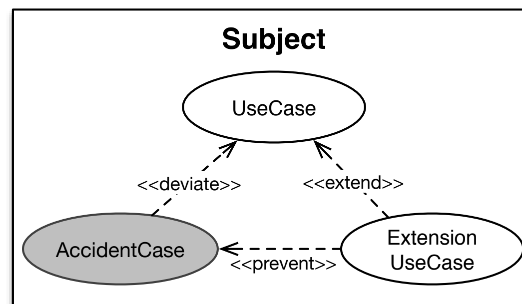


Figure 3.9: Prevent relationship.

3.4 Related Work

This section provides the related work on how undesired or unplanned behaviours are considered by requirements engineering techniques. Ellison et al. [35] introduce *intruders* and *intrusion scenarios* in their case study as part of a large-scale distributed health care system. The intrusion scenario is similar to an accident scenario, but they do not provide a diagrammatic notation, a specification, or guidelines for what constitutes an intrusion scenario.

McDermott and Fox [78] introduce the term *abuse case* as a way of eliciting security requirements; an abuse case defines an interaction between an actor and a system that

results in harm to a resource associated with one of the actors, one of the stakeholders, or the system itself. They capture the abuse cases and regular use cases in separate use case diagrams. This differs from our approach where we provide relationships between accident cases and regular use cases in the same use case diagram.

Sindre and Opdahl [99] introduce *misuse case* as a means to document conscious and active opposition in the form of a goal that a hostile agent intends to achieve, but which the organisation perceives as detrimental to some of its goals. Misuse cases introduces the *threatens* relationships which is perhaps closest in meaning to the accident case with *deviate* relationship. Both misuse case and abuse case have strong inclination to security. The accident case, on the other hand, is focused towards safety concerns.

Allenby and Kelly [7] describe a method for eliciting and analysing safety requirements for aero-engine control systems, using what they call *hazard-mitigating use cases*. In comparison to misuse case and abuse case, they do not suggest the use of *negative agents*, associated with their use cases. The motivation of their method is similar to the accident case. Their method is to tabulate the failures, their causes, types, and effects, and then possible mitigations. However, since their *hazard-mitigating use cases* describe potentially catastrophic failures and their effects, it seems reasonable to define them explicitly from use cases, as in *accident cases*.

Apart from use case based requirements analysis, undesired or negative behaviours have been considered for goal-oriented requirements engineering. Van Lamsweerde [109] and his co-workers on the KAOS approach have proposed *goal-obstacle* analysis. Anton and Potts [8] have used *goals* and *obstacles* to relate desired and undesired behaviour under a goal-hierarchy. Our approach has investigated UML use cases as it widely used in industry and its notations are familiar to practitioners, in comparison to these requirements capture techniques.

3.5 Summary & Discussion

UML use cases has been extended with a new use case type: accident case, for the explicit representation of safety concerns. As UML use cases are used during the early stages of development for defining and analysing system behaviour, this extension provides a platform to communicate accidents identified in the safety analysis with regards to deviations from the desired system behaviour. The accident case allows the requirements analysis to differentiate between the desired and undesired behaviour of the system.

The *deviate* relationship has been introduced in the use case to indicate how the accident case may deviate a use case. Analysing use cases in relation with *deviations*

from accident case support activities that involve *prevention* of adverse accident scenarios,ones with undesired consequences and the promotion of favourable scenarios that limit the severity of such consequences. The relationship *prevent* was introduced to allow use cases to control or limit the severity of an accident case they control. The semantics and notation for this extension to the use case model has been provided.

Chapter 4

Formal Use Cases

4.1 Introduction

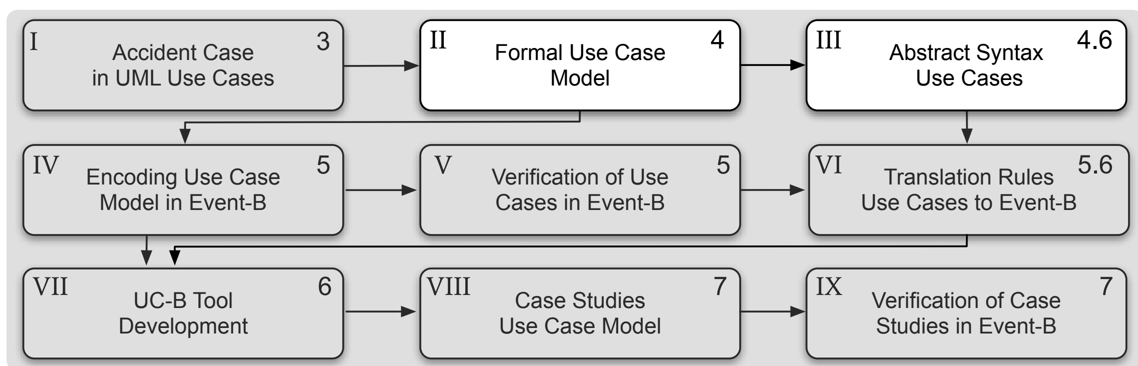


Figure 4.1: Thesis Roadmap for Chapter 4.

In this chapter, the specifications of use cases are *enhanced* to support a language with precise semantics. Figure 4.1 highlights which part of the roadmap this chapter implements. Use cases are a popular method for capturing behavioural requirements of the software system. The informality of use cases is an advantage at the early stages, however, informal requirements can be easily misinterpreted. It is difficult, if not impossible, to check whether the behaviour captured by the use cases satisfies the agreement of the stakeholders involved.

As discussed in Chapter 1, informal methods are limited to review-based analysis. Since their notations are generally incapable of expressing behaviour, the results of the analysis relies only on the properties of the artefact description, not the properties of the artefact itself. Errors committed in the course of preparing the use case document may have far reaching consequences [51]. Left undetected, these errors may later manifest

in the design or implementation phases, where the cost of fixing the same errors are more expensive.

As stated in the Section 2.1 (Background), as a primary artefact in the requirements documentation, UML use cases [19] often appear in two complementary forms:

- A *use case diagram* that provides an easy-to-understand illustration of the *subject*, *actors* and *use cases*. The use case diagrams have strictly formalized syntax.
- An informal document or plain text, often called a *use case specification* [9], used to specify each use case with a *contract* (pre-conditions, post-conditions, and invariants) and *scenarios* (interactions between actors and subject to achieve the contract). There is no agreed formal syntax or semantics for the use case specification.

The means for specifying the contents of a single use case is not agreed upon at all. The UML definition [19] just states that *"a use case can be described in plain text, using operations, in activity diagrams, by a state-machine, or by other behaviour description techniques..."*. In this chapter, an *enhancement* of the use case specification is provided that allows it to be written in a language with precise semantics and logic for reasoning. Inference based on the formally specified use cases allows for the verification of the desired properties in the use case. The gap between informal and formal methods can be reduced by adopting a dual representation in the specification where informal and formal notation is allowed to co-exist.

To implement this, a *use case model* for UML use cases is proposed. The use case model provides the specifications for detailing the concepts of UML use cases, namely the subject, actors and use cases. The specifications allows a dual representation of its content, with both informal and formal notation. The formal notation is based on Event-B's mathematical language [1]. The use case model is not meant as a replacement of the use case diagram that illustrates UML use cases. Instead, the use case model allows each artefact introduced in the use case diagram to be specified with both informal and formal notation.

The layout of this chapter is as follows. Section 4.2 gives an overview of the use case model. Sections 4.3 and 4.4 describe how the subject, actors and use cases in the use case model are represented in the use case model. The abstract syntax for the use case model is provided in Section 4.5. Finally, the related work and summary on this approach to formalising use cases is provided in Section 4.6 and 4.7.

4.2 Use Case Model

The key concepts associated with UML use cases are *actors*, *subject* and *use cases* (use cases refers to a use case type: use case, extension use case or accident case). The subject is the system under consideration to which the use cases apply; the actors model entities that are outside the system; and the use cases capture the interaction between the actors and the subject (e.g., by exchanging signals and data), to achieve some desired functionality. These concepts are illustrated in a use case diagram as seen in Figure 4.2. A *use case model* is introduced, that allows the concepts of UML use cases to be represented by specifications that have syntax and semantics.

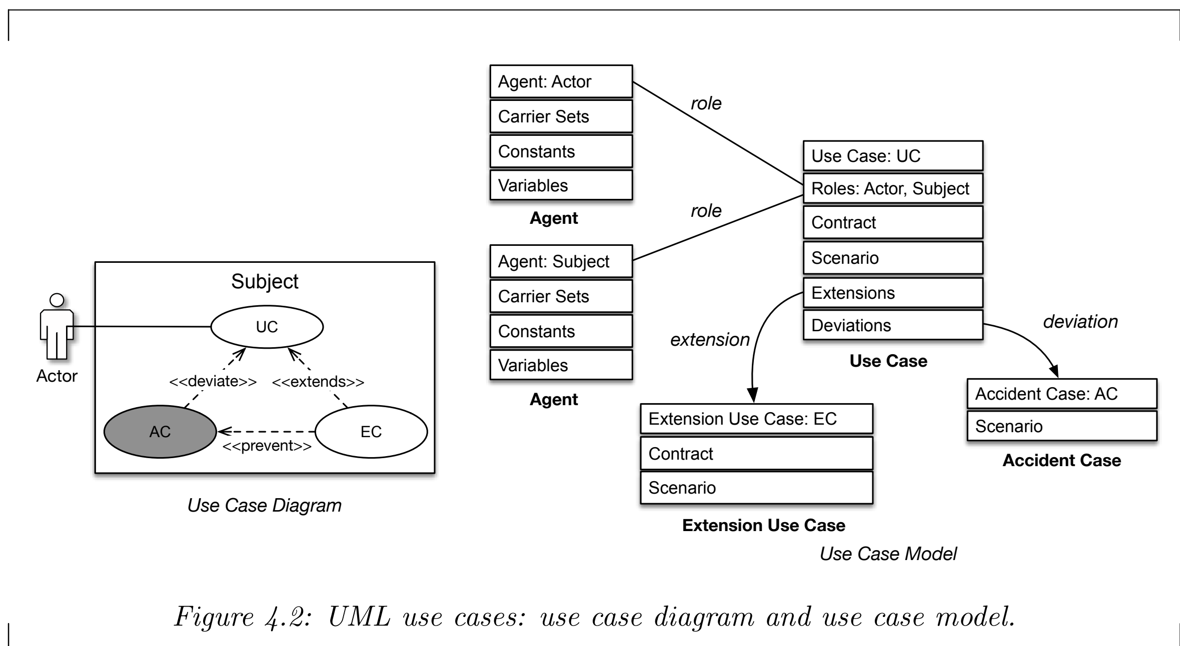


Figure 4.2: UML use cases: use case diagram and use case model.

The actors and subject are represented as *agents* in the use case model. An agent defines information or data relevant to the domain of the actor or subject that the agent represents. These agents play a *role* in use cases, where the information that is defined by the agent is used to detail the specification of the use cases. The specification of the agent is described in Section 4.3 with the *role* relationship.

In the use case model, a use case and extension use case is represented by a specification that contains a contract and scenario. The accident case however is specified with only a scenario. The relationships *extends* and *deviates* are introduced as an of *extensions* and *deviations* in the use case specification. Specifications of the use case, extension use case and accident case are described in Section 4.4.

4.3 Agent

The actors and subject are represented as *agents* in the use case model. An agent models the data or information relevant to the domain of the actor or the subject it represents. An agent is made up of five elements as seen in Figure 4.3.

Each element is described as follows:

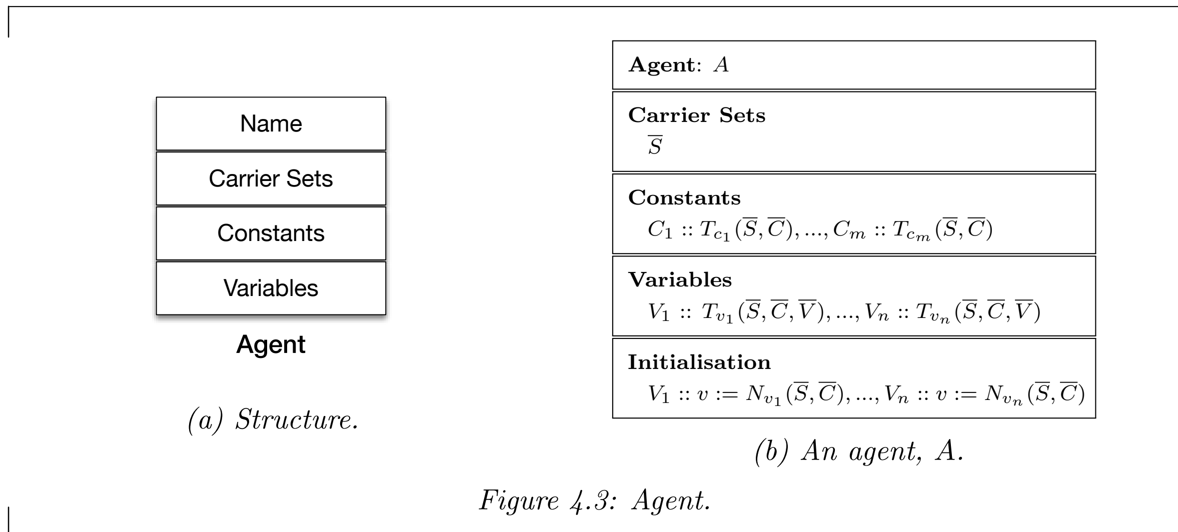
Name The name of the actor or subject the agent represents.

Carrier sets $\overline{S}$ denotes a list of carrier sets such that $\overline{S} = \{S_1, ..., S_l\}$. Each carrier set is represented by a name. The only requirement concerning such sets is that they are to be non-empty.

Constants $\overline{C}$ denotes a set of constants such that $\overline{C} = \{C_1, ..., C_m\}$. The syntactic form for declaring a constant C_i (where $1 \leq i \leq m$) is $C_i :: T_{c_i}(\overline{S}, \overline{C})$. In this case, $T_{c_i}(\overline{S}, \overline{C})$ is a *predicate* that denotes the type of the constant C_i.

Variables $\overline{V}$ denotes a set of constants such that $\overline{V} = \{V_1, ..., V_n\}$. The syntactic form for declaring a constant V_i (where $1 \leq i \leq n$) is $V_i :: T_{v_i}(\overline{S}, \overline{C}, \overline{V})$.In this, $T_{v_i}(\overline{S}, \overline{C}, \overline{V})$ is a *predicate* that denotes the type of the variable V_i.

Initialisation For each variable V_i an initialisation $v := N_{v_1}(\overline{S}, \overline{C})$ is provided. It denotes an assignment. These initialisation could take either the form of a deterministic or non-deterministic assignment as seen in Figure 2.11.



(a) Structure.

(b) An agent, A.

Figure 4.3: Agent.

A *predicate* is expressed within the language of first order predicate calculus with equality extended with set theory. It is the *predicate language* used by Event-B's mathematical language in [1]. The syntax for Event-B's mathematical language is provided in Appendix 1.

Example

In the water tank system, the actors (Water Tank, Sensor System, Pump, Drain) and the subject (Water Tank System) are introduced as *agents* in the use case model. This can be seen in Figure 4.4.

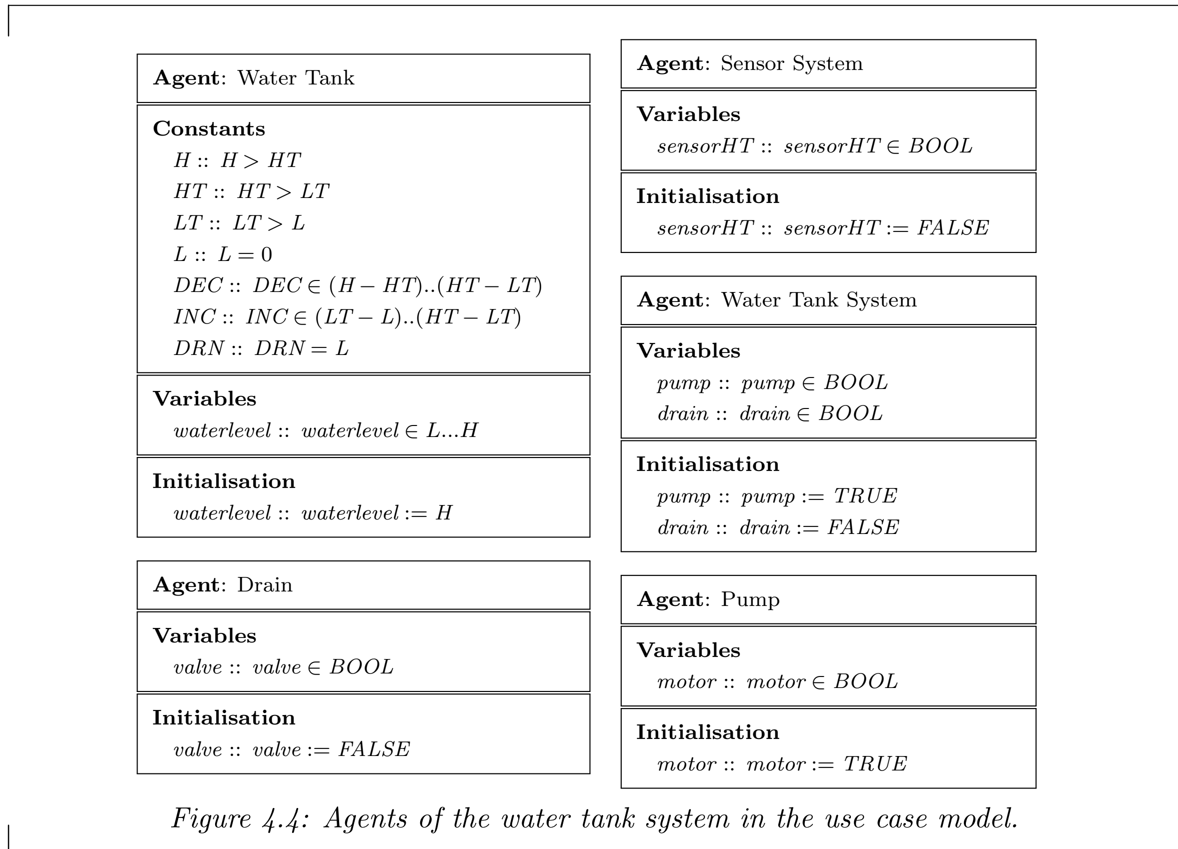


Figure 4.4: Agents of the water tank system in the use case model.

The Water Tank agent defines the limits and thresholds of the tank (L, H, LT, and HT) as constants, as they are not expected to be modified by the behaviour of the use cases. Their types specify important assumptions on the domain of the water tank, e.g. the high threshold if above the low threshold $HT > LT$. The water level in the tank is denoted by the variable *waterlevel* as its values are expected to change. It is of type, $waterlevel \in L..H$, where the water level is always expected to be between the L and H limits of the water tank. This variable is initialised to the value H. The constants, DEC and INC, denote a discrete representation in the decrease and increase of water level in the tank, respectively.

The agents Sensor System, Pump, Water Tank System, and Drain, introduce the variables, *sensorHT*, *pump*, *motor*, *drain*, *valve*. These variables are all of the type $BOOL$, where $TRUE$ indicates *activated*, and $FALSE$ indicate *deactivated*. These sets, constants and variables can be used to specify a use case in which the agent plays a *role* in.

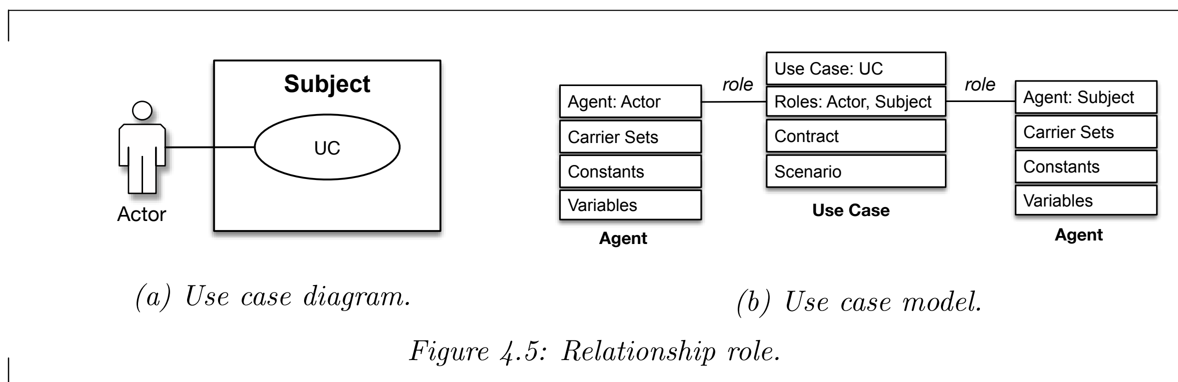
4.3.1 Role

In the use case model, an agent is related to a use case via the *role* relationship. This relationship plays an important part in allowing the specification of the use case to be detailed formally. The relationship allows for the *carrier sets*, *constants* and *variables*, defined by the agent, to be used to detail the specification of the use case. This relationship is used in the following:

Subject An agent that represent the subject (as seen in Figure 4.5a) will have the *role* relationship to the use case that belong to that subject (as seen in Figure 4.5b).

Actors The *association* between an actor and use case (indicated by a line in the use case diagram), introduce the relationship *role* between the agent and use case that corresponds to them in the use case model (as seen in Figure 4.5b).

2



(a) Use case diagram. *(b) Use case model.*

Figure 4.5: Relationship role.

4.4 Use Cases

In the use case model, a use case is made of three elements (as seen in Figure 4.6): (1) a name; (2) a contract; and (3) a scenario. The name of the use case the specification describes. The contract is an agreement with the stakeholders of *what* must be achieved by the use case. The scenario of the use case captures the interaction between the actor and the subject that describe *how* the contract is achieved. The elements, contract and scenario, are further discussed in the sub-sections below.

4.4.1 Contract

The structure of the contract is made of three elements (as seen in Figure 4.7): (1) pre-conditions; (2) post-conditions; and (3) invariants. Assuming that an agent A with

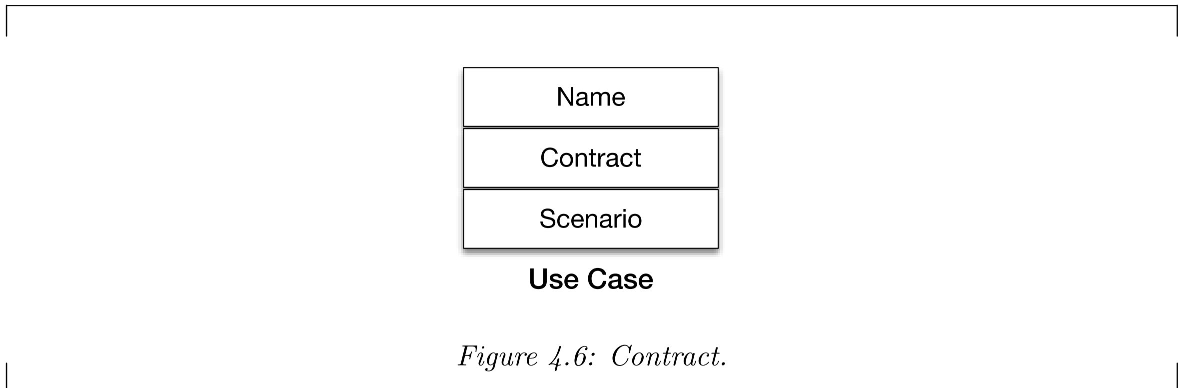


Figure 4.6: Contract.

carrier sets $\overline{S}$, constants $\overline{C}$, and variables $\overline{V}$, plays a *role* in the use case UC, then its contract can be specified as follows:

Pre-conditions A list of named predicates, collectively denoted by $P(\overline{S}, \overline{C}, \overline{V})$. These predicates state the conditions that are required to be true before the use case executes.

Post-conditions A list of named predicates, collectively denoted by $Q(\overline{S}, \overline{C}, \overline{V})$. These predicates state the conditions that are required to be true after the use case executes.

Invariants A list of named predicates, collectively denoted by $I(\overline{S}, \overline{C}, \overline{V})$. These predicates state the conditions that are required to be true throughout the execution of the use case.

The contract of the use case can be detailed formally using only the carrier sets $\overline{S}$, constants $\overline{C}$, and variables $\overline{V}$, of the agents that play a role in it. This requires a use case to have atleast one agent that plays a role in it, in order for its specification to be specified formally.

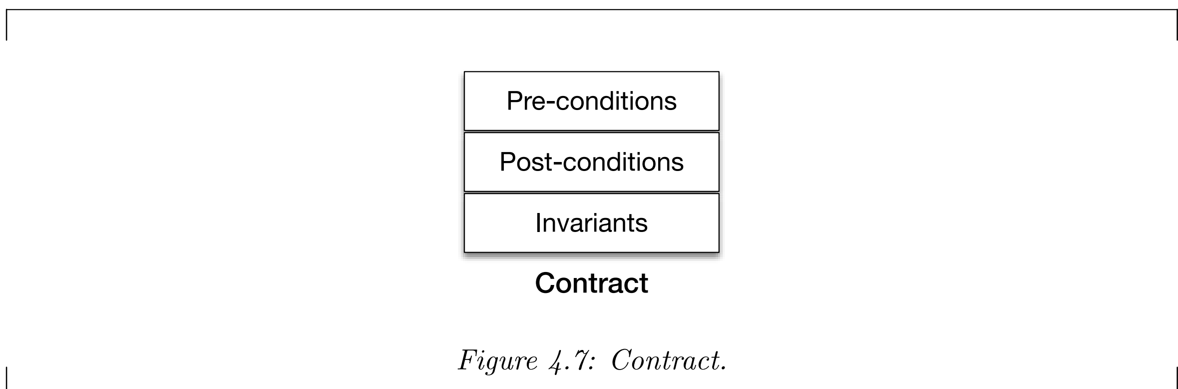


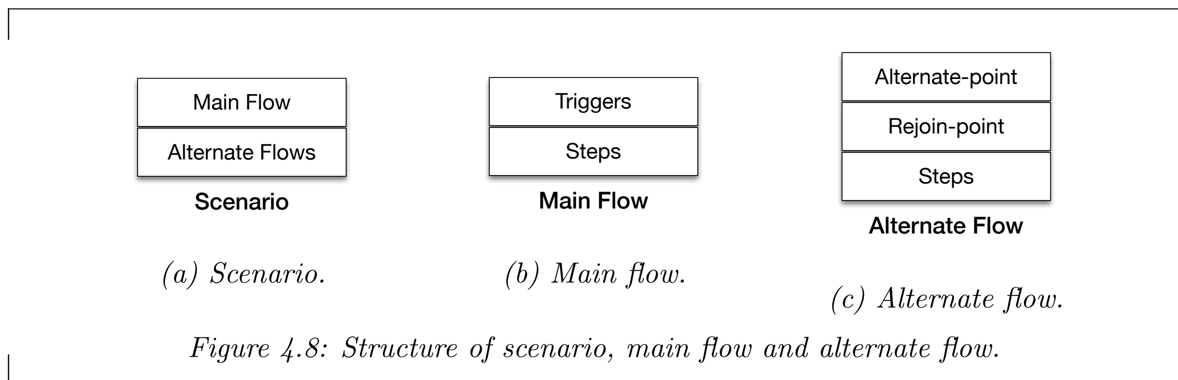
Figure 4.7: Contract.

4.4.2 Scenario

The structure for a scenario is made of two elements (as seen in Figure 4.8a): (1) a *main flow*, and (2) a collection of *alternate flows*. The alternate flows are optional, but there must be one main flow to describe the scenario of the use case. The structures of the main flow and alternate flow are seen in Figure 4.8b and 4.8c, respectively.

The main flow represents a "*sunny day*" scenario where there are no exceptions or failures in the interaction between the actors and the subject (agents) to achieve the contract. The main flow is made of two elements: (1) *triggers* and (2) *steps*, as seen in Figure 4.8b. The *triggers* are a list of named predicates, collectively denoted by $R(\overline{S}, \overline{C}, \overline{V})$. These predicates state the conditions that must be true in order for the main flow of the scenario to initiate execution. The *steps* element specify a sequence of individual steps. The specification for the *steps* element is described in Section 4.4.3.

The alternate flows are *optional*. They introduce a sequence of steps that also achieves the contract of use case, albeit, following different steps than those described in the main flow of the use case. These alternate flows capture *expected* errors (e.g. an ATM customer providing an incorrect PIN) in the interactions between the actors and subject. The structure of the alternate flow specifies an *alternate-point* and *rejoin-point*. The alternate-point specifies a step in the main flow of the use case, where the execution of the use case may *alternate* (instead of that step) to the first step of the alternate flow. The rejoin-point specifies a step in the main flow of the use case where the execution of the alternate flow returns after its sequence of steps have been executed.



(a) Scenario. *(b) Main flow.* *(c) Alternate flow.*

Figure 4.8: Structure of scenario, main flow and alternate flow.

4.4.3 Steps

The *steps* specify a sequence of steps, $U_1, ..., U_n$, where each step can be of either one of the following kinds: (1) action, (2) conditional, or (3) loop. Traditionally, branching in use cases are often shown by *alternate flows*. However, it is possible to specify

if (conditional) and **while** (loop) within the flow of the use case to introduce what is called *simple branching* [9]. The use of *simple branching* in a flow is desirable as it can reduce the total number of alternate flows specified in the use case. In this thesis, a use case flow is allowed to show branching in two ways: (1) simple branching create branches within the flow, namely, *conditionals* and *loops*, (2) complex branching specified by *alternate flows* written explicitly below the main flow.

Let U_i be a step such that $U_i \in U_1, ..., U_n$. Figure 4.9, describes the three different kinds of step that can be applied to U_i with their syntactic form and semantics.

Kind	**Syntactic Form**	**Description**
Action	U_i. $N_{u_i}(\overline{S}, \overline{C}, \overline{V})$	$N_{u_i}(\overline{S}, \overline{C}, \overline{V})$ is a generalised substitution that may take either form of a deterministic or non-deterministic assignment, as seen in Figure 2.11. The assignment may modify the state of variables $\overline{V}$, on the execution of the step U_i.
Conditional	U_i. **if** $C_{u_i}(\overline{S}, \overline{C}, \overline{V})$ **then** U_{i_1}. ... ⋮ U_{i_n}. ... U_{i+1}. ...	When the execution reaches a conditional step, U_i, if the predicate, $U_{u_i}(\overline{S}, \overline{C}, \overline{V})$, is *true*, then the (sub) steps, $U_{i_1}...U_{i_n}$, that belong to U_i, are allowed to execute. The execution then continues to step, U_{i+1}. If the predicate, $C_{u_i}(\overline{S}, \overline{C}, \overline{V})$, was *false*, then execution skips the steps, $U_{i_1}...U_{i_n}$, and executes the step, U_{i+1}.
Loop	U_i. **while** $C_{u_i}(\overline{S}, \overline{C}, \overline{V})$ **do** U_{i_1}. ... ⋮ U_{i_n}. ... U_{i+1}. ... ;	When the execution reaches a loop step, U_i, if the predicate, $C_{u_i}(\overline{S}, \overline{C}, \overline{V})$, is *true*, then the (sub) steps, $U_{i_1}...U_{i_n}$, that belong to U_i, is allowed to execute. The execution then returns back to the step, U_i. If the predicate, $C_{u_i}(\overline{S}, \overline{C}, \overline{V})$, was *false*, the execution skips the steps, $U_{i_1}...U_{i_n}$, and executes the step, U_{i+1}.

Figure 4.9: Kinds of steps: action, conditional, and loop.

The simple branching conditional and loop are described as follows:

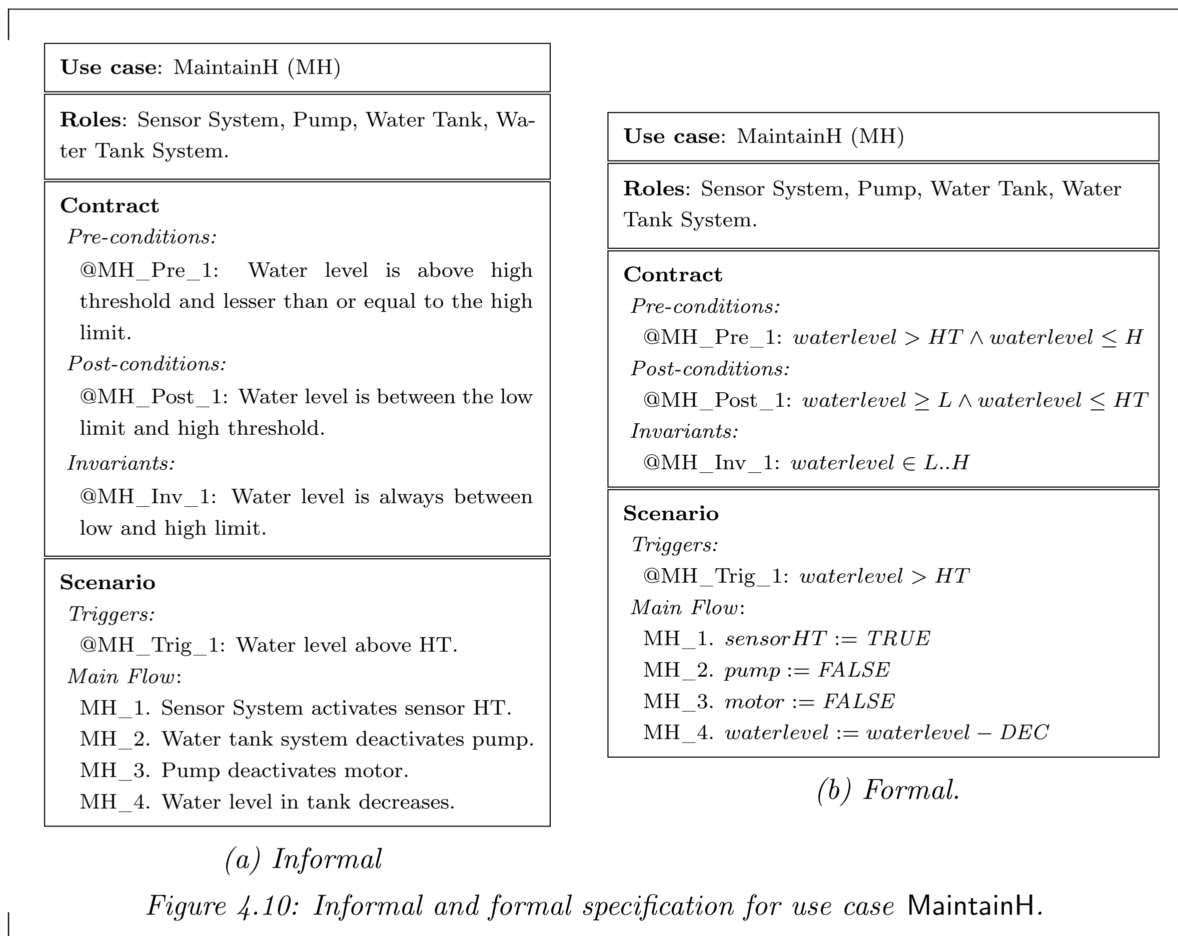
Conditional A conditional in the flow is introduced by a step with the prefix **if**. This step does not capture an action, but specifies a predicate that is either *true* or *false*. Under this step, is a collection of (sub-) steps that acts as the *body* to this conditional. This is clearly indicated with careful indentation and numbering. This removes the need for a closing statement, e.g. **end if**.

Loop Sometimes it is necessary to repeat an action several times within a flow of events. This does not occur very often in use case modelling [9], but it is useful

to provide a strategy to deal with it. The **while** keyword is used to model a sequence of actions in the flow of events that is performed while some condition is *true*. The syntactic form of a loop is similar to that of a conditional. However, the sub-steps of the loop execute until the loop predicate is false.

Example

The informal specification for the MaintainH use case (Figure 3.7) of the water tank system, is specified in the use case model as seen in Figure 4.10. The specification supports a dual representation of the requirements where the use case is detailed with both informal and formal notation. As the actors (Water Tank, Sensor System, Pump, Water Tank System) are *associated* with MaintainH, their corresponding agents have the relationship *role* with this use case (see Figure 4.4 for the agents). The contract of MaintainH is specified formally (Figure 4.10b) where the pre-condition, post-condition and invariant are specified formally via the predicates labelled @MH_Pre_1, @MH_Post_1 and @MH_Inv_1, respectively.



Use case: MaintainH (MH)

Roles: Sensor System, Pump, Water Tank, Water Tank System.

Contract

Pre-conditions:
@MH_Pre_1: Water level is above high threshold and lesser than or equal to the high limit.
Post-conditions:
@MH_Post_1: Water level is between the low limit and high threshold.
Invariants:
@MH_Inv_1: Water level is always between low and high limit.

Scenario

Triggers:
@MH_Trig_1: Water level above HT.
Main Flow:
MH_1. Sensor System activates sensor HT.
MH_2. Water tank system deactivates pump.
MH_3. Pump deactivates motor.
MH_4. Water level in tank decreases.

(a) Informal

Use case: MaintainH (MH)

Roles: Sensor System, Pump, Water Tank, Water Tank System.

Contract

Pre-conditions:
@MH_Pre_1: $waterlevel > HT \land waterlevel \leq H$
Post-conditions:
@MH_Post_1: $waterlevel \geq L \land waterlevel \leq HT$
Invariants:
@MH_Inv_1: $waterlevel \in L..H$

Scenario

Triggers:
@MH_Trig_1: $waterlevel > HT$
Main Flow:
MH_1. $sensorHT := TRUE$
MH_2. $pump := FALSE$
MH_3. $motor := FALSE$
MH_4. $waterlevel := waterlevel - DEC$

(b) Formal.

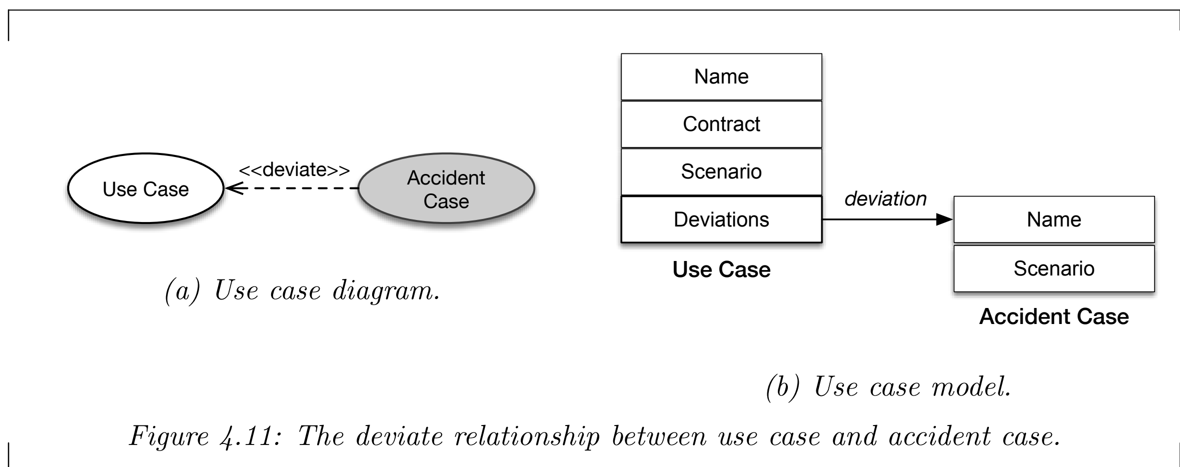
Figure 4.10: Informal and formal specification for use case MaintainH.

The constants and variables defined by the agents are used to formally specify the

contract and scenario of MaintainH. The pre-condition, post-condition and invariant are specified by predicates that clearly express the agreement of the stakeholders. The scenario specifies a main flow which captures a trigger and a sequence of steps (MH_1 to MH_4). Each step is of the action kind, that captures assignments that modify the variables of the agents that play a role in this use case. The execution of the main flow is required to satisfy the contract of the use case.

4.4.4 Accident Case

The relationship *deviate* is used to describe how an accident case may deviate a use case. The source of this relationship is the accident case and the target is the use case. Figure 4.11a provides a use case diagram of an accident case that deviates a use case via the *deviate* relationship. In the use case model, this is represented by the specification of the *use case* and *accident case*, as seen in Figure 4.11b.



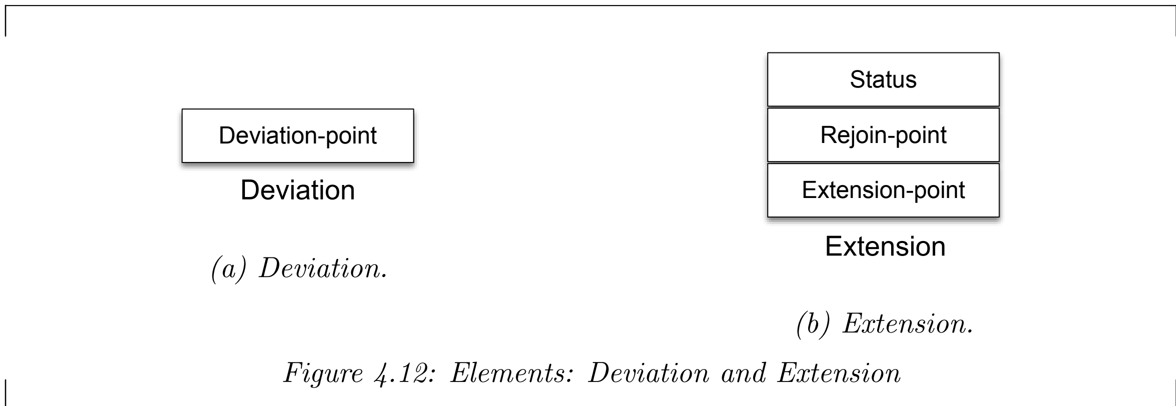
(a) Use case diagram.

(b) Use case model.

Figure 4.11: The deviate relationship between use case and accident case.

In the use case model, this deviate relationship introduces the *deviations* element in the structure of the use case. This element allows the use case to have a collection of *deviation* references to accident cases. The structure of the accident case is similar to that of the use case. However, it does not have a contract, and only specifies a name and a scenario, as seen in Figure 4.11b. The scenario of the accident case can be detailed formally using the carrier sets, constants and variables of the agents that play a role in the deviated use case.

The *deviation* element, as seen in Figure 4.12a, allows the use case to specify a *deviation-point*. This deviation-point is a reference to a step in the scenario of the use case (this includes steps in the main flow and alternate flows). This indicates that, during execution, at this step the scenario provided by the accident case is allowed to execute as a deviation from the scenario of the use case.



(a) Deviation.

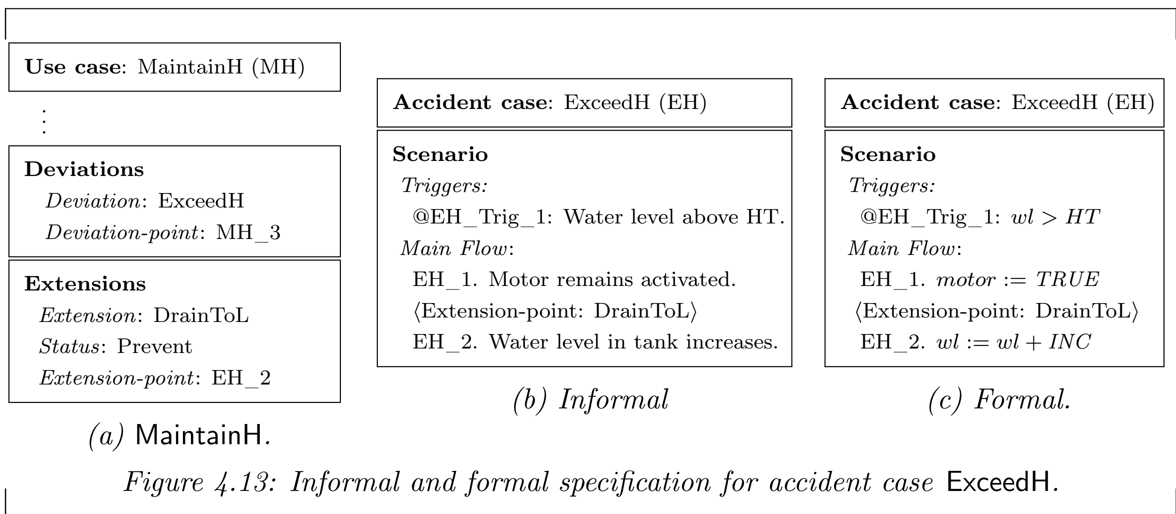
(b) Extension.

Figure 4.12: Elements: Deviation and Extension

Example

In the example of the water tank system, the ExceedH accident case deviates the use case MaintainH. In the use case model, the element *deviation* is introduced in the specification of MaintainH use case, as seen in Figure 4.13. This specifies a *deviation-point*, step MH_3 of MaintainH, which allows the scenario of the ExceedH accident case to be introduced as a deviation.

The specification of ExceedH only provides a scenario as it is an accident case. This scenario is seen in Figure 4.13, and is specified with both informal and formal notations. The deviation relation in the use case model allows the variables and constants defined by the agents that play a role in the use case MaintainH, to be used to specify the scenario of ExceedH.



(a) MaintainH.

(b) Informal

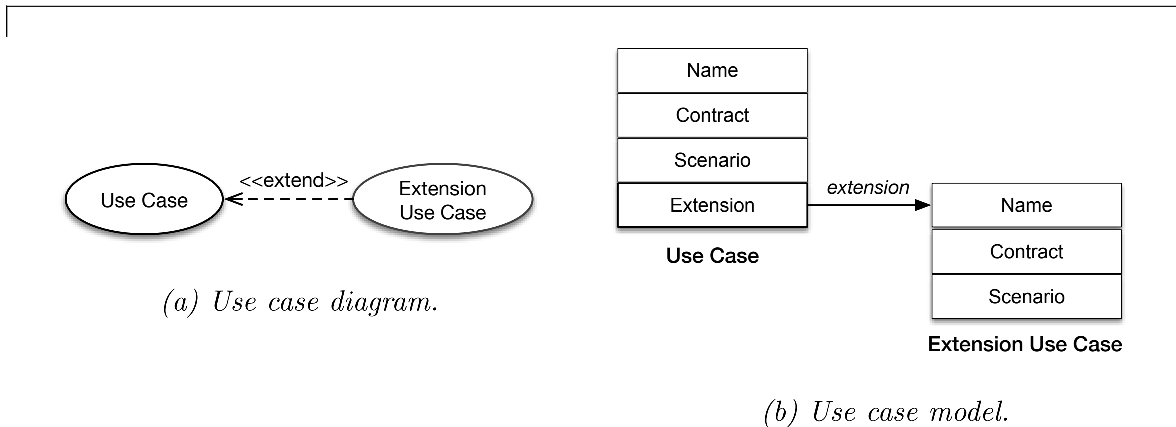
(c) Formal.

Figure 4.13: Informal and formal specification for accident case ExceedH.

4.4.5 Extension Use Case

An extends relationship from an extension use case to a use case is illustrated by a use case diagram in Figure 4.14a. The use case and extension use case are represented

in the use case model as seen in Figure 4.14b. When an extension use case intents to extend the behaviour of the use case, an element *extensions* is introduced in the structure of the use case. This allows the use case to specify one or more *extension*, which refers to the extension use case. This extension element represents the extends relationship in the use case model. The structure of the extension use case is the same as that of the use case. It specifies a name, a contract and a scenario.



(a) Use case diagram.

(b) Use case model.

Figure 4.14: The extend relationship between use case and extension use case.

The extension use case can be specified by the carrier sets, constants and variables, used to specify the parent use case. New agents may also be introduced that play a role in the extension use case. The *extension* element allows the use case to specify a *status*, *extension-point* and *rejoin-point*. Each of these elements are described as follows:

Status A status can be of two kinds: *ordinary* or *prevent*. It denotes the type of extension. By default, the status for an extension is *ordinary*. The status *prevent* is discussed in the following subsection.

Extension-point When the status is *ordinary*, the extension-point specifies a step S_e in the scenario of the use case that the extension element belongs to. The behaviour of the extension use case is inserted between steps S_e and S_{e-1}.

Rejoin-point When the status is *ordinary*, the rejoin-point specifies a step S_r in the scenario of the use case that the extension element belongs to. It specifies the step that is executed after the extension use case is complete. It is possible for the rejoin-point not to be specified. In this case the execution of the use case returns to the end of the use case.

The Prevent Status

When the status of the extension is *prevent*, it indicates that the extension use case is introduced as a means to prevent the behaviour of the accident case that deviates the use case from completing. The *prevent* status requires the *extension-point* to specify a step in the scenario of the accident case. The prevent relationship ensures that the accident case is not allowed to complete, i.e. the final step of the accident case must not be allowed to execute. When the status is *prevent*, *rejoin-point* must specify a step in the scenario of the use case that the extension element belongs to. This ensures that the behaviour of the extension use case is executed and returns to the scenario of the main use case.

Example

The element *extension* is introduced in the specification of MaintainH as shown in Figure 4.13a. It specifies the *status* and *extension-point* and refers to the extension use case DrainToL. The extension-point specifies a step EH_2 in scenario of the ExceedH accident case, as the status for the extension is *prevent.* This introduces the behaviour of the extension use case between the steps EH_1 and EH_2. Since the *rejoin-point* is not specified it returns to the end of the use case. This allows the execution to skip the step EH_2, preventing the accident scenario from not completing.

Extension Use Case: DrainToL (DL)
Roles: Drain.
Contract *Pre-conditions:* @DL_Pre_1: Pump has been deactivated and motor remains active. *Post-conditions:* @DL_Post_1: Water level is at L.
Scenario *Triggers:* @DL_Trig_1: Pump has been deactivated and motor remains active. *Main Flow:* DL_1. WTS activates drain. DL_2. Drain activates valve. DL_3. Water level in tank is drained.

(a) Informal

Extension Use Case: DrainToL (DL)
Roles: Drain.
Contract *Pre-conditions:* @DL_Pre_1: $pump = FALSE \wedge motor = TRUE$ *Post-conditions:* @DL_Post_1: $waterlevel = L$
Scenario *Trigger:* @DL_Trig_1: $pump = FALSE \wedge motor = TRUE$ *Main Flow:* DL_1. $drain := TRUE$ DL_2. $valve := TRUE$ DL_3. $waterlevel := DRN$

(b) Formal.

Figure 4.15: Informal and formal specification for extension use case DrainToLT.

The extension use case, DrainToL is specified with both formal and informal nota-

tion, as seen in Figure 4.15. The extension use case is specified with the carrier sets, constants and variables that were used to specify the MaintainH use case. The Drain agent plays a role in this extension use case, allow the specification to use the variables *drain* and *valve* and constant *DRN*, defined by the agent.

4.5 Abstract Syntax for Use Cases

This section provides the syntax in the structure of the use case specifications. Extended Backus-Naur Form (EBNF) [112], is used to describe the abstract syntax. This syntax is used to describe translation rules in Chapter 5 to encode it in Event-B. The meta-symbols for EBNF and their meaning are provided in Figure 4.16.

Symbol	**Meaning**
::=	is defined as
|	or
<>	non-terminal symbol

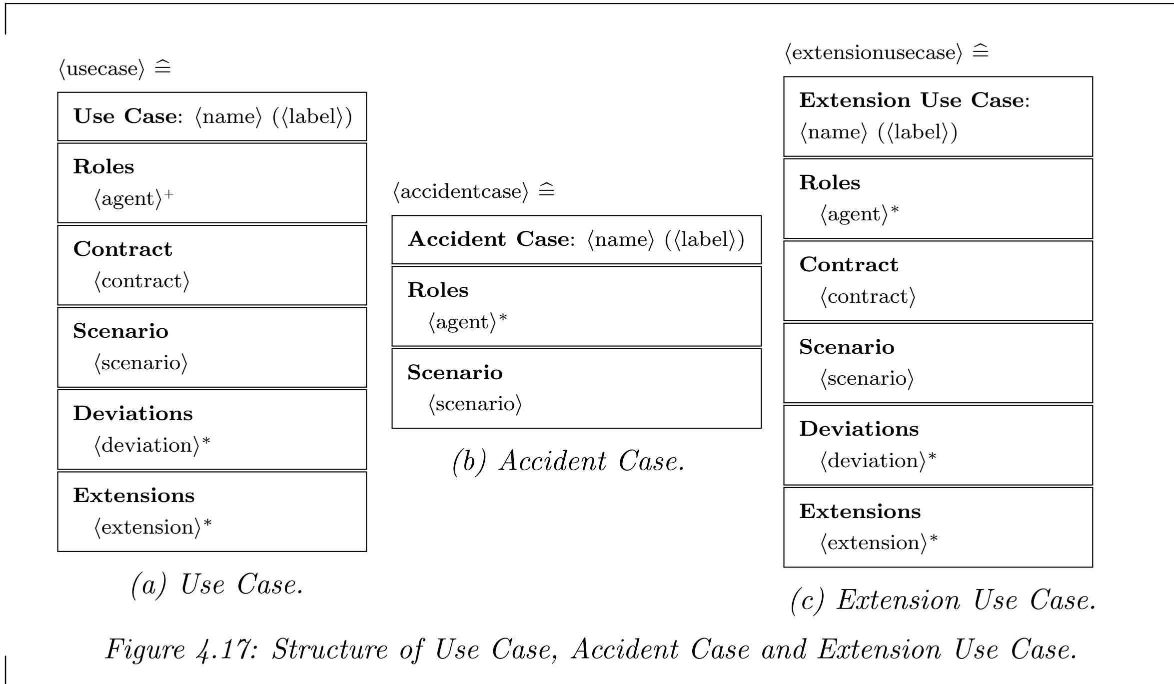
Symbol	**Meaning**
*	zero or more
+	one or more
?	zero or one

Figure 4.16: Meta-symbols for EBFN and their meanings.

Figure 4.17 describes the structure and syntax of the use case, extension use case and accident case. The structure of these were informally described in Section 4.4. The structure of the use case and extension use case are similar as they contain a *name*, *contract*, *scenario*, *extensions* and *deviations*. The *extension* and *deviation* are optional as there can be zero or more of them (*). However, the accident case only specifies a *name* and *scenario*. With regards to agents, the use case must specify atleast one agent ($^+$), while the extension use case and accident cases may specify zero or more (*) agents that play a role in them.

Figure 4.18 provides the structure and syntax used to define the *agent*, *contract* and *scenario*. The agent may specify zero or more *sets* and *constants*, but they must specify atleast one or more *variable* and *initialisation*. The *contract* of a use case or extension use case specifies *pre-conditions*, *post-conditions* and *invariants*. The invariants are optional but there must be atleast one or more pre-conditions and post-conditions. Each of these refer to a collection, for example, pre-conditions is defined as collection of *pre-condition*. The scenario specifies a *main flow* and optional *alternate flows*. The main flow is composed of *triggers* and a collection of *step**. The ⟨triggers⟩ may specify a collection of trigger.

Figure 4.19 provides the structure and syntax for the *alternate flow*, *deviation*, and *extension*. The alternate flows contains an *alternate-point*, steps (similar to a main



(a) Use Case.

(b) Accident Case.

(c) Extension Use Case.

Figure 4.17: Structure of Use Case, Accident Case and Extension Use Case.

flow), and an optional *rejoin-point*. The relations extension and deviation refer to extension use case and accident case via a name, respectively. The extension specifies a *status*, *extension-point*, and *rejoin-point*. The status is either of the form *ordinary*, *mitigate* or *prevent*. The deviation relation specifies the name of the accident case and a *deviation-point*. The pre-condition, post-condition, invariant, trigger, and condition capture a labelled predicate. The step, however, captures a labelled action.

As seen in Figure 4.18a the agents may specify sets, constants, variables and initialisations. The syntax for these are seen as follows. Each set, constant, variable and initialisation specifies an *identifier*. The *constant* and *variables* also specify a *predicate*. The initialisation specifies an *action*.

$\langle \textit{set} \rangle ::= \langle \textit{identifier} \rangle$

$\langle \textit{constant} \rangle ::= \langle \textit{identifier} \rangle$ '`::`' $\langle \textit{predicate} \rangle$

$\langle \textit{variable} \rangle ::= \langle \textit{identifier} \rangle$ '`::`' $\langle \textit{predicate} \rangle$

$\langle \textit{initialisation} \rangle ::= \langle \textit{identifier} \rangle$ '`::`' $\langle \textit{action} \rangle$

Each *step* may specify a label and action, or *conditional* or *loop*. The conditional and loop introduce the `if` and `while` constructs. They capture a *label*, *predicate* and *steps*.

$\langle \textit{step} \rangle ::= \langle \textit{label} \rangle$ '`:`' $\langle \textit{action} \rangle \mid \langle \textit{conditional} \rangle \mid \langle \textit{loop} \rangle$

$\langle \textit{conditional} \rangle ::= \langle \textit{label} \rangle$ '`:`' '`if`' $\{\langle \textit{predicate} \rangle\}$ '`then`' $\langle \textit{step} \rangle^{+}$

$\langle \textit{loop} \rangle ::= \langle \textit{label} \rangle$ '`:`' '`while`' $\{\langle \textit{predicate} \rangle\}$ '`do`' $\langle \textit{step} \rangle^{+}$

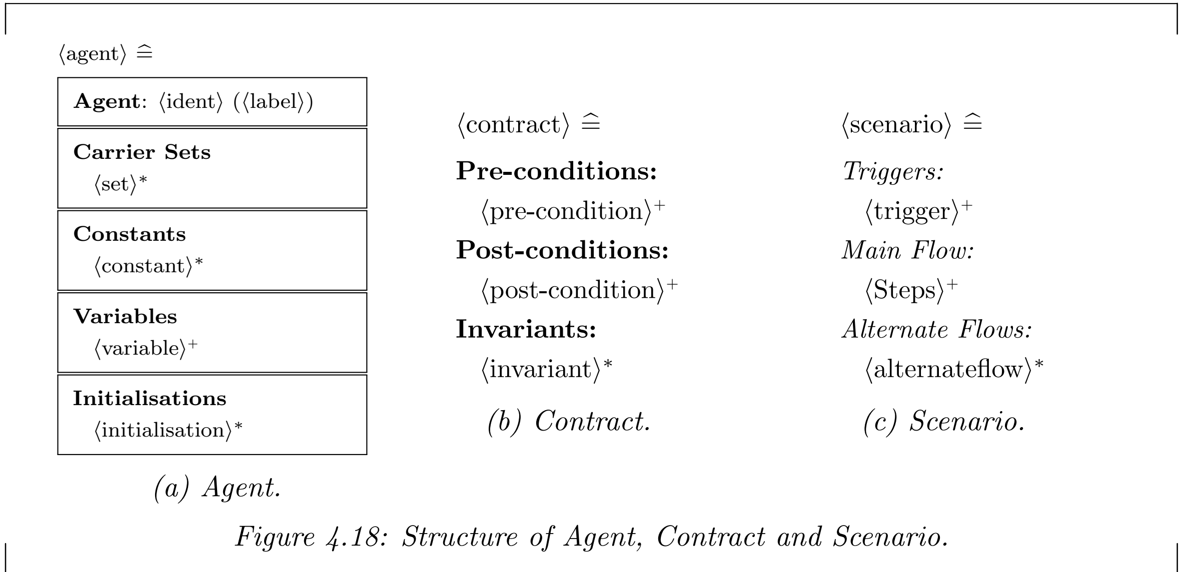


Figure 4.18: Structure of Agent, Contract and Scenario.

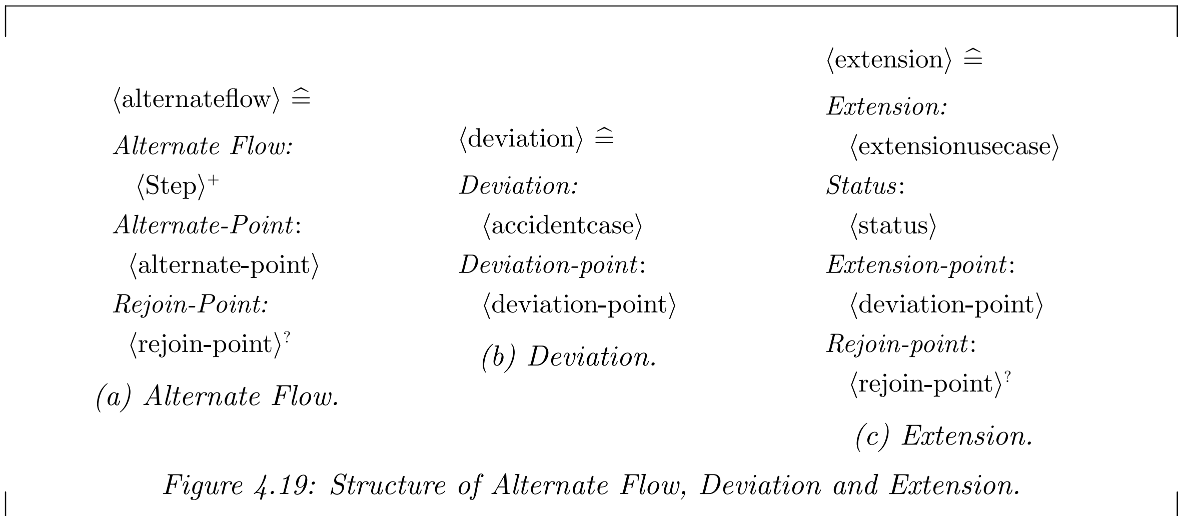


Figure 4.19: Structure of Alternate Flow, Deviation and Extension.

The syntax for the pre-condition, post-condition, invariant, trigger, condition and step are as follows. They all, apart from step, specify a *label* and a *predicate*. The step specifies a *label* and an *action*.

⟨pre-condition⟩ ::= *⟨label⟩* : *⟨predicate⟩*

⟨post-condition⟩ ::= *⟨label⟩* : *⟨predicate⟩*

⟨invariant⟩ ::= *⟨label⟩* : *⟨predicate⟩*

⟨trigger⟩ ::= *⟨label⟩* : *⟨predicate⟩*

The syntax for a predicate is specified in Appendix A. An action takes the syntactic form as seen in Figure 2.11, that may contain a predicate or expression. The syntax for an expression is provided in Appendix A.2. The predicate and expression language provided is based on Event-B's mathematical language [1].

⟨*alternate-point*⟩ ::= ⟨*label*⟩

⟨*deviation-point*⟩ ::= ⟨*label*⟩

⟨*extension-point*⟩ ::= ⟨*label*⟩

⟨*rejoin-point*⟩ ::= ⟨*label*⟩

A label or identifier is a sequence of characters that enjoy some special property, like referring to a letter or a digit.

4.6 Related Work

In this chapter, a formal use case model has been provided which allows the concepts for UML use cases to be detailed in a formal specification. This use case model is not meant as a replacement of the use case diagram but as an enhancement to it.

Hurlbut [49] provides a very thorough and detailed survey of selected issues concerning formalising of use cases. Pohl and Haumer [91] propose contextual models for representing and reasoning about scenario-based requirements. However, it has no formal model and consequently there is a lack of reliable mechanism for formal reasoning about the modelled system. In [98], Shen and Liu propose a rigorous review technique for use case diagrams. The pre- and post-condition of an use case are formalised in a HCL specification. In [15], Bartsch et al. describe an approach to check consistency between use case scenarios and sequence diagrams. Barrett et al., [14] includes the transition of use cases to finite state machines, however the formal model presented in our work is more detailed and also shows the specific context of the formal verification.

In [118], Zhao and Duan shows the formal analysis of use cases using the Petri nets [89] formalism. Overgaard and Palmkvist provide a formalisation of the relationships used within UML use cases [87], but the authors do not show how to analyse the use cases. OCL [110] is a text-based language that is part of the XML [20] standard and it is used to constraint the behaviour of UML elements. Unfortunately, OCL specifications cannot capture the interaction between actors. Therefore they are not as expressive as contracts.

Back et al. [11] propose the enhancement of use case diagrams with formal documents (contracts) using the refinement calculus [12]. In [47], Wolfgang et al. propose an approach for use cases to be specified in Abstract State Machine Language, and test cases are generated. In comparison to these approaches, the work presented in this section provides an enhancement UML use cases allowing its textual specifications to be written in a language with precise semantics. In addition, the extension to UML use cases via the accident case is taken into account in this formalisation of use cases.

4.7 Summary & Discussion

This chapter has introduced a *use case model* that allows concepts of UML use cases, namely, use cases (including accident case), actors and subject to be detailed in a formal specification. The structure and semantics for the specification has been described with relation to the *use case diagram*. The relationships, *deviate* and *extends*, have been taken into account in the use case model that allows the creation of extension use cases and accident cases, respectively. The formal language used is a derivation of Event-B's mathematical language. Using precise semantics to describe use cases is aimed at removing ambiguity and inconsistencies in the requirements. The abstract syntax for for the use cases is provided.

In the next chapter, an encoding of the use case model in Event-B is provided. The aim of this encoding is to use to verification tools provided by Event-B to automate the reasoning applied on the laws of the use case. The translation rules for this encoding in Event-B is provided with the use of the abstract syntax for use cases provided in this chapter.

Chapter 5

Encoding Use Cases in Event-B

5.1 Introduction

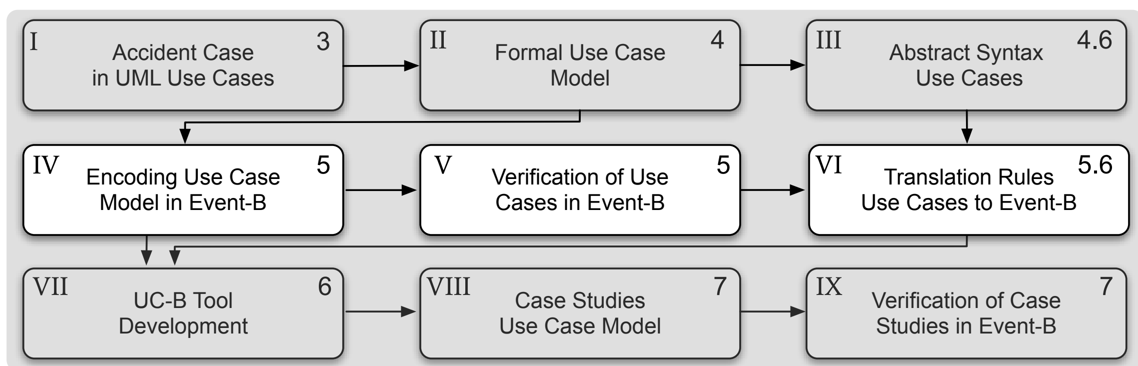


Figure 5.1: Thesis Roadmap for Chapter 5.

In this chapter, an encoding of the use cases in Event-B is provided. Figure 5.1 highlights which part of the roadmap this chapter implements. The purpose of this encoding is three fold: (1) to exploit an abstraction found is use cases that allow its behaviour to be encoded in the refinement-based formalism Event-B; (2) provide semantics for use cases in Event-B, and (3) harness the verification support provided by Event-B to prove that contracts are satisfied by scenarios, including those with prevented accident cases. As discussed in Section 2, Event-B supports a refinement-based approach, where its model represents different abstraction levels of the *system behaviour*; internal consistency and between the abstraction levels are ensured by *formal verification*.

Use cases provide a *goal hierarchy* [25] by its *contract* and *scenario* that provide an *abstraction* of the problem. A use case arises when a subject needs to interact with actors to achieve an overall *goal*. This goal is specified by the *contract* of the use case

as an agreement of *what* must be achieved. The scenario captures the interactions between the subject and actors to achieve this goal. Each step in the scenario, can be viewed as *sub-goals*, that act as steps taken towards achieving the overall goal. The scenario of the use case details how the overall goal is broken down into sub-goals represented by *steps*.

In addition, the scenario of the use case may have deviations and extensions that further introduce additional behaviour to the use case via *accident cases* and *extension use cases*. These additional behaviours can be considered as sub-goals introduced in the scenario of a use case, which may result in further abstractions of the overall problem. In this chapter, an argument is made for the use of this *goal hierarchy* (an abstraction of the problem) to help model behaviour of the use cases as an Event-B model via step-wise refinement. This is seen in Figure 5.2. Traversing through this hierarchy answers the following questions:

- Moving down the hierarchy answers *how* to show how a certain use case can be achieved.
- Moving up the hierarchy answers *why* and provides a rationale for why a certain use case exists.

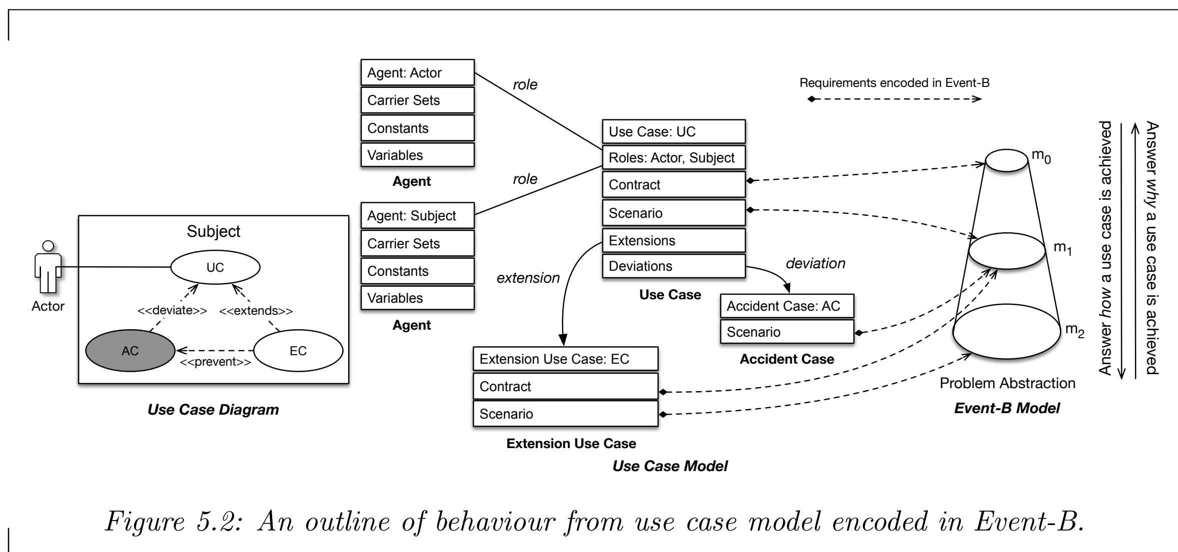


Figure 5.2: An outline of behaviour from use case model encoded in Event-B.

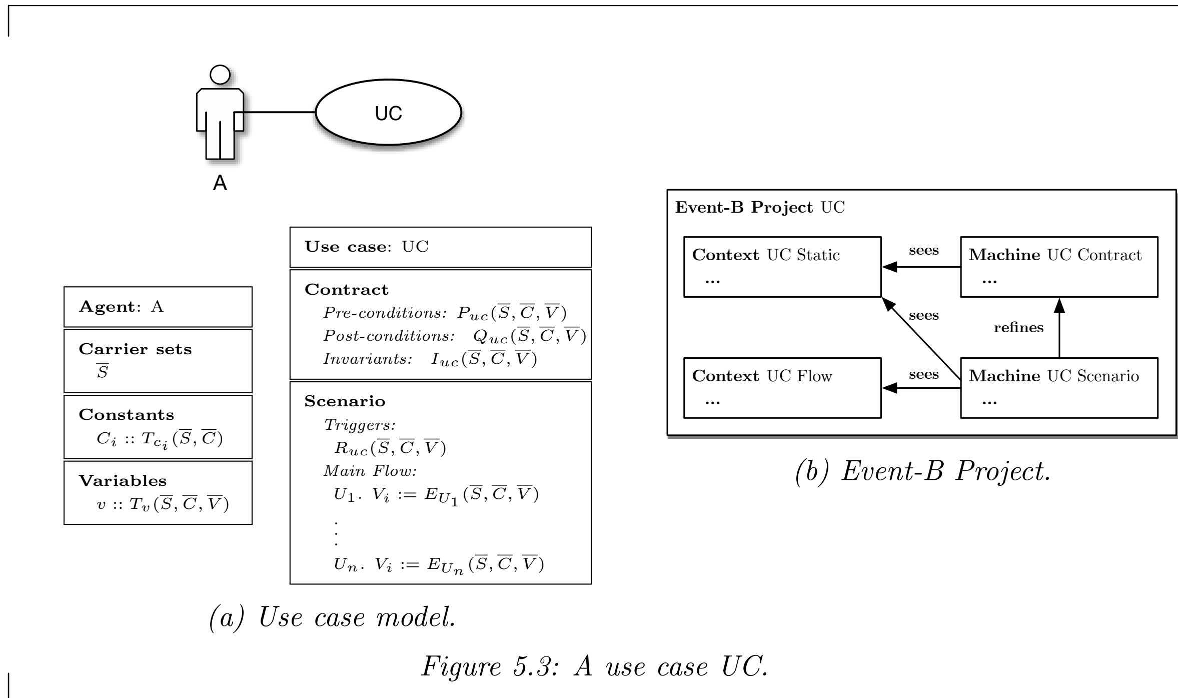
This chapter provides an encoding of use cases as Event-B models based on the *natural abstraction* found in the use cases structure and relationships. This encoding also automatically extracts *gluing invariants*. These invariants provide the link between the abstract and concrete representation that is needed to verify that each abstract behaviour is a correct simulation of its concrete behaviour. Providing sufficient, but provable, gluing invariants can be a significant task. The provision of gluing invariants

to discharge the proof obligations (POs) associated with a refinement is a significant step in providing verifiable models.

The encoding of use cases, accident cases and extension use cases in Event-B are provided in Section 5.2, 5.3 and 5.4, respectively. Each section describes the encoding along with the verification provided by Event-B. The encodings of alternate flows, loops and conditional are provided in 5.5. Finally, the translation rules for the encoding use case, extension use case and accident case is provided in Section 5.6.

5.2 Use Case

This section provides the encoding of a use case UC in Event-B. Figure 5.3b illustrates a use case diagram for the use case UC with an actor A that is associated to it, and their formal use case specifications. The actor A is represented as an agent in the use case model that plays a role in UC. The carrier sets, constants and variables declared by the agents are same as those provided in Section 4.3. These sets, constants and variables are used to specify the contract and scenario of the use case UC. The contract specifies the pre-conditions $P_{uc}(\overline{S}, \overline{C}, \overline{V})$, post-conditions $Q_{uc}(\overline{S}, \overline{C}, \overline{V})$ and invariants $I_{uc}(\overline{S}, \overline{C}, \overline{V})$ for the use case UC. Its scenario captures a main flow with steps $U_1, ..., U_n$ and triggers $R_{uc}(\overline{S}, \overline{C}, \overline{V})$. Each steps specifies an action that can modify the variable $\overline{V}$. The encoding of this use case in Event-B is provided in Section 5.2.1.



(a) Use case model.

(b) Event-B Project.

Figure 5.3: A use case UC.

5.2.1 Encoding in Event-B

The encoding for the use case, UC, is provided in Figure 5.4, where the contract is modelled by the abstract machine UC Contract. The content of the use case specification that are encoded in the Event-B model are highlighted (in grey). This machine aims to establish what is achieved by the use case. It is then *refined* by UC Scenario to introduce the scenario of the use case. The refinement introduced by this encoding aims to prove that the behaviour defined by the scenario achieves what is required by the contract.

The encoding also produces the context UC Static and UC Flow. The UC Static context models all the static aspects used to specify the use case, namely the carrier sets $\overline{S}$ and constants $\overline{C}$. The variables $\overline{V}$ that are used to specify the contract and scenario, in Figure 5.3a, are treated as abstract $\overline{V_a}$ and concrete $\overline{V_c}$ variables by the encoding as seen in Figure 5.4. These variables are described as follows:

Abstract Variables The variables that occur in the pre-condition and post-condition of the use case are denoted as abstract variables $\overline{V_a}$, such that $\overline{V_a}$ is a subset of $\overline{V}$. These variables are introduced in the abstract machine UC Contract.

Concrete Variables The encoding denotes the variables that occur in the scenario of the use case (triggers and steps) as concrete variables $\overline{V_c}$, such that $\overline{V_c}$ is a subset of $\overline{V}$. These concrete variables are introduced in machine UC Scenario. The variables $\overline{V_a}$ that occur in the pre- and post-condition also appear in the scenario. These concrete variables that correspond to $\overline{V_a}$ are denoted as $\overline{V_{c_a}}$.

The following subsection describe each of the components introduced by the encoding in the Event-B model.

UC Static

This context models the static aspects of the use case. The sets $\overline{S}$ and constants $\overline{C}$ that are defined by all the agent A (that plays a role in this UC), are introduced in this context. The types $T_c(\overline{S}, \overline{C})$ for these constants defined by the agent are introduced as axioms of the context. This context is *seen* by both machines UC Contract and UC Scenario. This allows the machines to use the sets and constants.

UC Contract

This machine models the *contract* of the use case. The contract specifies what the execution of the use case must achieve (*post-conditions*) given a promised or guaran-

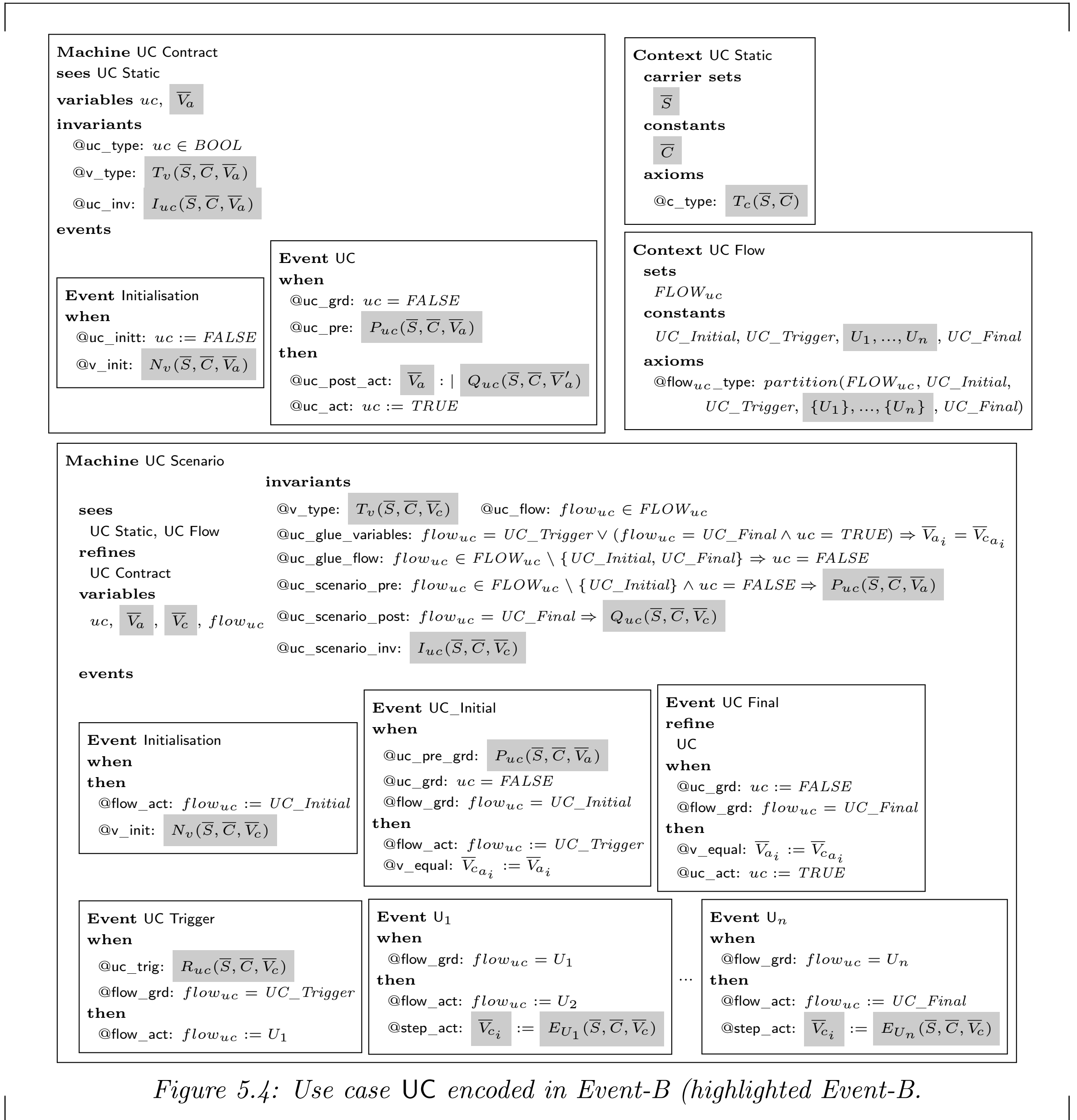


Figure 5.4: Use case UC *encoded in Event-B (highlighted Event-B.*

teed set of *pre-conditions*. By modelling the pre- and post-conditions in this abstract machine it is possible to establish *what* the use case achieves without specifying *how*.

The abstract variables, $\overline{V}_a$, that occur in the pre- and post-conditions of the use case are introduced as variables of the abstract machine, as seen in Figure 5.4. The types associated with the abstract variables are denoted by $T_v(\overline{S}, \overline{C}, \overline{V}_a)$ in the encoding. These are introduced as *invariants* (labelled @v_Type) in this the machine. This corresponds to typing invariants in Event-B. An event UC is introduced that represents a state transition of the use case, i.e. an atomic execution of the use case. An auxiliary boolean variable, *uc*, is introduced to indicate whether the use case has been executed. This variable is used in the guards and actions of the event, i.e. the action $uc := TRUE$,

is an indication that the use case has been executed, as seen in Figure 5.4. The encoding models the pre-conditions and postconditions in the event UC as follows:

Pre-conditions The encoding represents the pre-condition as $P_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ that contains the abstract variables $\overline{V_a}$. These predicates are modelled as the guards of the event UC. Doing so ensures that the event may only be enabled when the guaranteed or promised pre-conditions for the use case is *true*.

Post-conditions The encoding represents the post-condition as $Q_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ that contains the abstract variables V_a. This predicate is transformed to produce the non-deterministic assignment with before-after predicate, $\overline{V_a} :\mid Q_{uc}(\overline{S}, \overline{C}, \overline{V}'_a)$, where the variables V_a appears on the LHS of the such that operator, while all occurrence of $\overline{V_a}$ in the predicates $Q_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ is primed $\overline{V}'_a$ and introduced on the RHS. This establishes a state transition where the post-condition is achieved by the execution of this event.

The invariants $I_{uc}(\overline{S}, \overline{C}, \overline{V})$ in which only the abstract variables $\overline{V_a}$ occur are introduced as invariants $I_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ in the abstract machine. The action of the event UC, that achieves the post-conditions, is required to be within bound of the invariant specified. This machine establishes an abstraction of the use case that describes what is achieved by the use case, via the event UC, without specifying how.

UC Flow

This context establishes a type $FLOW_{uc}$, which aims to model the states in the scenario of the use case UC. A variable of this type can be used to simulate the execution of the scenario in the concrete machine, UC Scenario. The type is created by adding the name of $FLOW_{uc}$ in the sets section. It is an enumerated set where all the elements are known and defined as follows:

UC_Initial Variables of type, $FLOW_{uc}$, are initialised to this value.

UC_Trigger This value indicates a state in the scenario of the use case, where the trigger condition may be checked to initiate the execution of the main flow.

U$_1$,...,U$_n$ These values indicates the the steps in the flow where the execution of the use case is in.

UC_Final This value indicates that the execution of the final step in the use case scenario has been executed. Note that it does not indicate that the use case is complete.

The *partition* operation is used to express this type and is introduced as an axiom (labelled @uc_label).

UC Scenario

This machine models the scenario of use case UC. The scenario captures behaviour that describes *how* the use case achieves its contract. This machine refines the abstract machine UC Contract. The scenario is modelled by a collection of events, as seen in Figure 5.4.

The encoding introduces an auxiliary control flow variable $flow_{uc}$. This variable is of the type $FLOW_{uc}$, which was modelled as a set by the UC Flow context. This variable controls the event ordering that simulate the sequencing of events that correspond to the scenario of the use case. The concrete variables $\overline{V_c}$ are introduced in this machine along with the types associated with them $T_v(\overline{S}, \overline{C}, \overline{V_c})$. The abstract variables uc and $\overline{V_a}$ remain in this machine alongside the concrete variables, $\overline{V_c}$ and $flow_{uc}$. The gluing invariants labelled @uc_glue_variables and @uc_glue_flow are introduced to relate the abstract variables and concrete variables. The invariant @uc_glue_variables ensures that all the abstract variables $\overline{V_{a_i}}$ that correspond to concrete variables $\overline{V_{c_{a_i}}}$ (where $1 \leq i \leq x$ and x is the total number of corresponding variables), have the same values before and after the scenario executes. This is achieved by the following invariant:

$$flow_{uc} = UC_Trigger \vee (flow_{uc} = UC_Final \wedge uc = TRUE) \Rightarrow (\overline{V_{a_i}} = \overline{V_{c_{a_i}}}) \quad \text{(uc_glue_variables)}$$

The invariant @uc_glue_flow ensures that the use case can never be complete (indicated by $uc = FALSE$) during the execution of its scenario. The following invariant relates the states of the abstract and concrete variables uc and $flow_{uc}$:

$$flow_{uc} \in FLOW_{uc} \setminus \{UC_Initial, UC_Final\} \Rightarrow uc = FALSE \quad \text{(uc_glue_flow)}$$

The abstract event UC no longer exists in this machine. Instead, the following events are automatically introduced by the encoding. The event UC Final refines the abstract event UC, while the other events refine *skip*:

UC Initial This event models the initialises the scenario, where the pre-condition of the use case must be guaranteed. This is achieved by modelling $P_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ as the guard of the event. The event also ensures that the concrete variables $\overline{V_{c_a}}$ that correspond to the abstract variables $\overline{V_a}$, have the same values via the action labelled @v_equal. The execution of this event leads to the event UC Trigger via the auxiliary control flow variable.

UC Trigger This event models the triggers of the main flow $R_{uc}(\overline{S}, \overline{C}, \overline{V_c})$, as its guard (labelled @uc_trig). The trigger captures the states that are required to be true in order to initiate the execution of the main flow. The execution of this event leads the flow of the use case to the event that represents the first step, U_1.

U_1 to U_n Each step in the main flow is modelled as an event. The action associated with each step are introduced as actions of the events (labelled @step_act in each event). The auxiliary control flow variable $flow_{uc}$ mediates the order in which the events are executed to simulate the sequence as in the scenario. The execution of the event U_n leads to the event, UC Final.

UC Final This event represents the *end* of the use case. It refines the abstract event UC. This requires the execution of this event to achieve the post-condition. The concrete variables V_{c_a} are expected to have been modified by the events of the main flow, U_1 to U_n, to achieve the post-condition. The action $\overline{V}_{a_i} := \overline{V}_{c_{a_i}}$ is introduced that assigns the values of the concrete variables to the variables. This event refines the abstract event UC, while the other events refine *skip*.

In order to ensure that the scenario satisfies the contract of the use case, the invariants @uc_scenario_pre, @uc_scenario_post and @uc_scenario_inv, are introduced in this machine. The following invariant @uc_scenario_pre, ensures the pre-condition $P_{uc}(\overline{S}, \overline{C}, \overline{V_a})$ was guaranteed before the scenario executed:

$$flow_{uc} \in FLOW_{uc} \setminus \{UC_Initial\} \wedge uc = FALSE \Rightarrow P_{uc}(\overline{S}, \overline{C}, \overline{V_a}) \tag{uc_scenario_pre}$$

The invariant @uc_scenario_post introduce a constraint where concrete variables $\overline{V_{c_a}}$ that correspond to the abstract variables $\overline{V_a}$ achieve the post-condition at the end of the scenario. The scenario describe state transitions for the concrete variables $\overline{V_c}$. The post-condition is transformed to $Q_{uc}(\overline{S}, \overline{C}, \overline{V_{c_a}})$, where all occurrence of the abstract variable $\overline{V_a}$ is replaced by the corresponding concrete variables $\overline{V_{c_a}}$. This invariant ensures that when the final event S_n executes, the post-condition is required to be achieved for the concrete variables.

$$flow_{uc} = UC_Final \Rightarrow Q_{uc}(\overline{S}, \overline{C}, \overline{V_{c_a}}) \tag{uc_scenario_post}$$

Finally, all the invariant of the use case that contain the concrete variables are introduced as $I_{uc}(\overline{S}, \overline{C}, \overline{V_c})$. This invariant ensures that the behaviour defined by the

scenario maintains the invariants of the use case for the concrete variables.

$$I_{uc}(\overline{S}, \overline{C}, \overline{V_c}) \qquad \text{(uc_scenario_inv)}$$

This machine establishes *how* the the contract of the use case is achieved. The encoding uses the refinement to relate the behaviour specified by the scenario satisfies the constraints of what is required to be achieved by the contract.

5.2.2 Verification

UC Contract

Figure 5.5 describes the proof obligations produced in the abstract machine and their meaning to the contract of the use case. The main mathematical judgement applied on the abstract machine UC Contract is to determine whether the invariants of the use case (labelled @uc_inv) are maintained by the execution of event UC. That is, the post-condition that describes what is achieved by the use case is within bounds with the invariant of the use case. This is formulated by the *invariant preservation* proof obligation (INV). Proving UC/uc_inv/INV, ensures that the before-after predicate $Q_{uc}(\overline{S}, \overline{C}, \overline{V}'_a)$ associated with the event UC maintains the invariant $I_{uc}(\overline{S}, \overline{C}, \overline{V_a})$. The invariant preservation for $T_{v_a}(s, c, v_a)$ applied on the event UC produces the proof obligation UC/v_a_type/INV.

Proof Obligation	Proof failure meaning towards Contract
UC/uc_inv/INV	The post-condition is not specified within bounds of the invariants of the use case.
UC/v_type/INV	The post-condition is not specified within bounds of the variable type.

Figure 5.5: Defects identified by proof obligations in contracts.

UC Scenario

Figure 5.6 describes the main proof obligations produced in this machine and the meaning of their failure to the scenario of the use case. In the concrete machine UC Scenario, the abstract event UC is refined by the events that model the scenario of the use case. The event UC Final refines the abstract event UC (the other events that represent the scenario refine *skip*). Proof obligations associated with guard strengthening (GRD) and action simulation (SIM) for the concrete event UC Final is produced to ensure that

the concrete behaviour of the scenario satisfies the abstract behaviour of the contract. This aims to prove two things: (1) the pre-condition must be guaranteed before the execution of the use case scenario and (2) the post-condition must be achieved at the end of the use case scenario. The invariants @uc_scenario_pre and @uc_scenario_post help automatically prove the guard strengthening and action simulation proof obligation, as they ensure that the pre- and post-condition are provided before and after the execution of the use case scenario.

The invariant, @uc_scenario_post, ensures that any event that leads to the end of the use case, i.e. via the action $flow_{uc} := UC_Final$, must achieve the post-condition of the use case for the concrete variables. In the main flow the final step U_n leads to the end of the use case. This produces the proof obligation U_n/uc_scenario_post/INV that requires the final step to achieve the post-condition.

The invariants labelled uc_scenario_inv ensures that all the steps in the scenario maintains the invariants of the use case on the concrete variables v_{c_a}. Let U_i be a step in the scenario of the use case. The proof obligation U_i/uc_scenario_inv/INV for the invariant preservation ensures the steps maintain the invariants.

Proof Obligation	**Proof failure meaning towards Scenario**
U_n/uc_scenario_post/INV	A step leading to the end of the use case does not achieve the post-condition.
U_i/uc_scenario_inv/INV	A step U_i in the scenario of the use case does not satisfy the invariant of the use case.
U_i/v_type/INV	A step U_i in the scenario of the use case does not satisfy the type of the variable.

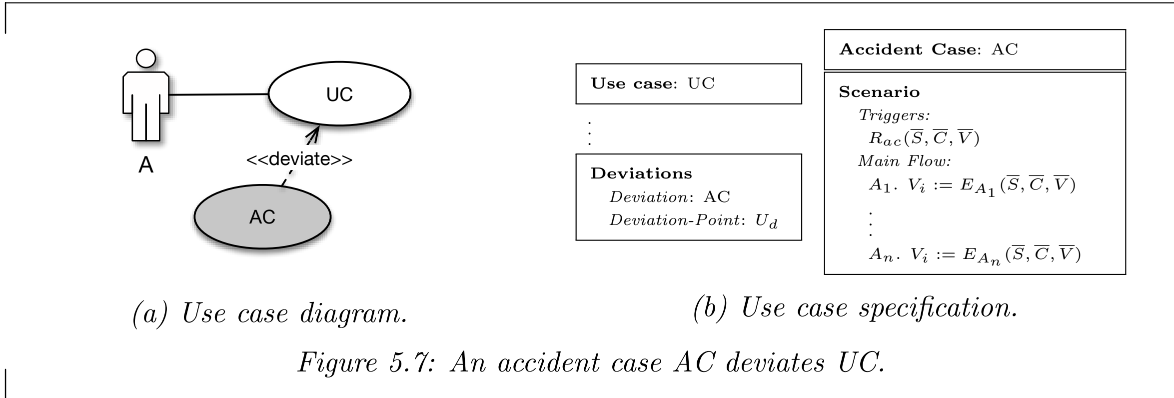
Figure 5.6: Defects identified by proof obligations in scenario.

5.3 Accident Case

The accident case introduces *undesired* behaviour that is expected to violate the contract of the use case it *deviates*. The use case UC from Figure 4.6, is updated with an accident case AC that deviates it, as seen in Figure 5.7. The use case diagram and the use case model are seen in Figure 5.7a and 5.7b, respectively.

The deviate relationship introduces the element *deviation* in the specification of the use case. This deviation provides a reference to the accident case AC and specifies a *deviation-point* at U_d where the scenario of the accident case can be introduced. The deviation-point U_d is some step in the scenario of the use case between steps U_1 to U_n. The accident case AC specifies a scenario that contains the steps A_1 to A_n. Each step

captures an action that modifies some variable V_i that belong to the variables $\overline{V}$. The carrier sets $\overline{S}$, constants $\overline{C}$ and variables $\overline{V}$, defined by the agent A and plays a role in use case UC, are used to specify the scenario of the accident case. The encoding of accident cases in Event-B is provided next.



(a) Use case diagram. *(b) Use case specification.*

Figure 5.7: An accident case AC deviates UC.

5.3.1 Encoding in Event-B

The encoding introduces the scenario of the accident case in the Event-B model that was produced for the use case UC it deviates (from Figure 5.4). The execution of the accident case depends on the deviated use case. This updated Event-B model is seen in Figure 5.8. The encoding introduces the scenario of the accident case alongside that of the use case. The accident case scenario is introduced in the existing machine UC Scenario. The context UC Flow is expanded with the steps of the accident case scenario.

The context UC Context models any new carrier sets and constants associated with the accident scenario. The encoding treats the variables that occur in the accident scenario as part of the concrete variables $\overline{V_c}$ from the encoding of the use case UC. That is, some of these variables ($\overline{V_{c_a}}$), used to specify the accident scenario, correspond to the abstract variables $\overline{V_a}$ specified in the contract of the use case UC.

UC Flow

The type $FLOW_{uc}$ is extended with the steps $A_1, ..., A_n$ of the accident scenario, as seen in Figure 5.8. This allows the auxiliary control flow variable of type, $FLOW_{uc}$, to introduce the scenario of the accident case as a deviation from the use case scenario. The steps are introduced as constants. The *partition* operation define these steps as part of the type $FLOW_{uc}$.

UC Scenario

The steps in the accident scenario $A_1, ..., A_n$ and its triggers are introduced as events in this machine. They exist along side the events that correspond to the scenario of the use case UC. These events are as follows:

AC Trigger This event deviates the execution in the scenario from the use case to that of the accident case. The triggers of the accident scenario are denoted as $R_{ac}(\overline{S}, \overline{C}, \overline{V_c})$ by the encoding. These triggers, and the deviation-point $flow_{uc} = U_d$, are introduced as guards of this event. These guards ensure that the accident scenario may only initiate when the deviation-point in the use case scenario is reached and the trigger conditions are *true*. The action of this event deviates the scenario of the use case to the first step in the accident scenario, i.e. $flow_{uc} := A_1$.

A_1 to A_n Each step of the accident scenario is introduced as an event. The actions of the step are modelled as actions of the event. The actions of the accident scenario is required to introduce undesired behaviour that violate some property of the use case it deviates. The execution of the event A_n that represents the final step of the accident scenario, leads to the end of the use case, i.e. $flow_{uc} := UC_Final$.

When the final event A_n executes, the post-condition of the use case UC is required to be achieved. The accident scenario is also required to maintain the invariant of the use case throughout its execution. However, as the accident scenario captures undesired behaviour, this is expected to violate the contract of the use case.

5.3.2 Verification

Introducing the accident scenario in the machine UC Scenario results in new proof obligations generated in the Event-B model to ensure consistency. Figure 5.9 describes some of these proof obligations that can be related to the scenario of the accident case. The events that model the scenario of the accident case are subjected to the invariants labelled @uc_scenario_post and @uc_scenario_inv, as seen in Figure 5.4. The invariant @uc_scenario_post requires the final step A_n of the accident scenario to achieve the post-condition of the use case. This is because the event A_n, which models this step, leads to the end of the use case. The proof obligation of this final event is expected to be fail, otherwise the accident scenario is not a correct deviation of the use case.

The invariants of the use case are required to be maintained by the steps of the accident scenario. This is checked by the proof obligation A_n/uc_scenario_post/INV. As the accident scenario is introduced as a deviation from the use case scenario, the pre-condition of the use case can be shown to be guaranteed.

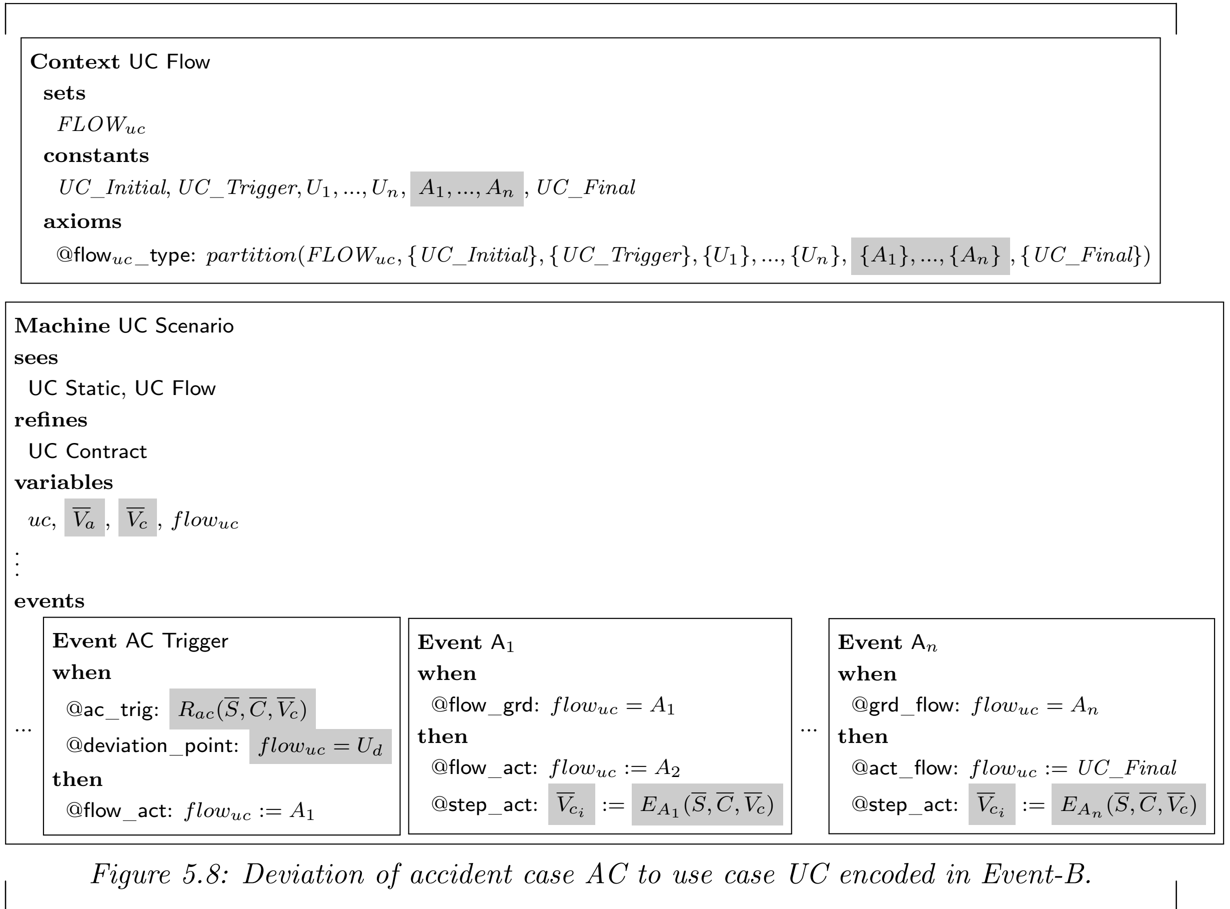


Figure 5.8: Deviation of accident case AC to use case UC encoded in Event-B.

5.4 Extension Use Case

An extension use case can introduce additional behaviour in the scenario of a use case. In Figure 5.10, an extension use case EC is introduced as an extension to the use case UC. The use case diagram illustrates this extension via the *extend* relationship as seen in Figure 5.10b with the use case specifications. This extension refers to the extension use case EC. It provides an extension-point U_e and a rejoin-point U_r. In this instance, the status of the extension is *ordinary*. Hence, the extension-point and rejoin-point are specified as some steps in the scenario of the use case (such as $U_e \in U_1, ..., U_n$ and $U_r \in U_1, ..., U_n$). The extension use case EC specifies a contract and a scenario, as seen in Figure 5.10. The encoding of this extension use case EC in the Event-B is provided next.

5.4.1 Encoding in Event-B

The extension use case is dependant of the use case it extends. This requires the encoding of the use case UC to be already provided, as seen in Figure 5.4. The encoding of the extension use case EC introduces the contract of the extension use case in the

Proof Obligation	Proof failure meaning towards accident scenario
A_n/uc_scenario_post/INV	Final step of accident scenario does not achieve post-condition. This PO is expected to fail unless the accident case is shown to be prevented.
A_i/uc_scenario_inv/INV	A step of the accident scenario where its action does not satisfy the invariants of the use case.
A_i/v_type/INV	A step of the accident scenario where its action does not satisfy the type defined by agent for a variable.

Figure 5.9: Defects related to an deviation identified by proof obligations.

machine UC Scenario as seen in Figure 5.11. The contract of EC is introduced between the steps in scenario of use case UC where the extension-point is specified. The encoding of the contract of EC models only *what* the extension use case may achieve, and does not specify *how*. The scenario of the extension use case is introduced in a new machine EC Scenario that refines the machine UC Scenario. This encoding determines a new set of abstract $\overline{V_a}$ and concrete $\overline{V_c}$ variables, as follows:

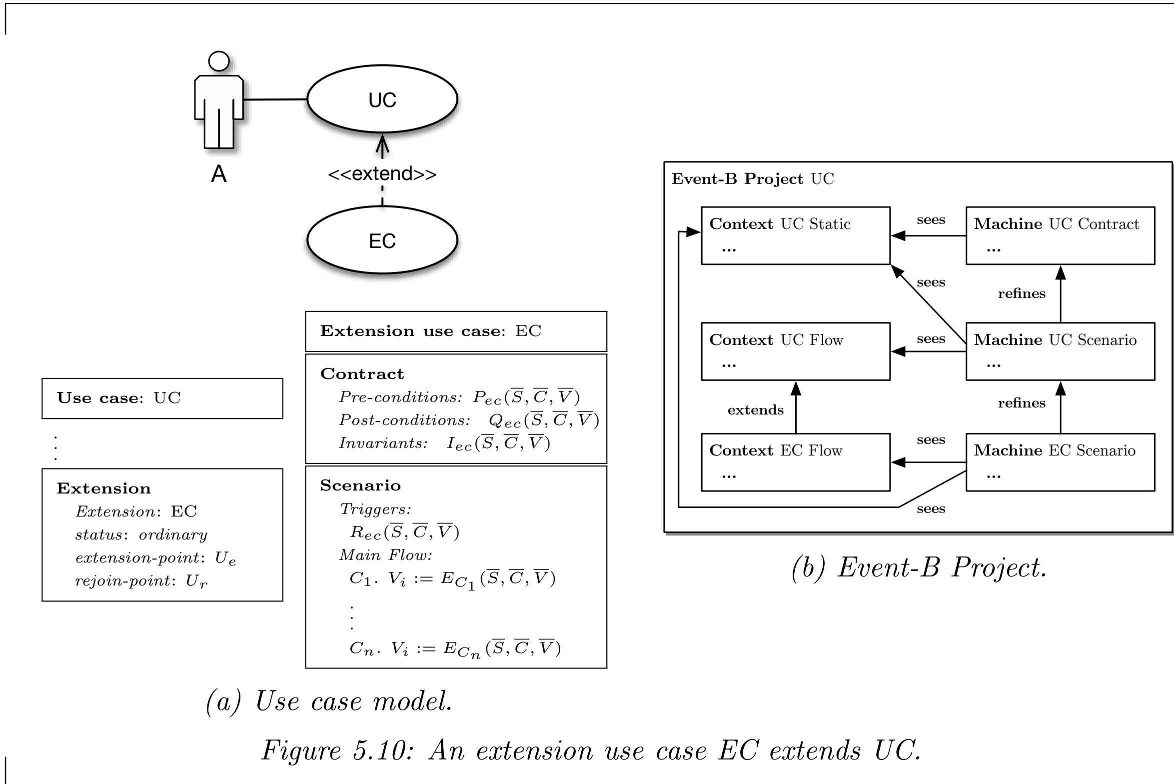
Abstract variables The existing variables of the UC Scenario machine (from Figure 5.4) excluding the auxiliary variables uc and $flow_{uc}$, are now treated as the abstract variables $\overline{V_a}$. These abstract variables also include the variables that occur in the pre-condition and post-condition in the contract of the extension use case EC.

Concrete variables The variables that occur in the scenario of the extension use case are now treated as the concrete variables $\overline{V_c}$.

All the static aspects of the use case and the extension use case have already been modelled in the context UC Static. The flow of the extension use case is modelled by the context EC Flow, which refines the context UC Flow. The new machine EC Scenario *sees* the contexts UC Static and EC Flow.

UC Scenario

The encoding introduces the contract of EC between the events U_e (extension-point) and U_{e-1}, in the scenario of UC. The encoding of the contract for the extension use case produces two events, EC and EC_FALSE. An auxiliary boolean variable ec is introduced that helps insert these two events between the events U_e and U_{e-1}. That is, the contract is introduced before the event that represents the specified extension-point U_e. The events related to the encoding of the extension use case in the use case scenario are as follows:



(a) Use case model.

(b) Event-B Project.

Figure 5.10: An extension use case EC extends UC.

EC The event models the contract of EC. The pre-conditions $P_{ec}(\overline{S}, \overline{C}, \overline{V_a})$ of EC was modelled as the guards of this event. The post-condition is introduced as the action. This is similar to the encoding of the contract of UC in the abstract machine. However, an additional guard $flow_{uc} = U_e$ and action $flow_{uc} := U_r$ are introduced to ensure that the event can be enabled only at the specified extension-point U_e and the execution of the event returns the flow to some step specified as the rejoin-point U_r.

EC_FALSE This models a negation of the pre-condition of EC, $\neg(P_{ec}(\overline{S}, \overline{C}, \overline{V_a}))$. If the pre-condition of EC is not guaranteed then the post-condition of the extension use case is not achieved. That is, the extension use case is not required to be executed at the extension-point. The action of this event returns the flow to step U_e.

U_e The event EC of the extension use case is required to be introduced before the execution of this event. To do so, an additional guard $ec = TRUE$, is introduced in the event U_e, as seen in Figure 5.11. This requires either the event EC or EC_FALSE to have been executed.

U_{e-1} The event U_{e-1} that is before U_e, has an additional action introduced $ec := FALSE$. This action ensures that either EC or EC_FALSE will be enabled after

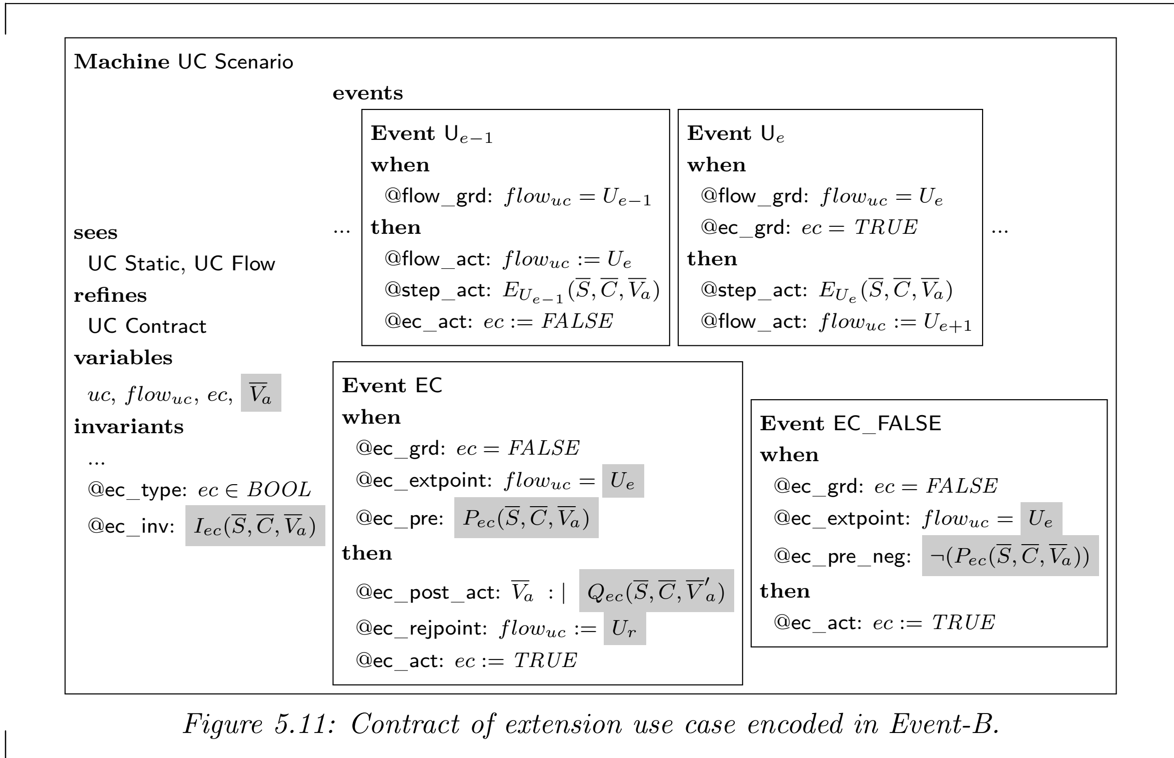


Figure 5.11: Contract of extension use case encoded in Event-B.

this step.

These events model the contract of the extension use case in the scenario of the use case UC, via the use of the extension-point. The invariants in the contract of the extension use case EC, in which the abstract variables $\overline{V_a}$ occur, are introduced as $I_{ec}(\overline{S}, \overline{C}, \overline{V_a})$. The events EC and EC_FALSE is required to maintain the invariants introduced for the use case scenario.

EC Scenario

The encoding introduces the scenario of the extension use case EC in machine EC Concrete, as seen in Figure 5.12. This encoding is similar to the encoding of the use case scenario in the UC Scenario machine (Figure 5.4), with a few differences. The following describes these similarities and differences:

Refines This machine *refines* UC Scenario in which the contract of the extension use case was introduced.

Variables The variables from the abstract machine remain in this machine. The new variables associated with the scenario of the extension use case, $flow_{ec}$ and $\overline{V_c}$ (concrete variables), are introduced in this machine. This is similar to the encoding of the use case scenario.

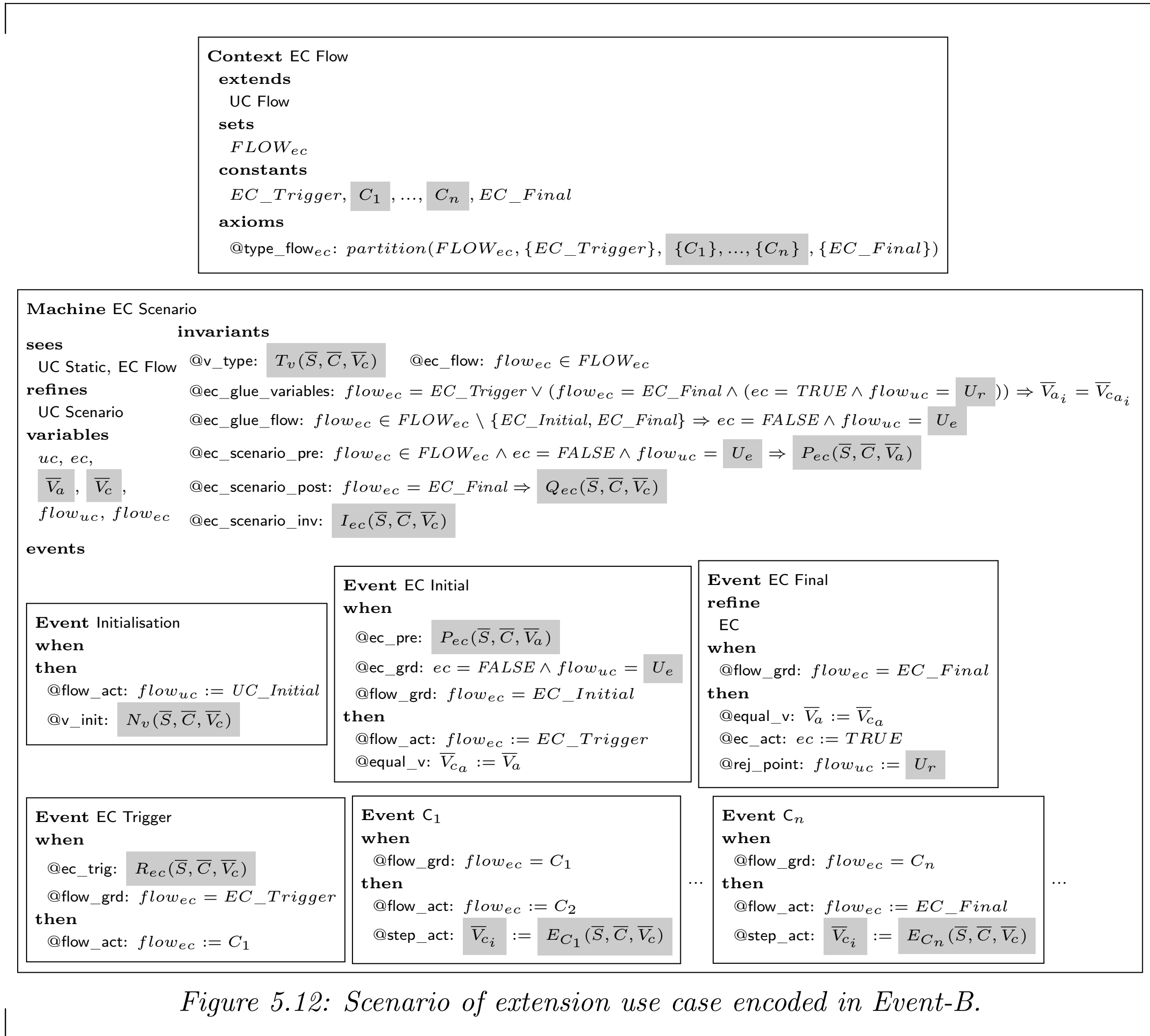


Figure 5.12: Scenario of extension use case encoded in Event-B.

Invariants The invariants introduced for the scenario of the extension use case are similar to those introduced for the use case scenario. The only difference is that $uc = TRUE$ corresponds to $ec = TRUE \wedge flow_{uc} = U_r$, while $uc = FALSE$ corresponds to $ec = FALSE \wedge flow_{uc} = U_e$. This is because the scenario of the extension use case is introduced between the scenario of the use case. This results in that the extension-point and rejoin-point are taken into account in the invariants.

Abstract Events All the events of the abstract machine are introduced in this machine, apart from the event name EC, which is refined by the events introduced in the scenario of the extension use case.

EC Initial This event models the initialisation of the extension use case scenario. An additional guard $flow_{uc} = U_e$ which ensures that the execution of the extension use case's scenario takes place at the extension point, is added.

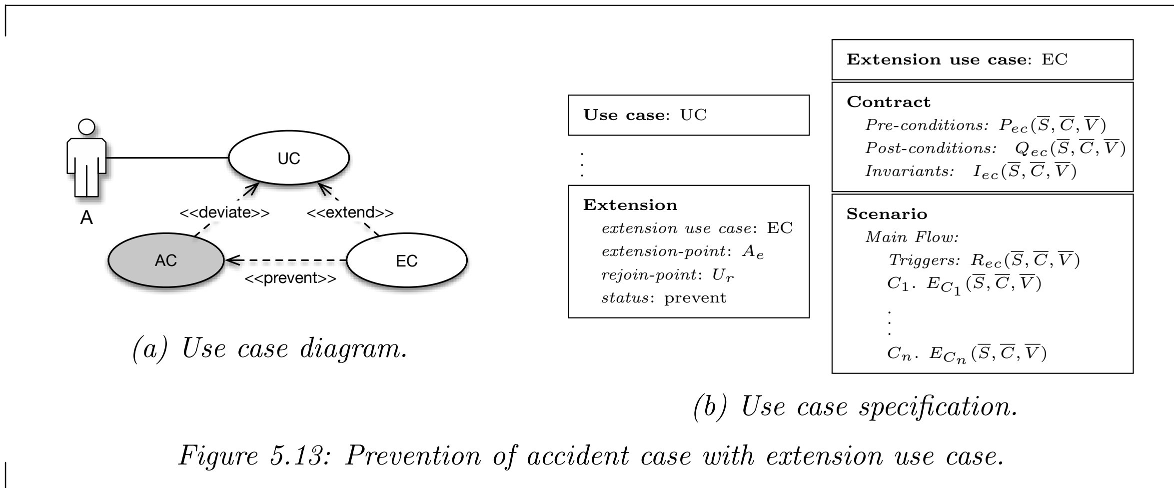
EC Final This event refines the abstract event EC. It has one additional action $flow_{uc} := U_r$, which ensures that after the extension use case's scenario has executed, the flow of UC rejoins at the step specified by the extension relation.

The encoding ensures that the abstract event EC that models the contract of the extension use case, is refined by the events introduced by its scenario.

Prevent

If the *status* of the extension is *prevent*, then the extension use case is introduced as a means to prevent an accident case from violating the contract of the deviated use case. In Figure 5.10, the extension use case EC extends the use case UC, by preventing any deviation to the accident case AC. This results the *extension-point* to specify some step A_e in the accident scenario. The *rejoin-point* in the extension specifies some step in the scenario of the use case or accident scenario. If the rejoin-point is not specified, then it returns to the end of the use case. The extension use case must always execute during the accident scenario. That is, the behaviour of the extension use case must prevent the remaining steps of the accident scenario to complete execution. The following invariant is introduced in the machine the contract of the extension is introduced, in this instance UC Scenario if the prevent status is provided:

$$\neg(ec = FALSE \wedge flow_{uc} = A_e \wedge \neg(P_{ec}(\overline{S}, \overline{C}, \overline{V_a})) \quad \text{(ec_prevent)}$$



(a) Use case diagram.

(b) Use case specification.

Figure 5.13: Prevention of accident case with extension use case.

5.4.2 Verification

The EC event introduces the contract of the extension use case within the scenario of the use case via the extension-point. Figure 5.14 describes what defects these proof

obligations produced in this machine and the meaning of their failure to the extension introduced in the use case scenario relates to in the use case. What is achieved by the contract of the extension use case must be shown to satisfy the invariants (@EC/uc_scenario_inv/INV) of the use case and the types (@EC/v_type/INV) of the concrete variables. If the rejoin-point is not specified, then the execution leads to the end of the use case. This requires the post-condition of the extension use case to achieve the post-condition of the use case it extends (@EC/uc_scenario_post/INV).

Proof Obligation	**Proof failure meaning towards EC**
EC/ec_scenario_post/INV	This PO is produced only if the rejoin-point of the extension leads to the end of the use case, i.e. $flow_{uc}$:= *UC_Final*. It indicates a defect where the post-condition of the extension use case is not established by its scenario.
EC/ec_scenario_inv/INV	Indicates that what is achieved by the extension use case (post-condition transformed to action) does not maintain the invariants of the use case it extends.
EC/v_c_type/INV	Indicates that what is achieved by the extension use case (post-condition transformed to action) does not maintain the type of the variable.

Figure 5.14: Defects related to an extension identified by proof obligations.

5.5 Encoding Branching in Event-B

As discussed in Chapter 4, branching in the scenario of use cases is supported by simple and complex branching. Simple branching provides the use of conditional and loop constructs within a flow. On the other hand, complex branching allows one or more alternate flows that are explicitly stated below the main flow of a scenario. This section provides the encoding of these styles of branching in the Event-B model.

Conditionals

Figure 5.15a describes a conditional if branching in the scenario of an use case model. The step U_i specifies a conditional branching with the predicate $C_{U_i}(\overline{S}, \overline{C}, \overline{V})$. The steps U_{i_1} to U_{i_n} act as sub-steps that belong to the step U_i. Each of these sub-steps capture an action. The encoding of this conditional step U_i and its sub-steps in Event-B is provided in Figure 5.15b.

The step U_i is modelled by two events if U_i and if U_i FALSE. The event if U_i is *enabled* when the execution of the scenario reaches the step U_i and the predicate

$C_{U_i}(\overline{S}, \overline{C}, \overline{V})$. The execution of this event leads to the sub-step U_{i_1}, and so on. The execution of the final sub-step U_{i_n} leads to the step U_{i+1}. On the other hand, the event if U_i FALSE is also enabled when the execution of the scenario reaches the step U_i. However, it captures a negation of predicate $\neg(C_{U_i}(\overline{S}, \overline{C}, \overline{V}))$ as its guard. This allows the execution of the scenario to skip the sub-steps that belong to U_i, and lead the execution to step U_{i+1}.

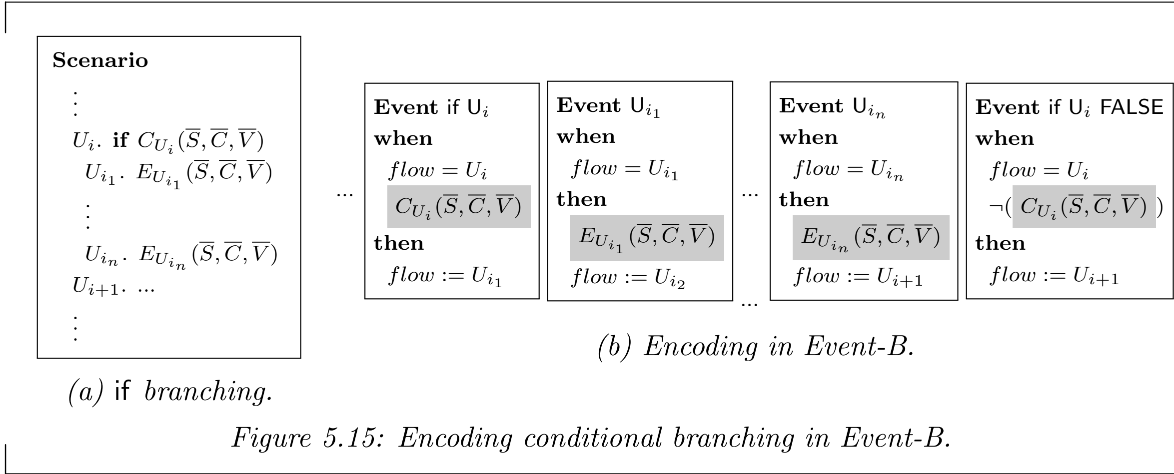


Figure 5.15: Encoding conditional branching in Event-B.

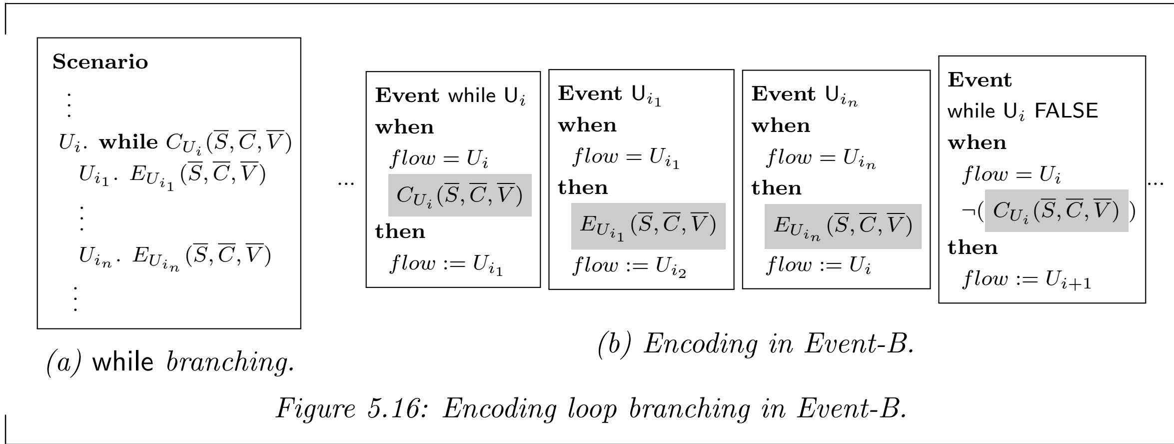
Loops

A simple branching provided by the *loop* construct allows a collection of steps to be repeated several times in the execution of the scenario. Figure 5.16a illustrates a loop construct for a step U_i. This step captures the predicate $C_{U_i}(\overline{S}, \overline{C}, \overline{V})$ and has a collection of sub-steps U_{i_1} to U_{i_n}. The structure of the *loop* is similar to that of the conditional seen in Figure 5.15a. However, the semantics of the constructs differ in the sense that the sub-steps of U_i are repeatedly executed as long as the predicate $C_{U_i}(\overline{S}, \overline{C}, \overline{V})$ is true when the execution reaches the step U_i. The encoding of the loop construct in Event-B, as seen in Figure 5.16b, is similar to that of the conditional. The only difference with respect to the encoding of the condition construct is that, the execution of the final sub-step U_{i_n} *returns* to the step U_i.

The event if U_i FALSE is enabled when the execution of the scenario reaches the step U_i. It captures a negation of predicate $\neg(C_{U_i}(\overline{S}, \overline{C}, \overline{V}))$ as its guard. This allows the execution of the scenario to break from the loop when the predicate is *false*, and leads to the step U_{i+1}.

Alternate Flow

Each scenario in the use case model may contain one or more alternate flows. These are alternative paths from the main flow in the scenario. This, unlike the flow of an



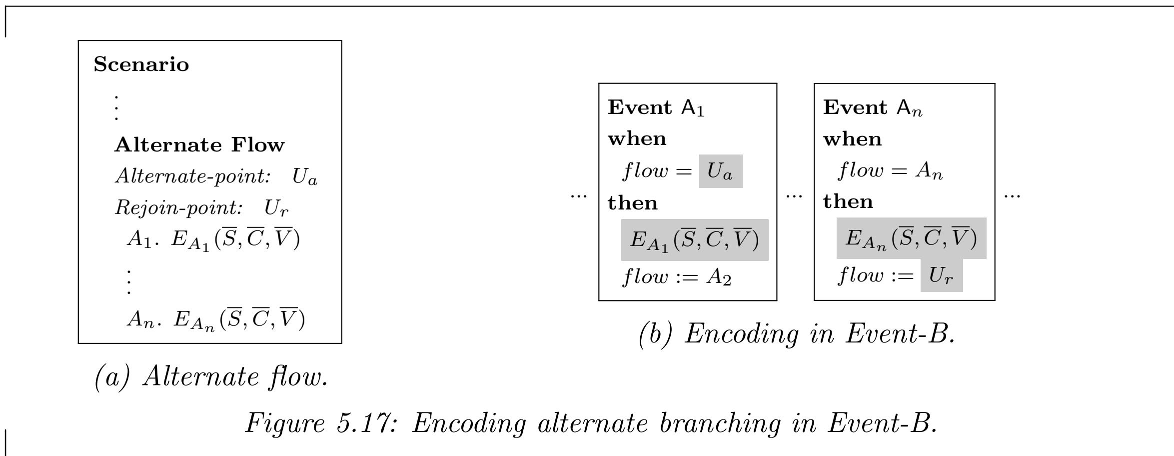
(a) while *branching.*

(b) Encoding in Event-B.

Figure 5.16: Encoding loop branching in Event-B.

accident scenario, is allowed to complete and does not require a prevention. A scenario always contains one main flow and any number of alternate flows. The key point of the alternate flow is that they frequently do not return to the main flow. Unlike the scenario of an accident, the alternate flow is expected to satisfy the contract of the use case. These alternate flows are appended to the end of the structure of a scenario, after the main flow. An alternate flow specifies an *alternate-point* that specifies a trigger in the main flow. The formal specification for an alternate flow is seen in Figure 5.17a, and its transformation to an Event-B model is seen in Figure 5.17b.

The alternate flow introduces the steps A_1 to A_n, each capturing their own actions. The alternate-point and rejoin-point are specified to some step in the use case the alternate flow belongs to, U_a and U_r, respectively. The encoding of the alternate flow in the Event-B model is seen in Figure 5.17b, where all the steps of the alternate flow are introduced as events A_1 to A_n. The guard of event A_1 ensures the event is enabled only at the specified alternate point, while the action of event A_n ensures the flow rejoins to some point in the flow after the alternate flow has completed execution.



(a) Alternate flow.

(b) Encoding in Event-B.

Figure 5.17: Encoding alternate branching in Event-B.

5.6 Translation Rules

This section provides the translation rules for encoding use cases in Event-B. The syntax used to describe the semantics in the translation is seen in Figure 5.18. The top row provides the semantic rule that has a rule number, indicated by Rule #. The rule numbers are used to refer to the translation rule in the text. The rule is also of a certain rule type, e.g. when in the text it is stated that RULETYPE is used, then it indicates that the rule with the set of *arguments* of the specified rule is called. If several rules of the same parameter are used, then both are used unless specified otherwise. The arguments are references to the abstract syntax provided in Section 4.5. The second row provide an unpacking of the argument, i.e. it provides what is defined by the argument. The third row provides the Event-B elements that are produced using the unpacked content of the use case.

TRule #: RULETYPE⟦⟨Usecase⟩⟧
⟨Usecase⟩
⟨Event-B⟩ | REFERENCE⟦⟨arguments⟩⟧

Figure 5.18: Syntactic form of translation rule.

Figure 5.19 provides an overview of the translation rules provided. The boxes denote a translation rule of a specific type and list of arguments. In total there are ten types of translation rules. Each of these translation rules are discussed in the following sub sections. The directed arrows indicate the translation rules produced from a source translation rule. The first translation rule in this tree PROJECT⟦⟨usecase⟩⟧.

5.6.1 Rule Type: Project

The rule type PROJECT is given a use case and an Event-B project is produced.

$$\text{PROJECT: Use Case} \hookrightarrow \text{Event-B Project}$$

Figure 5.20 describes the translation rule TRule 1 of rule type PROJECT that takes the argument ⟨usecase⟩. The middle row shows an unpacking of this argument to reveal that the use case contains a name, label, etc. The argument is unpacked only to show what is required by the translation rule. The bottom row shows the Event-B elements that are produced. This rule creates an Event-B project for the given use case, and within it produces two machines components and two context components.

The function used by the translation rule to create the Event-B components are as follows:

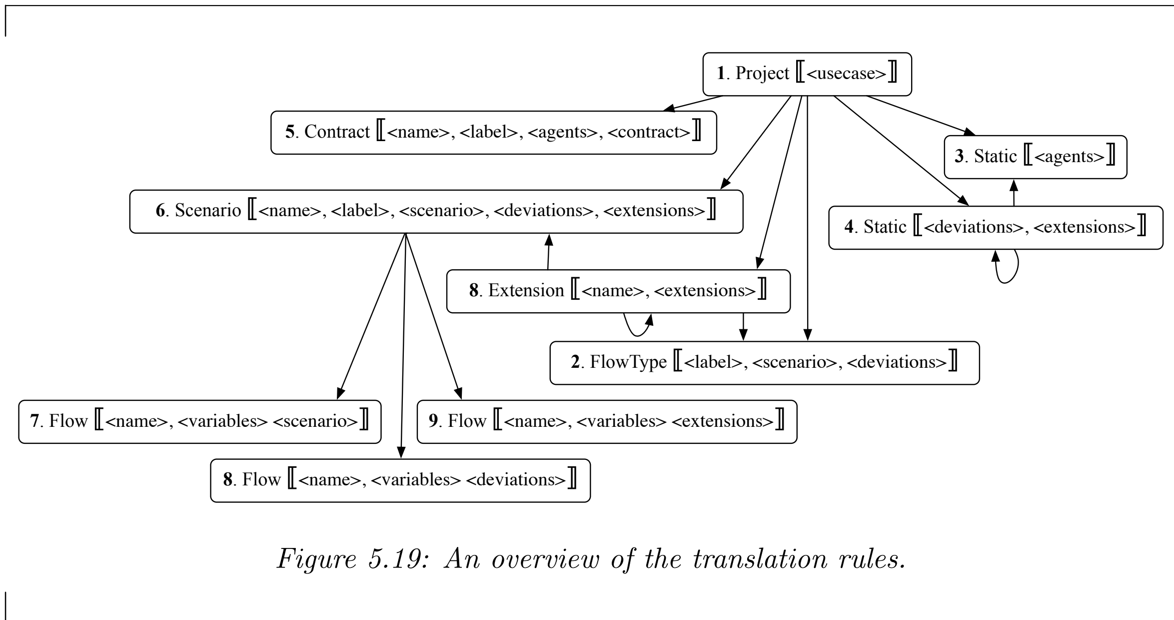


Figure 5.19: An overview of the translation rules.

getName(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨name⟩.

getNameContract(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨name⟩ with suffix "_Contract".

getNameStatic(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨name⟩ with suffix "_Static".

getNameFlow(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨name⟩ with suffix "_Flow".

getNameScenario(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨name⟩ with suffix "_Scenario".

createSees(⟨identifiers⟩): Creates *sees* element in the machine for each ⟨identifier⟩ in ⟨identifiers⟩.

createRefines(⟨identifiers⟩): Creates a *refines* element in the machine for each ⟨identifier⟩ in ⟨identifiers⟩.

createMachine(⟨identifiers⟩): Creates *machine* element in the machine for each ⟨identifier⟩ in ⟨identifiers⟩.

createContext(⟨identifiers⟩): Creates *context* element in the machine for each ⟨identifier⟩ in ⟨identifiers⟩.

createProject(⟨identifier⟩): Creates an Event-B *project* in the workspace of the UC-B for each ⟨identifier⟩.

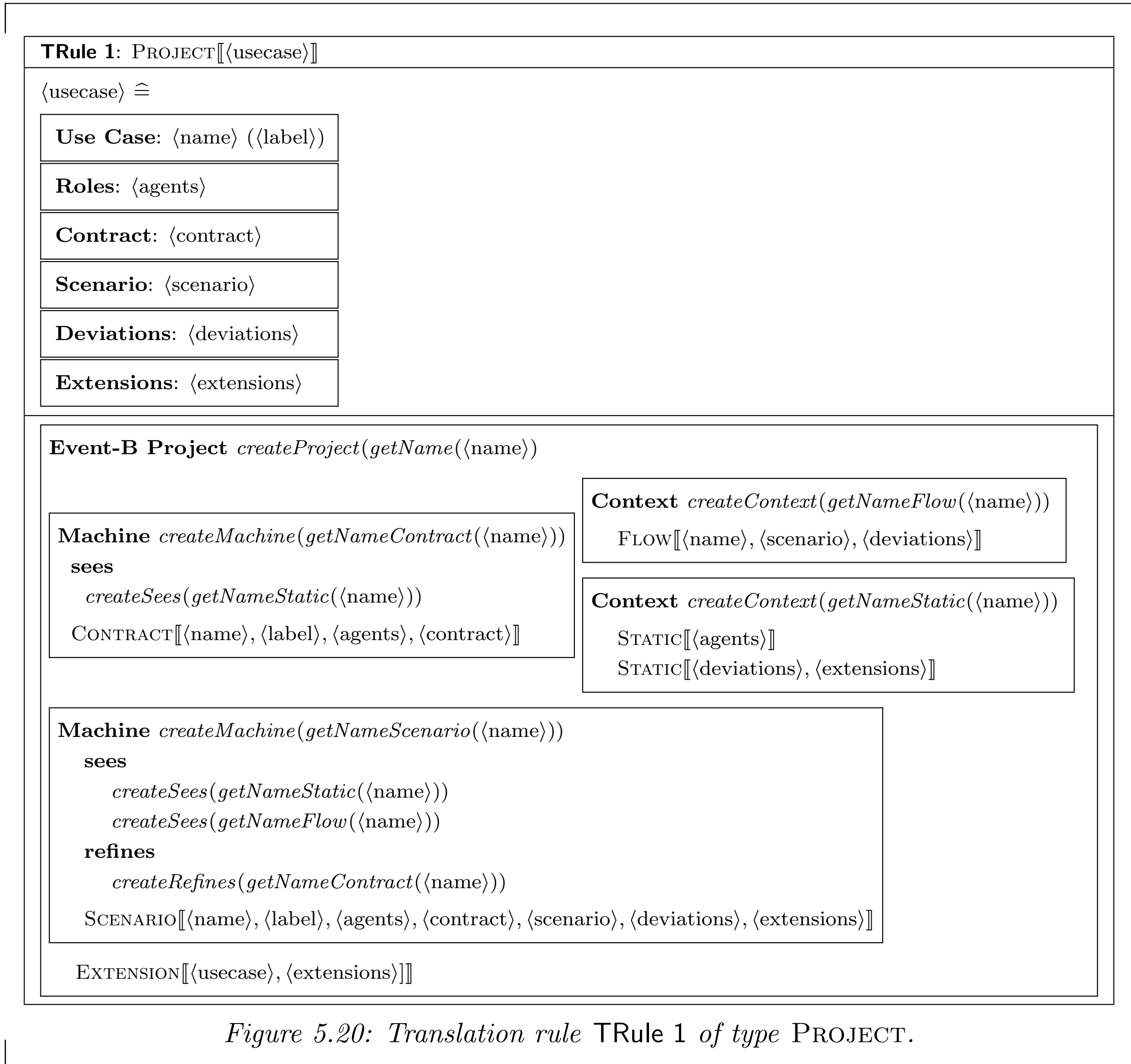


Figure 5.20: Translation rule TRule 1 *of type* PROJECT.

This translation rule has only created a skeleton of an Event-B project. However, within the components created are new translation rules that are applied. These new translation rules describe how content of the use case are used to create the Event-B elements. Each of these new translation rules are described in the following sections.

5.6.2 Rule Type: FlowType

The rule type FLOWTYPE models the scenario of the use case and any accident scenarios (deviations) in the context of Event-B that the rule belongs to.

FLOWTYPE: Scenario and Deviations ↪ Flow Type in Context

The translation rule TRule 2, as seen in Figure 5.21, takes the label, scenario and zero or more deviations as its arguments. It uses the following functions to produce the elements within this context component:

getNameFlowType(⟨label⟩): Returns an ⟨identifier⟩ with the literal value of ⟨label⟩ with suffix "_FLOW".

getNameFlowInitial(⟨label⟩): Returns an ⟨identifier⟩ with the literal value of ⟨label⟩ with suffix "_Initial".

getNameFlowTrigger(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨label⟩ with suffix "_Trigger".

getNameFlowFinal(⟨name⟩): Returns an ⟨identifier⟩ with the literal value of ⟨label⟩ with suffix "_Final".

getSteps(⟨scenario⟩, ⟨deviations⟩): The literal value of ⟨label⟩ in each step of the scenario and deviations are returned as a list of ⟨identifiers⟩.

createPartitionFlowType(⟨label⟩, ⟨scenario⟩, ⟨deviations⟩): Creates a predicate that denotes the type of the flow using the partition operator.

createSet(⟨identifiers⟩): Creates a *set* in the context component for each ⟨identifier⟩ in ⟨identifiers⟩.

createConstant(⟨identifiers⟩): Creates a *constant* in the context component for each ⟨identifier⟩ in ⟨identifiers⟩.

createAxiom(⟨predicates⟩): For each ⟨predicate⟩ in ⟨predicates⟩ it creates the axiom in the context component.

TRule 2: FLOWTYPE⟦⟨label⟩, ⟨scenario⟩, ⟨deviations⟩⟧
⟨label⟩ ⟨scenario⟩ ⟨deviations⟩
sets *createSet*(*getFlowType*(⟨label⟩)) **constants** *createConstant*(*getNameFlowInital*(⟨label⟩)) *createConstant*(*getNameFlowTrigger*(⟨label⟩)) *createConstant*(*getSteps*(⟨scenario⟩, ⟨deviations⟩)) *createConstant*(*getNameFlowFinal*(⟨label⟩)) **axioms** *createAxiom*(*createPartitionFlowType*(⟨label⟩, ⟨scenario⟩, ⟨deviations⟩))

Figure 5.21: Translation rule TRule 2 *of type* FLOW.

This translation rule creates a type based on the scenario of the use case and any accident scenarios that are referenced the deviations associated with this use case.

5.6.3 Rule Type: Static

The rule type STATIC generates all the static aspects, namely sets and constants, defined by the agents that play a role in the use case are introduced in the context component of the Event-B model. This also includes the accident cases and extension use cases that are related to the use case via the extensions and deviation relationships.

$$\textsc{Static}\text{: Agents} \hookrightarrow \text{Sets and Constants in Context}$$

The translation rule TRule 3, as seen in Figure 5.22, takes the one or more agents as its argument. The middle row shows an unpacking of an agent that reveals the all the sets and constants defined by the agent. The translation rule uses the following functions to create the static aspects in the context components:

getSets(⟨sets⟩): For each ⟨set⟩ in ⟨sets⟩ it returns the identifier ⟨set⟩.⟨identifier⟩.

getConstants(⟨constants⟩): For each ⟨constant⟩ in ⟨constant⟩ it returns the identifier ⟨constant⟩.⟨identifier⟩.

getConstantsType(⟨constants⟩): For each ⟨constant⟩ in ⟨constant⟩ it returns the predicate ⟨constant⟩.⟨predicate⟩.

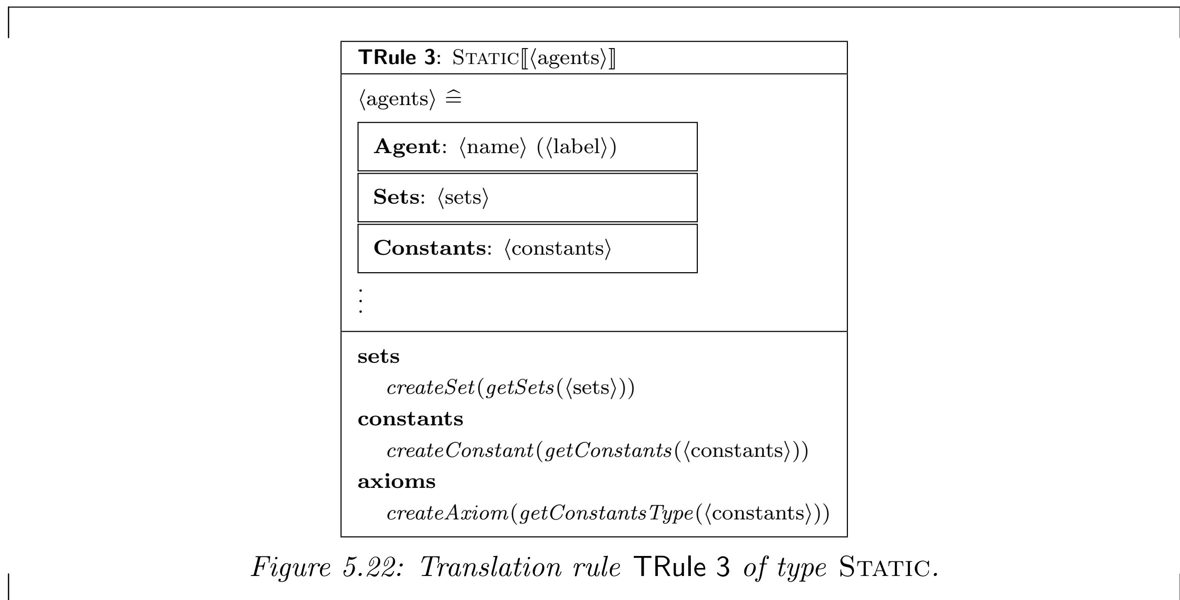


Figure 5.22: Translation rule TRule 3 *of type* STATIC.

The translation rule Trule 4 as seen in Figure 5.23 is also of the rule type STATIC. This translation rule does not produce any Event-B elements. The argument of this rule contains zero or more deviations and extensions. Its middle row reveals the agents that may belong to these deviations and extensions. The rule applies the translation rules STATIC⟦⟨agents⟩⟧ for all the agents that belong to the accident case and extension

use case that belong to the deviations and extensions, respectively. The extensions may also contain further deviations and extensions. STATIC⟦⟨deviations⟩, ⟨extensions⟩⟧ translation rule is applied again for these deviations and extension that belong to the extension use case.

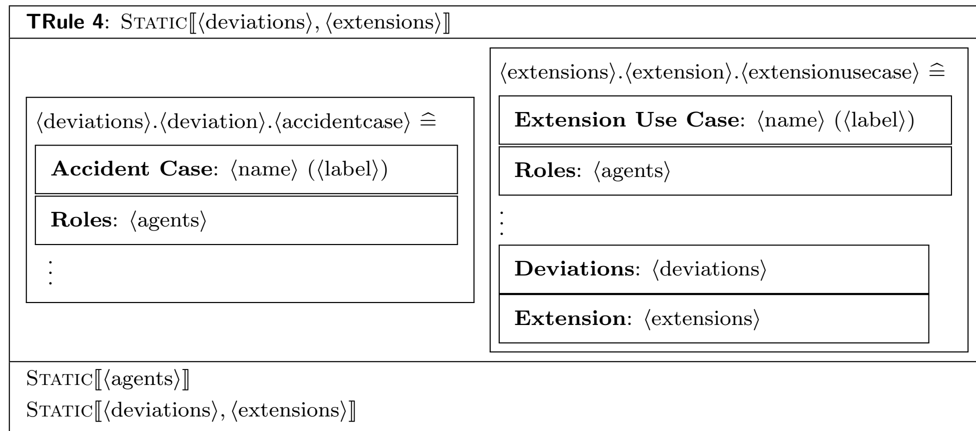


Figure 5.23: Translation rule TRule 4 *of type* STATIC.

5.6.4 Rule Type: Contract

The rule type CONTRACT introduces the behaviour of what the use case achieves in the machine component of Event-B.

CONTRACT: Contract ↪ Contract modelled in Machine

The translation rule TRule 5, as seen in Figure 5.24, takes the label, name, agents and contract as the arguments. The middle row shows the content of the arguments unpacked that are used by the translation rule. The translation rule creates two events, one of this is the INITIALISATION event. The following functions are used by this rule to create the elements within the machine components:

createAuxiliaryVariable(⟨label⟩): Creates an auxiliary boolean variable using the literal value of ⟨label⟩ that contains an identifier, predicate (that denotes its type) and initialisation. For example, a label UC would create a variable with identifier UC, type $UC \in BOOL$, and initialisation $UC := FALSE$.

getAbstractVariables(⟨agents⟩, ⟨pre-conditions⟩, ⟨post-conditions⟩): Returns a collection of variables in the agents that occur in the predicates of the ⟨pre-conditions⟩ and ⟨post-conditions⟩. These variables are treated as the abstract variables.

getVariableType(⟨variables⟩): For each ⟨variable⟩ in ⟨variables⟩ this function returns the predicate of the variable that denotes its type.

getAssociatedInvariants(⟨variables⟩, ⟨invariants⟩): This function returns the ⟨predicates⟩ for each of the ⟨invariants⟩ where the variables are found to occur.

getVariableInitialisation(⟨variables⟩): This function returns the ⟨action⟩ that denotes the initialisation for each ⟨variable⟩ in ⟨variables⟩.

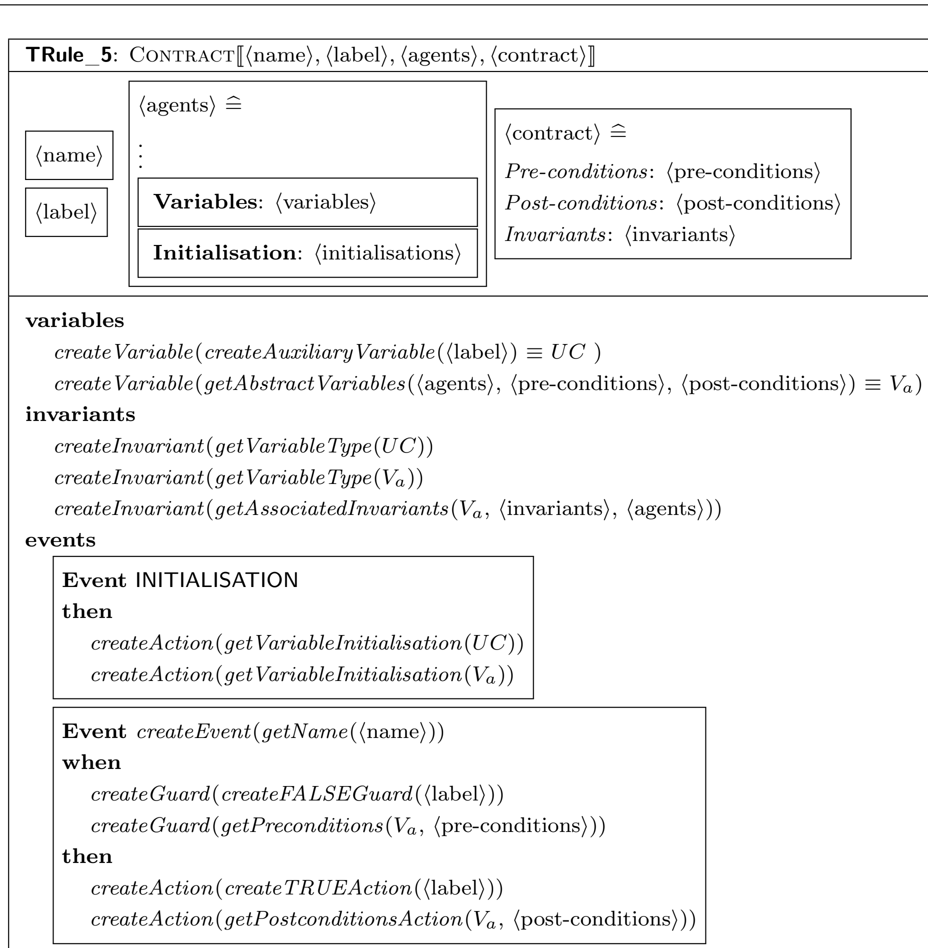


Figure 5.24: Translation rule TRule 5 *of type* CONTRACT.

createFALSEGuard(⟨label⟩): Creates an *action* using the label, e.g. for a label UC an action *UC* := *FALSE* is produced.

createTRUEAction(⟨label⟩): Creates an *action* in the event that assigns the value *true* for the auxiliary variable, e.g. for a label UC an action *UC* := *TRUE* is produced.

getPreconditions(⟨variables⟩, ⟨pre-conditions⟩): Returns a list of predicates for all the pre-conditions provided. If the variables provided have a corresponding abstract

variable, these variables that occur in the pre-condition are updated with the name of the refined variable.

getPostconditionstsAction(⟨variables⟩, ⟨post-conditions⟩): Creates a becomes such that action with the post-conditions.

createVariable(⟨identifier⟩): For each ⟨identifier⟩ in ⟨identifiers⟩ it creates a variable in the machine.

createInvariant(⟨predicates⟩): For each ⟨predicate⟩ in ⟨predicates⟩ it creates an invariant in the machine.

createEvent(⟨identifiers⟩): For each ⟨identifier⟩ in ⟨identifiers⟩ it creates an event in the machine.

createAction(⟨actions⟩): For each ⟨action⟩ in ⟨actions⟩ it creates a action in the event.

createGuard(⟨predicates⟩): For each ⟨predicate⟩ in ⟨predicates⟩ it creates a guard in the event.

5.6.5 Rule Type: Scenario

The rule type SCENARIO models the behaviour of a scenario in the Event-B machine component. The scenario may contain extensions and deviations that refer to other extension use cases and accident cases. This is taken into account by the semantics.

SCENARIO: Scenario ↪ Scenario modelled in Machine

The translation rule TRule 6, as seen in Figure 5.25, takes the name, label, agents, contract, scenario, deviations and extensions as the argument. The middle row shows the unpacking of these arguments that are used by the translation rule to produce the Event-B elements. The following functions are used by this rule to create the elements within the machine components:

getAbstractMachineVariables(): Returns a collection of variables that were introduced in the abstract machine.

getConcreteVariables(⟨variables⟩, ⟨agents⟩, ⟨scenario⟩, ⟨deviations⟩, ⟨extensions⟩): A list of variables that occur in the scenario, deviations and the pre-conditions and post-conditions of any extension use cases is returned. These variables are treated as the concrete variables. The first argument ⟨variables⟩ of this function denotes a list of abstract variables. For any concrete variable that corresponds to the

abstract variable has its identifier suffixed to indicate that it is a concrete representation of the abstract variables. The concrete variable keeps note of the variable that it refines.

createFlowVariable(⟨label⟩): Creates a auxiliary flow variable using the literal value of ⟨label⟩ that contains an identifier, predicate (that denotes its type) and initialisation. For example, a label *UC* would create a variable with identifier *UC_flow*, type $UC_flow \in UC_FLOW$, and initialisation $UC_flow := UC_Initial$.

createGlueFlow(⟨label⟩): Creates an ⟨predicate⟩ that denotes the gluing invariant between the concrete control flow variable and the abstract auxiliary boolean variable using the label provided.

createGlueVariables(⟨label⟩, ⟨variables⟩): Creates a ⟨predicate⟩ in the machine that denotes the gluing invariant between the abstract and concrete variables.

createScenarioPrecondition(⟨label⟩, ⟨pre-condition⟩): Creates a *predicate* that denotes the invariant for the pre-condition to be established before the flow of the use case may execute.

createScenarioPostcondition(⟨label⟩, ⟨variables⟩, ⟨post-condition⟩): Creates a *predicate* in the machine that denotes the invariant that states the post-condition is established once the flow has finished its execution.

createAbstractMachineEvents(⟨name⟩): Creates an *extended* event for each event from the abstract machine apart from the event that has the same name as literal value of ⟨name⟩.

createFlowInitialGuard(⟨label⟩): Creates a *predicate* in the event that states the initial value the control flow auxiliary variable, e.g. for a label UC a guard $UC_flow = UC_Initial$ is produced.

createFlowFinalGuard(⟨label⟩): Creates a *predicate* in the event that states the final value the control flow auxiliary variable, e.g. for a label UC a guard $UC_flow = UC_Final$ is produced.

createFlowTriggerAction(⟨label⟩): Creates an *action* in the event that assigns the trigger value the control flow auxiliary variable, e.g. for a label UC an action $UC_flow := UC_Trigger$ is produced.

createRefinesEvent(⟨name⟩): Creates a *refine* element in the event with the literal value of the argument ⟨name⟩ as the identifier.

createConcreteEqualsAction(⟨variables⟩): The argument ⟨variables⟩ is list of concrete variables. Some of the concrete variables have a corresponding abstract variable. This function returns a collection of actions that assign the value of abstract variable to the concrete variable.

createAbstractEqualsAction(⟨variables⟩): The argument ⟨variables⟩ is list of concrete variables. Some of the concrete variables have a corresponding abstract variable. This function returns a collection of actions that assign the value of concrete variable to the abstract variable.

Figure 5.25: Translation rule TRule 6 *of type* SCENARIO.

5.6.6 Rule Type: Flow

The rule type FLOW models a sequence of steps in a use case as a sequence of events in the machine component of Event-B.

$$\text{FLOW: Sequence of Steps} \hookrightarrow \text{Events in Machine}$$

The translation rule TRule 7, as seen in Figure 5.26, takes the label, variables, and scenario as the argument. The middle row provides an unpacking of these arguments that are used by the translation rule to produce the Event-B elements. The following

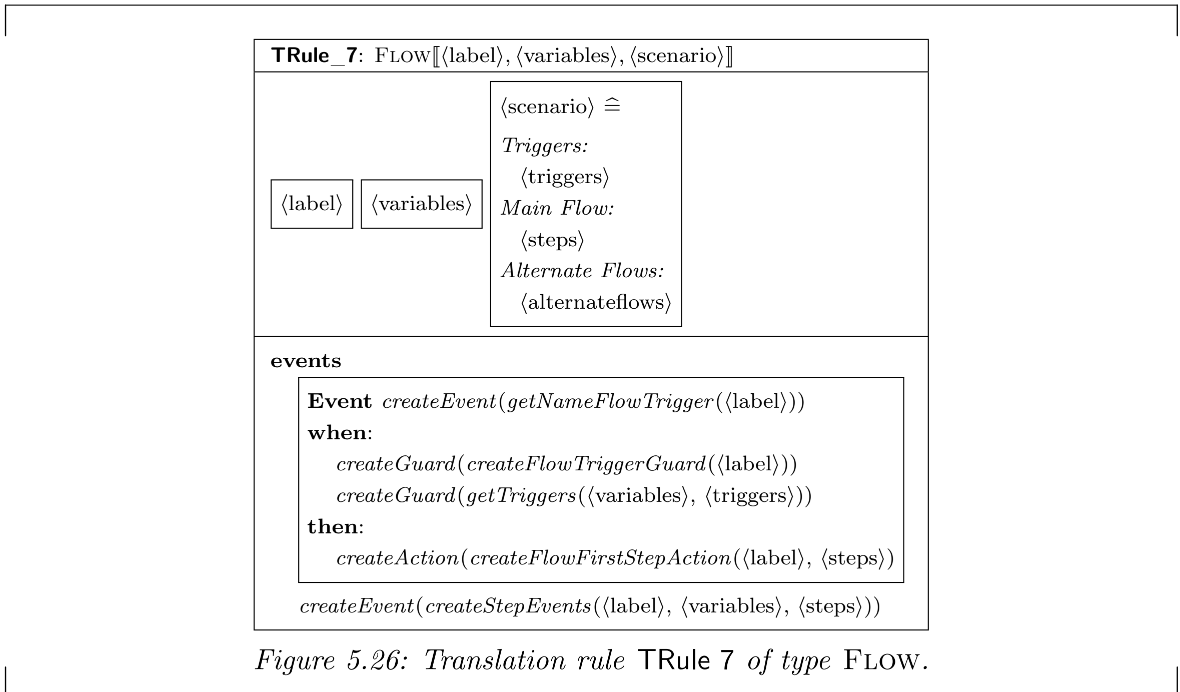


Figure 5.26: Translation rule TRule 7 *of type* FLOW.

functions are used in the translation rule to create the events that model the scenario.

createFlowTriggerGuard(⟨label⟩): Creates a *predicate* in the event that states the trigger value the control flow auxiliary variable, e.g. for a label UC a guard $UC_flow = UC_Trigger$ is produced.

getTriggers(⟨variables⟩, ⟨triggers⟩): For each predicate ⟨trigger⟩.⟨predicate⟩ in ⟨triggers⟩ is returned as a collection of predicates ⟨predicate⟩.

createFlowFirstStepActtion(⟨label⟩, ⟨steps⟩): Creates an *action* that assigns the value of the first step ⟨steps⟩ to the control flow auxiliary variable, e.g. for a label UC and the first step S an action $UC_flow := S$ is produced.

createStepEvents(⟨label⟩, ⟨variables⟩, ⟨steps⟩): Creates an event for each step ⟨step⟩ in ⟨steps⟩ in the machine.

createDeviationPointGuard(⟨label⟩, ⟨deviation-point⟩): Creates a guard in the event that states the control flow variable has the value of the deviation-point, e.g. for a label UC and deviation-point S_d a guard $UC_flow := S_d$ is produced.

createExtensionPointGuard(⟨label⟩, ⟨deviation-point⟩): Creates a guard in the event that states the control flow variable has the value of the extension-point, e.g. for a label UC and extension-point S_e a guard $UC_flow := S_e$ is produced.

The translation rule TRule 8, as seen in Figure 5.27, takes the label, variables, and deviations as the argument. The middle row provides an unpacking of these arguments. The deviation refers to an accident case that contains a scenario, with triggers and a main flow.

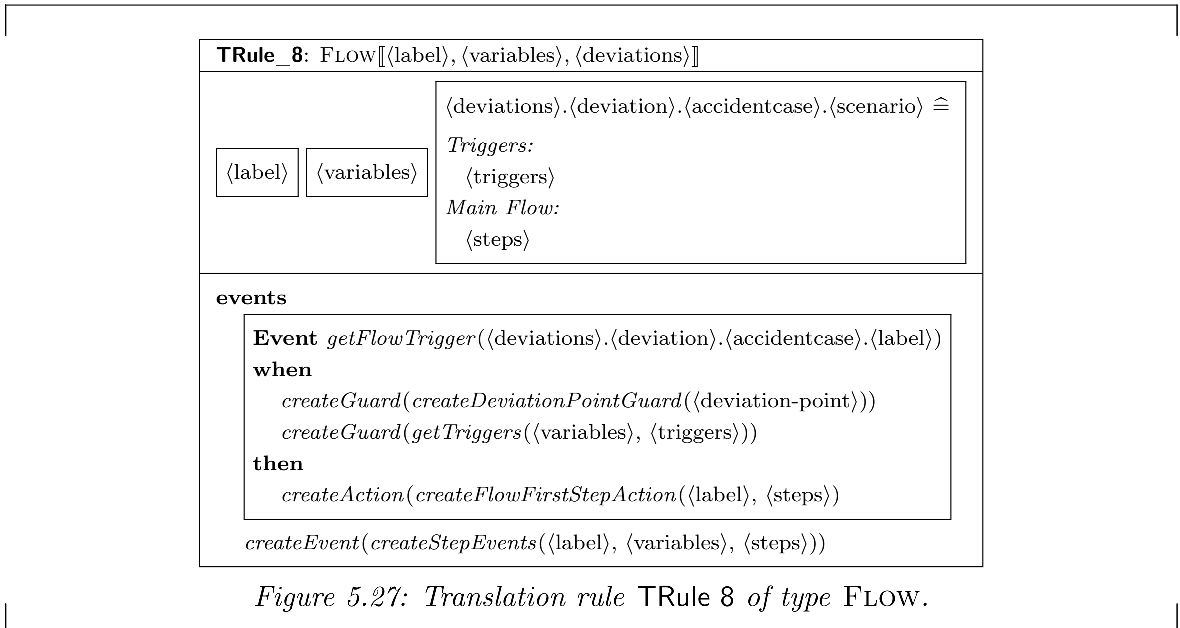


Figure 5.27: Translation rule TRule 8 *of type* FLOW.

The translation rule TRule 9, as seen in Figure 5.28, takes the label, extension and variables as the argument. The middle row provides an unpacking of these arguments. The extension refers to an extension use case. It specifies an extension-point and rejoin-point. Only the contract of the extension use case is provided, as the scenario is not required by this translation rule.

5.6.7 Rule Type: Extension

The rule type EXTENSION introduces the extensions as refinement in the Event-B model. The scenarios of the extension use cases are introduced in the Event-B model by this rule type.

EXTENSION: Extensions ↪ Extension Scenario in Event-B

Figure 5.28: Translation rule TRule 9 *of type* Flow.

The translation rule TRule 10, as seen in Figure 5.29, takes the label and extensions as the argument. The middle row provides an unpacking of these arguments. The translation rules produces a machine and context component. The *sees* and *refines* relationships are created for the machine to relate it to context and machine components. It uses translation rules of types Scenario, Flow and Extension, with the arguments of the extension use case. This results in the scenario of the extension use case being introduced in the Event-B project. If the extension use case contains extensions, this results in further refinement in the Event-B model.

Figure 5.29: Translation rule TRule 10 *of type* EXTENSION.

5.7 Related Work

Several groups have investigated a rigorous approach to capturing UML use cases [65, 96,111]. In comparison, the novelty of our approach comes from the use of refinement to introduce key abstractions that are captured naturally by the structure of the use case specification and its relationship to other use cases. In [96], Soussa and Russo provide a mapping from the flow of a use case to operations in B. They rely upon the flow to be written in accordance to a transaction pattern between the actor and the system as follows: (1) an actors request action; (2) a system data validation action; (3) a system expletive action; and finally (4) a system response action. We consider this pattern would require the designer to focus more on the solution rather than understanding the problem domain, which steps away from some of the benefits and simplicity of using UML use cases. In [111], Whittle presents a precise notation for specifying use cases based on three levels of abstraction: use case charts, scenario charts and interaction diagrams. The motivation for this approach is similar to ours which also considers the use of negative scenarios. However, we have focused on adding rigour to the *textual*

specification of use cases which is commonly used in industry. In [65], Klimek and Szwed refer to the formal analysis of the use case diagrams. They propose a formal model of use cases that provide two methods of formal analysis and verification: the first one based on a state exploration represents a model checking approach, and the second one refers to the symbolic reasoning using formal methods of temporal logic. In comparison to this formalisation of use cases, the approach presented in this thesis takes into consideration the formalisation of undesired behaviour via the accident case and its relationship to regular use cases.

Control flow between events in Event-B is typically modelled implicitly via variables and event guards. While this fits well with Event-B refinement, it can make models involving sequencing of events more difficult to specify and understand than if control flow was explicitly specified. Atomicity Decomposition (AD) diagram introduced by Fathabadi. et al [40], provide a graphical notation that is capable of representing relationships between abstract and concrete events explicitly. Using the AD approach has another advantage which is that we can represent event ordering explicitly. The AD diagrams are based on JSD diagrams from Jackson. The Event-B models produced from the AD diagrams use auxiliary variables to help mediate the execution of events. In the approach of encoding UML use cases in Event-B, an auxiliary control flow variable is introduced to help ensure the execution of events in the Event-B model correspond to the ordering of steps in the use case scenario.

5.8 Summary & Discussion

This chapter has provided the encoding of use cases as an Event-B model. This has also taken into account deviations from accident case and extensions from extension use cases in the encoding into Event-B. The encoding has utilised an abstraction found in the structure and relationships of a use case. In order to relate the abstract and concrete layers, gluing invariants were identified that establish the relationship between the layers. In addition, invariants for the scenario were identified that helps establish that the scenarios satisfy the pre-condition, post-condition and invariants of the contract. The encoding of simple and complex branching in the scenario of use case to Event-B has been provided. Verification of the generated Event-B models have been discussed with emphasis on specific proof obligations that provide identification of defects to the use case specification. An overview of the translation rules have been provided. These are used in Chapter 6, which implements the UC-B tool that automatically produces Event-B models from formally specified use cases use cases.

Chapter 6

UC-B: Tool Development

6.1 Introduction

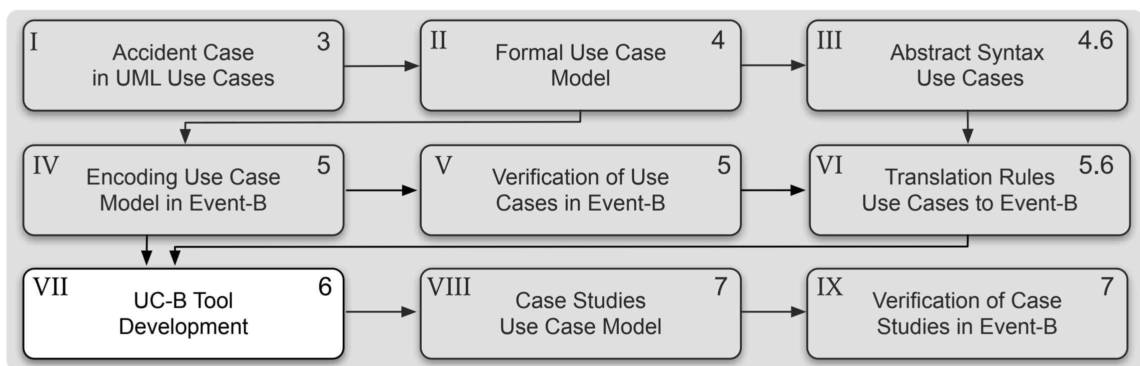


Figure 6.1: Thesis roadmap for Chapter 6.

One of the contributions in this thesis is the development of a prototype plug-in UC-B for the Rodin platform [4]. The aim of UC-B is to enable practitioners to adopt a *light-weight* approach in the use of formal methods during requirements analysis via use case modelling. The development of this plug-in make use of the extensibility features provided by Rodin as a result of its eclipse-based installation. This enables the use of Event-B's mathematical language to specify use cases formally, as well support for the automatic generation of Event-B from a source use case. The implementation of this tool is based on the structure of the use case model and the translation rules to Event-B, provided in Chapter 4 and 5. Figure 6.1 highlights which part of the roadmap this chapter implements.

Historically, formal methods have been viewed as a pure alternative to traditional development methodologies, demanding a revolutionary change in industry to adopt

them [6, 59]. This approach has been documented to not be realistic because: (1) often only parts of the systems would benefit formal methods and (2) the skill level required to cope with techniques for full formal development would be expensive.

As discussed in Chapter 1, lightweight approach in the use of formal methods has been a new trend, where they are targeted primarily on the early stages of development and are focused towards defect detection through rigorous examination. This research has focused in the development of the plug-in UC-B for Rodin. UC-B is aimed to be a pragmatic lightweight approach that allow the textual specification of use cases to be detailed in using Event-B's mathematical language, while enabling the automatic generation of corresponding Event-B models. The generated Event-B models are subjected to automatic verification tools on Rodin that check for defects in the behaviour specified by the use cases.

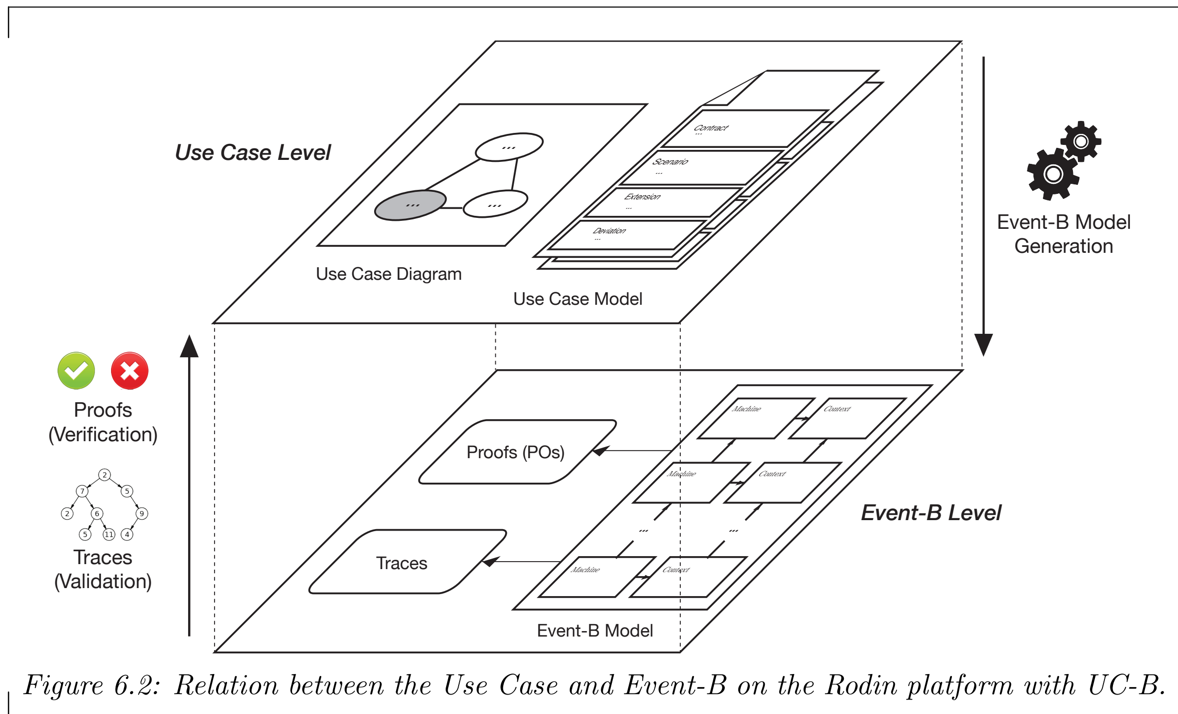


Figure 6.2: Relation between the Use Case and Event-B on the Rodin platform with UC-B.

In Figure 6.2, an overview of the implementation of UC-B on the Rodin platform is provided. The aim of UC-B is to introduce a level for use case modelling that is placed on top of the existing Event-B development environment. These two levels are described as follows:

Use Case Level At the use case level, the focus is placed on the authoring and management of the use case model. The specification of the use cases is supported by a dual representation of informal and formal notation. This brings precision and clarity in specifying use cases. At this level, a formally specified use case can be subjected for translation to an Event-B model.

Event-B Level The generated Event-B models are immediately subjected to the verification tools supported by Event-B, provers that help identify inconsistencies in the model via proof obligations (POs). The pass and fail of these proof obligations defects in the specification of the use case, at the use case level. In addition, tools such ProB support activities to identifying traces that aid in the validation of the use cases (scenarios) using the generated Event-B model.

In this chapter, the goals for UC-B are first discussed in Section 6.2. The architecture and technologies used to implement these goals are provided in Section 6.3.

6.2 Goals

The following specifies the goals required in the development of UC-B:

- UC-B will be an application that can be installed on the Rodin platform.
- UC-B must allow use case models to be authored and managed on the Rodin platform.
  - The use case model must support the use case types: use case, extension use case and accident case.
  - The use case model must support specification of contract and scenario for use cases.
  - The use case model shall support consistent and automatic labelling of use case model elements.
- UC-B must enable use case specification to be detailed with both informal and formal notation.
  - The formal notation must be based on the mathematical language of Event-B [1].
  - The informal and formal notation shall coexist side-by-side in the specification of the use case.
- UC-B must support the automatic generation of an Event-B model from formally specified use cases.

6.3 Architecture & Technologies

This section describes the technologies employed in building UC-B and its architecture, as seen in Figure 6.3. The framework proof traceability (dotted box) has not been implemented at this stage, and is addressed as part of the future work in Chapter 8.

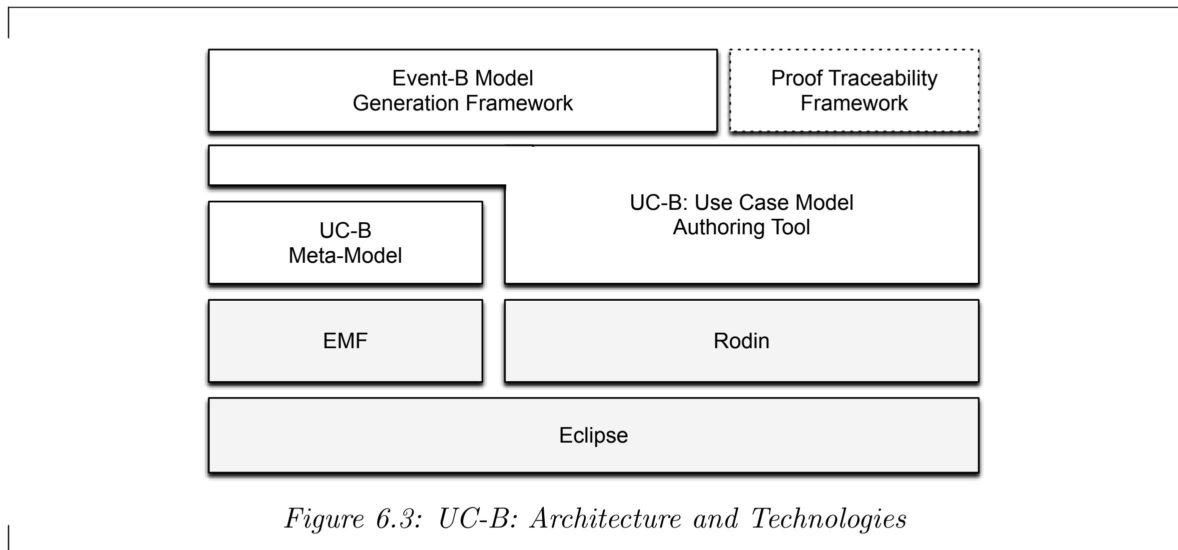


Figure 6.3: UC-B: Architecture and Technologies

6.3.1 Eclipse

Eclipse [34] is a platform for general purpose applications with an extensible plug-in system. It is known as an integrated development environment (IDE) for Java development, although the Java IDE is just one specialised application of the platform. Eclipse employs plug-ins in order to provide all of its functionality on top of the run-time system, this is based on Equinox, an OSGi standard compliant implementation. The Eclipse platform provides facilities for workspace management, GUI building, a help system, team support and more. These components are examples of plug-ins. Plug-ins may provide extension points, to which other plug-ins may connect via extensions. A typical Eclipse installation contains hundreds of extensions.

6.3.2 The Eclipse Modelling Framework (EMF)

The Eclipse Modelling Framework [103] is a modelling and code generation facility. EMF provides tools and runtime support to produce Java code for the model and adaptor classes that enable viewing and command-based editing of the model. EMF is attractive for UC-B as it is modular and it takes care of many mundane tasks in GUI development. An EMF-application typically consists of three layers:

Model The model layer contains the data model and is stored in the form of an Ecore model [103]. The Ecore model can be either generated from scratch or imported. Customisation of the Ecore model include namespace, containment of elements and other. There is a corresponding Genmodel (Generator Model) that allows fine-tuning of the generated code for Model, Edit, Editor and Tests. Applied to the model layer, it generates the Java code for the data model.

Edit The Edit layer consists of so-called *ItemProviders*, which represent the bridge between the data model and a GUI. The *ItemProviders* can provide an alternative structure of the data. It is not unusual that the structure of the data model differs from the structure in the GUI. *ItemProviders* also provide basic information like labels and icons. They also collect the properties that are presented in the property view and collect the properties that are presented in the property view and collect the commands that a use can perform on a data element. Finally, they provide facilities to support Undo/Redo, Copy, Cut and Paste, Drag and Drop, and more. The *ItemProvider* code is also customised through and generated by the Genmodel.

Editor EMF can also generate code for an Eclipse-based editor. Such an editor is generic, in the sense that it can be driven by any set of ItemProviders. The editor support many standard features that one would expect of a modern model editor.

The development of the UC-B uses EMF to establish the meta-model for constructing a use case model on the Rodin platform. The edit and editor layers are used to author and mange the data structured on use case meta-model. The meta-model for UC-B is discussed in Section 6.3.3, and the editor is discussed in Section 6.3.4.

6.3.3 UC-B Meta-model

The UC-B meta-model defines the data structure of UC-B projects. The model is structured into three packages for clarity. The core package contains a structure of abstract meta-classes so that models can be treated generically as far as possible. There are two sub-packages contained with the core page: one for agents and one for use cases.

Abstract Basis

The meta-model is built upon a structure of meta-classes. Abstract meta-classes are the ones that cannot be directly instantiated. Their instances are always instances of their

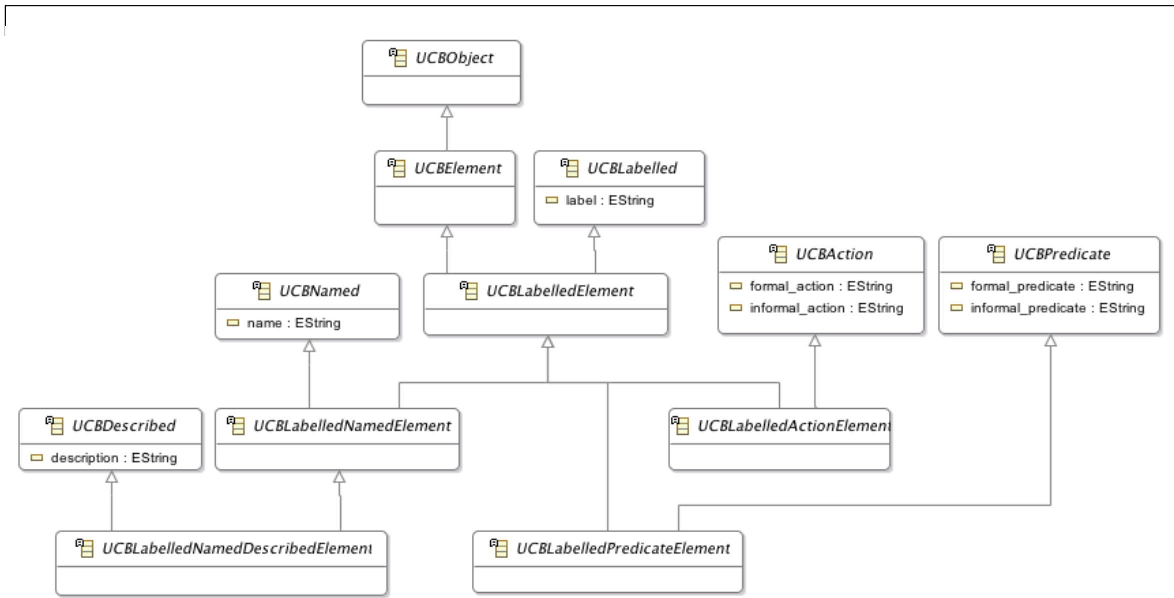


Figure 6.4: UC-B: Abstract meta-class

concrete sub-meta-classes. The abstract meta-classes are useful because they enable a feature to be defined once in the meta-model and then be used, via inheritance, by many concrete meta-classes. Apart from making it easier to maintain the meta-model, this makes it possible to write code that is more generic, since it can work on features without knowing which concrete meta-class the instance it is working on belongs to. To easily distinguish the abstract meta-classes from concrete ones, the names of the abstract meta-class are prefixed with *UCB*.

For abstract classes, the convention of including all the features that are inherited by that meta-class within the name is followed. For example, *UCBLabelledPredicateElement* inherits from *UCBLabelledElement* and *UCBPredicate*. The abstract meta-classes which are to be used to define concrete meta-classes are arranged in a hierarchy. Each feature is contained in a meta-class outside of this hierarchy. This provides a flexible choice of how to access model objects, either at a point in the hierarchy to generalise over parts of its structure, or via the feature containers to generalise over all elements that may own that feature. The root of all meta-classes in the UC-B meta-model is the abstract base class *UCBObject*. *UCBObject* extends the EMF class *EObject*. A description of the base meta-classes are provided in Figure 6.5.

Project

Figure 6.6 illustrates how UC-B projects are modelled in the UC-B meta-model. This provides support for the UC-B authoring tool (editor) to author and manage the contents of a UC-B project. A project contains a collection of *UC* which are generalisation

Abstract Meta Class	**Description**
UCBLabelledNamedDescribedElement	Provides a common basis to have an element with a label, name and description.
UCBLabelledPredicateElement	Provides a string attribute, *formal_predicate*, for an Event-B mathematical predicate, it also provides the attribute *informal_predicate*, which corresponds to informal description of the predicate using natural language. The element also inherits a label.
UCBLabelledActionElement	Provides a string attribute, *formal_action*, for an Event-B mathematical substitution, it also provides the attribute *informal_predicate*, that corresponds to informal description of the predicate using natural language. The element also inherits a label.

Figure 6.5: Abstract meta-classes and their description.

for use case, extension use case and accident case, and *Agents*. The class *Agent* and *UC* inherits the abstract meta-class *UCBLabelledNamedDescribedElement*. This allows agents and use cases to be specified with a *label*, *name* and *description*.

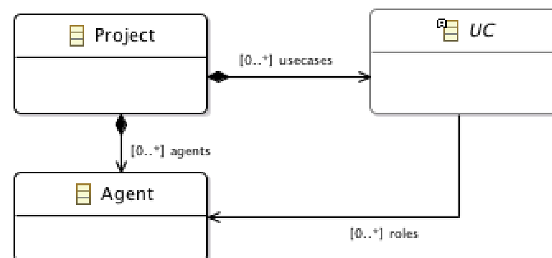


Figure 6.6: UC-B: Project

Agent

Figure 6.7 illustrates how agents are modelled in UC-B. The *Agent* class may contain carrier sets, constants, variables. The class *CarrierSet* may have a collection of *Element* objects. The benefit of creating the set and its elements, is that the Event-B model generation automatically creates an axiom that enumerates the set with the elements using the *partition* operator. For example, a user may define a set *DOOR* with the elements *Open* and *Closed* in UC-B, as seen in Figure 6.8a. In Event-B the *elements* are introduced as constants in the context, in which the set *DOOR* belongs to. The *partition* operator is used to enumerate the set with the elements, as seen in Figure 6.8b.

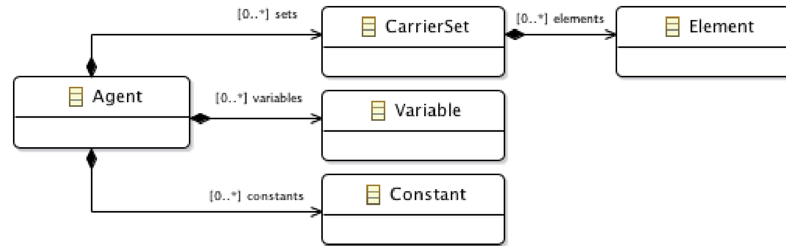
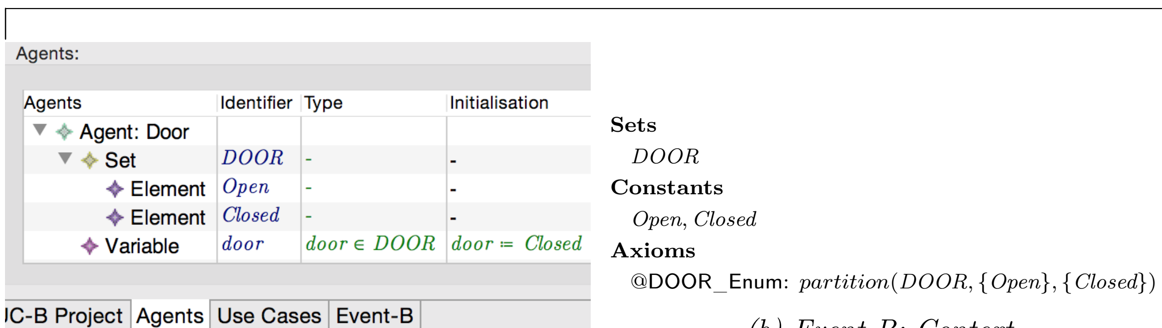


Figure 6.7: UC-B meta-class: Agent



(b) Event-B: Context.

(a) UC-B: Agent.

Figure 6.8: Example: Agent Door *with ennumerated set DOOR.*

The class *Constant* and *Variable* inherit the abstract meta-class *UCBLabelledPredicateElement*. This allows each constant and variable to specify a label (the identifier), a predicate to describe its type, and an informal description. The identifier specified via the label is required to be unique. In addition, the variable has a string *initialisation*, that allows it to specify an assignment.

Use Cases

Figure 6.7 illustrates how use cases are modelled in UC-B. The abstract class *UC* may contain a scenario. This abstract meta-class is inherited by *UseCase* and *AccidentCase*, which allows use cases and accident cases may contain a scenario. The *UseCase* may also contain a *Contract*. The *Contract* may contain a collection of preconditions, postconditions and invariants. The class *Precondition*, *Postcondition* and *Invariant* inherit the abstract meta-class *UCBLabelledPredicate*. This allows an instance of these classes to specify a label, a formal predicate, and an informal description.

The *UseCase* may also contain deviations and extensions that relate the use case to *AccidentCase* and *ExtensionUsecase*. The *ExtensionUsecase* inherits the class *UseCase*, i.e. it is also allowed to have a contract, scenario, deviations and extensions. The class *Scenario* may contain one main flow and a collection of alternate flows. The *MainFlow*

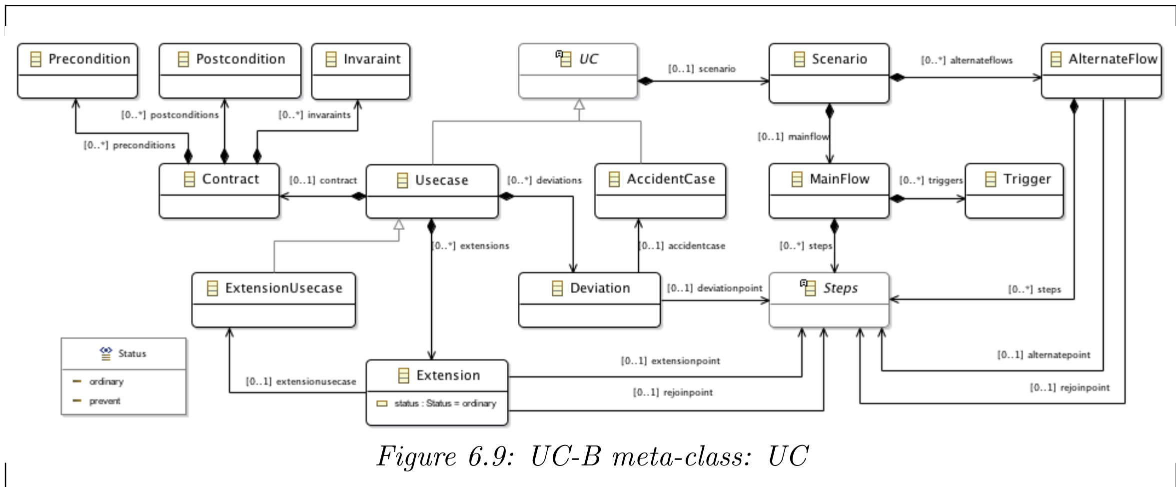


Figure 6.9: UC-B meta-class: UC

and *AlternateFlow* may contain a collection of steps. The *AlternateFlow* may specify a *alternatepoint* and *rejoinpoint* that refer to a step. The element *extension* may specify a status that can have the values *Ordinary* or *Prevent*.

Steps

The abstract meta-class, *Steps*, is inherited by *Step*, *If* and *While*. The meta-class *Step* inherits *UCBLabelledAction*, which allows each Step to specify labelled action, with formal assignment and informal description. The meta-classes *If* and *While* inherit the abstract-meta class *UCBLabelledPredicate*. This allows these meta-classes to specify a label, predicate and informal description. The meta-classes may also contain a collection of steps themselves.

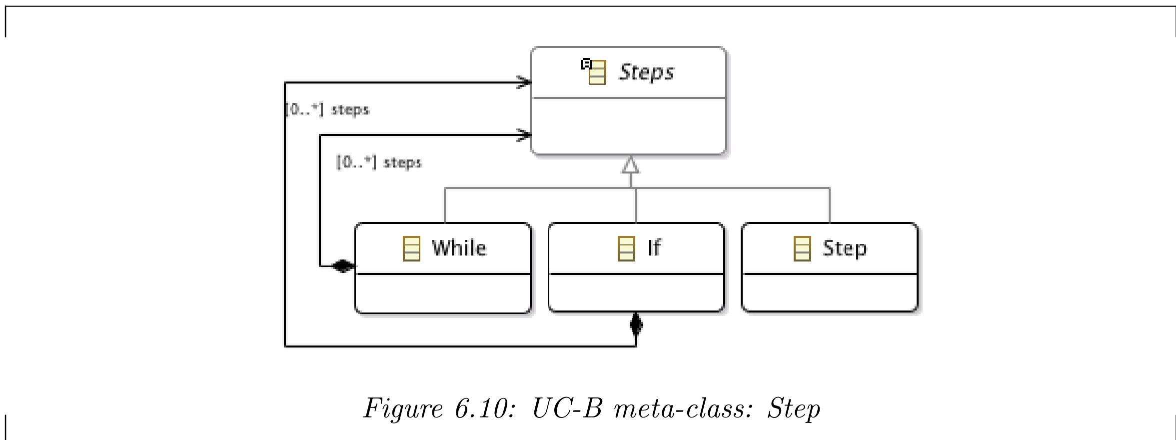


Figure 6.10: UC-B meta-class: Step

6.3.4 UC-B

An UC-B editor (extended from the editor produced by EMF) is provided to allow for the authoring and management of use case model based on the UC-B meta-model

provided in Section 6.3.3. The Rodin APIs allow for the use of Event-B's mathematical language in detailing the use case model formally, and support for the generation of an Event-B model from a target use case.

Am UC-B editor provides four *views* to help manage the use case model: (1) UC-B Project, (2) Agents, (3) Use Cases and (4) Event-B model generation. Each of these views help the user focus on authoring and managing parts of the use case model.

UC-B Project

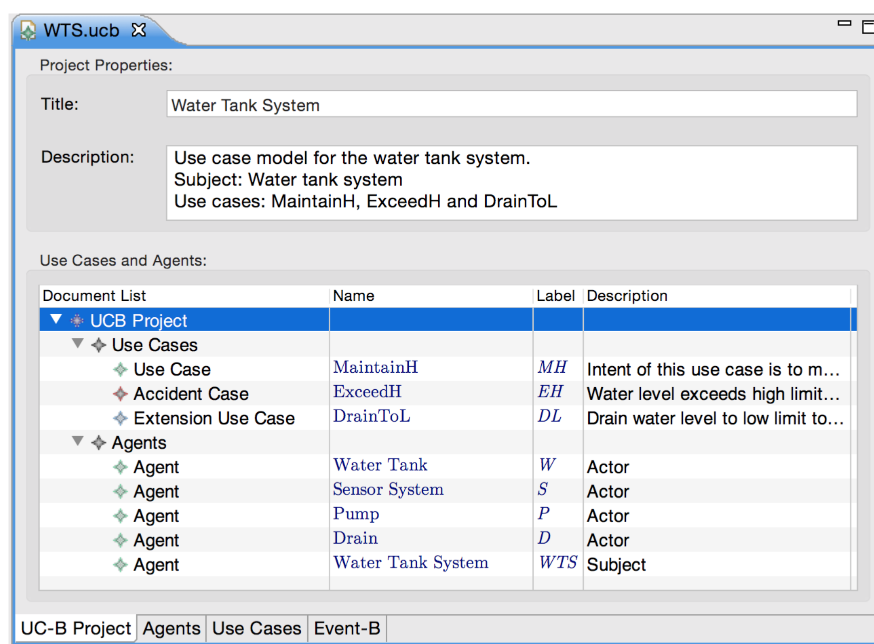


Figure 6.11: UC-B: Project for water tank system.

Figure 6.11 provides a screen shot of the UC-B project view for the water tank system. This view allows the user to provide a *title* and *description* for the use case model created. It also provides a list of the use cases and agents that belong to the project. In this section, new agents and use cases can be added to and removed from the project. At this view, only the *title*, *label* and *description* of the use cases and agents can be modified. The tool provides a static check that ensures that the label for any use case or agent created is unique, upon saving any changes to the project. The creation of use cases support the types: use case, accident case and extension use case.

WTS.ucb

Agents:

Agents	Identifier	Type	Initialisation	Description
Agent: Water Tank				
Constant	L	$L = 0$	-	Low limit on water tank.
Constant	LT	$LT > L$	-	Low threshold on water tank.
Constant	HT	$HT > LT$	-	High threshold on water tank.
Constant	H	$H > HT$	-	High limit on water tank.
Constant	DEC	$DEC \in (H-HT)..(HT-LT)$	-	A decrease in water level.
Constant	INC	$INC \in (LT-L)..(HT-LT)$	-	An increase in water level.
Constant	DRN	$DRN = L$	-	Drain set at low level in water tank
Variable	$waterlevel$	$waterlevel \in L..H$	$waterlevel = H$	Water level in the water tank.
Agent: Sensor System				
Variable	$sensorHT$	$sensorHT \in BOOL$	$sensorHT = FALSE$	High threshold sensor reading on..
Agent: Pump				
Variable	$motor$	$motor \in BOOL$	$motor = TRUE$	Motor in pump component.
Agent: Drain				
Variable	$valve$	$valve \in BOOL$	$valve = FALSE$	Exit valve on water tank.
Agent: Water Tank System				
Variable	$pump$	$pump \in BOOL$	$pump = TRUE$	WTS signal to pump component.
Variable	$drain$	$drain \in BOOL$	$drain = FALSE$	WTS signal to drain component.

UC-B Project | Agents | Use Cases | Event-B

Figure 6.12: UC-B: View for agents.

Agents

Every agent created in the project view appears in the *agent view*, as seen in Figure 6.12. Here, the user is allowed to define carrier sets, constants and variables for the agent. The constants and variables allow a type to be specified using Event-B's predicate language. The tool ensures that identifiers for carrier sets, constants and variables are unique. A static check is performed to ensure that the formal language used is syntactically correct and the identifiers used have been defined. For each variable, there is also a specification for its initialisation. All elements created for the agents may specify an informal description.

Use Cases

The use cases view allows for each use case created to be specified with both informal and formal notation, as seen in Figure 6.13. The view allows the user to select a created use case, via a drop-down list of all the created use cases. Once selected, the content of the use case, namely its contract (unless its an accident case) and scenario is provided. The use case and extension use case can further introduce *extensions* and *deviations* that refer to extension use case and accident cases, respectively. The contract and scenario can be specified using both informal and formal notation. Allowing these notations to co-exist enable precision in the requirements documentation while still maintaining ease of communication. A use case can be specified formally only with the declared sets, variables and constants defined by agents that play a *role* in the use case. The use case view allows the use case to specify agents that play a role in it.

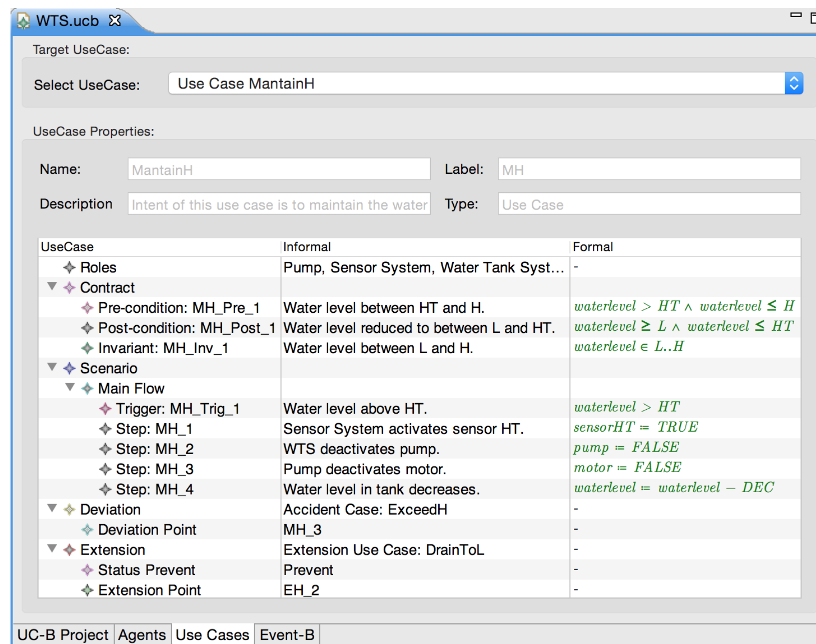


Figure 6.13: UC-B: View for use cases.

Event-B Model Generation

Once the use cases have been specified, the Event-B view allows for a target use case to be provided for Event-B model generation. The translation rules provided in Chapter 5 are used to generate the Event-B model from a target use case. The tool creates a project with the title of the target use case, and creates the machines and contexts components of the Event-B model that correspond to the use case, based on the translation rules. Rodin provers and syntax checks run automatically on the generated Event-B model providing an immediate display of errors or inconsistencies found at the Event-B level. Animation tools, such as ProB can be used to identify *traces* that help to validate the use cases against the generated Event-B model.

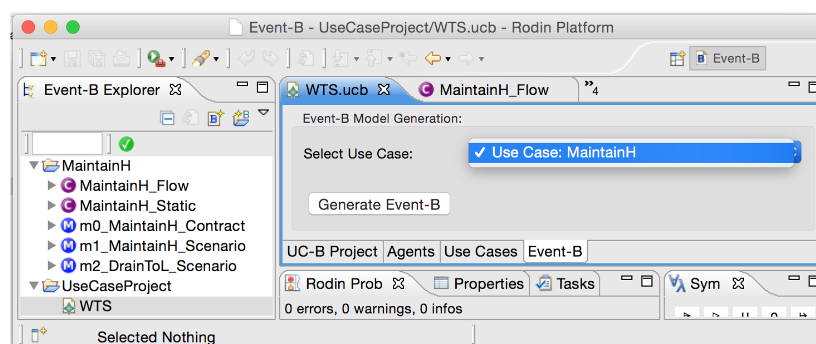


Figure 6.14: Event-B model generation.

6.4 Summary & Discussion

The Rodin platform, as an Event-B tool, serves as a host for the UC-B plug-in developed to give tool support to author and manage use case models. The theory underpinning the UC-B plug-in has been presented in Chapter 3, Chapter 4 and Chapter 5; The applications to case studies will be presented in Chapter 7. The tool benefits from features of EMF, to create a use case model on the Rodin platform. The APIs provided by Rodin allow for the use of the Event-B language in specifying the use case models formally, and automate the generation of the Event-Bs from a target, formally specified use case. We consider developing a graphical user interface for the use case diagram, in diagrammatic view, as future work. The automatic generation of Event-B from a formally specified use case, is aimed to decrease the effort of modelling in Event-B while allowing the user to focus on specifying use cases.

Chapter 7

Case Studies & Evaluation

7.1 Introduction

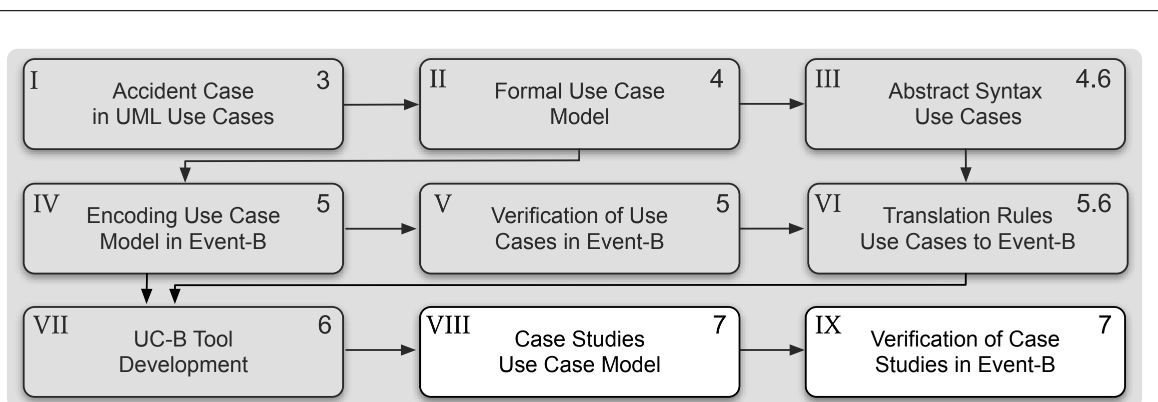


Figure 7.1: Thesis roadmap for Chapter 7.

This chapter evaluates the appraoch via a set of case studies. UC-B and the verification support provided by their corresponding Event-B model are used to support the evaluation. Figure 7.1, highlights which part of the roadmap that is being implemented. The case studies will cover: (1) the use case types: use case, accident case and extension use case; (2) simple and complex branching in scenarios; and (3) the extension types: *ordinary* and *prevent*. The case studies are as follows:

UC1 In Section 7.2, the *water tank system* (WTS), our running example, is discussed. The use case model has a use case, accident case and extension use case. The extension use case covers the use of the *prevent* relationship.

UC2 In Section 7.3, a case study of a *train door control system* (TDCS) is provided. The use case model has a use case, accident case and extension use case. It

covers the use of simple branching via the conditional if within the scenario of a use case. The extension use case covers the use of the *prevent* relationship.

UC3 In Section 7.4, a case study of an *automated teller machine* (ATM) is provided. It covers the use of complex branching via alternate flows, and the use an *ordinary* extension type in the use case.

UC4 In Section 7.5, a simplified case study *sense and avoid* (SAA) provided by BAE Systems is discussed. It covers the use of use case, accident case and extension use case. The extension use case is introduced to *prevent* the occurrence of the accident introduced by the accident case.

Each case study comprises of a use case model that contain the informal and formal specification of use cases, the corresponding Event-B model generated for the source use case, and the verification support provided by Event-B. The generated Event-B models for these case studies are provided in Appendix B. In addition, the state charts generated via Pro-B, associated with each Event-B model is provided in Appendix C. The traces generated by Pro-B help validate the Event-B model against the source use cases.

7.2 Case Study UC1: Water Tank System

The case study for the water tank system has been gradually introduced as a running example in this thesis. In this section, the case study is discussed as a whole, with the use case diagram, use case specifications and its corresponding Event-B model.

As discussed in Chapter 2, the aim of the water tank system is to maintain the *water level* between the high (H) and low (L) limits of the *water tank*, via the use of a *controller* (referred to as the water tank system), as seen in Figure 7.2. To achieve this intent, the controller interacts with two external components, namely the *sensor system* and *pump*. The sensor system monitors the water level in the tank with respect to the high threshold (HT) and low threshold (LT) sensor readings. Based on these readings, the controller either *activates* or *deactivates* the pump. When the pump is active, its motor is switched *on*, which subsequently *increases* the water level in the tank. On the other hand when the pump is deactivated, its motor is switched *off* which then allows the water level in the tank gradually *decrease*.

In addition, the controller also interacts with a *drain* component that is introduced as a safety control structure. In the event of a failure in the pump component, the controller may *activate* the drain, which subsequently *opens* an exit valve. This *exit*

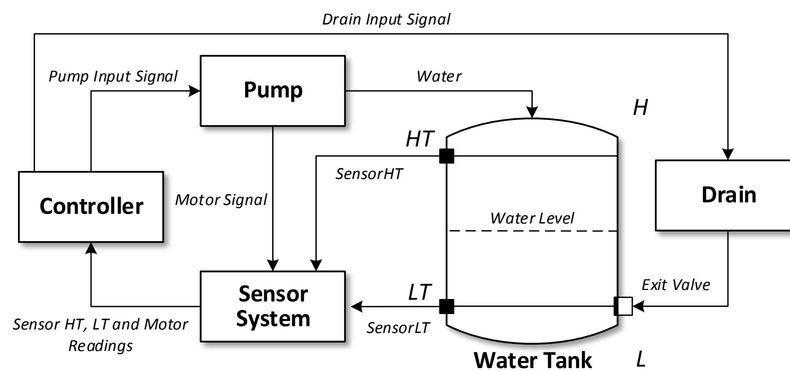


Figure 7.2: A description of the water tank system.

valve is located at the base of the water tank, at the low limit (L). When the exit valve is open the water level is reduced to the low limit in the event of a component failure.

Use Case Diagram

The use case diagram for the water tank system can be seen in Figure 7.3. It contains the use case MaintainH, the accident case ExceedH and extension use case DrainToL. The intent of MaintainH is to maintain the water level in the water tank below the high limit. The actors Water Tank, Pump and Sensor System, play a role in this use case to achieve this functionality. This use case is deviated by the accident case ExceedH via the deviate relationship.

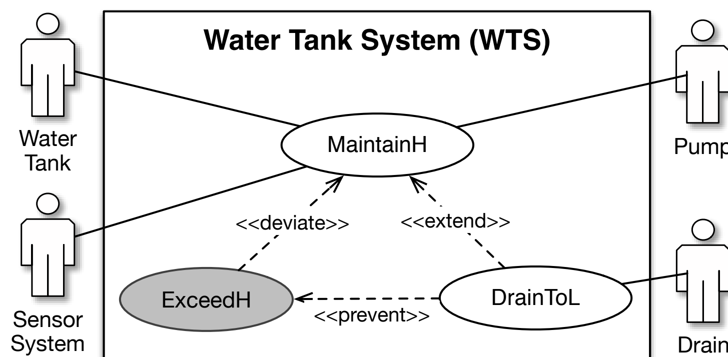


Figure 7.3: Water tank system use case diagram.

The accident case introduces undesired behaviour that allows the water level to exceed above the high limit. In order to prevent this accident, the extension use case DrainToL is introduced that extends the functionality of MaintainH by preventing the final outcome of the accident case ExceedH. The actor Drain plays a role in this extension

use case. The extension use case DrainToL introduces the interaction between the water tank system and the drain component in order to reduce the water level to the low limit of the tank.

7.2.1 Use Case Model

Agents

The actors and subject in the use case diagram of the water tank system are represented by *agents* in the use case model. These agents define the carrier sets, constants and variables that are required to specify the use cases MaintainH, ExceedH and DrainToL. This can be seen in Figure 7.4,

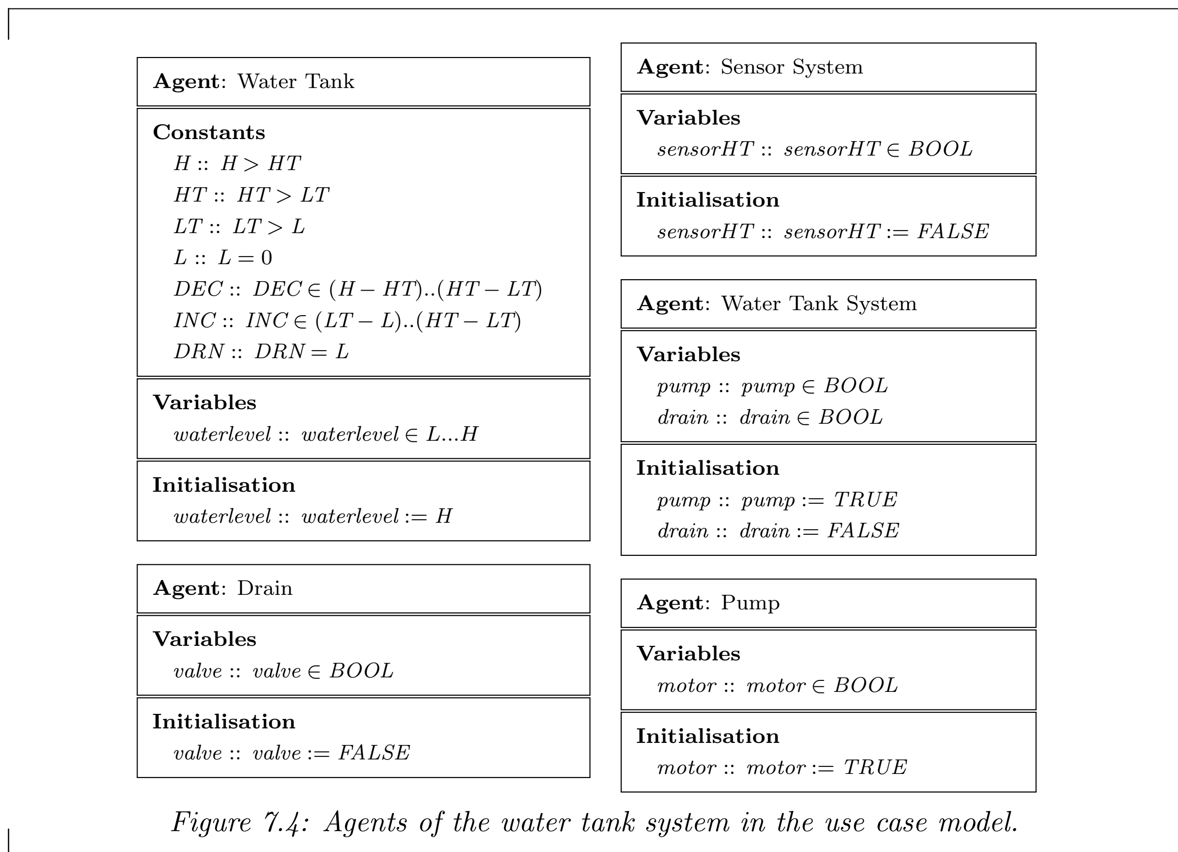


Figure 7.4: Agents of the water tank system in the use case model.

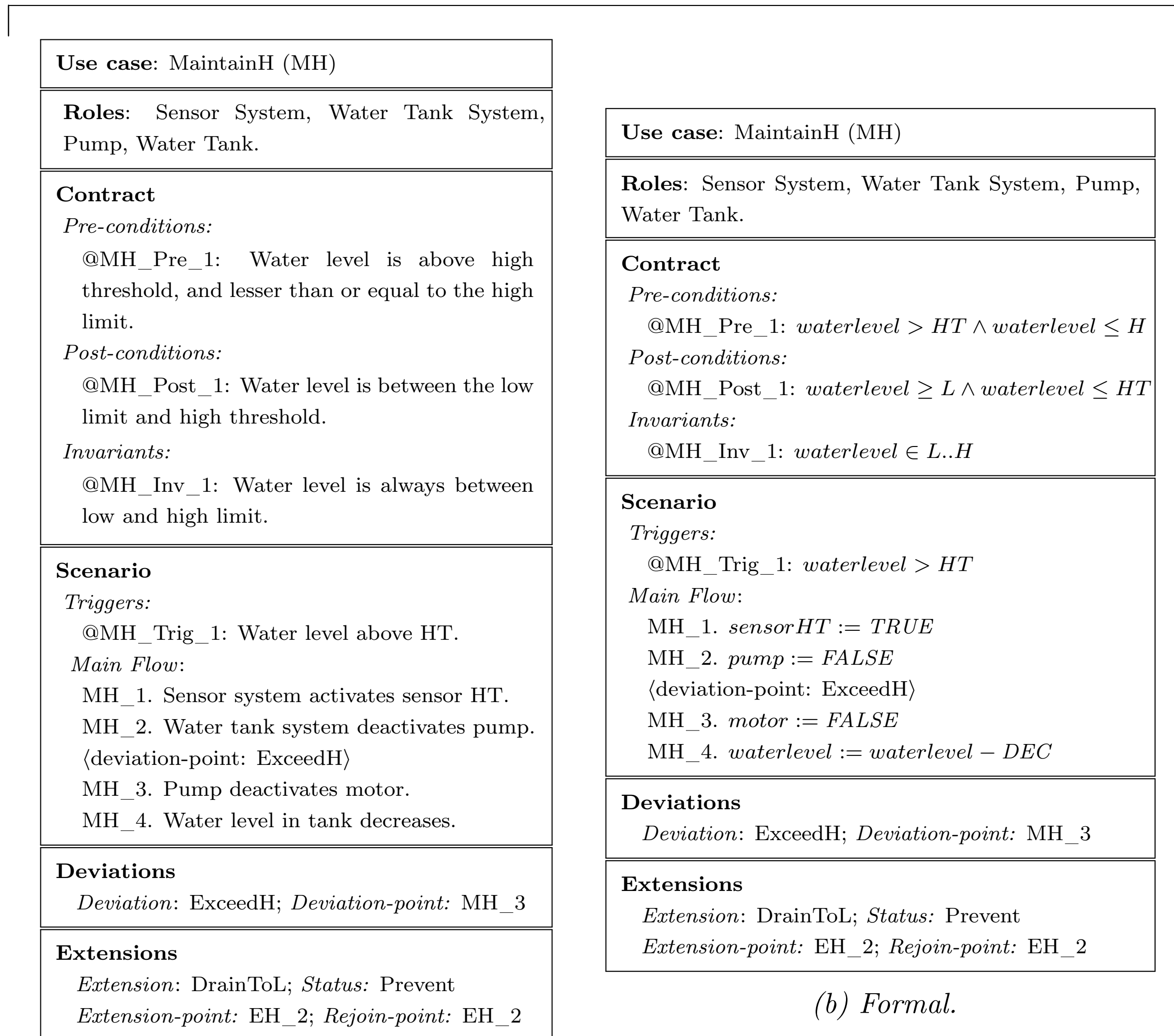
As discussed in Chapter 4, the Water Tank agent defines the limits and thresholds of the tank (L, H, LT, and HT) as constants, as they are not expected to be modified by the behaviour of the use cases. Their types specify important assumptions on the domain of the water tank, e.g. the high threshold if above the low threshold $HT > LT$. The water level in the tank is denoted by the variable *waterlevel* as its values are expected to change. It is of type, $waterlevel \in L..H$, where the water level is always expected to be between the L and H limits of the water tank. This variable is initialised

to the value H. The constants DEC and INC, denote a discrete representation in the decrease and increase of the water level in the tank, respectively.

The agents Sensor System, Pump, Water Tank System, and Drain, introduce the variables, *sensorHT*, *pump*, *motor*, *drain*, *valve*. These variables are all of the type $BOOL$, where $TRUE$ indicates *activated*, and $FALSE$ indicate *deactivated*. These sets, constants and variables can be used to specify a use case in which the agent plays a *role* in.

Use Case Specification

The informal and formal specification for the MaintainH use case is seen in Figure 7.5.



Use case: MaintainH (MH)

Roles: Sensor System, Water Tank System, Pump, Water Tank.

Contract

Pre-conditions:

@MH_Pre_1: Water level is above high threshold, and lesser than or equal to the high limit.

Post-conditions:

@MH_Post_1: Water level is between the low limit and high threshold.

Invariants:

@MH_Inv_1: Water level is always between low and high limit.

Scenario

Triggers:

@MH_Trig_1: Water level above HT.

Main Flow:

MH_1. Sensor system activates sensor HT.
MH_2. Water tank system deactivates pump.
⟨deviation-point: ExceedH⟩
MH_3. Pump deactivates motor.
MH_4. Water level in tank decreases.

Deviations

Deviation: ExceedH; *Deviation-point:* MH_3

Extensions

Extension: DrainToL; *Status:* Prevent
Extension-point: EH_2; *Rejoin-point:* EH_2

(a) Informal

Use case: MaintainH (MH)

Roles: Sensor System, Water Tank System, Pump, Water Tank.

Contract

Pre-conditions:

@MH_Pre_1: $waterlevel > HT \land waterlevel \leq H$

Post-conditions:

@MH_Post_1: $waterlevel \geq L \land waterlevel \leq HT$

Invariants:

@MH_Inv_1: $waterlevel \in L..H$

Scenario

Triggers:

@MH_Trig_1: $waterlevel > HT$

Main Flow:

MH_1. $sensorHT := TRUE$
MH_2. $pump := FALSE$
⟨deviation-point: ExceedH⟩
MH_3. $motor := FALSE$
MH_4. $waterlevel := waterlevel - DEC$

Deviations

Deviation: ExceedH; *Deviation-point:* MH_3

Extensions

Extension: DrainToL; *Status:* Prevent
Extension-point: EH_2; *Rejoin-point:* EH_2

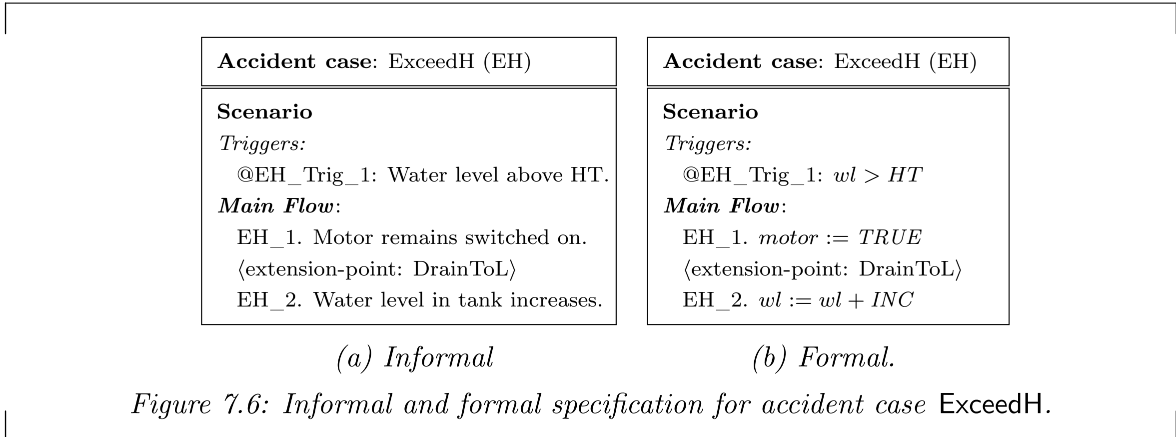
(b) Formal.

Figure 7.5: Informal and formal specification for use case MaintainH.

As the actors Water Tank, Sensor System, Pump, Drain, are *associated* with MaintainH, their corresponding agents have the *role* relationship with this use case. The

constants and variables defined by the agents are used to specify the contract and scenario of MaintainH formally. The pre-condition, post-condition, and invariant, are specified by predicates that clearly express the agreement of the stakeholders. The scenario specifies a main flow, that captures a trigger and a sequence of steps (MH_1 to MH_4). Each step is of the kind action, that captures assignments that modify the variables of the agents that play a role in this use case. The execution of the main flow is required to satisfy contract of the use case.

The specification of ExceedH only provides a scenario as it is an accident case. This scenario is seen in Figure 7.6 is specified with both informal and formal notation. The deviation relation in the use case model, allows the variables and constants defined by the agents that play a role in the use case MaintainH, to be used to specify the scenario of ExceedH.



(a) Informal *(b) Formal.*

Figure 7.6: Informal and formal specification for accident case ExceedH.

The element *extension* is introduced in the specification of MaintainH, as seen in Figure 7.5. It specifies the *status* and *extension-point* and refers to the extension use case DrainToL. The extension-point specifies a step, EH_2, in the scenario of the accident case ExceedH, as the status for the extension is *prevent*. This introduces the behaviour of the extension use case between the steps EH_1 and EH_2. The rejoin-point returns the flow back to the accident scenario at EH_2. By introducing this extension use case the water level is drained to the low limit. This prevents the water level from exceeding the high limit, even after pump increasing the water level at step EH_2.

The DrainToL extension use case is specified with both formal and informal notation, as seen in Figure 7.7. The extension use case is specified with the carrier sets, constants and variables that were used to specify MaintainH. The Drain agent plays a role in this extension use case, which allows the specification to use the variables *drain* and *valve* and constant DRN, defined by the agent.

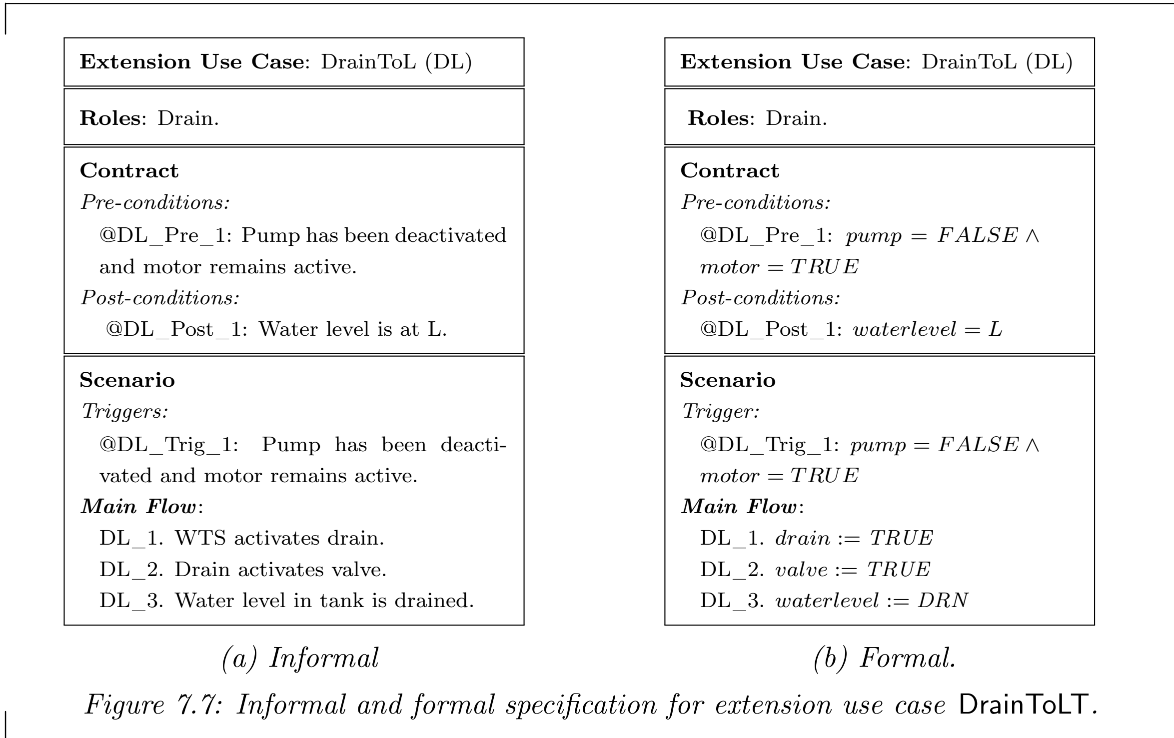


Figure 7.7: Informal and formal specification for extension use case DrainToLT*.*

7.2.2 Event-B

The Event-B model for MaintainH has three machine layers: m0_MaintainH_Contract, m1_MaintainH_Scenario, and m2_DrainToL_Scenario. Each machine layer introduces new variables and events that model the use case MaintainH, along with any extensions. The generated Event-B model also contains three contexts: MH_Static, MH_Flow and EH_Flow. The context MH_Static models all the static aspects (constants and sets) associated with the use case including the extension use case and accident case that are related to it. The contexts MH_Flow and EH_Flow model a type for the scenario of MaintainH and DrainToL. The state charts produced by ProB for this Event-B model is seen in Appendix C.1.

m0_MaintainH_Contract

This machine models the contract of the MaintainH use case. The variable *waterlevel* is introduced in this machine as it occurs in the pre-condition (@MH_Pre_1) and post-condition (@MH_Post_1). In addition, there is an auxiliary boolean variable, *MH*, introduced by the encoding to denote the execution of the use case. The machine contains an event MaintainH, which models the pre-condition and post-condition, as its guard and action. The post-condition (@MH_Post_1) which is predicate, is transformed to a non-deterministic action, where all occurrence of the variable *waterlevel*

(highlighted) are *primed* (after value), on the RHS of operator $:|$, as follows:

$$waterlevel :|\ waterlevel' \geq L \wedge waterlevel' \leq HT$$

(action MH_Post_Act in event MaintainH)

The invariant, labelled @MH_Inv_1, constraints the variable *waterlevel* to be always between the high (H) and low (L) limits. The main mathematical judgement made in this abstract machine is ensure that what is achieved by the post-condition of the use case maintains the constraints of this invariant. The following invariant preservation proof obligation is produced for event MaintainH for invariant @MH_Inv_1, and is automatically proved by the provers at the Event-B level:

$$L = 0 \wedge L < LT \wedge LT < HT \wedge HT < H$$
$$waterlevel' \geq L \wedge waterlevel' \leq HT$$
$$\vdash$$
$$waterlevel' \in L..H$$

(MaintainH/MH_Inv_1/INV)

Proving this PO establishes *what* is achieved by the use case is within bounds of the invariant. This machine is refined to introduce the scenario of MaintainH.

m1_MaintainH_Scenario

This machine introduces the scenario of MaintainH. This takes into account the *deviation* from the accident case ExceedH and the *extension* from DrainToL that aims to *prevent* the scenario of the accident case from resulting in an accident. The variables introduced in this machine are separated by the encoding, as follows:

Abstract variables The variables *MH* and *waterlevel*, which were introduced for the event MaintainH in the abstract machine, are treated as *abstract variables* in this machine.

Concrete variables Variables *sensorHT*, *pump*, *motor*, *waterlevel_m1* and *MH_flow* are introduced to model the scenario of MaintainH. The gluing invariants labelled @MH_Glue_Variables and @MH_Glue_Flow, are introduced to relate the concrete variables *waterlevel_m1* and *MH_flow* to their corresponding abstract variables *waterlevel* and *MH*.

The scenario is introduced as events that modify the concrete variables via its actions. The event MaintainH_Final *refines* the abstract event MaintainH. The gluing

invariants help to automatically discharge the guard strengthening (GRD) and action simulation (SIM) POs.

The invariants @MH_Scenario_Post and @MH_Scenario_Inv, are introduced to ensure that the scenario that modify the concrete variable, *waterlevel_m1*, achieve the post-condition (@MH_Post_1) and maintain the invariant (@MH_Inv_1) for the use case. In these invariants, all occurrence of the abstract variable *waterlevel* is replaced by *waterlevel_m1* (highlighted):

$$MH_flow = MH_Final \Rightarrow \boxed{waterlevel_m1} \geq L \land \boxed{waterleve_m1} \leq HT \tag{MH_Scenario_Post}$$

$$\boxed{waterlevel_m1} \in L..H \tag{MH_Scenario_Inv}$$

The invariant @MH_Scenario_Post introduces a constraint where the events that lead to the end of the use case, i.e. having the action $MH_flow := MH_Final$, are required to achieve the post-condition (@MH_Post_1) for the concrete variable *waterlevel_m1*. This constraint is placed on the events MH_4 and EH_2 as they lead to the end of the use case, producing the invariant preservation proof obligations MH_4/MaintainH_Scenario_Post/INV and EH_4/MaintainH_Scenario_Post/INV.

The PO MH_4/MaintainH_Scenario_Post/INV requires that the action of the event (step) MH_4 decreases the water level ($waterlevel_m1 := waterlevel_m1 - DEC$) to ensure that the level has been reduced to some value between the high threshold (HT) and low limit (L). The PO is as follows:

$$\begin{array}{l} MH_flow = MH_4 \\ waterlevel_m1 \in L..H \\ \vdash \\ waterlevel_m1 - DEC \geq L \land waterlevel_m1 - DEC \leq HT \end{array} \tag{MH_4/MaintainH_Scenario_Post/INV}$$

To help prove this PO the invariant labelled @MH_StepAssert_1 was manually introduced to help denote that the water level remained above the high threshold from the steps from MH_1 and MH_4. That is, after the scenario triggered, the water level remained above the high threshold during the interactions between the Sensor System, WTS and Pump, till the decrease took place. Automating the invariant discovery for these manually introduced invariants is part of the future work, which is discussed in

Chapter 8.

$$MH_flow \in \{MH_1, MH_2, MH_3, MH_4\} \Rightarrow (waterlevel_m1 > HT)$$
(MH_StepAssert_1)

The PO EH_2/MaintainH_Scenario_Post/INV is produced to ensure that the final step of the accident scenario achieves the post-condition. However, the final step introduces an action that *increases* the water level in the tank (due the failure of the pump component) which is expected to not achieve the post-condition of MaintainH.

In order to prevent the water level from exceeding the high limit, the extension use case DrainToL is introduced before the step EH_2 via an *extension-point*. This introduces the event DrainToL and DrainToL_FALSE in the scenario of the accident case, before step EH_2. This extension use case drains the water level to the low limit (L) which prevents the increase of water level above H. Due to the *prevent* status of the extension, the invariant @DL_Prevent is introduced to ensure that the event DrainToL always executes during the execution of the accident scenario. This is achieved by ensuring that the event DrainToL_FALSE is never enabled:

$$\neg(DL = FALSE \wedge MH_flow = EH_2 \wedge \neg(pump = FALSE \wedge motor = TRUE))$$
(DL_Prevent)

The event DrainToL models the pre-condition (@DL_Pre_1) and post-condition (@DL_Post_1) of the extension use case as its guard and action, respectively, based on the encoding for the extension use case. This machine is later refined to introduce the scenario of @DrainToL. The execution of this event achieves the post-condition (@DL_Post_1) that reduces the water level to the low limit (L). This allows the PO for event EH_4 to maintain the post-condition as the increase of the water level from the low level is proved to be below the high threshold.

$$MH_flow = EH_2$$
$$waterlevel_m1 = L$$
$$\vdash$$
$$waterlevel_m1 + INC \geq L \wedge waterlevel_m1 + INC \leq HT$$
(EH_2/MaintainH_Scenario_Post/INV)

To help prove this PO, the invariant @MH_AssertStep_2 was manually introduced to ensure that water level was at the low level before the event EH_2 was executed. Again, the automatic identification of these invariants that are introduced manually

form part of the future work, discussed in Chapter 8.

$$MH_flow = EH_2 \Rightarrow waterlevel_m1 = L \qquad \text{(MH_AssertStep_2)}$$

The invariant preservation POs for invariant @MH_Scenario_Post was automatically discharged for events MH_4, EH_2, and DrainToL, as the actions were shown to decrease the water level to some point between the low and high limits of the water tank. This machine establishes the scenario of MaintainH is consistent with its contract, with inclusion of the deviation of the accident case with the prevention of the extension use case.

m2_DrainToL_Scenario

The machine m2_DrainToL_Scenario *refines* m1_MaintainH_Scenario to introduce the scenario of DrainToL. The abstract event DrainToL is refined by the events that model the scenario of the extension use case. The variables in this machine are distinguished as follows:

Abstract variables The variables $waterlevel_m1$, $pump$, $motor$, DL and MH_flow that are associated with the abstract event DrainToL are treated as the abstract variables.

Concrete variables The variables $drain$, $valve$, $pump_m2$, $motor_m2$, $waterlevel_m2$ and DL_flow are the concrete variables associated with the scenario of DrainToL. The gluing invariants labelled @DL_Glue_Variables and @DL_Glue_Flow, are introduced to relate the concrete variables $waterlevel_m2$, $pump_m2$, $motor_m2$ and DL_flow to their corresponding abstract variables.

Note, the invariant (@MH_Inv_1) of MaintainH is considered an invariant of its extension use case DrainToL. These invariants are as follows, where all occurrence of the abstract variables are replaced by corresponding concrete variables (highlighted):

$$DL_flow = DL_Final \Rightarrow \boxed{waterlevel_m2} = L \qquad \text{(DL_Scenario_Post)}$$

$$\boxed{waterlevel_m2} \in L..H \qquad \text{(DL_Scenario_Inv)}$$

The invariant @DL_Scenario_Post ensures the event @DL_3 that leads to the end of the extension use case, via the action $DL_flow := DL_Final$, achieves the post-condition where the water level is reduced to the low limit (@DL_Post_1). This produces the invariant preservation PO, DL_3/DL_Scenario_Post/INV. This PO is

automatically discharged as the final step, which reduces the water level to the level of the drain, i.e the low limit where the drain is set $DRN = L$.

The PO DL_3/DL_Scenario_Inv/INV is also automatically proved as the action to drain to water level to the low limit is within the constraints of the invariant @DL_Scenario_Inv. This machines reveals more of the system with respect to the operation of the drain and valve. The refinement by this machine ensures that the scenario of the extension use case is consistent with its contract.

7.3 Case Study UC2: Train Door Control System

The train door control system (TDCS) provides the functionality to open train doors based on the request of the train operator. The use case diagram for TDCS is provided in Figure 7.8. It provides the use case OpenDoor and the external actors Train, Door, and Operator that are associated with it. The OpenDoor use case defines the behaviour to open the train doors provided based on the request of the train operator. This case study describes the use of *simple branching* via a conditional **if** in the use case scenario.

A potential accident in the operation of this system would be for a passenger to fall off a moving train (this accident is labelled PassengerFallsOffMovingTrain). This accident could result from a hazardous action for the train doors to be opened while the train is moving (an environmental condition). This cause of the accident PassengerFallsOffMovingTrain can be written as follows:

Door opened (Act.) + Train is moving (Cond.) ⇒ PassengerFallsOffMovingTrain

This accident is introduced in the use case diagram as seen in Figure 7.8 as an accident case PassengerFallsOffMovingTrain. A safety requirement for this system is for the train doors to always remain closed while the train is moving. The behaviour defined by the accident case is expected to violate this safety requirement. However, an extension use case EmergencyBraking is introduced to prevent this accident. This extension use case interacts with the actor Brake System which can reduce the train speed to stationary if it detects a fault that could result in the accident. The specification for these use cases are provided in Section 7.3.1.

The use case model for this case study is provided in Section 7.3.1 and the generated Event-B model for the use case OpenDoor is discussed in Section 7.3.2.

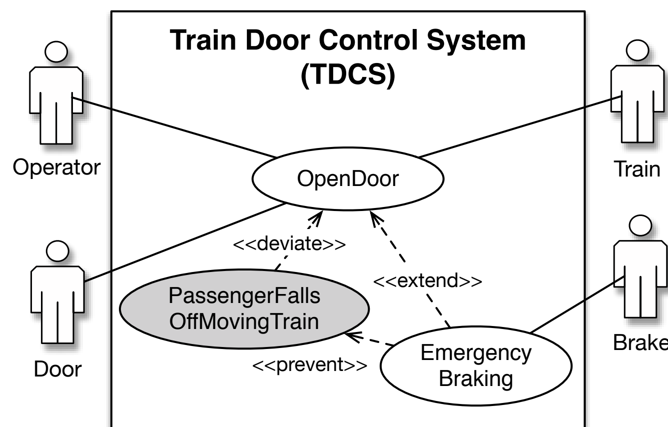


Figure 7.8: Use case diagram for Train Door Control System.

7.3.1 Use Case Model

Agents

The actors and subject in the use case diagram of TDCS are represented by *agents* in the use case model, as seen in Figure 7.9. These agents are described as follows:

Door This agent can be regarded as the provider of information on the current state of the door. The agent introduces an enumerated set *DOOR* with the elements *Open*, *Opening* and *Closed*, and a variable *door*. This variable is of type *DOOR* allowing it to have a value of either *Open*, *Opening* or *Closed* (the state of the door "closing" is not a necessary abstraction for this case study and hence not considered to keep the case study simple). This variable denotes the state of the train doors.

Train This agent can be regarded as the provider of information on the current speed of the train. The agent introduces an enumerated set *SPEED* with the elements *Stationary* and *Moving*, and a variable *t_speed*. This variable is of type *SPEED* allowing it to have a value of either *Stationary* or *Moving* that indicates the current train speed.

Operator This agent introduces the variable *request_door* of type *BOOL*. When this variable has the value *TRUE* indicates a request from the operator to open train door, while the value *FALSE*, indicates the operator requests close the train doors.

Brake This agent introduces the variable *brake* of type *BOOL*. The value $TRUE$ for this variable denotes the activation of the emergency brake, while the value $FALSE$ when the value is $FALSE$ denotes the emergency brake is not activated.

TDCS This agent introduces the variable *door_cmd* of type *BOOL*. The variable *door_cmd* denotes the command issued by TDCS to open the train door, i.e. when the value *TRUE*.

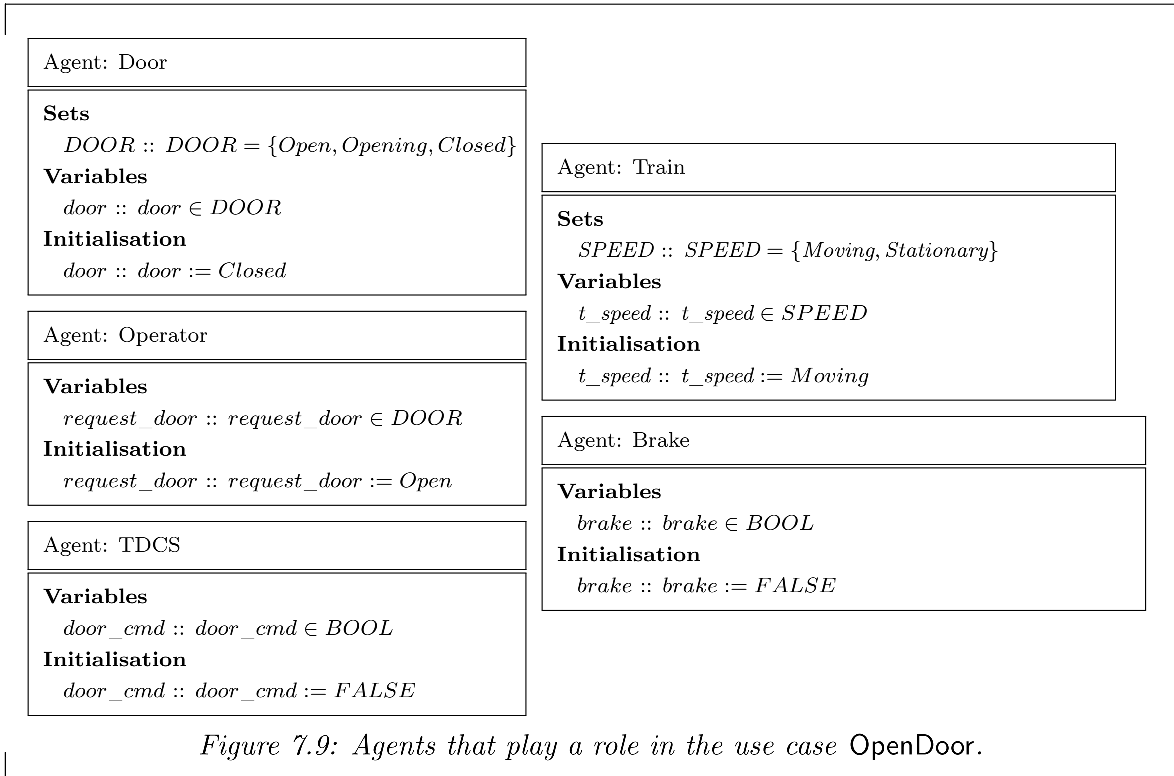


Figure 7.9: Agents that play a role in the use case OpenDoor.

Use Case Specification

The specification for the OpenDoor use case is provided in Figure 7.10. The contract of the use case specifies the pre-condition (@OD_Pre_1) where the door is to be guaranteed to be closed before the use case is executed. The execution of the use case achieves the post-condition (@OD_Post_1) where the train door is open when the train is stationary, or it remains closed as the train is moving. The invariant (@OD_Inv_1) specifies the safety requirement where the door must never be open when the train is moving. This property must be maintained throughout the execution of the use case.

The scenario of OpenDoor specifies a main flow that triggers (@OD_Trig_1) when door is closed and the operator has requested the train doors to open. The step labelled OD_1 introduces a simple branching via conditional where if the speed sensor has read the train is stationary, then the sub-steps OD_1_1, OD_1_2 and OD_1_2 may execute sequentially. If this condition is *false*, the execution of the scenario skips the sub-steps and leads to the end of the use case. This creates a branch in the scenario

Use case: OpenDoor (OD)
Roles: Train, Operator, Door, Speed Sensor.
Contract *Pre-conditions:* OD_Pre_1: Train door is closed. *Post-conditions:* @OD_Post_1: Train is stationary and door is open or train is moving and door remains closed. *Invariants:* @OD_Inv_1: Door must never be open while train is moving.
Scenario *Triggers:* @OD_Trig_1: Operator requests to open door. *Main Flow*: ⟨deviation-point: DoorOpensWhileTrainMoving⟩ OD_1: **if** Train speed is stationary **then** OD_1_1. TDCS issues open door command. OD_1_2. Door starts to open. OD_1_3. Door opened.
Deviations *Deviation*: DoorOpensWhileTrainMoving *Deviation-point:* OD_2
Extensions *Extension*: EmergencyBraking; *Status:* Prevent *Extension-point:* DT_2; *Rejoin-point:* DT_2

(a) Informal

Use case: OpenDoor (OD)
Roles: Train, Operator, Door, Speed Sensor.
Contract *Pre-conditions:* @OD_Pre_1: $door = Closed$ *Post-conditions:* @OD_Post_1: $(t_speed = Stationary \wedge door = Open) \vee (t_speed = Moving \wedge door = Closed)$ *Invariants:* @OD_Inv_1: $\neg(t_speed = Moving \wedge door = Open)$
Scenario *Triggers:* @OD_Trig_1: $door = Closed \wedge request_door = Open$ *Main Flow*: ⟨deviation-point: DoorOpensWhileTrainMoving⟩ OD_1: **if** $t_speed = Stationary$ **then** OD_1_1. $door_cmd := TRUE$ OD_1_2. $door := Opening$ OD_1_3. $door := Open$
Deviations *Deviation*: DoorOpensWhileTrainMoving *Deviation-point:* OD_2
Extensions *Extension*: EmergencyBraking; *Status:* Prevent *Extension-point:* DT_2; *Rejoin-point:* DT_2

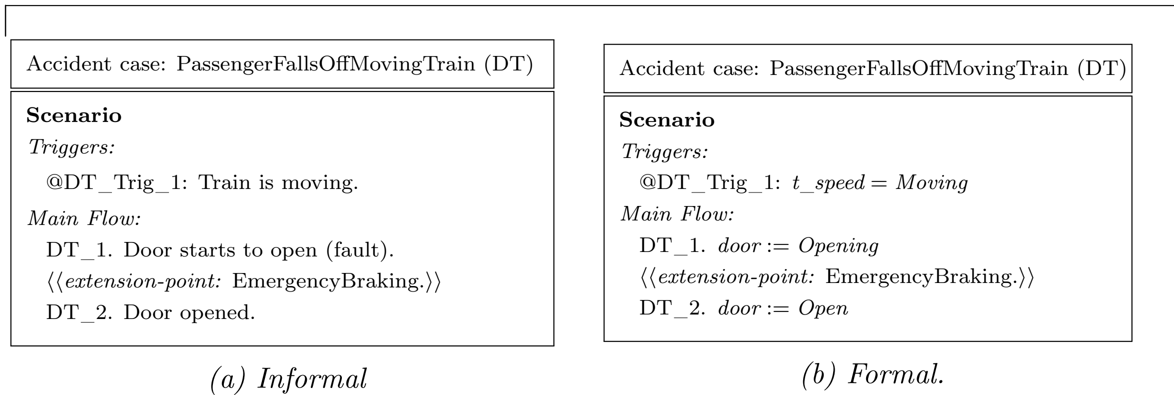
(b) Formal.

Figure 7.10: Informal and formal specification for use case OpenDoor.

of the use case, and the choice in this branching is based on the train location and train speed.

The specification for the accident case PassengerFallsOffMovingTrain is provided in Figure 7.11. It captures a simple accident scenario where the doors begins to open due to a fault introduced in step DT_1. The subsequent action is for the door to be fully opened at step DT_2. This final step in the accident scenario leads to an accident provided by the environmental condition, which is that the train is moving, is set to *true*. This environmental condition is introduced via the trigger condition @DT_Trig_1 that allows the accident scenario to only deviate the OpenDoor use case when the train is moving. In order to prevent this accident, the extension use case EmergencyBraking is introduced between the steps DT_1 and DT_2 via an extension-point.

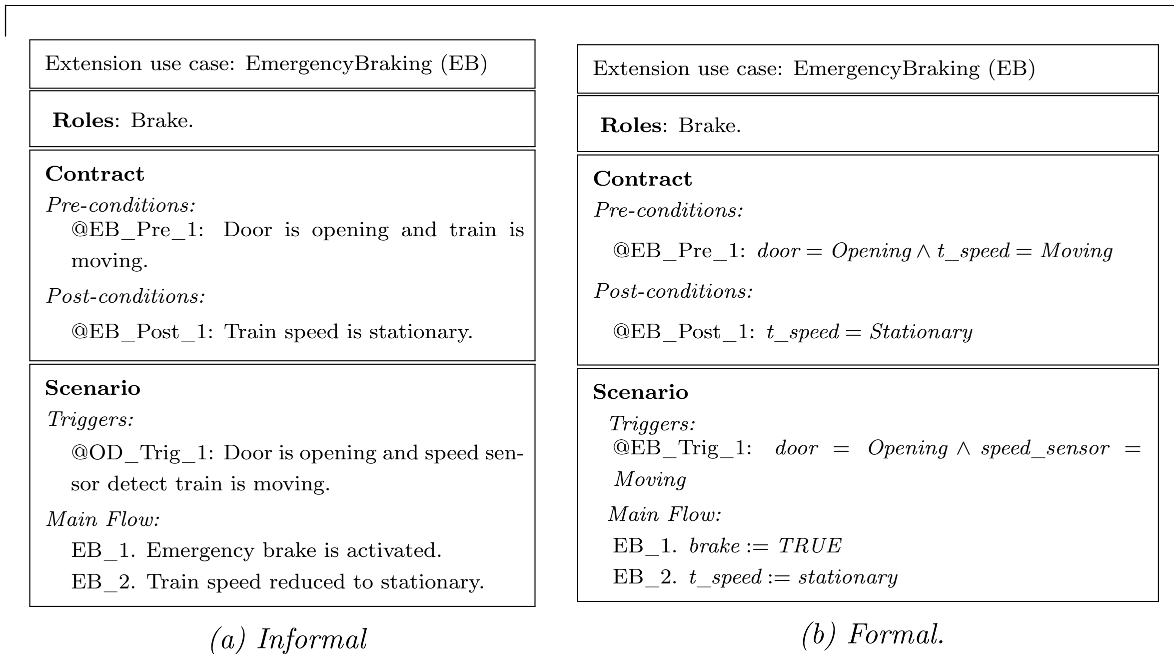
The specification for the extension use case EmergencyBraking is provided in Figure



(a) Informal (b) Formal.

Figure 7.11: Informal and formal specification for use case PassengerFallsOffMovingTrain.

7.12. It introduces an additional functionality to stop the train speed to stationary in the event of a potential accident. The behaviour of this extension use case is performed provided that the train doors begin to open while the train is moving (pre-condition @EB_Pre_1). The extension use case introduces the scenario where the emergency brake is activated at step EB_1. This results in the the train speed being reduced to stationary at steps EB_2.



(a) Informal (b) Formal.

Figure 7.12: Informal and formal specification for extension use case EmergencyBraking.

7.3.2 Event-B

The Event-B model produced for OpenDoor has three machines: m0_OpenDoor_Contract, m1_OpenDoor_Scenario, m1_OpenDoor_Scenario; and two contexts: OpenDoor_Static

and OpenDoor_Flow. The full details of these are given in Appendix B.2. The context OpenDoor_Static models the enumerated sets and constants defined by the agents that are associated with the OpenDoor use case. The context OpenDoor_Flow models the type that is used to simulate the scenario. The state charts produced by ProB for this Event-B model is seen in Appendix C.2.

m0_OpenDoor_Contract

The contract of OpenDoor is introduced in this machine. The event OpenDoor models the pre-condition (@OD_Pre_1) and post-condition (@OD_Post_1) as its guard and action, respectively. The transformation of the post-condition to the action takes the following form, where all occurrence of the variables *door* and *t_speed* are primed.

$$door, t_speed :\mid (t_speed' = Stationary \land door' = Open) \lor (t_speed' = Moving \land door' = Closed)$$

(action OD_Post_Act in event OpenDoor)

The invariant, labelled @OD_Inv_1, is introduced as an invariant of the machine as it contains the variable *door*. The main mathematical judgement made in this abstract machine is to ensure that the action that may *open* the train door must only do so when the train is *stationary*. The invariant preservation proof obligation OpenDoor/OD_Inv_1/INV is produced to ensure that this property is maintained by the event OpenDoor.

$$(t_speed' = Stationary \land door' = Open) \lor$$
$$(t_speed' = Moving \land door' = Closed)$$
$$\vdash$$
$$\neg(t_speed' = Moving \land door' = Open)$$

(OpenDoor/OD_Inv_1/INV)

This PO is automatically proved. It establishes that the use case to open the train doors of the constraint imposed by the invariant of the use case. This abstract machine for OpenDoor establishes *what* the use case achieves, without specifying *how*.

m1_OpenDoor_Scenario

This machine models the scenario of OpenDoor and *refines* the abstract machine m0_OpenDoor_Contract. It introduces the variables and events associated with detailing the scenario. The variables introduced in this machine are separated by the encoding, as follows:

Abstract variables The variables *OD*, *door* and *t_speed* that were introduced for the event OpenDoor in the abstract machine are treated as *abstract variables* in this machine.

Concrete variables The variables *request_door*, *door_m1*, *t_speed_m1* and *OD_flow* are introduced to model the scenario of MaintainH. The gluing invariants labelled @OD_Glue_Variables and @OD_Glue_Flow, are introduced to relate the concrete variables *t_speed_m1*, *door_m1* and *OD_flow* to their corresponding abstract variables *t_speed*, *door* and *OD*, respectively.

Events are introduced that model the scenario of OpenDoor in this machine. The invariants @OD_Scenario_Post and @OD_Scenario_Inv are introduced to ensure that the events that model the scenario maintains the invariants and post-condition of OpenDoor. In these invariants, the concrete variable *door_m1* (highlighted) replaces all occurrences of its corresponding abstract variable *door*, as follows:

$$OD_flow = OD_Final \Rightarrow (t_speed_m1 = Stationary \wedge door_m1 = Open) \vee (t_speed_m1 = Moving \vee door_m1 = Closed) \quad \text{(OD_Scenario_Post)}$$

$$\neg(t_speed_m1 = Moving \wedge door_m1 = Open) \quad \text{(OD_Scenario_Inv)}$$

As the main flow has a *simple branching* (at step @OD_1) via the conditional **if**, there are two paths that lead to the end of the use case. The step @OD_1 is modelled by two events OD_1_If and OD_1_If_False. The event OD_1 is *enabled* when the execution reaches $flow_{od} = OD_1$ and the *condition* of the step OD_1 is *true*. The execution of this event goes through the event sequence OD_1_1, OD_1_2 and finally to the *end* of the use case, i.e. action *OD_flow* := *OD_Final*. On the other hand, the event OD_1_If_False leads directly to the end of the use case, i.e. action *OD_flow* := *OD_Final*. The condition (predicate) specified for step @OD_1 is negated (highlighted) and introduced as the guard for event OD_2_If_False, as follows:

$$\neg(speed_sensor = Stationary) \quad \text{(A guard of event OD_1_If_False)}$$

The invariants labelled @OD_StepAssert_1 and @OD_StepAssert_2, ensure certain properties are maintained over certain steps. For example, the speed sensor readings

remain the same over steps OD_1_1, OD_1_2 and OD_1_3.

$$OD_flow \in \{OD_1_1, OD_1_2, OD_1_3\} \Rightarrow t_speed_m1 = Stationary \quad \text{(OD_StepAssert_3)}$$

The INV POs for invariant @OD_Scenario_Post on events OD_1_If_False and OD_1_3 are required to ensure that the door must only be opened when the train is stationary and aligned. However, the only information available for events @OD_1_If_False and @OD_1_3 are the readings from the speed sensor. For event OD_1_If_False it is known that the speed sensor have detected that the train is moving and the door remains closed.

$$\begin{aligned}
&OD_flow = OD_2\\
&door_m1 = Closed\\
&\neg(t_speed_m1 = Moving)\\
&\vdash\\
&(t_speed_m1 = Stationary \land door_m1 = Open) \lor\\
&(t_speed_m1 = Moving \land door_m1 = Closed)
\end{aligned} \quad \text{(OD_1_If_False/OD_Scenario_Post/INV)}$$

The POs for invariant @OD_Inv_1 on events OD_2_2 and OD_2_If_False are automatically discharged. The scenario of the accident case is introduced by steps DT_Trigger, DT_1 and DT_2. The event DT_2 is required to achieve the post-condition of the use case as it leads to the end of the use case scenario. The action of this final event opens the train door due to a fault. The extension use case is introduced to prevent this accident scenario by introducing the two events EmergencyBraking and EmergencyBraking_FALSE between events DT_1 and DT_2.

The event EmergencyBraking models the pre-condition (@EB_Pre_1) and post-condition (@EB_Post_1) of the extension use case as its guard and action, respectively, based on the encoding for the extension use case. This machine is later refined to introduce the scenario of @EmergencyBraking. The execution of this event achieves the post-condition (@EB_Post_1) that reduces the train speed to stationary via the emergency brake. This allows the PO for event DT_2 to prove that the post-condition, in which the action to open the train doors does not violate the safety constraint and the train speed is stationary.

m2_EmergencyBraking_Scenario

This machine m2_EmergencyBraking_Scenario *refines* m1_OpenDoor_Scenario to introduce the scenario of EmergencyBraking. The abstract event EmergencyBraking is refined by the events that model the scenario of the extension use case. In this refinement, the variable *brake* is introduced. The steps in the scenario of the extension use case EB_1, EB_2 and EB_3 are introduced as events.

The invariant @EB_Scenario_Post ensures that the event @EB_3, which leads to the end of the extension use case, achieves the post-condition where the train speed is required to be stationary (@EL_Post_1). This produces the invariant preservation PO, EB_3/EB_Scenario_Post/INV. This PO is automatically discharged as the final step ensures the train speed is stationary via the reading from the speed sensor.

7.4 Case Study UC3: Automated Teller Machine

An *automated teller machine* (ATM) is a banking subsystem that provides bank customers with access to financial transactions in a public space without the need for a cashier or bank teller. A customer may use the ATM for services such as to check balance, cash withdrawal, depositing funds, etc. The ATM case study is a popular example used to describe UML use cases. The case study used in this thesis is partially based on the one found in [47]. A use case diagram for an ATM can be seen in Figure 7.13. In the use case diagram, the *withdrawal* service is taken into account via the use case Withdraw. The use case is associated with the actors Customer and Bank that are external to the system.

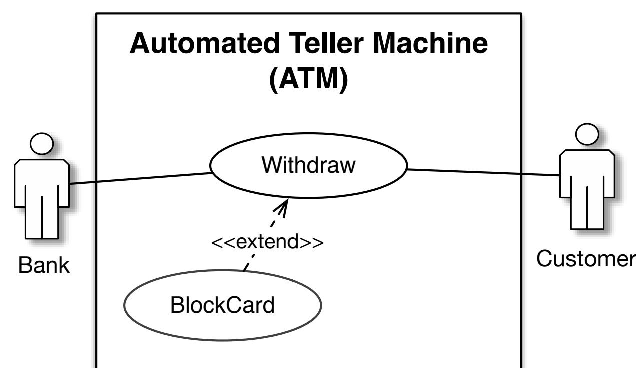


Figure 7.13: Use case diagram for the automated teller machine (ATM).

The withdrawal service is allowed to initiate when the customer card is inserted into the ATM. It captures the functionality to dispense a sufficient withdrawal request

from the customer. The customer is authenticated via a PIN (Personal Identification Number) registered to the bank account of the customer. The customer is allowed to enter the PIN incorrectly only two times. When the third attempt is incorrect, the customer card is withheld and the customer account is blocked. The extension use case BlockCard extends Withdraw to introduce this additional functionality.

The purpose of this case study is twofold: (1) is to use *complex branching* in the scenario of the use case via *alternate flows*, and (2) to use an *ordinary* extension use case opposed to ones used for *mitigate* and *prevent* accident cases. The use case model for this case study is provided in Section 7.4.1 and the Event-B model generated for the use case Withdraw is described in Section 7.4.2

7.4.1 Use Case Model

Agents

The actors and subject in the use case diagram Customer, Bank and ATM are introduced as *agents* in the use case model, as seen in Figure 7.14. These agents are described as follows:

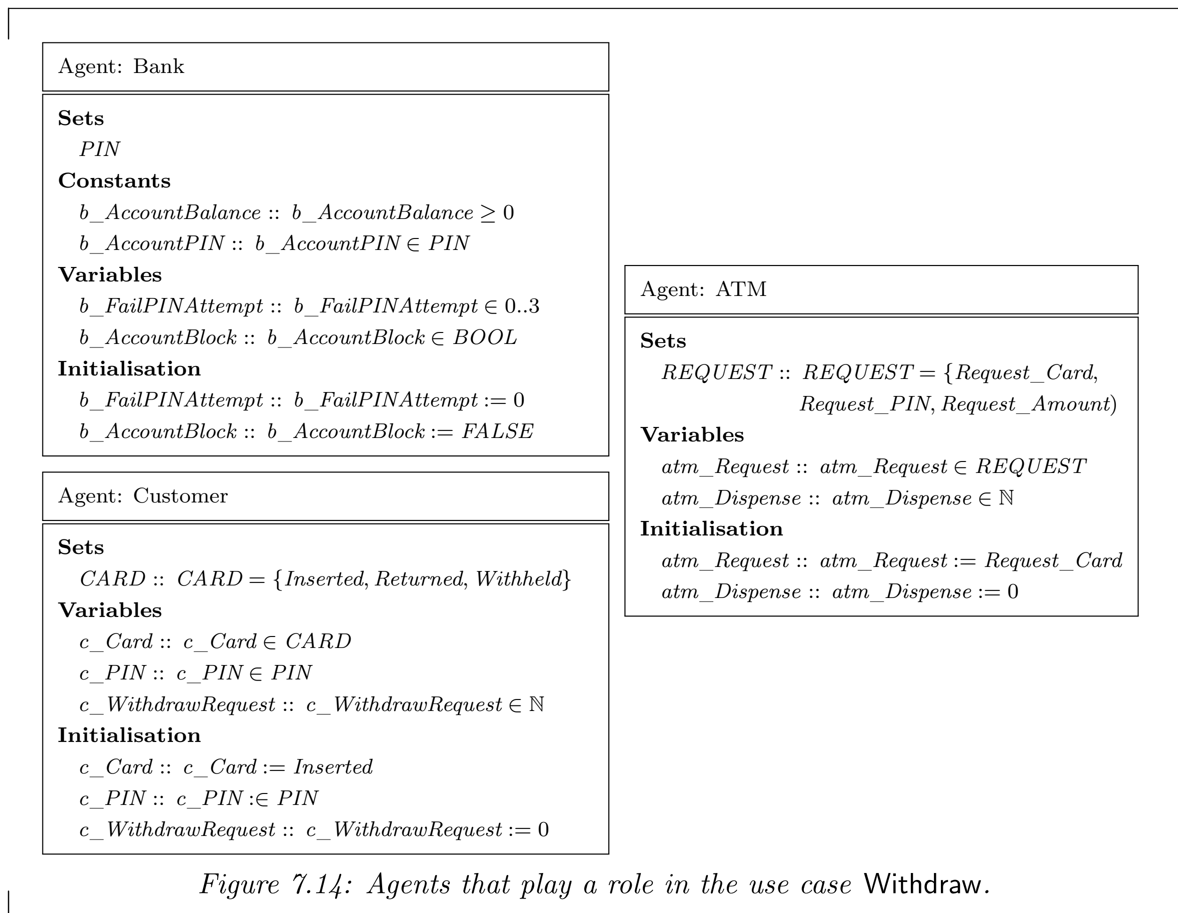


Figure 7.14: Agents that play a role in the use case Withdraw.

Bank This agent introduces the constants *b_AccountBalance* and *b_AccountPIN* that denote a balance and PIN registered to the bank account of the customer. The set *PIN* represents a collection of PINs where, $b_AccountPIN \in PIN$, registers one PIN to the account. The variable *b_FailPINAttempt* denotes the number of registered failed PIN attempts by the customer on the account with its value ranging from 0 to 3. The variable *b_AccountBlock* is of type *BOOL*, where *TRUE* indicates the account is blocked, while *FALSE* indicate that it is not blocked.

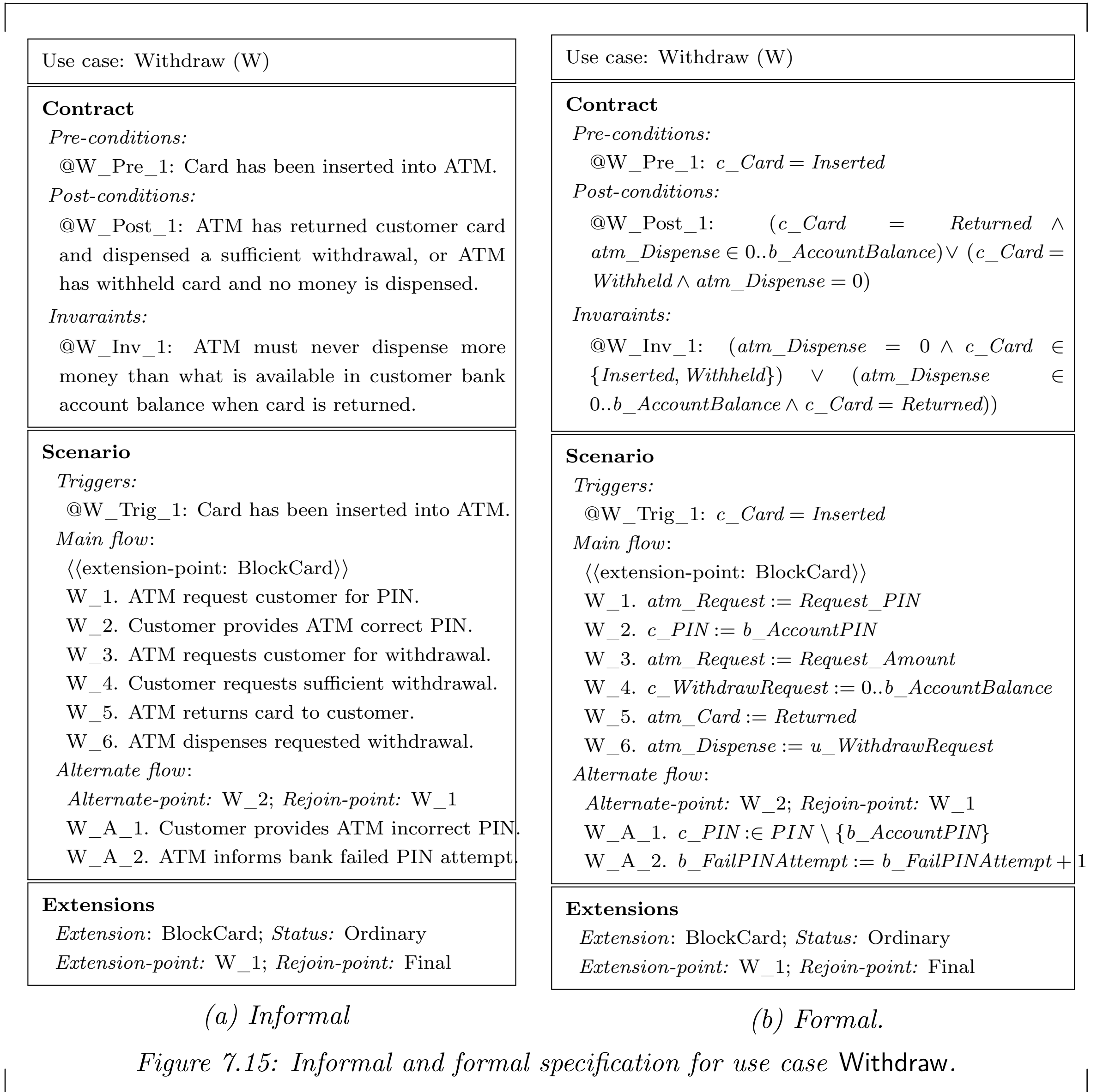
Customer The customer defines the variables *c_PIN* and *c_WithdrawRequest*, which denote the PIN and amount for withdrawal the customer may provide to the ATM, respectively. The variable *c_Card* is the card provided to the ATM. It is of type *CARD*, which is an enumerated set where the card provided is either *Inserted*, *Returned* or *Withheld*.

ATM The ATM agent introduces the variables *atm_Request* and *atm_Dispense*. Variable *atm_Request* is used to indicate the requests that the ATM can make to the customer during the withdrawal service. This variable is of type *REQUEST* which is an enumerated set of values *Request_Card*, *Request_PIN*, *Request_Withdrawal*. The *atm_Dispense* variable is a numerical value that represents the amount of money dispensed by the ATM during the withdrawal service.

Use Cases

The informal and formal specification for the use case Withdraw is provided in Figure 7.15. The contract specifies a pre-condition (@W_Pre_1), which requires that the customer to have inserted the card into the ATM for the withdrawal service. The invariant (@W_Inv_1) specifies an important constraint where the ATM must only dispense an amount within the limit of the customer bank account balance, and if the customer card is withheld, then no money must be dispensed. The post-condition (@W_Post_1) states that the customer card is returned and some amount of money within the limit of the customer's account balance is dispensed, or the card is withheld and no money is dispensed.

The scenario for Withdraw specifies a *main flow* and an *alternate flow*. The main flow specifies an *sunny day* scenario where the interaction between customer and the ATM have no errors or exceptions. For example, at step W_2 and W_4, the customer provides the correct PIN and requests a sufficient amount for withdrawal (within the bank account balance). However, the alternate flow introduces an *alternate-point* at step W_2, where the steps W_A_1 and W_A_2 are introduced. These steps capture the interaction where the customer provides an incorrect PIN and the ATM informs



(a) Informal

Use case: Withdraw (W)

Contract

Pre-conditions:

@W_Pre_1: Card has been inserted into ATM.

Post-conditions:

@W_Post_1: ATM has returned customer card and dispensed a sufficient withdrawal, or ATM has withheld card and no money is dispensed.

Invaraints:

@W_Inv_1: ATM must never dispense more money than what is available in customer bank account balance when card is returned.

Scenario

Triggers:

@W_Trig_1: Card has been inserted into ATM.

Main flow:

⟨⟨extension-point: BlockCard⟩⟩

W_1. ATM request customer for PIN.

W_2. Customer provides ATM correct PIN.

W_3. ATM requests customer for withdrawal.

W_4. Customer requests sufficient withdrawal.

W_5. ATM returns card to customer.

W_6. ATM dispenses requested withdrawal.

Alternate flow:

Alternate-point: W_2; *Rejoin-point:* W_1

W_A_1. Customer provides ATM incorrect PIN.

W_A_2. ATM informs bank failed PIN attempt.

Extensions

Extension: BlockCard; *Status:* Ordinary

Extension-point: W_1; *Rejoin-point:* Final

(b) Formal

Use case: Withdraw (W)

Contract

Pre-conditions:

@W_Pre_1: $c_Card = Inserted$

Post-conditions:

@W_Post_1: $(c_Card = Returned \land atm_Dispense \in 0..b_AccountBalance) \lor (c_Card = Withheld \land atm_Dispense = 0)$

Invaraints:

@W_Inv_1: $(atm_Dispense = 0 \land c_Card \in \{Inserted, Withheld\}) \lor (atm_Dispense \in 0..b_AccountBalance \land c_Card = Returned))$

Scenario

Triggers:

@W_Trig_1: $c_Card = Inserted$

Main flow:

⟨⟨extension-point: BlockCard⟩⟩

W_1. $atm_Request := Request_PIN$

W_2. $c_PIN := b_AccountPIN$

W_3. $atm_Request := Request_Amount$

W_4. $c_WithdrawRequest := 0..b_AccountBalance$

W_5. $atm_Card := Returned$

W_6. $atm_Dispense := u_WithdrawRequest$

Alternate flow:

Alternate-point: W_2; *Rejoin-point:* W_1

W_A_1. $c_PIN :\in PIN \setminus \{b_AccountPIN\}$

W_A_2. $b_FailPINAttempt := b_FailPINAttempt + 1$

Extensions

Extension: BlockCard; *Status:* Ordinary

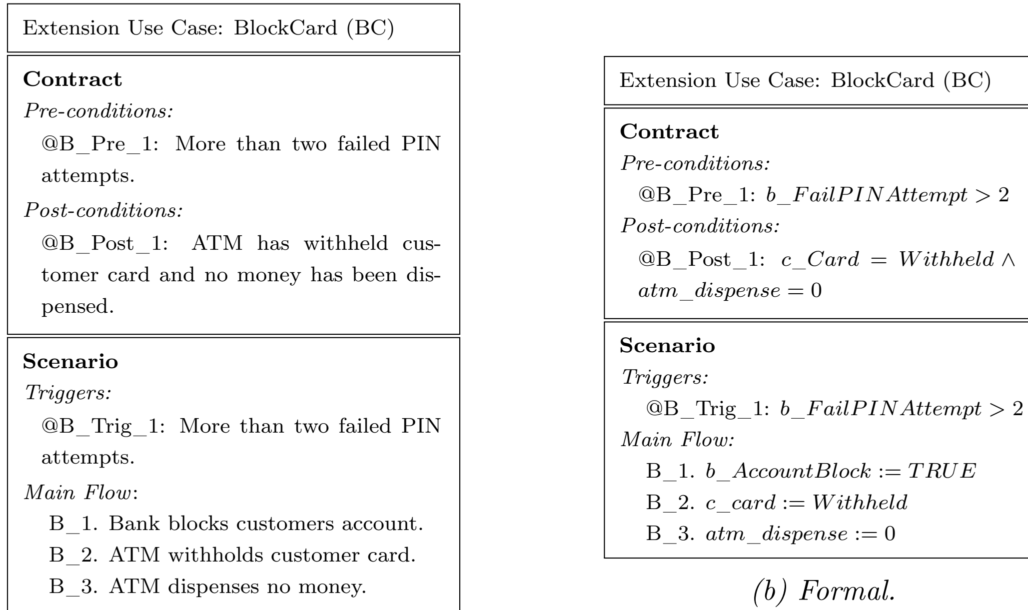
Extension-point: W_1; *Rejoin-point:* Final

(a) Informal *(b) Formal.*

Figure 7.15: Informal and formal specification for use case Withdraw.

the bank of the failed PIN attempt. The alternate flow specifies a *rejoin-point* where the flow returns to the main flow at step W_1, where the customer is requested to enter the PIN again.

An extension is introduced in Withdraw that refers to the extension use case Block-Card. The specification for this extension use case is seen in Figure 7.16. This extension specifies an *extension-point* at step W_1 in Withdraw. That is, the behaviour of the extension use case is introduced before this step. The extension use case may execute when the number of failed PIN attempts exceed two, as denoted by the pre-condition (@B_Pre_1) of the extension use case. The post-condition (@B_Post_1) requires the ATM to have dispensed no money and the customer card withheld. The scenario of the extension use case ensures that this post-condition is achieved and also interacts with



(a) Informal

(b) Formal.

Figure 7.16: Informal and formal specification for extension use case BlockCard.

the bank to block the customer bank account B_1. The *rejoin-point* for this extension is specified at Final, which indicates the execution returns to the ends of the use case Withdraw.

7.4.2 Event-B

The Event-B model produced for Withdraw has three machines: m0_Withdraw_Contract, m1_Withdraw_Scenario and m1_BlockCard_Scenario. It also contains three contexts: Withdraw_Static, Withdraw_Flow and BlockCard_Flow, as seen in Appendix B.3. The context Withdraw_Static captures the enumerated sets and constants defined by the agents that are associated with detailing Withdraw and BlockCard. The context Whitdraw_Flow models the type $FLOW_W$ for the scenario of Withdraw, and the context BlockCard_Flow models the type $FLOW_BC$ for the scenario of BlockCard. The state charts produced by ProB for this Event-B model is seen in Appendix C.3.

m0_Withdraw_Contract

This machine models the contract of the Withdraw use case. The event Withdraw is enabled when the pre-condition @W_Pre_1 of the card inserted in ATM is *true*. The execution of this event achieves the post-condition @W_Post_1, where the customer card is returned and some money has been dispensed or the card has been withheld

and no money has been dispensed. The transformation of the post-condition to the action takes the following form, where all occurrence of the variables *atm_Dispense* and *c_Card*, are primed (highlighted), as follows:

$$atm_Dispense, c_Card :|(c_Card' = Returned \wedge atm_Dispense' \in 0..b_AccountBalance) \vee (c_Card' = Withheld \wedge atm_Dispense' = 0)$$

(action W_Post_Act in event Withdraw)

The invariant, @W_Inv_1, contains the variables *atm_Dispense* and *c_Card* and therefore is introduced in the abstract machine. The main mathematical judgement made in this abstract machine is to ensure that what is achieved by the post-condition (@W_Post_1) of the use case maintains the constraints of this invariant. The following invariant preservation PO is produced for event Withdraw and is automatically proved:

$$(c_Card' = Returned \wedge atm_Dispense' \in 0..b_AccountBalance) \vee$$
$$(c_Card' = Withheld \wedge atm_Dispense' = 0)$$
$$\vdash$$
$$(atm_Dispense' = 0 \wedge c_Card' \in \{Inserted, Withheld\}) \vee$$
$$(atm_Dispense' \in 0..b_AccountBalance \wedge c_Card' = Returned))$$

(Withdraw/W_Inv_1/INV)

m1_Withdraw_Scenario

This machine models the scenario of Withdraw, and *refines* the abstract machine. The abstract event Withdraw is refined by the events that model its scenario. The abstract and concrete variables identified by the encoding of the scenario for this machine, are as follows:

Abstract variables The variables *W*, *atm_Dispense* and *c_card* that were introduced as part of the abstract event Withdraw are treated as *abstract variables* in this machine.

Concrete variables The variables *atm_Request*, *c_PIN*, *b_FailPINAttempt*, *c_Card_m1*, *atm_Dispense_m1* and *W_flow*, are the concrete variables associated with the scenario. The gluing invariants @W_Glue_Variables and @W_Glue_Flow are introduced to relate the concrete variables *atm_Dispense_m1*, *c_Card_m1* and *W_flow* to their corresponding abstract variables *atm_Dispense*, *c_Card*, and *W* respectively. The gluing invariants are introduced to relate the concrete and

abstract variables.

The encoding introduces the main flow and alternate flow of Withdraw in this machine. The extension use case BlockCard is introduced before the step W1 via the extension-point. It is modelled by two events: BlockCard and BlockCard_False. The following invariants W_Scenario_Post and W_Scenario_Inv ensure that the post-condition and invariant for the concrete variables c_Card_m1 and $atm_Dispense_m1$ (highlighted) are achieved by the events that model the scenario:

$$\begin{aligned} W_flow = W_Final \Rightarrow (& c_Card_m1 = Returned \wedge \\ & atm_Dispense_m1 \in 0..b_AccountBalance) \vee \\ (& c_Card_m1 = Withheld \wedge atm_Dispense_m1 = 0) \end{aligned} \quad \text{(W_Scenario_Post)}$$

$$\begin{aligned} & (atm_Dispense_m1 = 0 \wedge c_Card_m1 \in \{Inserted, Withheld\}) \vee \\ & (atm_Dispense_m1 \in 0..b_AccountBalance \wedge c_Card_m1 = Returned)) \end{aligned} \quad \text{(W_Scenario_Inv)}$$

The alternate flow introduces steps that captures the interaction where the customer provides an incorrect PIN and the bank is informed of the failed PIN attempts. BlockCard models the pre-condition (@BC_Pre_1) and post-condition (@BC_Post_1) of the extension use case. This event BlockCard ensures that when the customer has three failed PIN attempts, then the customer card is withheld and no money is dispensed. This extension use case is executed and the flow returns to the end of the use case, as specified by the *rejoin-point*, via action $W_flow := W_Final$. This results in the following proof obligation for BlockCard in order to ensure the post-condition is achieved by the scenario.

$$\begin{aligned} & W_flow = W_1, W_flow' = W_Final \\ & atm_Dispense_m1' = 0 \wedge c_Card_m1' = Withheld \\ & \vdash \\ & (atm_Dispense_m1' = 0 \wedge c_Card_m1' \in \{Inserted, Withheld\}) \vee \\ & (atm_Dispense_m1' \in 0..b_AccountBalance \wedge c_Card_m1' = Returned)) \end{aligned} \quad \text{(BlockCard/W_Scenario_Post/INV)}$$

The insertion of the extension use case before step W_1 is important as it prevents the number of failed PIN attempts from being incremented more than three times by

the event W_A_2 in the alternate flow. The following PO is produced to ensure the type $b_FailPINAttempt \in 0..3$ is maintained by event W_A_2 that increments this variable.

$$W_flow = W_A_2$$
$$b_FailPINAttempt \in 0..3$$
$$\vdash$$
$$b_FailPINAttempt + 1 \in 0..3 \qquad \text{(W_A_2/atm_FailPINAttempt_Type/INV)}$$

The invariant W_StepAssert_1 was introduced manually to state that the execution of the steps W_1, W_2 and W_A_2 ensures that the number of failed PIN attempts (highlighted) is *not* more than two. This invariant is maintained due to the insertion of the extension use case which ensures that to reach step *W_1* the number of failed PIN attempts is not more than 2.

$$(BC = TRUE \wedge W_flow = W_1 \Rightarrow \neg(b_FailPINAttempt > 2)) \wedge$$
$$(W_flow \in \{W_2, W_A_2\} \Rightarrow \neg(b_FailPINAttempt > 2)) \qquad \text{(W_StepAssert_1)}$$

The PO W_6/W_Scenario_Post was generated for the final step (W_6) in the main flow to ensure that it achieves the post-condition. This results in the following PO which is automatically discharged.

$$W_flow = W_6$$
$$c_Card_m1 = Returned$$
$$\vdash$$
$$(c_Card_m1 = Returned \wedge c_WithdrawRequest \in 0..b_AccountBalance) \vee$$
$$(c_Card_m1 = Withheld \wedge c_WithdrawRequest = 0)$$
$$\text{(W_6/W_Scenario_Post/INV)}$$

The invariant W_StepAssert_2 was manually introduced to state that the steps W_5 and W_6 ensure that the withdrawal requested by the customer was within the limit of the account (highlighted), and the customer card was returned before the money was dispensed.

$$(W_flow \in \{W_5, W_6\} \Rightarrow c_WithdrawRequest \in 0..b_AccountBalance) \wedge$$
$$(W_flow = W_6 \Rightarrow c_Card_m1 = Returned) \qquad \text{(W_StepAssert_2)}$$

This invariant helps to prove the W_6/W_Scenario_Post/INV PO which ensures that the scenario achieves the post-condition for of Withdraw is satisfied.

m2_BlockCard_Scenario

The machine m1_Withdraw_Scenario is refined by machine m2_BlockCard_Scenario to introduce the scenario of the extension use case BlockCard. The abstract event BlockCard is refined by the events that model the scenario of BlockCard. The encoding of the scenario in this machine distinguishes the variables as follows:

Abstract variables The variables *b_FailPINAttempt*, *atm_dispense_m1*, *c_card_m1*, *BC* and *W_flow* that are associated with the abstract event BlockCard and are therefore treated as *abstract variables* in this machine.

Concrete variables The variables *b_BlockCard*, *b_FailPINAttempt_m2*, *c_Card_m2*, *atm_Dispense_m2*, and *BC_flow* are introduced in this machine as the concrete variables associated with the scenario. The gluing invariants @BC_Glue_Variables, @BC_Glue_Flow are introduced to relate the concrete variables *atm_Dispense_m2*, *b_FailPINAttempt_m2*, *c_Card_m2*, *BC_flow* to their corresponding abstract variables.

The invariants @BC_Scenario_Post ensures that the post-condition on the concrete variables is maintained by the scenario of the extension use case. The extension use case BlockCard inherits the invariants W_Inv_1 of its parent use case Withdraw. The invariant BC_Scenario_Inv ensures that this invariant of the use case is maintained by the events that model the scenario of the extension use case.

$$BC_flow = BC_Final \land W_flow = W_Final \Rightarrow (c_Card_m2 = Withheld \land atm_Dispense_m2 = 0) \qquad \text{(BC_Scenario_Post)}$$

$$(atm_Dispense_m2 = 0 \land c_Card_m2 \in \{Inserted, Withheld\}) \lor (atm_Dispense_m2 \in 0..b_AccountBalance \land c_Card_m2 = Returned)) \qquad \text{(BC_Scenario_Inv)}$$

The post-condition of BlockCard requires that the customer card to be withheld and no money to be dispensed, as the number of failed PIN attempts have exceeded two. The scenario of the extension use case informs the bank to block the card at step B_1, while steps B_2 and B_3 withhold the card and dispense no money. The invariant @BC_StepAssert_1 is introduced manually that ensures the state of the card

being withheld is maintained till the final step B_3. The invariant @BC_StepAssert_2 ensures no money has been dispensed by the ATM till step BC_2.

$$BC_flow = BC_3 \Rightarrow c_Card_m2 = Withheld \qquad \text{(BC_StepAssert_1)}$$

$$BC_flow \in \{BC_Trigger, BC_1, BC_2\} \Rightarrow atm_Dispense_m2 = 0 \qquad \text{(BC_StepAssert_2)}$$

These invariants help the proof obligations associated with the invariant and post-condition of the extension use case to be automatically proved. This machine establishes the consistency of the scenario of the extension use case with its contract.

7.5 Case Study UC4: Sense and Avoid

Sense and Avoid (SAA) is system designed to, where possible, give authority and responsibility for aerial collision avoidance to the pilot of an Unmanned Aerial Vehicle (UAV). SAA is developed as part of the UK's ASTRAEA[1] (Autonomous Systems Technology Related Airborne Evaluation & Assessment) project. While UAVs are not currently permitted to routinely fly in non-segregated UK airspace, the ASTRAEA project has helped to establish some of the concepts of operation of a UAV in non-segregated airspace by developing and demonstrating a synthetic UAV in a controlled environment. BAE Systems[2] is one of the industry project partners of ASTRAEA. One of their objectives has been to determine the requirements and design of a UAV avionics system. SAA forms one of the requisite technologies as part of this project.

In SAA, the *sense* capability enables the UAV to sense (using visual, radar, co-operative transponder or other advanced technologies) all other air traffic in the airspace or ground based obstacles, and to determine whether any air traffic poses a potential conflict. The *avoid* capability enables the UAV to take action to circumvent an impending collision in situations where a loss of separation has occurred. This system provides the UAV pilot with data to determine any course or altitude change to avoid intruders or to autonomously manoeuvre the UAV to eliminate the conflict. In order to provide this service for sense and avoid, the *safe separation* and *collision avoidance zones* are established, as follows:

Safe Separation (SS) SS is a zone around the ownship aircraft is defined (normally 0.5nm Horizontal and 500ft Vertical radii). Safe Separation is maintained if all

[1] ASTRAEA Project Home - `http://astraea.aero`.
[2] BAE Systems[3] `http://www.baesystems.com/`

other air vehicles remain outside this defined zone along its future flight path. The SAA system will detect if and when Safe Separation is predicted to be compromised and then warn the UAV pilot who may then take action to remedy the situation.

Collision Avoidance (CA) An emergency Collision Avoidance zone is defined around the ownership aircraft (normally 500 ft Horizontal and 350ft Vertical radii). The SAA system will ensure that this zone always remains free of all other air vehicles along its flight path. Should other safety provisions fail and it is predicted that the collision avoidance zone will be breached; the SAA system will autonomously manoeuvre to maintain safety.

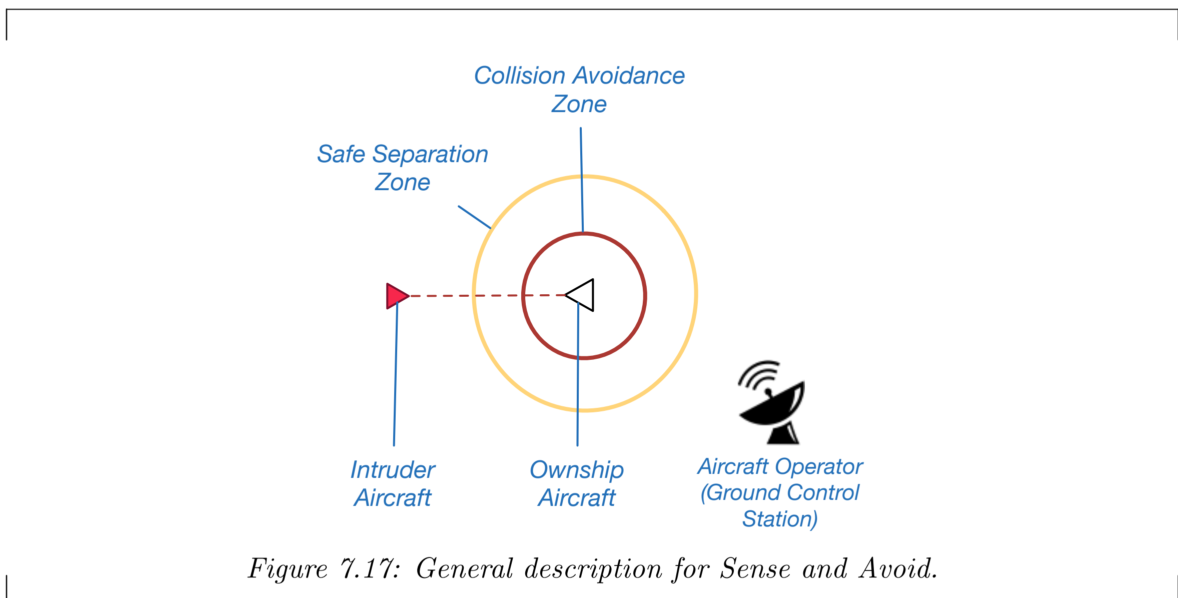


Figure 7.17: General description for Sense and Avoid.

Figure 7.17 provides an informal description of these zones for an ownship (UAV) aircraft. The general requirements for the sense and avoid system are as follows:

- Determination of the risk of loss of *safe separation* with other airborne objects.
- Calculate a plan as avoidance manoeuvre, if necessary, capable of ensuring breach of safe separation is avoided.
- Advice on both risk and avoidance manoeuvres to the decision making authority (aircraft operator) for acceptance or rejection, depending on the urgency of the risk.
- Autonomous *collision avoidance* action in the absence of timely intervention by the decision making authority, if necessary.

Figure 7.18, provides the use case diagram for sense and avoid. The use case diagram introduces the use case SafeSeparation, accident case CollisionWithIntruderAircraft, and extension use case CollisionAvoidance. This case study of Sense and Avoid is simplified, and only take into account the interaction between the actors Intruder Aircraft, Ownship Aircraft, Aircraft Operator and the (subject) Sense and Avoid. Interaction with external entities, such as the Air Traffic Controller (ATC), are not taken into account to reduce the complexity of this case study. The intent of those use cases are as follows:

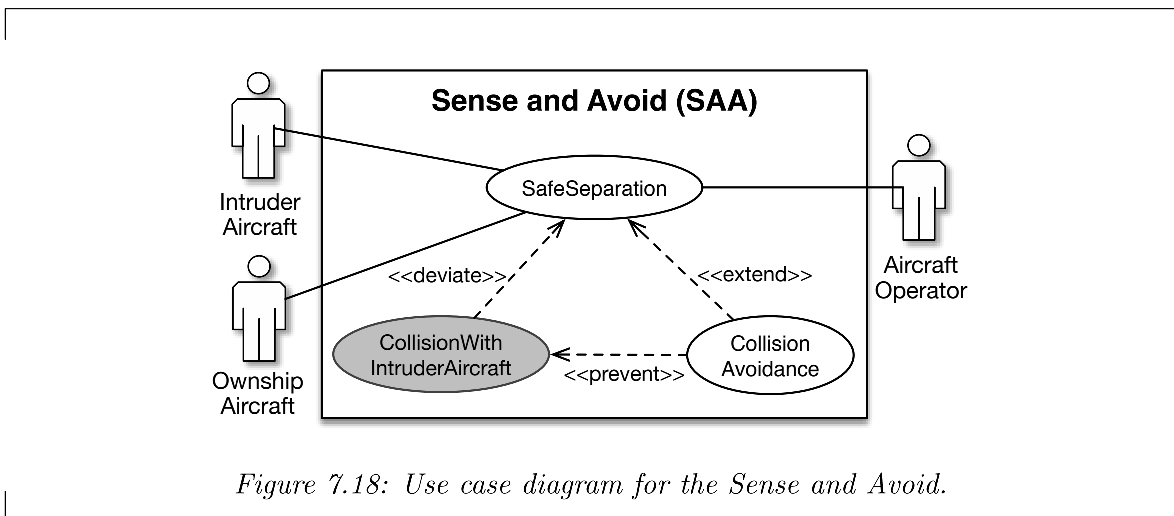


Figure 7.18: Use case diagram for the Sense and Avoid.

SafeSeparation The intent of this use case is to provide service to maintain the safe separation (SS) for an ownship aircraft in the event of an intruder aircraft is detected. The use case is associated with the actors, Ownship Aircraft, Intruder Aircraft, and Aircraft Operator, to achieve this functionality.

CollisionWithIntruderAircraft This accident case introduces a deviation from the functionality of the SafeSeparation use case, where undesired behaviour in the failure to maintain safe separation is provided. Allowing this accident case to complete results in the loss of separation between the intruder and ownship aircraft. There are no new actors introduced by the accident case.

CollisionAvoidance This extension use case is introduced as a means to provide an emergency service of collision avoidance (CA) in the event of failure in the functionality to provide safe separation. This extension use case *extends* SafeSeparation, where any occurrence of its deviating accident case, CollisionWithIntruderAircraft, is *prevented*.

These actors and use cases from the use case diagram, as seen in Figure 7.18, are further detailed in the use case model.

7.5.1 Use Case Model

Agents

The actors and subject in the use case diagram are: Intruder Aircraft, Ownship Aircraft, Aircraft Operator and Sense and Avoid (subject). These can be as seen in Figure 7.18, are introduced as *agents* in the use case model (Figure 7.19). These agents are described as follows:

Intruder Aircraft This can be regarded as a provider of information about external non-cooperative airborne entities to the SAA. The agent introduces the variable *intruder* of type *BOOL*. When this variable has the value *TRUE*, it indicates that an intruder aircraft has been detected.

Sense and Avoid This is the system under consideration. It provides a risk and a plan for forecast of loss of separation or forecast collision with an intruder aircraft. This agent provides the variables *ss_risk* and *ca_risk*, of type *RISK*. The value *Significant* for either of these variables, indicates that the SAA has determined the risk to be significant for the zone they represent. The variables *ss_plan* and *ca_plan* are of type *BOOL*, where *TRUE* indicates that the SAA has provided either a safe separation or collision avoidance plan to the aircraft operator.

Aircraft Operator This agent can be regarded as the user of the SAA. Its role is to receive and send command control data. The agent provides the variables *ss_response* and *ca_response*, which are of type *RESPONSE*. The values of these variables indicate the communication between the aircraft operator and the SAA. The aircraft operator may *accept*, *reject* or remains *idle* with respect to the safe separation or collision avoidance plan being provided by the SAA.

Ownship Aircraft This agent can be regarded as the vehicle that hosts the SAA system. It introduces the variable *separation* of type *BOOL*. The value *TRUE* for a variable denotes that the separation of the ownship aircraft is maintained, and *FALSE* denotes that the separation is lost. The variables *ss_breach* and *ca_breach* are of type *BOOL*. If they are *TRUE* then the safe separation and collision avoidance zones are breached. The variable *mission* denotes the current plan being performed by the ownship aircraft. This is of type *PLAN*. This set is enumerated with elements: *On_Route*, *SS_Plan* and *CA_Plan*.

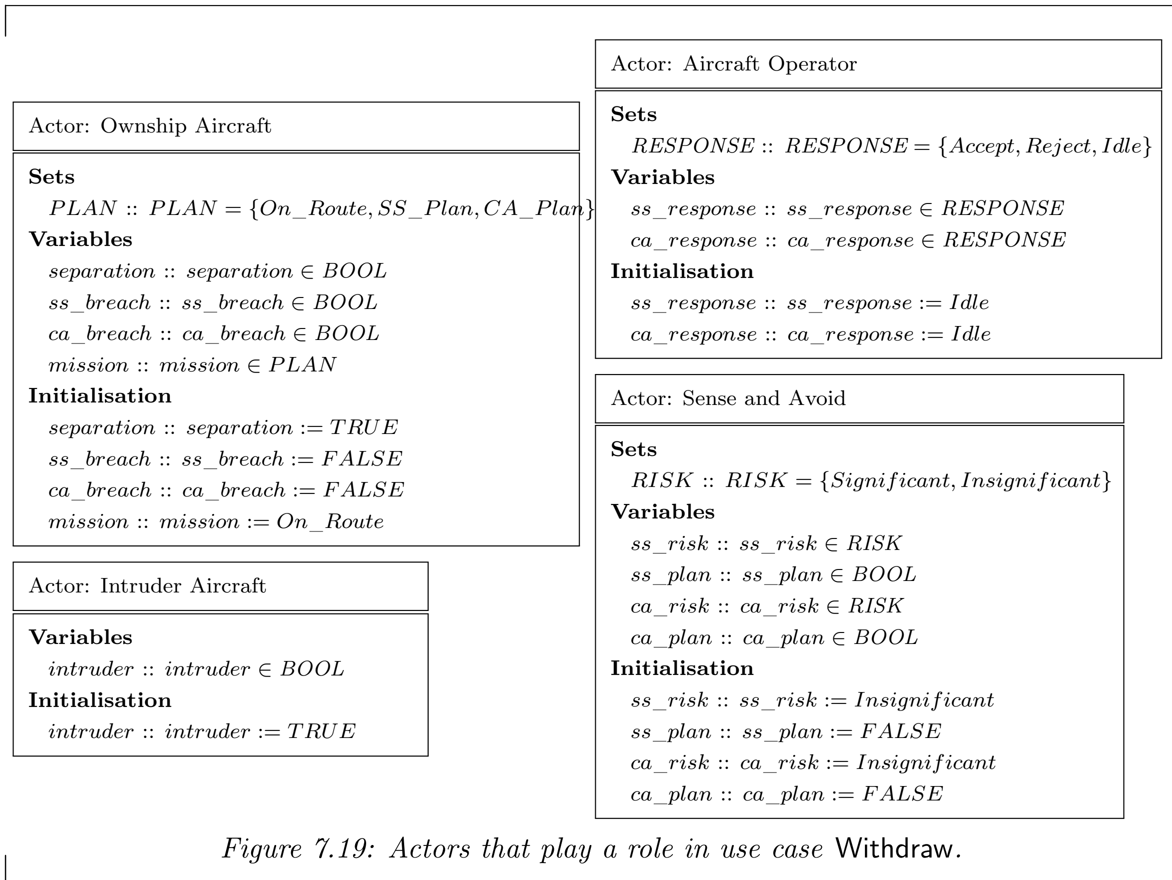


Figure 7.19: Actors that play a role in use case Withdraw.

Use Cases

The use case specification for SafeSeparation is provided in Figure 7.20. The functionality of SafeSeparation is performed when an intruder aircraft has been detected (@SS_Pre_1). The execution of this use case must ensure that the intruder aircraft is no longer a threat and the separation between the ownship and intruder aircraft is maintained (@SS_Post_1). The invariants in the contract explicitly state a constraint where separation must always be provided (@SS_Inv_1). It also states the relationships in the breach of the safe separation and collision avoidance zones with respect to the overall separation provided, by invariants @SS_Inv_2, @SS_Inv_3 and @SS_Inv_4.

The scenario of SafeSeparation provides a *main flow* where the expected interaction to maintain safe separation is provided. There can be alternate flows to this main flow, but these are excluded to maintain the complexity of this case study. The main flow introduces a scenario where the risk from the intruder aircraft to breach safe separation zone is determined to be significant (step SS_1).This results in a safe separation plan being produced by the SAA, which is provided to the aircraft operator (step SS_2). The aircraft operator accepts the plan (step SS_3) as the main flow describes an ideal

Use case: SafeSeparation (SS)

Contract
Pre-conditions:
@SS_Pre_1: Threat from intruder aircraft detected.
Post-conditions:
@SS_Post_1: No threat from intruder aircraft and separation between ownship and intruder aircraft is maintained.
Invariants:
@SS_Inv_1: Separation must always be provided.
@SS_Inv_2: Separation is provided when SS zone and CA zone are not breached, or if CA zone is not breached.
@SS_Inv_3: Separation lost when CA zone is breached.
@SS_Inv_4: CA zone cannot be breached without breach of SS zone.

Scenario
Triggers:
@S_Trig_1: Threat from intruder aircraft detected.
Main Flow:
SS_1. SAA determines risk to SS is significant.
SS_2. SAA generates plan to maintain SS.
SS_3. Aircraft operator accepts SS plan.
SS_4. Ownship performs SS plan as manoeuvre.
SS_5. Threat from intruder aircraft is mitigated.

Deviation
Accident case: CollisionWithIntruderAircraft
Deviation-point: SS_2

Extension
Extension use case: CollisionAvoidance
Status: Prevent;
Extension-point: F_2; *Rejoin-point:* Final

(a) Informal

Use case: SafeSeparation (SS)

Contract
Pre-conditions:
@SS_Pre_1: $intruder = TRUE$
Post-conditions:
@SS_Post_1: $intruder = FALSE \wedge separation = TRUE$
Invariants:
@SS_Inv_1: $separation = TRUE$
@SS_Inv_2: $separation = TRUE \Leftrightarrow (ss_breach = FALSE \wedge ca_breach = FALSE) \vee (ss_breach = TRUE \wedge ca_breach = FALSE)$
@SS_Inv_3: $separation = FALSE \Leftrightarrow ca_breach = TRUE$
@SS_Inv_4: $\neg(ca_breach = TRUE \wedge ss_breach = FALSE)$

Scenario
Triggers:
@SS_Trig_1: $intruder = TRUE$
Main Flow
SS_1. $ss_risk := Significant$
SS_2. $ss_plan := TRUE$
SS_3. $ss_response := Accept$
SS_4. $mission := SS_Plan$
SS_5. $intruder := FALSE$

Deviation
Accident case: CollisionWithIntruderAircraft
Deviation-point: SS_2

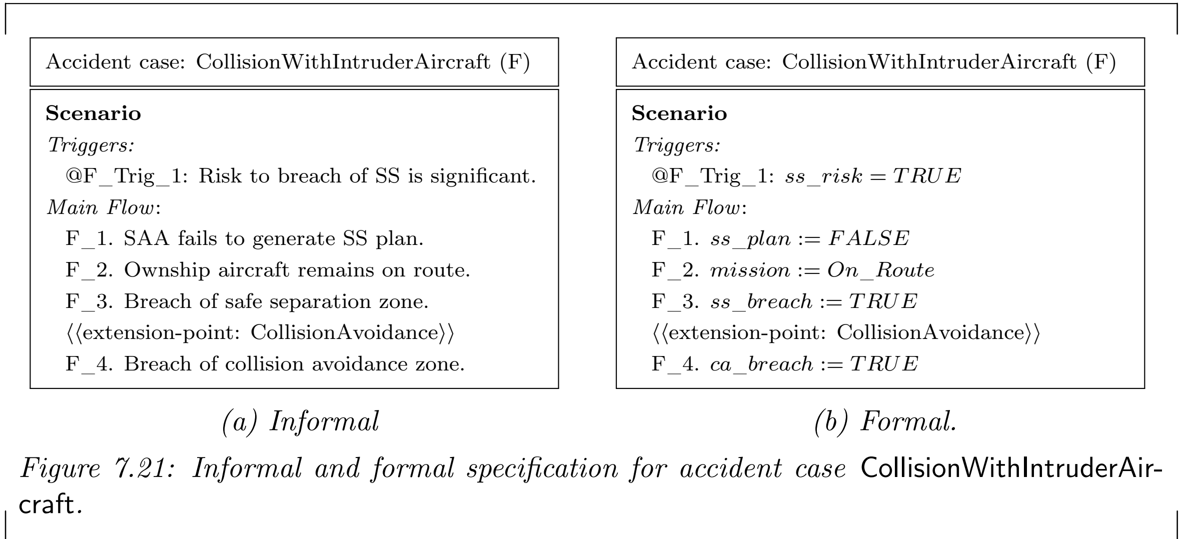
Extension
Extension use case: CollisionAvoidance
Status: Prevent
Extension-point: F_2; *Rejoin-point:* Final

(b) Formal.

Figure 7.20: Specification for use case SafeSeparation*.*

scenario, and the ownship aircraft performs the safe separation plan (step SS_4) to maintain the safe separation zone. This subsequently removes the threat from intruder aircraft (step SS_5).

The accident case CollisionWithIntruderAircraft is introduced as a *deviation* of SafeSeparation at step SS_1. The scenario of the accident case may only trigger (@F_Trig_1) when the risk to breach the safe separation zone is significant. It introduces an accident scenario where the SAA fails to produce a safe separation plan (step F_1), and the ownship aircraft remains on route (step F_2). This results in the safe separation zone being determined by SAA to be breached (step F_3), and subsequently the breach of the collision avoidance zone (step F_5). Allowing the accident scenario to complete



(a) Informal *(b) Formal.*

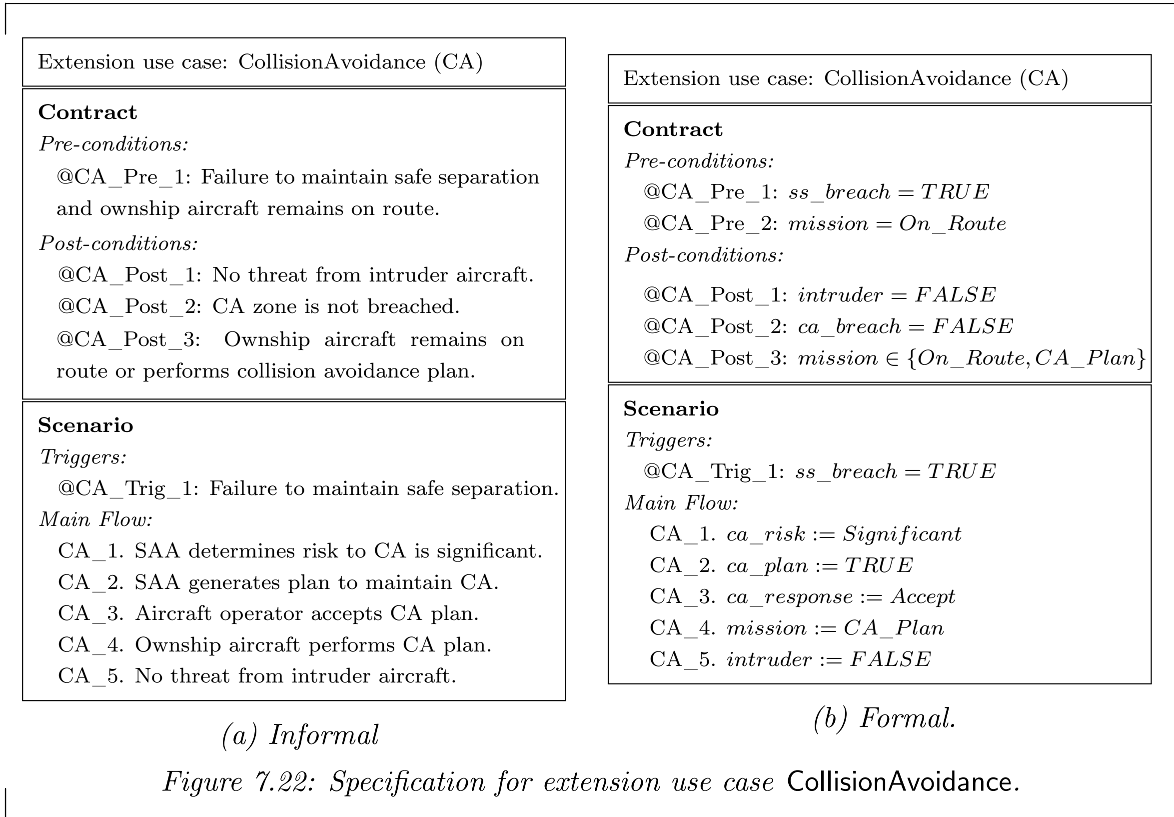
Figure 7.21: Informal and formal specification for accident case CollisionWithIntruderAircraft.

will result in the loss of separation between the ownship and intruder aircraft.

An *extension* is provided to the SafeSeparation use case, where any occurrence of the accident case CollisionWithIntruderAircraft is *prevented* by the extension use case CollisionAvoidance. This extension use case is introduced before the step F_4 via the *extension-point*. The functionality of CollisionAvoidance is performed given the breach of safe separation zone (@CA_Pre_1) and the ownship aircraft remains on route (@CA_Pre_2). The execution of CollisionAvoidance ensures that the threat from intruder aircraft is averted (@CA_Post_1) and the collision avoidance zone is not breached (@CA_Post_2). The functionality of collision avoidance may result in the ownship aircraft either performing a collision avoidance plan or remaining on route (@CA_Post_3). The extension use case specifies a *rejoin-point* that returns the execution to the end of the *SafeSeparation* use case. The extension use case is introduced to prevent the final step of the accident scenario from completing execution.

7.5.2 Event-B

The Event-B model produced for SafeSeparation is provided in Appendix B.4. This takes into account the *deviation* to the accident case CollisionWithIntruderAircraft, and extension from the extension use case CollisionAvoidance. The context SafeSeparation_Static models all the sets and constants associated with the SafeSeparation (this includes the accident case and extension use case). The context SafeSeparation_Flow and CollisionAvoidance_Flow model the type for the scenario of SafeSeparation and CollisionAvoidance, respectively. The state charts produced by ProB for this Event-B model is seen in Appendix C.4.



(a) Informal

(b) Formal.

Figure 7.22: Specification for extension use case CollisionAvoidance*.*

m0_SafeSeparation_Contract

In this machine, only the variables *intruder* and *separation* are introduced as they occur in the pre-condition (@SS_Pre_1) and post-condition (@SS_Post_1) of SafeSeparation. An event SafeSeparation models the pre-condition and post-condition as its guard and action respectively. The event is enabled when the intruder aircraft is detected, and its execution results in the threat from an intruder aircraft to be averted and the separation to be maintained. The post-condition @SS_Post_1, which is a predicate, is transformed to the action, where all occurrence of the variables *intruder* and *separation* are primed (highlighted) on the RHS of the : | operator, as follows :

$$intruder, separation :| \; intruder' = FALSE \land separation' = TRUE$$

(action SS_Post_Act in event SafeSeparation)

The invariant labelled @SS_Inv_1, is introduced in this machine. It establishes a constraint where the separation must always be provided. The proof obligation produced to ensure that the SafeSeparation event maintains this invariant is automatically proved as the post-condition @SS_Post_1 ensures the separation is provided. The invariants @SS_Inv_2 and @SS_Inv_3 are not introduced in this machine as they contain variables that do not occur in the pre-conditions and post-conditions of the use

case, i.e. *ss_breach* and *ca_breach*. These variables and invariants are later introduced in later refinement.

m1_SafeSeparation_Scenario

The machine m0_SafeSeparation_Contract is refined by by this machine to introduce the scenario. The accident scenario of CollisionWithIntruderAircraft and the contract of CollisionAvoidance are taken into account via the *deviation* and *extension* relationships. The variables in this machine are treated as abstract and concrete variables as follows:

Abstract variables The variables *SS*, *separation* and *intruder* that are associated with modelling the abstract event SafeSeparation are treated as the abstract variables.

Concrete variables The variables *ss_risk*, *ss_plan*, *mission*, *ss_response*, *ss_breach*, *ca_breach*, *intruder_m1* and *SS_flow* are introduced as *concrete* variables in this machine. The gluing invariants @SS_Glue_Variables and @SS_Glue_Flow are used to relate concrete variable $intruder_m1$ and SS_Flow to their corresponding abstract variable *intruder* and SS, respectively.

The invariants @SS_Scenario_Inv and SS_Scenario_Post are introduced to ensure that the post-condition and invariants of the use case Separation are satisfied by the scenario. The invariant @SS_Inv_2 and @SS_Inv_3, which were not introduced in the abstract machine, are introduced as part of the invariant SS_Scenario_Inv. The invariant @SS_Scenario_Inv relate the abstract variable *separation* to the concrete variables *ss_breach* and *ca_breach*.

$$SS_flow = SS_Final \Rightarrow intruder_m1 = FALSE \wedge separation = TRUE \quad \text{(SS_Scenario_Post)}$$

The events SS_5 and F_4 (final steps) that lead to the end of the use case are required to ensure that the post-condition is achieved. The SS_5/SS_Scenario_Post/INV PO for event SS_5 is automatically proved, as the final step of the main flow ensure there is no threat from the intruder via its action and, that there is no breach of safe separation or collision avoidance zones in the main flow.

$$\begin{aligned}
&(separation = TRUE) \wedge \\
&(separation = TRUE \Leftrightarrow (ss_breach = FALSE \wedge ca_breach = FALSE) \\
&\vee (ss_breach = TRUE \wedge ca_breach = FALSE)) \wedge \\
&(separation = FALSE \Leftrightarrow ca_zone = FALSE) \wedge \\
&\neg(ca_breach = TRUE \wedge ss_breach = FALSE) \qquad \text{(SS_Scenario_Inv)}
\end{aligned}$$

On the other hand, event F_4 from the accident, does not achieve the post-condition or maintain the invariant, as its action introduces a breach of safe separation and the collision avoidance zones. In order to prevent the accident scenario from completing the extension use case CollisionAvoidance was introduced before the step F_4 via an extension-point. This extension use case is modelled by two events: CollisionAvoidance and CollisionAvoidance_False. The event CollisionAvoidance models the pre-conditions (@CA_Pre_1 and @CA_Pre_2) and post-conditions (@CA_Post_1 , @CA_Post_2 and @CA_Post_3) of the extension use case as its guards and actions, respectively. As the extension use case is of type *prevent*, the invariant CA_Prevent is introduced. It negates the guards of the event CollisionAvoidance_False, and the guards of the events from the extension-point to the end of the accident scenario, i.e. in this case step F_4.

$$\neg(CA = FALSE \wedge SS_flow = F_3 \wedge \neg(ss_breach = TRUE) \wedge \neg(mission = On_Route)) \wedge \neg(SS_flow = F_3 \wedge CA = TRUE) \qquad \text{(CA_Prevent)}$$

This invariant introduces a constraint where the CollisionAvoidance extension use case always executes during the scenario of the accident case, and ensures that the final step of the accident case is not allowed to execute. This required the extension use case to be inserted at the correct step that enables it to capture the failure conditions. The execution of the extension use case returns the flow back to the main flow of the use case it extends, in this case at the end of SafeSeparation ($SS_Flow := SS_Final$) as the rejoin point. The post-condition of the extension use case ensures that the threat from the intruder no longer exists and the collision avoidance zone is maintained. The proof obligations generated to ensure that the events SS_5 and CollisionAvoidance satisfy the invariant @SS_Scenario_Inv and @SS_Scenario_Post are automatically proved. This machine establishes the scenario of SafeSeparataion is consistent with the contract.

m2_CollisionAvoidance_Scenario

The machine m1_SafeSeparation_Scenario is refined by this machine to introduce the scenario of the extension use case, CollisionAvoidance. The abstract event Collision-Avoidance is refined by the events that model the scenario of the extension use case. The variables in this machine are distinguished into:

Abstract variables The variables *ca_breach*, *mission*, *SS_Flow*, *CA* and *intruder_m1* that are associated with the abstract event CollisionAvoidance are treated as the abstract variables.

Concrete variables The variables *ca_risk*, *ca_plan*, *ca_response*, *CA_Flow* , *mission_m2*,

ca_breach_m2 and *intruder_m2*, which are associated with the scenario, are introduced as *concrete* variables in this machine. Invariants @CA_Glue_Variables and @CA_Glue_Flow are introduced to relate the concrete variable *ca_breach_m2*, *intruder_m2*, *mission_m2* and *CA_Flow* with their corresponding to the abstract variables.

The invariants @CA_Scenario_Inv and @CA_Scenario_Post are introduced to ensure that the post-condition and invariants of the extension use case are satisfied by the events that model the scenario of CollisionAvoidance. The invariant @CA_Scenario_Inv takes into account the invariants @SS_Inv_1, @SS_Inv_2 and @SS_Inv_3 of the parent use case SafeSeparation. The invariant replaces all occurrence of the abstract variable *ca_breach* with the concrete variables *ca_breach_m2* (highlighted):

$$CA_flow = CA_Final \Rightarrow intruder_m2 = FALSE \wedge ca_breach_m2 = FALSE \wedge mission_m2 \in \{On_Route, CA_Plan\} \quad \text{(CA_Scenario_Post)}$$

$$\begin{aligned} &(separation = TRUE) \wedge \\ &(separation = TRUE \Leftrightarrow (ss_breach_m2 = FALSE \wedge ca_breach_m2 = FALSE) \\ &\vee (ss_breach_m2 = TRUE \wedge ca_breach_m2 = FALSE)) \wedge \\ &(separation = FALSE \Leftrightarrow ca_zone_m2 = FALSE) \wedge \\ &\neg(ca_breach_m2 = TRUE \wedge ss_breach_m2 = FALSE) \end{aligned} \quad \text{(CA_Scenario_Inv)}$$

These invariants impose constraints on the events that model the scenario of the extension use case. The invariant CA_Scenario_Post places a constraint on the final step CA_5 that requires the post-condition to be achieved by the execution of this event. The action of event CA_5 ensures that the intruder aircraft is no longer a threat. It is necessary to show that the mission performed by the ownship aircraft is a collision avoidance plan. The invariant CA_StepAssert_1 is introduced to ensure that the step CA_4 provides the mission with the plan for collision avoidance.

$$CA_flow = CA_5 \Rightarrow mission_m2 = CA_Plan \quad \text{(CA_StepAssert_1)}$$

This approach of formalising UML use cases was discussed with engineers working within the Intelligence Systems team at BAE Systems, Warton. These engineers routinely used UML use cases to define and analyse system behaviour during the early stages in their systems development process. They found the dual representation of

the use case specification with informal and formal notation helped bring precision to the use cases while maintaining ease of communication. The generation of an Event-B model from a formally specified use case helped provide formal assurance in the behaviour specified by the use cases via proof obligations. However, it was difficult for practitioners, who were not familiar with the Event-B modelling environment, to manually relate failed proof obligations back to the use case specification. Further work was required to automatically help relate failed or undischarged proof obligations at the Event-B level back to the level of the use case specification. This is addressed as part of the future work in Section . Finally, the use of model checking tool Pro-B on the generated Event-B models helped animate the execution of steps in the use case scenario, which provided a better understanding of the use cases. Also, the production of statecharts from the Pro-B tool helped visualise the execution of the different paths in a scenario of use case.

7.6 Summary & Discussion

In this chapter four case studies for UC-B have been introduced. The case studies have covered the different use case types, use case, extension use case and accident case, as well as simple and complex branching within the scenario of the use case. Each case study has been examined with respect the verification provided by the proof obligations generated at the level of the Event-B model. The evaluation describe properties of the requirements that are checked by the formal analysis. In this evaluation some auxiliary invariants were required to be manually introduced to prove some of the generated proof obligations.

The industrial project partnership provided the opportunity to discuss the approach of formalising UML use cases with systems engineers working within the Intelligence Systems team at BAE Systems, Warton. They found the enhancement of the informal use case specification with the formal counterpart provided precision while detailing the use cases. This helped tackle issues with ambiguity while detailing the use cases. The automatic generation of Event-B models from formally specified use cases helped provide formal assurance in the behaviour specified in the use cases. However, it was difficult for practitioners who were not familiar with the Event-B modelling environment to manually relate failed proof obligations back to the use case specification. Further work was required to automatically relate failed proof obligations at the Event-B level back to the level of the use case specification. This is addressed as part of the future work in Chapter 8.

Chapter 8

Future Work & Conclusion

8.1 Contributions revisited

Here we elaborate upon the contributions of the thesis that were outlined in Chapter 1:

- **Accident case**: UML use cases have been extended with the notion of accident case in Chapter 3. This extension enables undesired behaviour identified from the safety analysis to be considered at an early stage in the requirements analysis process. The *deviate* relationship is provided that allows an accident case to be introduced as a deviation from the desired behaviour of a use case. The accident case specifies an accident scenario. The *prevent* relationship is introduced. It provides a mechanism to use an extension use case to introduce additional behaviour that may prevent an accident scenario from completing. This extension of UML use cases with accident case provides a platform for systems and safety engineers to communicate appropriate design recommendations at an early stage of the systems development process.

- **Use case model**: In Chapter 4, a use case model was introduced that allows the textual specification of use cases to be specified formally using Event-B's mathematical language. The use case model provides a specifications for the actor, subject and use cases. The specification aims to reduce the gap between informal and formal methods by allowing a dual representation of requirements with both informal and formal notation. Writing the specification in a precise language removes ambiguity, while maintaining a corresponding informal description provides ease of communication. The abstract syntax for the use cases in the use case model is provided.

- **Encoding use cases in Event-B**: Given a formally specified use case, Chapter 5 has provided an encoding of the use case as an Event-B model. The generic structure of a use cases can be viewed in terms of various levels of abstraction. This has been used to dictate the structure of refinement in the corresponding Event-B model. The encoding has identified gluing invariants that relate the abstract and concrete states in the Event-B model. Providing sufficient but provable gluing invariants can be a significant task. The encoding also takes into account the use case types, accident case and extension use case. The translation rules for encoding the use case model to the Event-B model has been provided.

- **Tool development**: The Rodin platform has been extended to support the authoring and management of a use case model in Chapter 6. The tool, UC-B, enables the specifications to be detailed using the mathematical language of Event-B as well as corresponding informal descriptions in natural language. This dual representation of the specification enables the user to better relate informal and formal artefacts. The translation rules for encoding the use case in an Event-B model, is implemented by the tool. This supports the generation of an Event-B model given a formally specified use case. The generated Event-B model is subjected to Rodin's automatic provers and syntax checkers to ensure the model produced is correct. The aim of this implementation is to reduce the formal modelling effort while allowing the use case modeller to benefit from the use of formal methods during requirements analysis.

- **Case studies**: Evidence of formally specified use cases and their encoding in Event-B has been provided through four case studies in Chapter 7. The case studies have covered: the different types of use cases (use case, extension use case and accident case); simple (conditional) and complex branching (alternate flow) in the scenario; and the deviate and prevent relationship. The use case model of these case studies have been provided that describe how the concepts of use cases are specified formally. Their verification provided by proof obligations generated in the Event-B model describe if the behaviour specified by the use case is consistent, i.e. that the scenario satisfies the contract.

8.2 Limitations

The following describe the limitations in the approach proposed in this thesis:

- **Include use case**: In UML use cases, the use case type *includes* [19] is used to modularise common parts of behaviours of two or more use cases. The use case

model provided in this thesis (Chapter 4) does not treat this type of use case as it was not essential for the case studies being examined. However, in order to make the approach more accessible to practitioners, all use case types must be considered.

- **Auxiliary invariants**: The generated Event-B model often require additional auxiliary invariants to ensure that certain properties are maintained over the steps in the scenario of a use case. These invariants are not crucial to the requirements specification. At this stage, these auxiliary invariants are manually inserted in the Event-B model in order to help prove some of the proofs obligations. The aim of UC-B is to allow the user to be concerned only with the artefacts in the use case specification and not the Event-B model.
- **Traceability**: A generated Event-B model from a source use case produces many proof obligations that provide an indication towards defects in the use case specification. At this stage, the user is required to manually relate failed proof obligations back to the use case specification. The UC-B tool automatically provides labels at the use case level, and these are used used to label the generated Event-B model elements.
- **Redundant variables**: The encoding of a formal use case in Event-B often results in a variable at the use case level having more than one corresponding variables at the Event-B level. This is because the encoding does not replace all abstract variables with concrete variables in the Event-B model. The abstract variables remain along side the concrete variables, and gluing invariants used to relate their states. However, this results in the user having to cope with more variables in the Event-B model.

8.3 Future Work

The work described in the thesis has opened several opportunities for future.

Include Use Case

As part of the future work, the approach will be extended to take into account the includes relationship in UML use cases. This introduced the includes use case in the formal use case model and provide its encoding in Event-B. The relation between the include use case and accident case is also required to be examined. Taking into account the includes use case would provide the use case modeller with more flexibility in documenting and analysing the requirements.

Auxiliary Invariants

The generated Event-B models require auxiliary invariants to help prove some proof obligations. These invariants often specify constraints over a series of steps in the scenario of the use case. At this stage, they are introduced manually by the user once the Event-B model has been generated. This can be time consuming and difficult for users who are primarily interested identifying defects in the requirements. As part of the future work, it would be possible for these auxiliary invariants to be specified at the use case level. These auxiliary invariants could be included as an assertion for each step. These assertions could the be generated in the Event-B model to state properties that are required to be *true* over that step in the execution of the use case scenario.

Traceability

In Chapter 5, proof obligations at the Event-B level have been identified and the meaning of their failure to the use case specification have been described. This work can be further extended by tool support to relate the failed proof obligation to the use case specification. Mechanising the traceability between proofs generated in the Event-B model to the parent use case could help to quickly identify defects in the specification of the use case. At this stage, the user is required to manually related failed proofs between the generated Event-B model and use case specification.

Conformance Testing with Formal Use Cases

Use cases can be used to guide test case generation [84]. Our formalisation of use cases potentially enables a systematic method to identify scenarios that could be used for conformance testing. In comparison to classical test engineering, a much higher degree of automation could be achieved by this work. This future work may provide scenarios for black-box conformance testing from formally specified use cases. The test case generation could computes all paths to states that can be reached. A path may be terminated when the end of the use case is visited. The test generation based on use cases would remain an interactive process, as a human test engineer would still be required to assign priorities to test sequences.

A Methodology for Accident Case

As discussed in Chapter 1, the requirements and safety analysis process are often performed in an ad-hoc manner [73]. While the safety process remains entirely separate from the requirements definition process, the problem associated with incompleteness in requirements with regards to safety is unlikely to be resolved. The extension to

UML use cases with the accident case allow for the safety-related artefacts *accidents*, *hazards* and *accident scenarios* from the safety analysis to be taken into account into the requirements analysis process. This enables the artefacts that originated from the safety analysis to be captured and treated in a manner consistent with other requirements applicable at the development phase, as recommended in ARP4754 [10], which provides guidelines for development of civil aircraft and systems.

A future work for the accident case is to develop and evaluate a methodology that aims to bridge the requirements and safety analysis process, using this extension of the accident case. An initial development for this methodology is provided in Figure 8.1. The methodology aims to state: 'what' steps to take, 'how' these steps are to be performed and most importantly the reasons 'why' the methodology follow those steps in the suggested order. Defining such a methodology is aimed at alleviating some of the current discontinuities that exist between the requirements and safety process, and improve the confidence in the systematic identification of safety-related functional requirements.

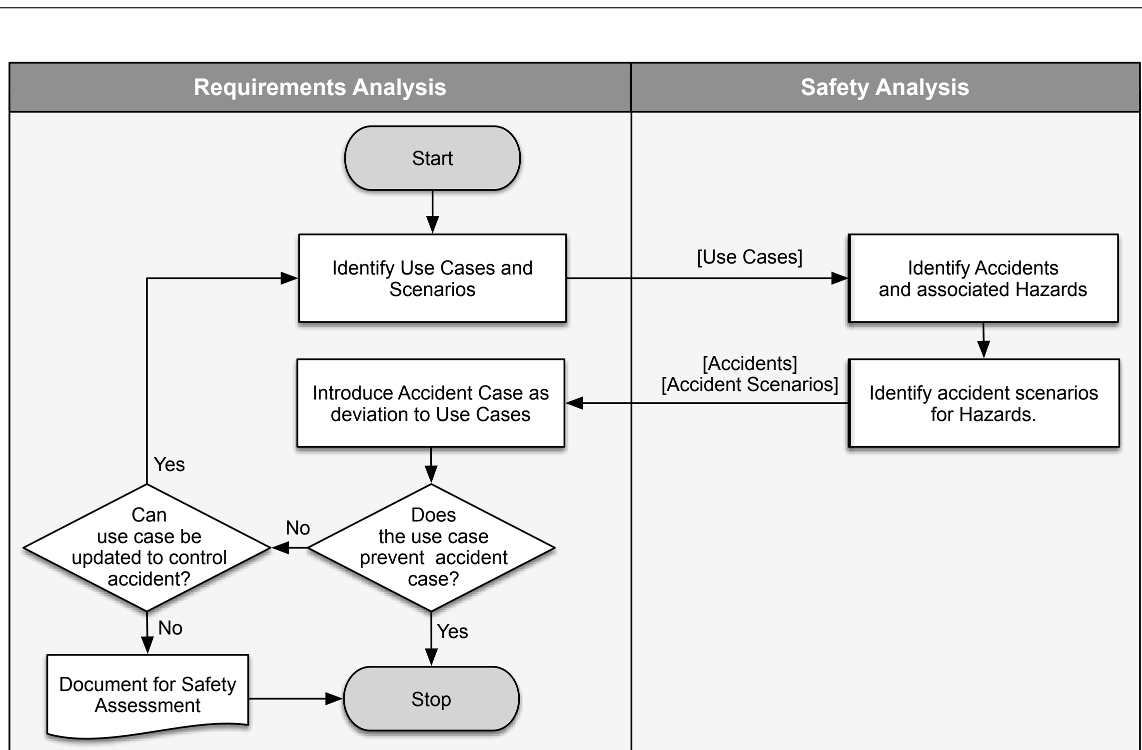


Figure 8.1: Future work: A methodology to bridge requriements and safety analysis process using accident case.

UML Profile

A profile [42] in the UML provides a generic extension mechanism for customizing UML models for particular domains and platforms. Extension mechanisms allow refining standard semantics in strictly additive manner, preventing them from contradicting standard semantics. The future work for UC-B will aim towards create a profile for UML use cases to make the extension with accident cases conform to the UML standard.

Tool Development

The future work aims to investigate the use of the approach in a live development project in order to examine how the tool can take into account the evolution of requirements. In addition, the integration of UC-B with other established UML tools such as Papyrus [68] (eclipse-based UML modelling tool) can help better relate the use case specification from UC-B to other UML diagrams such as sequence diagrams and activity diagrams. The integration of UC-B with Papyrus will aim to keep the UML model (from Papyrus) synchronised with UC-B use case specifications, i.e. creating a UML use case diagram (use cases, actors and subject) in Papyrus would automatically generate the corresponding elements in UC-B. In Papyrus, only the use case diagram would be shown while the content of the use case specification could be managed and enriched with UC-B.

8.4 Concluding Remarks

We have developed an approach that helps to bridge the gap between informal use cases and formal modelling. Moreover, our approach has extended UML use cases with the notion of accident cases with aim of defining system behaviour with context to safety. We believe that this work makes a contribution to a broader goal of making formal methods more accessible to industry. The work presented in this thesis provides a step as part of an on-going effort to help in the industrial adoption of formal methods and of a more specific effort to consider safety concerns.

Appendix **A**

Syntax of Event-B Mathematical Language

This appendix present the syntax of predicates and of expressions used in the mathematical language of Event-B [79].

A.1 Predicate Language

The grammar used for predicates is defined as follows:

⟨*predicate*⟩	::= { ⟨*quantifier*⟩ } ⟨*unquantified-predicate*⟩
⟨*quantifier*⟩	::= ‘$\forall$’ ⟨*ident-list*⟩ ‘$\cdot$’
	| ‘$\exists$’ ⟨*ident-list*⟩ ‘$\cdot$’
⟨*ident-list*⟩	::= ⟨*ident*⟩ { ‘,’ ⟨*ident*⟩ }
⟨*unquantified-predicate*⟩	::= ⟨*simple-predicate*⟩ [‘$\Rightarrow$’ ⟨*simple-predicate*⟩]
	| ⟨*simple-predicate*⟩ [‘$\Leftrightarrow$’ ⟨*simple-predicate*⟩]
⟨*simple-predicate*⟩	::= ⟨*literal-predicate*⟩ { ‘$\wedge$’ ⟨*literal-predicate*⟩ }
	| ⟨*literal-predicate*⟩ { ‘$\vee$’ ⟨*literal-predicate*⟩ }
⟨*literal-predicate*⟩	::= { ‘$\neg$’ ⟨*atomic-predicate*⟩
⟨*atomic-predicate*⟩	::= ‘$\bot$’
	| ‘$\top$’
	| ‘`finite`’ ‘(’ ⟨*expression*⟩ ‘)’
	| ⟨*pair-expression*⟩ ⟨*relop*⟩ ⟨*pair-expression*⟩
	| ‘(’ ⟨*predicate*⟩ ‘)’
⟨*relop*⟩	::= ‘$=$’ | ‘$\neq$’ | ‘$\in$’ | ‘$\notin$’ | ‘$\subset$’ | ‘$\not\subset$’ | ‘$\subseteq$’ | ‘$\not\subseteq$’ | ‘$<$’ | ‘$\leq$’ | ‘$>$’ | ‘$\geq$’

A.2 Expression Language

The grammar used for expressions is defined as follows:

$\langle \textit{expression} \rangle$ ::= $\langle \textit{expression} \rangle$ $\langle \textit{binary-operator} \rangle$ $\langle \textit{expression} \rangle$
| $\langle \textit{unary-operator} \rangle$ $\langle \textit{expression} \rangle$
| $\langle \textit{expression} \rangle$ ‘-1’
| $\langle \textit{expression} \rangle$ ‘`[`’ $\langle \textit{expression} \rangle$ ‘`]`’
| $\langle \textit{expression} \rangle$ ‘`(`’ $\langle \textit{expression} \rangle$ ‘`)`’
| ‘λ’ $\langle \textit{ident-pattern} \rangle$ ‘$\cdot$’ $\langle \textit{predicate} \rangle$ ‘`|`’ $\langle \textit{expression} \rangle$
| $\langle \textit{quantifier} \rangle$ $\langle \textit{ident-list} \rangle$ ‘$\cdot$’ $\langle \textit{predicate} \rangle$ ‘`|`’ $\langle \textit{expression} \rangle$
| $\langle \textit{quantifier} \rangle$ $\langle \textit{expression} \rangle$ ‘$\cdot$’ $\langle \textit{predicate} \rangle$
| ‘$\{$’ $\langle \textit{ident-list} \rangle$ ‘$\cdot$’ $\langle \textit{predicate} \rangle$ ‘`|`’ $\langle \textit{expression} \rangle$ ‘$\}$’
| ‘$\{$’ [$\langle \textit{expression} \rangle$ ‘`|`’ $\langle \textit{predicate} \rangle$ ‘$\}$’
| ‘`bool`’ ‘`(`’ $\langle \textit{predicate} \rangle$ ‘`)`’
| ‘$\{$’ [$\langle \textit{expression-list} \rangle$] ‘$\}$’
| ‘`(`’ $\langle \textit{expression} \rangle$ ‘`)`’
| ‘$\varnothing$’
| ‘$\mathbb{Z}$’ | ‘$\mathbb{N}$’ | ‘$\mathbb{N}_1$’
| ‘`BOOL`’ | ‘`TRUE`’ | ‘`FALSE`’
| $\langle \textit{ident} \rangle$
| $\langle \textit{integer-literal} \rangle$

$\langle \textit{binary-operator} \rangle$ ::= ‘$\leftrightarrow$’ | ‘$\trel$’ | ‘$\srel$’ | ‘$\pfun$’ | ‘$\rightarrow$’ | ‘$\pinj$’ ’ | ‘$\tinj$’ | ‘$\psur$’ | ‘$\tsur$’ | ‘$\tbij$’ | ‘$\cap$’
| ‘$\cup$’ | ‘$\setminus$’ | ‘$\times$’ | ‘$\parallel$’ | ‘$\otimes$’ | ‘;’ | ‘$\ovl$’ | ‘$\circ$’ | ‘$\triangleleft$’ | ‘$\domsub$’ | ‘$\triangleright$’ | ‘$\ransub$’ |
‘$..$’ | ‘$+$’ | ‘$-$’ | ‘$\div$’ | ‘$*$’ | ‘`mod` ’ | ‘$\frown$’

$\langle \textit{unary-operator} \rangle$ ::= ‘$--$’ | ‘`card`’ | ‘$\mathbb{P}$’ | ‘$\mathbb{P}_1$’ | ‘`union`’ | ‘`inter`’ | ‘`dom`’ | ‘`ran`’ |
‘`prj` | ‘`id`’

$\langle \textit{quantifier} \rangle$ ::= ‘$\bigcap$’ | ‘$\bigcup$’

$\langle \textit{ident-pattern} \rangle$ ::= $\langle \textit{ident-pattern} \rangle$ ‘$\mapsto$’ $\langle \textit{ident-pattern} \rangle$
| ‘`(`’ $\langle \textit{ident-pattern} \rangle$ ‘`)`’
| $\langle \textit{ident} \rangle$

$\langle \textit{expression-list} \rangle$ ::= $\langle \textit{expression-list} \rangle$ ‘`,`’ $\langle \textit{expression} \rangle$
| $\langle \textit{expression} \rangle$

Appendix B

Case Studies: Event-B Model

B.1 UC1: Water Tank System

An Event-B Specification of MaintainH_Static

CONTEXT MaintainH_Static
CONSTANTS
$H, HT, LT, L, DEC, INC, DRN$

AXIOMS
`H_Type` : $H > HT$
`HT_Type` : $HT > LT$
`LT_Type` : $LT > L$
`L_Type` : $L = 0$
`DEC_Type` : $DEC \in (H - HT) \mathrel{..} (HT - LT)$
`INV_Type` : $INC \in (LT - L) \mathrel{..} (HT - LT)$
`DRN_Type` : $DRN = L$
END

An Event-B Specification of m0_MaintainH_Contract

MACHINE m0_MaintainH_Contract
SEES MaintainH_Static
VARIABLES
$MH, waterlevel$

INVARIANTS
`waterlevel_Type` : $waterlevel \in \mathbb{N}$
`MH_Inv_1` : $waterlevel \in L \mathrel{..} H$
`MH_Type` : $MH \in BOOL$

EVENTS
Initialisation

begin

waterlevel_Init : $waterlevel := H$

MH_Init : $MH := FALSE$

end

Event $MaintainH \;\widehat{=}$

when

MH_Grd : $MH = FALSE$

MH_Pre_1 : $waterlevel > HT \land waterlevel \leq H$

then

MH_Act : $MH := TRUE$

MH_Post_Act : $waterlevel :| waterlevel' \geq L \land waterlevel' \leq HT$

end

END

An Event-B Specification of MaintainH_Flow

CONTEXT MaintainH_Flow

SETS

MH_FLOW

CONSTANTS

$MH_Initial, MH_Trigger, MH_1, MH_2, MH_3, MH_4, MH_Final, EH_1, EH_2$

AXIOMS

MH_FLOW_Type : $partition(MH_FLOW, \{MH_Initial\}, \{MH_Trigger\}, \{MH_1\}, \{MH_2\}, \{MH_3\}, \{MH_4\}, \{EH_1\}, \{EH_2\}, \{MH_Final\})$

END

An Event-B Specification of m1_MaintainH_Scenario

MACHINE m1_MaintainH_Scenario

REFINES m0_MaintainH_Contract

SEES MaintainH_Static, MaintainH_Flow

VARIABLES

$MH, DL, MH_flow, waterlevel, waterlevel_m1, sensorHT, pump, motor$

INVARIANTS

MH_flow_Type : $MH_flow \in MH_FLOW$

DL_Type : $DL \in BOOL$

sensorHT_Type : $sensorHT \in BOOL$

pump_Type : $pump \in BOOL$

motor_Type : $motor \in BOOL$

MH_Glue_Variables : $MH_flow = MH_Trigger \lor (MH_flow = MH_Final \land MH = TRUE) \Rightarrow (waterlevel_m1 = waterlevel)$

MH_Glue_Flow : $MH_flow \in MH_FLOW \setminus \{MH_Initial, MH_Final\} \Rightarrow MH = FALSE$

MH_Scenario_Pre : $MH_flow \in MH_FLOW \setminus \{MH_Initial\} \land MH = FALSE \Rightarrow (waterlevel > HT \land waterlevel \leq H)$

MH_Scenario_Inv : $waterlevel_m1 \in L \,..\, H$

MH_Scenario_Post : $MH_flow = MH_Final \Rightarrow waterlevel_m1 \geq L \land waterlevel_m1 \leq HT$

DL_Prevent : $\neg(DL = FALSE \land MH_flow = EH_2 \land \neg(pump = FALSE \land motor = TRUE))$

MH_StepAssert_1 : $MH_flow \in \{MH_1, MH_2, MH_3, MH_4\} \Rightarrow (waterlevel_m1 > HT)$

MH_StepAssert_2 : $(MH_flow \in \{MH_3, EH_1\} \Rightarrow pump = FALSE)$

MH_StepAssert_3 : $(MH_flow = EH_2 \land DL = TRUE \Rightarrow waterlevel_m1 = L)$

EVENTS
Initialisation
extended

begin
waterlevel_Init : $waterlevel := H$
MH_Init : $MH := FALSE$
MH_Flow_Init : $MH_flow := MH_Initial$
DL_Init : $DL := FALSE$
waterlevel_m1_Init : $waterlevel_m1 := H$
motor_Init : $motor := TRUE$
pump_Init : $pump := TRUE$
sensorHT_Init : $sensorHT := FALSE$
end

Event *MaintainH_Initial* $\widehat{=}$
when
MH_Flow_Grd : $MH_flow = MH_Initial$
MH_Pre_1 : $waterlevel > HT$
MH_Grd : $MH = FALSE$
then
MH_Flow_Act : $MH_flow := MH_Trigger$
waterlevel_Equal : $waterlevel_m1 := waterlevel$
end

Event *MaintainH_Trigger* $\widehat{=}$
when
MH_Flow_Grd : $MH_flow = MH_Trigger$
MH_Trig_1 : $waterlevel_m1 > HT$
then
MH_Flow_Act : $MH_flow := MH_1$
end

Event *MH_1* $\widehat{=}$
when
MH_Flow_Grd : $MH_flow = MH_1$
then
MH_Flow_Act : $MH_flow := MH_2$
MH_1_Act : $sensorHT := TRUE$
end

Event *MH_2* $\widehat{=}$
when
MH_Flow_Grd : $MH_flow = MH_2$
then
MH_Flow_Act : $MH_flow := MH_3$
MH_2_Act : $pump := FALSE$
end

Event *MH_3* $\widehat{=}$
when
MH_Flow_Grd : $MH_flow = MH_3$

then
    `MH_Flow_Act` : $MH_flow := MH_4$
    `MH_3_Act` : $motor := FALSE$
end

Event *MH_4* $\widehat{=}$
when
    `MH_Flow_Grd` : $MH_flow = MH_4$
then
    `MH_Flow_Act` : $MH_flow := MH_Final$
    `MH_3_Act` : $waterlevel_m1 := waterlevel_m1 - DEC$
end

Event *ExceedH_Trigger* $\widehat{=}$
when
    `EH_DeviationPoint` : $MH_flow = MH_3$
    `EH_Trig_1` : $waterlevel_m1 > HT$
then
    `MH_Flow_Act` : $MH_flow := EH_1$
end

Event *EH_1* $\widehat{=}$
when
    `MH_Flow_Grd` : $MH_flow = EH_1$
then
    `MH_Flow_Act` : $MH_flow := EH_2$
    `DL_Act` : $DL := FALSE$
    `EH_1_Act` : $motor := TRUE$
end

Event *DrainToL* $\widehat{=}$
when
    `DL_Grd` : $DL = FALSE$
    `DL_Pre_1` : $pump = FALSE \wedge motor = TRUE$
    `DL_ExtensionPoint` : $MH_flow = EH_2$
then
    `DL_RejoinPoint` : $MH_flow := EH_2$
    `DL_Act` : $DL := TRUE$
    `DL_Post_Act` : $waterlevel_m1 :| (waterlevel_m1' = L)$
end

Event *DrainToL_False* $\widehat{=}$
when
    `DL_Grd` : $DL = FALSE$
    `DL_Pre_1_Neg` : $\neg(pump = FALSE \wedge motor = TRUE)$
    `DL_ExtensionPoint` : $MH_flow = EH_2$
then
    `DL_Act` : $DL := TRUE$
end

Event *EH_2* $\widehat{=}$
when
    `MH_Flow_Grd` : $MH_flow = EH_2$
    `DL_Grd` : $DL = TRUE$
then
    `MH_Flow_Act` : $MH_flow := MH_Final$
    `EH_2_Act` : $waterlevel_m1 := waterlevel_m1 + INC$

end

Event $MaintainH_Final \widehat{=}$
refines $MaintainH$

when

MH_Flow_Grd : $MH_flow = MH_Final$

MH_Grd : $MH = FALSE$

then

MH_Act : $MH := TRUE$

waterlevel_Equal : $waterlevel := waterlevel_m1$

end

END

An Event-B Specification of DrainToL_Flow

CONTEXT DrainToL_Flow
SETS

DL_FLOW

CONSTANTS

$DL_Initial, DL_Trigger, DL_1, DL_2, DL_3, DL_Final$

AXIOMS

DL_FLOW_Type : $partition(DL_FLOW, \{DL_Initial\}, \{DL_Trigger\}, \{DL_1\}, \{DL_2\}, \{DL_3\}, \{DL_Final\})$

END

An Event-B Specification of m2_DrainToL_Scenario

MACHINE m2_DrainToL_Scenario
REFINES m1_MaintainH_Scenario
SEES MaintainH_Static, MaintainH_Flow, DrainToL_Flow
VARIABLES

$MH, DL, MH_flow, DL_flow, waterlevel, waterlevel_m1, pump,$

$sensorHT, motor, waterlevel_m2, pump_m2, motor_m2, drain, valve$

INVARIANTS

DL_flow_Type : $DL_flow \in DL_FLOW$

pump_m2_Type : $pump_m2 \in BOOL$

motor_m2_Type : $motor_m2 \in BOOL$

valve_Type : $valve \in BOOL$

drain_Type : $drain \in BOOL$

DL_Glue_Variables : $DL_flow = DL_Trigger \vee (DL_flow = DL_Final \wedge DL = TRUE \wedge MH_flow = EH_2) \Rightarrow waterlevel_m2 = waterlevel_m1$

DL_Glue_Flow : $DL_flow \in DL_FLOW \setminus \{DL_Initial, DL_Final\} \Rightarrow DL = FALSE \wedge MH_flow = EH_2$

DL_Scenario_Pre : $DL_flow \in DL_FLOW \setminus \{DL_Initial\} \wedge DL = FALSE \wedge MH_flow = EH_2 \Rightarrow pump = FALSE \wedge motor = TRUE$

DL_Scenario_Inv : $waterlevel_m2 \in L \,..\, H$

DL_Scenario_Post : $DL_flow = DL_Final \Rightarrow waterlevel_m2 = L$

EVENTS

Initialisation
extended

begin
waterlevel_Init : $waterlevel := H$
MH_Init : $MH := FALSE$
MH_Flow_Init : $MH_flow := MH_Initial$
DL_Init : $DL := FALSE$
waterlevel_m1_Init : $waterlevel_m1 := H$
motor_Init : $motor := TRUE$
pump_Init : $pump := TRUE$
sensorHT_Init : $sensorHT := FALSE$
DL_flow_Init : $DL_flow := DL_Initial$
waterlevel_m2_Init : $waterlevel_m2 := H$
drain_Init : $drain := FALSE$
valve_Init : $valve := FALSE$
pump_Init_m2 : $pump_m2 := TRUE$
motor_Init_m2 : $motor_m2 := TRUE$
end

Event *MaintainH_Initial* $\widehat{=}$
extends *MaintainH_Initial*
when
MH_Flow_Grd : $MH_flow = MH_Initial$
MH_Pre_1 : $waterlevel > HT$
MH_Grd : $MH = FALSE$
then
MH_Flow_Act : $MH_flow := MH_Trigger$
waterlevel_Equal : $waterlevel_m1 := waterlevel$
end

Event *MaintainH_Trigger* $\widehat{=}$
extends *MaintainH_Trigger*
when
MH_Flow_Grd : $MH_flow = MH_Trigger$
MH_Trig_1 : $waterlevel_m1 > HT$
then
MH_Flow_Act : $MH_flow := MH_1$
end

Event *MH_1* $\widehat{=}$
extends *MH_1*
when
MH_Flow_Grd : $MH_flow = MH_1$
then
MH_Flow_Act : $MH_flow := MH_2$
MH_1_Act : $sensorHT := TRUE$
end

Event *MH_2* $\widehat{=}$
extends *MH_2*
when
MH_Flow_Grd : $MH_flow = MH_2$
then
MH_Flow_Act : $MH_flow := MH_3$
MH_2_Act : $pump := FALSE$

end

Event *MH_3* $\widehat{=}$
extends *MH_3*

when

`MH_Flow_Grd` : $MH_flow = MH_3$

then

`MH_Flow_Act` : $MH_flow := MH_4$

`MH_3_Act` : $motor := FALSE$

end

Event *MH_4* $\widehat{=}$
extends *MH_4*

when

`MH_Flow_Grd` : $MH_flow = MH_4$

then

`MH_Flow_Act` : $MH_flow := MH_Final$

`MH_3_Act` : $waterlevel_m1 := waterlevel_m1 - DEC$

end

Event *ExceedH_Trigger* $\widehat{=}$
extends *ExceedH_Trigger*

when

`MH_Flow_Grd` : $MH_flow = MH_3$

`EH_Trig_1` : $waterlevel_m1 > HT$

then

`MH_Flow_Act` : $MH_flow := EH_1$

end

Event *EH_1* $\widehat{=}$
extends *EH_1*

when

`MH_Flow_Grd` : $MH_flow = EH_1$

then

`MH_Flow_Act` : $MH_flow := EH_2$

`DL_Act` : $DL := FALSE$

`EH_1_Act` : $motor := TRUE$

end

Event *DrainToL_Init* $\widehat{=}$

when

`DL_Grd` : $DL = FALSE$

`DL_ExtPoint` : $MH_flow = EH_2$

`DL_Pre_1` : $pump = FALSE \wedge motor = TRUE$

`MP_Flow_Grd` : $DL_flow = DL_Initial$

then

`MP_Flow_Act` : $DL_flow := DL_Trigger$

`waterlevel_Equal` : $waterlevel_m2 := waterlevel_m1$

`pump_Equal` : $pump_m2 := pump$

`motor_Equal` : $motor_m2 := motor$

end

Event *DrainToL_Trigger* $\widehat{=}$

when

`DL_Flow_Grd` : $DL_flow = DL_Trigger$

`DL_Trig_1` : $motor_m2 = TRUE \wedge pump_m2 = FALSE$

then

DL_Flow_Act : $DL_flow := DL_1$

end

Event *DL_1* ≙
when
DL_Flow_Grd : $DL_flow = DL_1$

then
DL_Flow_Act : $DL_flow := DL_2$
DL_1_Act : $drain := TRUE$

end

Event *DL_2* ≙
when
DL_Flow_Grd : $DL_flow = DL_2$

then
DL_Flow_Act : $DL_flow := DL_3$
DL_2_Act : $valve := TRUE$

end

Event *DL_3* ≙
when
DL_Flow_Grd : $DL_flow = DL_3$

then
DL_Flow_Act : $DL_flow := DL_Final$
DL_3_Act : $waterlevel_m2 := DRN$

end

Event *DrainToL_Final* ≙
refines *DrainToL*
when
DL_Flow_Grd : $DL_flow = DL_Final$
DL_Grd : $DL = FALSE$
DL_ExtPoint : $MH_flow = EH_2$

then
DL_Act : $DL := TRUE$
DL_RejPoint : $MH_flow := EH_2$
waterlevel_Equal : $waterlevel_m1 := waterlevel_m2$

end

Event *DrainToL_False* ≙
extends *DrainToL_False*
when
DL_Grd : $DL = FALSE$
DL_Pre_1_Neg : $\neg(pump = FALSE \land motor = TRUE)$
DL_ExtPoint : $MH_flow = EH_2$

then
DL_Act : $DL := TRUE$

end

Event *EH_2* ≙
extends *EH_2*
when
MH_Flow_Grd : $MH_flow = EH_2$
DL_Grd : $DL = TRUE$

then
MH_Flow_Act : $MH_flow := MH_Final$
EH_2_Act : $waterlevel_m1 := waterlevel_m1 + INC$

end

Event $MaintainH_Final \widehat{=}$
extends $MaintainH_Final$

when

`MH_Flow_Grd` : $MH_flow = MH_Final$

`MH_Grd` : $MH = FALSE$

then

`MH_Act` : $MH := TRUE$

`waterlevel_Equal` : $waterlevel := waterlevel_m1$

end

END

B.2 UC2: Train Door Control System

An Event-B Specification of OpenDoor_Static

CONTEXT OpenDoor_Static
SETS

$DOOR, SPEED, LOCATION$

CONSTANTS

$Open, Opening, Closed, Moving, Stationary$

AXIOMS

`DOOR_Enum` : $partition(DOOR, \{Open\}, \{Opening\}, \{Closed\})$

`SPEED_Enum` : $partition(SPEED, \{Stationary\}, \{Moving\})$

END

An Event-B Specification of m0_OpenDoor_Contract

MACHINE m0_OpenDoor_Contract
SEES OpenDoor_Static
VARIABLES

OD

$door$

t_speed

INVARIANTS

`OD_Type` : $OD \in BOOL$

`door_Type` : $door \in DOOR$

`OD_Inv_1` : $\neg(door = Open \land t_speed = Moving)$

`t_speed_Type` : $t_speed \in SPEED$

EVENTS
Initialisation

begin

`door_Init` : $door := Closed$

`OD_Init` : $OD := FALSE$

`t_speed_Init` : $t_speed := Stationary$

end

Event $OpenDoor \widehat{=}$

when

`OD_Grd` : $OD = FALSE$

`OD_Pre_1` : $door = Closed$

then

`OD_Act` : $OD := TRUE$

`OD_Post_Act` : $door, t_speed$

$:|$

$(t_speed' = Stationary \wedge door' = Open)$

$\vee$

$(t_speed' = Moving \wedge door' = Closed)$

end

END

An Event-B Specification of OpenDoor_Flow

CONTEXT OpenDoor_Flow

SETS

OD_FLOW

CONSTANTS

$OD_Initial, OD_1, OD_1_1, OD_1_2, OD_1_3, OD_Trigger, OD_Final, PT_1, PT_2$

AXIOMS

`OD_FLOW_Type` : $partition(OD_FLOW, \{OD_Initial\}, \{OD_Trigger\}, \{OD_1\}, \{OD_1_1\}, \{OD_1_2\}, \{OD_1_3\}, \{PT_1\}, \{PT_2\}, \{OD_Final\})$

END

An Event-B Specification of m1_OpenDoor_Scenario

MACHINE m1_OpenDoor_Scenario

REFINES m0_OpenDoor_Contract

SEES OpenDoor_Static, OpenDoor_Flow

VARIABLES

$OD, OD_flow, door, door_m1, request_door, door_cmd, t_speed, t_speed_m1, EB$

INVARIANTS

`OD_flow_Type` : $OD_flow \in OD_FLOW$

`operator_request_Type` : $request_door \in BOOL$

`door_cmd_Type` : $door_cmd \in BOOL$

`door_m1_Type` : $door_m1 \in DOOR$

`t_speed_m1_Type` : $t_speed_m1 \in SPEED$

`OD_Glue_Variables` : $OD_flow = OD_Trigger \vee (OD_flow = OD_Final \wedge OD = TRUE) \Rightarrow door = door_m1$

`OD_Glue_Flow` : $OD_flow \in OD_FLOW \setminus \{OD_Initial, OD_Final\} \Rightarrow OD = FALSE$

`OD_Scenario_Pre` : $OD_flow \in OD_FLOW \setminus \{OD_Initial\} \wedge OD = FALSE \Rightarrow door = Closed$

`OD_Scenario_Inv` : $\neg(door_m1 = Open \wedge t_speed_m1 = Moving)$

`OD_Scenario_Post` : $OD_flow = OD_Final \Rightarrow (t_speed_m1 = Stationary \wedge door_m1 = Open) \vee (t_speed_m1 = Moving \wedge door_m1 = Closed)$

`EB_Type` : $EB \in BOOL$

OD_Prevent : $\neg((EB = FALSE) \wedge (OD_flow = PT_2) \wedge \neg$
$(door_m1 = Opening \wedge t_speed_m1 = Moving))$

OD_StepAssert_1 : $OD_flow \in \{OD_1\} \Rightarrow door_m1 = Closed$

OD_StepAssert_2 : $OD_flow \in \{OD_1_1, OD_1_2, OD_1_3\} \Rightarrow t_speed_m1 = Stationary$

OD_StepAssert_3 : $OD_flow = OD_Initial \Rightarrow t_speed = t_speed_m1$

OD_StepAssert_4 : $OD_flow = PT_2 \wedge EB = TRUE \Rightarrow t_speed_m1 = Stationary$

OD_StepAssert_5 : $OD_flow = PT_1 \Rightarrow t_speed_m1 = Moving$

EVENTS

Initialisation

extended

begin

door_Init : $door := Closed$

OD_Init : $OD := FALSE$

t_speed_Init : $t_speed := Stationary$

OD_flow_Init : $OD_flow := OD_Initial$

door_m1_Init : $door_m1 := Closed$

operator_request_Init : $request_door := TRUE$

door_cmd_Init : $door_cmd := FALSE$

t_speed_m1_Init : $t_speed_m1 := Stationary$

EB_Init : $EB := FALSE$

end

Event $OpenDoor_Initial \widehat{=}$

when

OD_flow_Grd : $OD_flow = OD_Initial$

OD_Grd : $OD = FALSE$

OD_Pre_1 : $door = Closed$

then

OD_flow_Act : $OD_flow := OD_Trigger$

door_m1_Equal : $door_m1 := door$

t_speed_m1_Equal : $t_speed_m1 := t_speed$

end

Event $OpenDoor_Trigger \widehat{=}$

when

OD_flow_Grd : $OD_flow = OD_Trigger$

OD_Trig_1 : $request_door = TRUE \wedge door_m1 = Closed$

then

OD_flow_Act : $OD_flow := OD_1$

end

Event $OD_1_If \widehat{=}$

when

OD_flow_Grd : $OD_flow = OD_1$

OD_3_Grd : $t_speed_m1 = Stationary$

then

OD_flow_Act : $OD_flow := OD_1_1$

end

Event $OD_1_If_False \widehat{=}$

when

OD_flow_Grd : $OD_flow = OD_1$

`OD_3_Grd_Neg` : $\neg(t_speed_m1 = Stationary)$

then

`OD_flow_Act` : $OD_flow := OD_Final$

end

Event *OD_1_1* ≙

when

`OD_flow_Grd` : $OD_flow = OD_1_1$

then

`OD_flow_Act` : $OD_flow := OD_1_2$

`OD_2_1_Act` : $door_cmd := TRUE$

end

Event *OD_1_2* ≙

when

`OD_flow_Grd` : $OD_flow = OD_1_2$

then

`OD_flow_Act` : $OD_flow := OD_1_3$

`OD_2_2_Act` : $door_m1 := Opening$

end

Event *OD_1_3* ≙

when

`OD_flow_Grd` : $OD_flow = OD_1_3$

then

`OD_flow_Act` : $OD_flow := OD_Final$

`OD_2_2_Act` : $door_m1 := Open$

end

Event *OpenDoor_Final* ≙

refines *OpenDoor*

when

`OD_flow_Grd` : $OD_flow = OD_Final$

`OD_Grd` : $OD = FALSE$

then

`OD_Act` : $OD := TRUE$

`door_Equal` : $door := door_m1$

`t_speed_Equal` : $t_speed := t_speed_m1$

end

Event *PT_Trigger* ≙

when

`OD_flow_Grd` : $OD_flow = OD_1$

`DeviationPoint` : $t_speed_m1 = Moving$

then

`OD_flow_Act` : $OD_flow := PT_1$

end

Event *PT_1* ≙

when

`OD_flow_Grd` : $OD_flow = PT_1$

then

`OD_flow_Act` : $OD_flow := PT_2$

`PT_1_Act` : $door_m1 := Opening$

`EB_Act` : $EB := FALSE$

end

Event *EmergencyBraking* $\widehat{=}$
when
EB_Grd : $EB = FALSE$
EB_ExtesnionPoint : $OD_flow = PT_2$
EB_Pre_1 : $(door_m1 = Opening \land t_speed_m1 = Moving)$
then
EB_Post_Act : $t_speed_m1 :| t_speed_m1' = Stationary$
EB_Act : $EB := TRUE$
EB_RejoinPoint : $OD_flow := PT_2$
end
Event *EmergencyBraking_FALSE* $\widehat{=}$
when
EB_Grd : $EB = FALSE$
EB_ExtesnionPoint : $OD_flow = PT_2$
EB_Pre_1_Negate : $\neg(door_m1 = Opening \land t_speed_m1 = Moving)$
then
EB_Act : $EB := TRUE$
end
Event *PT_2* $\widehat{=}$
when
OD_flow_Grd : $OD_flow = PT_2$
EB_Grd : $EB = TRUE$
then
OD_flow_Act : $OD_flow := OD_Final$
PT_1_Act : $door_m1 := Open$
end
END

An Event-B Specification of EmergencyBraking_Flow

CONTEXT EmergencyBraking_Flow
EXTENDS OpenDoor_Flow
SETS
EB_FLOW

CONSTANTS
$EB_Initial, EB_Trigger, EB_1, EB_2, EB_3, EB_Final$

AXIOMS
EB_FLOW_Type : $partition(EB_FLOW, \{EB_Initial\}, \{EB_Trigger\}, \{EB_1\}, \{EB_2\}, \{EB_3\}, \{EB_Final\})$
END

An Event-B Specification of m2_EmergencyBraking_Scenario

MACHINE m2_EmergencyBraking_Scenario
REFINES m1_OpenDoor_Scenario
SEES OpenDoor_Static, EmergencyBraking_Flow
VARIABLES
$OD, OD_flow, door, door_m1, request_door, door_cmd, t_speed, EB_flow, brake, t_speed_m1, t_speed_m2, EB$

INVARIANTS

EB_flow_Type : $EB_flow \in EB_FLOW$

t_speed_m2_Type : $t_speed_m2 \in SPEED$

brake_Type : $brake \in BOOL$

EB_Scenario_Pre : $EB_flow \in EB_FLOW \setminus \{EB_Initial\} \wedge$
$EB = FALSE \wedge OD_flow = PT_2 \Rightarrow (door_m1 = Opening \wedge t_speed_m1 = Moving)$

EB_Scenario_Post : $EB_flow = EB_Final \Rightarrow (t_speed_m2 = Stationary)$

EB_Scenario_Inv : $EB_flow \in EB_FLOW \setminus \{EB_Initial, EB_Final\} \Rightarrow$
$\neg(door_m1 = Open \wedge t_speed_m2 = Moving)$

EB_Glue_Variables : $(EB_flow = EB_Trigger \wedge OD_flow = PT_2) \vee$
$(EB_flow = EB_Final \wedge EB = TRUE \wedge OD_flow = PT_2) \Rightarrow (t_speed_m2 = t_speed_m1)$

EB_Glue_Flow : $EB_flow \in EB_FLOW \setminus \{EB_Initial, EB_Final\} \Rightarrow$
$EB = FALSE \wedge OD_flow = PT_2$

EB_StepAssert_1 : $EB_flow = EB_3 \Rightarrow t_speed_m2 = Stationary$

EVENTS

Initialisation

extended

begin

door_Init : $door := Closed$

OD_Init : $OD := FALSE$

t_speed_Init : $t_speed := Stationary$

OD_flow_Init : $OD_flow := OD_Initial$

door_m1_Init : $door_m1 := Closed$

operator_request_Init : $request_door := TRUE$

door_cmd_Init : $door_cmd := FALSE$

t_speed_m1_Init : $t_speed_m1 := Stationary$

EB_Init : $EB := FALSE$

EB_flow_Init : $EB_flow := EB_Initial$

t_speed_m2_Init : $t_speed_m2 := Stationary$

brake_Init : $brake := FALSE$

end

Event *OpenDoor_Initial* $\widehat{=}$

extends *OpenDoor_Initial*

when

OD_flow_Grd : $OD_flow = OD_Initial$

OD_Grd : $OD = FALSE$

OD_Pre_1 : $door = Closed$

then

OD_flow_Act : $OD_flow := OD_Trigger$

door_m1_Equal : $door_m1 := door$

t_speed_m1_Equal : $t_speed_m1 := t_speed$

end

Event *OpenDoor_Trigger* $\widehat{=}$

extends *OpenDoor_Trigger*

when

OD_flow_Grd : $OD_flow = OD_Trigger$

OD_Trig_1 : $request_door = TRUE \wedge door_m1 = Closed$

then

OD_flow_Act : $OD_flow := OD_1$

end

Event $OD_1_If \widehat{=}$
extends OD_1_If
when
`OD_flow_Grd` : $OD_flow = OD_1$
`OD_3_Grd` : $t_speed_m1 = Stationary$
then
`OD_flow_Act` : $OD_flow := OD_1_1$
end

Event $OD_1_If_False \widehat{=}$
extends $OD_1_If_False$
when
`OD_flow_Grd` : $OD_flow = OD_1$
`OD_3_Grd_Neg` : $\neg(t_speed_m1 = Stationary)$
then
`OD_flow_Act` : $OD_flow := OD_Final$
end

Event $OD_1_1 \widehat{=}$
extends OD_1_1
when
`OD_flow_Grd` : $OD_flow = OD_1_1$
then
`OD_flow_Act` : $OD_flow := OD_1_2$
`OD_2_1_Act` : $door_cmd := TRUE$
end

Event $OD_1_2 \widehat{=}$
extends OD_1_2
when
`OD_flow_Grd` : $OD_flow = OD_1_2$
then
`OD_flow_Act` : $OD_flow := OD_1_3$
`OD_2_2_Act` : $door_m1 := Opening$
end

Event $OD_1_3 \widehat{=}$
extends OD_1_3
when
`OD_flow_Grd` : $OD_flow = OD_1_3$
then
`OD_flow_Act` : $OD_flow := OD_Final$
`OD_2_2_Act` : $door_m1 := Open$
end

Event $OpenDoor_Final \widehat{=}$
extends $OpenDoor_Final$
when
`OD_flow_Grd` : $OD_flow = OD_Final$
`OD_Grd` : $OD = FALSE$
then
`OD_Act` : $OD := TRUE$
`door_Equal` : $door := door_m1$
`t_speed_Equal` : $t_speed := t_speed_m1$
end

Event $PT_Trigger \widehat{=}$

extends *PT_Trigger*

when

`OD_flow_Grd` : $OD_flow = OD_1$

`DeviationPoint` : $t_speed_m1 = Moving$

then

`OD_flow_Act` : $OD_flow := PT_1$

end

Event *PT_1* ≙

extends *PT_1*

when

`OD_flow_Grd` : $OD_flow = PT_1$

then

`OD_flow_Act` : $OD_flow := PT_2$

`PT_1_Act` : $door_m1 := Opening$

`EB_Act` : $EB := FALSE$

end

Event *EB_Initial* ≙

when

`EB_flow_Grd` : $EB_flow = EB_Initial$

`EB_Grd` : $EB = FALSE$

`EB_ExtensionPoint` : $OD_flow = PT_2$

`EB_Pre_1` : $(door_m1 = Opening \land t_speed_m1 = Moving)$

then

`EB_flow_Act` : $EB_flow := EB_Trigger$

`t_speed_m2_Equal` : $t_speed_m2 := t_speed_m1$

end

Event *EB_Trigger* ≙

when

`EB_flow_Grd` : $EB_flow = EB_Trigger$

then

`EB_flow_Act` : $EB_flow := EB_1$

end

Event *EB_1* ≙

when

`EB_flow_Grd` : $EB_flow = EB_1$

then

`EB_flow_Act` : $EB_flow := EB_2$

`EB_1_Act` : $brake := TRUE$

end

Event *EB_2* ≙

when

`EB_flow_Grd` : $EB_flow = EB_2$

then

`EB_flow_Act` : $EB_flow := EB_Final$

`EB_2_Act` : $t_speed_m2 := Stationary$

end

Event *EB_Final* ≙

refines *EmergencyBraking*

when

`EB_Grd` : $EB = FALSE$

`EB_flow_Grd` : $EB_flow = EB_Final$

EB_ExtensionPoint : $OD_flow = PT_2$

then

EB_Act : $EB := TRUE$

t_speed_m1_Equal : $t_speed_m1 := t_speed_m2$

EB_RejoinPoint : $OD_flow := PT_2$

end

Event *EmergencyBraking_FALSE* $\widehat{=}$
extends *EmergencyBraking_FALSE*

when

EB_Grd : $EB = FALSE$

EB_ExtesnionPoint : $OD_flow = PT_2$

EB_Pre_1_Negate : $\neg(door_m1 = Opening \wedge t_speed_m1 = Moving)$

then

EB_Act : $EB := TRUE$

end

Event *PT_2* $\widehat{=}$
extends *PT_2*

when

OD_flow_Grd : $OD_flow = PT_2$

EB_Grd : $EB = TRUE$

then

OD_flow_Act : $OD_flow := OD_Final$

PT_1_Act : $door_m1 := Open$

end

END

B.3 UC3: Automated Teller Machine

An Event-B Specification of Withdraw_Static

CONTEXT Withdraw_Static
SETS

$PIN, REQUEST, CARD$

CONSTANTS

$b_AccountBalance, b_AccountPIN, Request_PIN,$

$Request_Withdrawal, Request_Card, Inserted, Returned, Withheld$

AXIOMS

DISPLAY_Enum : $partition(REQUEST, \{Request_Card\}, \{Request_PIN\}, \{Request_Withdrawal\})$

CARD_Enum : $partition(CARD, \{Inserted\}, \{Returned\}, \{Withheld\})$

b_AccountBalance_Type : $b_AccountBalance \geq 0 \wedge b_AccountBalance = 1$

bank_AccountPIN_Type : $b_AccountPIN \in PIN$

END

An Event-B Specification of m0_Withdraw_Contract

MACHINE m0_Withdraw_Contract
SEES Withdraw_Static

VARIABLES

$W, c_Card, atm_Dispense$

INVARIANTS

`W_Type` : $W \in BOOL$

`c_Card_Type` : $c_Card \in CARD$

`atm_Dispense_Type` : $atm_Dispense \in \mathbb{N}$

`W_Inv_1` : $(atm_Dispense = 0 \wedge c_Card \in \{Inserted, Withheld\}) \vee$
$(c_Card = Returned \wedge atm_Dispense \in 0 \,..\, b_AccountBalance)$

EVENTS

Initialisation

begin

`W_Init` : $W := FALSE$

`c_Card_Init` : $c_Card := Inserted$

`atm_Dispense_Init` : $atm_Dispense := 0$

end

Event *Withdraw* $\widehat{=}$

when

`W_Grd` : $W = FALSE$

`W_Pre_1` : $c_Card = Inserted$

then

`W_Act` : $W := TRUE$

`W_Post_Act` : $atm_Dispense, c_Card :|$
$(c_Card' = Returned \wedge atm_Dispense' \in 0 \,..\, b_AccountBalance) \vee$
$(c_Card' = Withheld \wedge atm_Dispense' = 0)$

end

END

An Event-B Specification of Withdraw_Flow

CONTEXT Withdraw_Flow

SETS

W_FLOW

CONSTANTS

$W_Initial, W_Trigger, W_1, W_2, W_3, W_4, W_5, W_6,$

W_A_1, W_A_2, W_Final

AXIOMS

`W_FLOW_Type` : $partition(W_FLOW, \{W_Initial\}, \{W_Trigger\}, \{W_1\}, \{W_2\},$
$\{W_3\}, \{W_4\}, \{W_5\}, \{W_6\}, \{W_A_1\}, \{W_A_2\}, \{W_Final\})$

END

An Event-B Specification of m1_Withdraw_Scenario

MACHINE m1_Withdraw_Scenario

REFINES m0_Withdraw_Contract

SEES Withdraw_Static, Withdraw_Flow

VARIABLES

W

BC

W_flow

$atm_Dispense$

$atm_Request$

c_PIN

$c_WithdrawRequest$

c_Card

$b_FailPINAttempt$

$atm_Dispense_m1$

c_Card_m1

INVARIANTS

`W_flow_Type` : $W_flow \in W_FLOW$

`BC_Type` : $BC \in BOOL$

`atm_Request_Type` : $atm_Request \in REQUEST$

`atm_Dispense_m1_Type` : $atm_Dispense_m1 \in \mathbb{N}$

`c_PIN_Type` : $c_PIN \in PIN$

`c_WithdrawRequest_Type` : $c_WithdrawRequest \in \mathbb{N}$

`c_Card_m1_Type` : $c_Card_m1 \in CARD$

`b_FailPINAttempt_Type` : $b_FailPINAttempt \in 0 \,..\, 3$

`W_Manual_1` : $(W_flow \in \{W_5, W_6\} \Rightarrow c_WithdrawRequest = b_AccountBalance)$
$\wedge$
$(W_flow \in \{W_6\} \Rightarrow c_Card_m1 = Returned)$

`W_StepAssert_1` : $W_flow = W_A_2 \Rightarrow \neg(b_FailPINAttempt > 2)$

`W_Manual_2_1` : $W_flow = W_2 \Rightarrow \neg(b_FailPINAttempt > 2)$

`Withdraw_Manual_3` : $BC = TRUE \wedge W_flow = W_1 \Rightarrow \neg(b_FailPINAttempt > 2)$

`W_Glue_Variables` : $W_flow = W_Trigger \vee (W_flow = W_Final \wedge W = TRUE) \Rightarrow$
$(c_Card_m1 = c_Card) \wedge (atm_Dispense_m1 = atm_Dispense)$

`W_Glue_Flow` : $W_flow \in W_FLOW \setminus \{W_Initial, W_Final\} \Rightarrow W = FALSE$

`W_Scenario_Pre` : $W_flow \in W_FLOW \setminus \{W_Initial\} \wedge W = FALSE \Rightarrow c_Card = Inserted$

`W_Scenario_Post` : $W_flow = W_Final \Rightarrow$
$(c_Card_m1 = Returned \wedge atm_Dispense_m1 \in 0 \,..\, b_AccountBalance) \vee$
$(c_Card_m1 = Withheld \wedge atm_Dispense_m1 = 0)$

`W_Scenario_Inv` : $(atm_Dispense_m1 = 0 \wedge c_Card_m1 \in \{Inserted, Withheld\}) \vee$
$(c_Card_m1 = Returned \wedge atm_Dispense_m1 \in 0 \,..\, b_AccountBalance)$

`W_Step_Aeert_1` : $W_flow \in \{W_1, W_2, W_A_2\} \Rightarrow atm_Dispense_m1 = 0$

EVENTS

Initialisation

extended

begin

`W_Init` : $W := FALSE$

`c_Card_Init` : $c_Card := Inserted$

`atm_Dispense_Init` : $atm_Dispense := 0$

`W_flow_Init` : $W_flow := W_Initial$

`atm_Request_Init` : $atm_Request := Request_Card$

`atm_Dispense_m1_Init` : $atm_Dispense_m1 := 0$

`c_PIN_Init` : $c_PIN := b_AccountPIN$
`b_FailPINAttempt_Init` : $b_FailPINAttempt := 0$
`c_WithdrawRequest_Init` : $c_WithdrawRequest := 0$
`c_Card_m1_Init` : $c_Card_m1 := Inserted$
`BC_Init` : $BC := FALSE$

end

Event *Withdraw_Initial* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_Initial$
`W_Grd` : $W = FALSE$
`W_Pre_1` : $c_Card = Inserted$
then
`W_flow_Act` : $W_flow := W_Trigger$
`c_Card_m1_Equal` : $c_Card_m1 := c_Card$
`atm_Dispense_m1_Equal` : $atm_Dispense_m1 := atm_Dispense$
end

Event *Withdraw_Trigger* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_Trigger$
`W_Trig_1` : $c_Card_m1 = Inserted$
then
`W_flow_Act` : $W_flow := W_1$
`BC_Act` : $BC := FALSE$
end

Event *W_1* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_1$
`BF_Grd` : $BC = TRUE$
then
`W_flow_Act` : $W_flow := W_2$
`W_1_Act` : $atm_Request := Request_PIN$
end

Event *W_2* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_2$
then
`W_flow_Act` : $W_flow := W_3$
`W_2_Act` : $c_PIN := b_AccountPIN$
end

Event *W_A_1* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_2$
then
`W_flow_Act` : $W_flow := W_A_2$
`W_A_1_Act` : $c_PIN :\in PIN \setminus \{b_AccountPIN\}$
end

Event *W_A_2* $\widehat{=}$
when
`W_flow_Grd` : $W_flow = W_A_2$
then

`W_flow_Act` : $W_flow := W_1$
`W_A_2_Act` : $b_FailPINAttempt := b_FailPINAttempt + 1$
`BC_Act` : $BC := FALSE$

end

Event *BlockCard* ≙
when
`BC_ExtensionPoint` : $W_flow = W_1$
`BC_Grd` : $BC = FALSE$
`BC_Pre_1` : $b_FailPINAttempt > 2$
then
`BC_RejoinPoint` : $W_flow := W_Final$
`BC_Act` : $BC := TRUE$
`BC_Post_Act` : $c_Card_m1, atm_Dispense_m1 :|$
$c_Card_m1' = Withheld \land atm_Dispense_m1' = 0$
end

Event *BlockCard_FALSE* ≙
when
`BC_ExtensionPoint` : $W_flow = W_1$
`BC_Grd` : $BC = FALSE$
`BC_Pre_1_Neg` : $\neg(b_FailPINAttempt > 2)$
then
`BC_Act` : $BC := TRUE$
end

Event *W_3* ≙
when
`W_flow_Grd` : $W_flow = W_3$
then
`W_flow_Act` : $W_flow := W_4$
`W_3_Act` : $atm_Request := Request_Withdrawal$
end

Event *W_4* ≙
when
`W_flow_Grd` : $W_flow = W_4$
then
`W_flow_Act` : $W_flow := W_5$
`W_4_Act` : $c_WithdrawRequest := b_AccountBalance$
end

Event *W_5* ≙
when
`W_flow_Grd` : $W_flow = W_5$
then
`W_flow_Act` : $W_flow := W_6$
`W_5_Act` : $c_Card_m1 := Returned$
end

Event *W_6* ≙
when
`W_flow_Grd` : $W_flow = W_6$
then
`W_flow_Act` : $W_flow := W_Final$
`W_6_Act` : $atm_Dispense_m1 := c_WithdrawRequest$

end

Event *Withdraw_Final* $\widehat{=}$
refines *Withdraw*

when

`W_flow_Grd` : $W_flow = W_Final$

`W_Grd` : $W = FALSE$

then

`W_Act` : $W := TRUE$

`atm_Dispense_Equal` : $atm_Dispense := atm_Dispense_m1$

`c_Card_Equal` : $c_Card := c_Card_m1$

end

END

An Event-B Specification of BlockCard_Flow

CONTEXT BlockCard_Flow
SETS

BC_FLOW

CONSTANTS

$BC_Initial, BC_Trigger, BC_1, BC_2, BC_3, BC_Final$

AXIOMS

`BC_FLOW_Type` : $partition(BC_FLOW, \{BC_Initial\}, \{BC_Trigger\}, \{BC_1\}, \{BC_2\}, \{BC_3\}, \{BC_Final\})$

END

An Event-B Specification of m2_BlockCard_Scenario

MACHINE m2_BlockCard_Scenario
REFINES m1_Withdraw_Scenario
SEES Withdraw_Static, Withdraw_Flow, BlockCard_Flow
VARIABLES

$W, BC, W_flow, atm_Dispense, atm_Request, c_PIN, c_WithdrawRequest,$

$c_Card, b_FailPINAttempt, b_AccountBlock, atm_Dispense_m1, c_Card_m1,$

$BC_flow, c_Card_m2, atm_Dispense_m2$

INVARIANTS

`BC_flow_Type` : $BC_flow \in BC_FLOW$

`c_Card_m2_Type` : $c_Card_m2 \in CARD$

`atm_Dispense_m2_Type` : $atm_Dispense_m2 \in 0 \,..\, b_AccountBalance$

`b_AccountBlock_Type` : $b_AccountBlock \in BOOL$

`BC_Glue_Variables` : $(BC_flow = BC_Final \land BC = TRUE \land W_flow = W_Final) \Rightarrow (c_Card_m2 = c_Card_m1) \land (atm_Dispense_m2 = atm_Dispense_m1)$

`BC_Glue_Flow` : $BC_flow \in BC_FLOW \setminus \{BC_Initial, BC_Final\} \Rightarrow BC = FALSE \land W_flow = W_1$

`BC_Scenario_Pre` : $BC_flow \in BC_FLOW \setminus \{BC_Initial\} \land BC = FALSE \land W_flow = W_1 \Rightarrow b_FailPINAttempt > 2$

`BC_Scenario_Post` : $BC_flow = BC_Final \land W_flow = W_1 \Rightarrow (c_Card_m2 = Withheld \land atm_Dispense_m2 = 0)$

BC_Scenario_Inv : $(atm_Dispense_m2 = 0 \land c_Card_m2 \in \{Inserted, Withheld\}) \lor$
$(c_Card_m2 = Returned \land atm_Dispense_m2 \in 0 \mathinner{..} b_AccountBalance)$

BC_StepAssert_1 : $BC_flow = BC_3 \Rightarrow c_Card_m2 = Withheld$

BC_StepAssert_2 : $BC_flow \in \{BC_Trigger, BC_1, BC_2\} \Rightarrow atm_Dispense_m2 = 0$

BC_StepAssert_3 : $(W_flow \in \{W_1, W_2, W_A_2\} \Rightarrow atm_Dispense_m1 = 0) \land$
$(W_flow \in W_FLOW \setminus \{W_Final\} \land BC = TRUE \Rightarrow \neg(BC_flow = BC_Final))$

EVENTS

Initialisation

extended

begin

W_Init : $W := FALSE$

c_Card_Init : $c_Card := Inserted$

atm_Dispense_Init : $atm_Dispense := 0$

W_flow_Init : $W_flow := W_Initial$

atm_Request_Init : $atm_Request := Request_Card$

atm_Dispense_m1_Init : $atm_Dispense_m1 := 0$

c_PIN_Init : $c_PIN := b_AccountPIN$

b_FailPINAttempt_Init : $b_FailPINAttempt := 0$

c_WithdrawRequest_Init : $c_WithdrawRequest := 0$

c_Card_m1_Init : $c_Card_m1 := Inserted$

BC_Init : $BC := FALSE$

act1 : $BC_flow := BC_Initial$

act2 : $atm_Dispense_m2 := 0$

act3 : $c_Card_m2 := Inserted$

act4 : $b_AccountBlock := FALSE$

end

Event *Withdraw_Initial* $\widehat{=}$

extends *Withdraw_Initial*

when

W_flow_Grd : $W_flow = W_Initial$

W_Grd : $W = FALSE$

W_Pre_1 : $c_Card = Inserted$

then

W_flow_Act : $W_flow := W_Trigger$

c_Card_m1_Equal : $c_Card_m1 := c_Card$

atm_Dispense_m1_Equal : $atm_Dispense_m1 := atm_Dispense$

end

Event *Withdraw_Trigger* $\widehat{=}$

extends *Withdraw_Trigger*

when

W_flow_Grd : $W_flow = W_Trigger$

W_Trig_1 : $c_Card_m1 = Inserted$

then

W_flow_Act : $W_flow := W_1$

BC_Act : $BC := FALSE$

act1 : $BC_flow := BC_Initial$

end

Event *W_1* $\widehat{=}$

extends *W_1*
    when
        W_flow_Grd : $W_flow = W_1$
        BF_Grd : $BC = TRUE$
    then
        W_flow_Act : $W_flow := W_2$
        W_1_Act : $atm_Request := Request_PIN$
    end

Event *W_2* $\widehat{=}$
extends *W_2*
    when
        W_flow_Grd : $W_flow = W_2$
    then
        W_flow_Act : $W_flow := W_3$
        W_2_Act : $c_PIN := b_AccountPIN$
    end

Event *W_A_1* $\widehat{=}$
extends *W_A_1*
    when
        W_flow_Grd : $W_flow = W_2$
    then
        W_flow_Act : $W_flow := W_A_2$
        W_A_1_Act : $c_PIN :\in PIN \setminus \{b_AccountPIN\}$
    end

Event *W_A_2* $\widehat{=}$
extends *W_A_2*
    when
        W_flow_Grd : $W_flow = W_A_2$
    then
        W_flow_Act : $W_flow := W_1$
        W_A_2_Act : $b_FailPINAttempt := b_FailPINAttempt + 1$
        BC_Act : $BC := FALSE$
        act1 : $BC_flow := BC_Initial$
    end

Event *BlockCard_FALSE* $\widehat{=}$
extends *BlockCard_FALSE*
    when
        BC_ExtPoint : $W_flow = W_1$
        BC_Grd : $BC = FALSE$
        BC_Pre_1_Neg : $\neg(b_FailPINAttempt > 2)$
    then
        BC_Act : $BC := TRUE$
    end

Event *W_3* $\widehat{=}$
extends *W_3*
    when
        W_flow_Grd : $W_flow = W_3$
    then
        W_flow_Act : $W_flow := W_4$
        W_3_Act : $atm_Request := Request_Withdrawal$
    end

Event $W_4 \widehat{=}$
extends W_4
when
`W_flow_Grd` : $W_flow = W_4$
then
`W_flow_Act` : $W_flow := W_5$
`W_4_Act` : $c_WithdrawRequest := b_AccountBalance$
end

Event $W_5 \widehat{=}$
extends W_5
when
`W_flow_Grd` : $W_flow = W_5$
then
`W_flow_Act` : $W_flow := W_6$
`W_5_Act` : $c_Card_m1 := Returned$
end

Event $W_6 \widehat{=}$
extends W_6
when
`W_flow_Grd` : $W_flow = W_6$
then
`W_flow_Act` : $W_flow := W_Final$
`W_6_Act` : $atm_Dispense_m1 := c_WithdrawRequest$
end

Event $Withdraw_Final \widehat{=}$
extends $Withdraw_Final$
when
`W_flow_Grd` : $W_flow = W_Final$
`W_Grd` : $W = FALSE$
then
`W_Act` : $W := TRUE$
`atm_Dispense_Equal` : $atm_Dispense := atm_Dispense_m1$
`c_Card_Equal` : $c_Card := c_Card_m1$
end

Event $BlockCard_Initial \widehat{=}$
when
`BC_flow_Grd` : $BC_flow = BC_Initial$
`BC_Grd` : $BC = FALSE$
`BC_ExtensionPoint` : $W_flow = W_1$
`BC_Pre_1` : $b_FailPINAttempt > 2$
then
`BC_flow_Act` : $BC_flow := BC_Trigger$
`c_Card_m2_Equal` : $c_Card_m2 := c_Card_m1$
`atm_Dispense_m2_Equal` : $atm_Dispense_m2 := atm_Dispense_m1$
end

Event $BlockCard_Trigger \widehat{=}$
when
`BC_flow_Grd` : $BC_flow = BC_Trigger$
`BC_Trig_1` : $b_FailPINAttempt > 2$
then
`act1` : $BC_flow := BC_1$
end

Event $BC_1 \triangleq$
when
BC_flow_Grd : $BC_flow = BC_1$
then
BC_flow_Act : $BC_flow := BC_2$
BC_1_Act : $b_AccountBlock := TRUE$
end

Event $BC_2 \triangleq$
when
BC_flow_Grd : $BC_flow = BC_2$
then
BC_flow_Act : $BC_flow := BC_3$
BC_2_Act : $c_Card_m2 := Withheld$
end

Event $BC_3 \triangleq$
when
BC_flow_Grd : $BC_flow = BC_3$
then
BC_flow_Act : $BC_flow := BC_Final$
BC_3_Act : $atm_Dispense_m2 := 0$
end

Event $BlockCard_Final \triangleq$
refines $BlockCard$
when
BC_flow_Grd : $BC_flow = BC_Final$
BC_Grd : $BC = FALSE$
W_flow_Grd : $W_flow = W_1$
then
BC_Act : $BC := TRUE$
BC_RejoinPoint : $W_flow := W_Final$
c_Card_m1_Equal : $c_Card_m1 := c_Card_m2$
atm_Dispense_m1_Equal : $atm_Dispense_m1 := atm_Dispense_m2$
end
END

B.4 UC4: Sense and Avoid

An Event-B Specification of SafeSeparation_Static

CONTEXT SafeSeparation_Static
SETS
$RESPONSE, PLAN$

CONSTANTS
$Accept, Reject, Idle, SS_Plan, CA_Plan, On_Route$

AXIOMS
RESPONSE_Enum : $partition(RESPONSE, \{Accept\}, \{Reject\}, \{Idle\})$
PLAN_Enum : $partition(PLAN, \{SS_Plan\}, \{CA_Plan\}, \{On_Route\})$
END

An Event-B Specification of m0_SafeSeparation_Contract

MACHINE m0_SafeSeparation_Contract
SEES SafeSeparation_Static
VARIABLES
$SS, intruder, separation$

INVARIANTS
`SS_Type` : $SS \in BOOL$
`intruder_Type` : $intruder \in BOOL$
`separation_Type` : $separation \in BOOL$
`SS_Inv_1` : $separation = TRUE$

EVENTS
Initialisation
extended

begin
`S_Init` : $SS := FALSE$
`intruder_Init` : $intruder := TRUE$
`separation_Init` : $separation := TRUE$
end

Event $SafeSeparation \mathrel{\widehat{=}}$
when
`SS_Pre_1` : $intruder = TRUE$
`SS_Grd` : $SS = FALSE$
then
`SS_Act` : $SS := TRUE$
`SS_Post_Act` : $intruder, separation : |intruder' = FALSE \wedge separation' = TRUE$
end
END

An Event-B Specification of SafeSeparation_Flow

CONTEXT SafeSeparation_Flow
SETS
SS_FLOW

CONSTANTS
$SS_Initial, SS_Trigger, SS_1, SS_2, SS_3, SS_4$
$SS_5, F_1, F_2, F_3, F_4, SS_Final$

AXIOMS
`SS_FLOW_Type` : $partition(SS_FLOW, \{SS_Initial\}, \{SS_Trigger\}, \{SS_1\}, \{SS_2\}, \{SS_3\}, \{SS_4\}, \{SS_5\}, \{F_1\}, \{F_2\}, \{F_3\}, \{F_4\}, \{SS_Final\})$
END

An Event-B Specification of m1_SafeSeparation_Scenario

MACHINE m1_SafeSeparation_Scenario
REFINES m0_SafeSeparation_Contract

SEES SafeSeparation_Static, SafeSeparation_Flow
VARIABLES

$SS, CA, SS_flow, intruder, separation, ss_risk, ss_plan$

$ss_response, ss_breach, ca_breach, intruder_m1, mission$

INVARIANTS

`CA_Type` : $CA \in BOOL$

`SS_flow_Type` : $SS_flow \in SS_FLOW$

`ss_risk_Type` : $ss_risk \in BOOL$

`ss_response_Type` : $ss_response \in RESPONSE$

`ss_plan_Type` : $ss_plan \in BOOL$

`ss_breach_Type` : $ss_breach \in BOOL$

`mission_Type` : $mission \in PLAN$

`intruder_m1_Type` : $intruder_m1 \in BOOL$

`ca_breach_Type` : $ca_breach \in BOOL$

`SS_Scenario_Inv` : $(separation = TRUE \Leftrightarrow (ss_breach = FALSE \wedge ca_breach = FALSE) \vee (ss_breach = TRUE \wedge ca_breach = FALSE)) \wedge (separation = FALSE \Leftrightarrow ca_breach = TRUE) \wedge (\neg(ca_breach = TRUE \wedge ss_breach = FALSE)) \wedge (separation = FALSE \Leftrightarrow ca_breach = TRUE)$

`SS_Scenario_Pre` : $SS_flow \in SS_FLOW \setminus \{SS_Initial\} \wedge SS = FALSE \Rightarrow intruder = TRUE$

`SS_Scenario_Post` : $SS_flow = SS_Final \Rightarrow intruder_m1 = FALSE \wedge separation = TRUE$

`SS_Glue_Variables` : $SS_flow = SS_Trigger \vee (SS_flow = SS_Final \wedge SS = TRUE) \Rightarrow intruder = intruder_m1$

`SS_Glue_Flow` : $SS_flow \in SS_FLOW \setminus \{SS_Initial, SS_Final\} \Rightarrow SS = FALSE$

`Prevent_CA` : $\neg(CA = FALSE \wedge SS_flow = F_4 \wedge \neg(ss_breach = TRUE \wedge mission = On_Route)) \wedge \neg(SS_flow = F_4 \wedge CA = TRUE)$

`SS_StepAssert_1` : $SS_flow = F_3 \Rightarrow mission = On_Route$

EVENTS
Initialisation

extended

begin

`S_Init` : $SS := FALSE$

`intruder_Init` : $intruder := TRUE$

`separation_Init` : $separation := TRUE$

`CA_Init` : $CA := FALSE$

`SS_flow_Init` : $SS_flow := SS_Initial$

`ss_risk_Init` : $ss_risk := FALSE$

`ss_response_Init` : $ss_response := Idle$

`ss_plan_Init` : $ss_plan := FALSE$

`ss_breach_Init` : $ss_breach := FALSE$

`intruder_m1_Init` : $intruder_m1 := TRUE$

`ca_breach_Init` : $ca_breach := FALSE$

`mission_Init` : $mission := On_Route$

end

Event *SafeSeparation_Initial* $\widehat{=}$

when

`SS_Grd` : $SS = FALSE$

`SS_flow_Grd` : $SS_flow = SS_Initial$

`SS_Pre_1` : $intruder = TRUE$

then
`SS_flow_Act` : $SS_flow := SS_Trigger$
`intruder_m1_Equal` : $intruder_m1 := intruder$
end

Event *SafeSeparation_Trigger* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_Trigger$
`SS_Trig_1` : $intruder_m1 = TRUE$
then
`SS_flow_Act` : $SS_flow := SS_1$
end

Event *SS_1* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_1$
then
`SS_flow_Act` : $SS_flow := SS_2$
`SS_1_Act` : $ss_risk := TRUE$
end

Event *SS_2* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_2$
then
`SS_flow_Act` : $SS_flow := SS_3$
`SS_2_Act` : $ss_plan := TRUE$
end

Event *SS_3* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_3$
then
`SS_flow_Act` : $SS_flow := SS_4$
`SS_3_Act` : $ss_response := Accept$
end

Event *SS_4* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_4$
then
`SS_flow_Act` : $SS_flow := SS_5$
`SS_4_Act` : $mission := SS_Plan$
end

Event *SS_5* $\widehat{=}$
when
`SS_flow_Grd` : $SS_flow = SS_5$
then
`SS_flow_Act` : $SS_flow := SS_Final$
`SS_5_Act` : $intruder_m1 := FALSE$
end

Event *SafeSeparation_Final* $\widehat{=}$
refines *SafeSeparation*
when
`SS_Grd` : $SS = FALSE$
`SS_flow_Grd` : $SS_flow = SS_Final$

then

`SS_Act` : $SS := TRUE$

`intruder_equal` : $intruder := intruder_m1$

end

Event *SafeSeperationFailure_Trigger* $\widehat{=}$

when

`SS_flow_Grd` : $SS_flow = SS_2$

`F_Trig_1` : $ss_risk = TRUE$

then

`SS_flow_Act` : $SS_flow := F_1$

end

Event *F_1* $\widehat{=}$

when

`SS_flow_Grd` : $SS_flow = F_1$

then

`SS_flow_Act` : $SS_flow := F_2$

`F_1_Act` : $ss_plan := FALSE$

end

Event *F_2* $\widehat{=}$

when

`SS_flow_Grd` : $SS_flow = F_2$

then

`SS_flow_Act` : $SS_flow := F_3$

`F_2_Act` : $mission := On_Route$

end

Event *F_3* $\widehat{=}$

when

`SS_flow_Grd` : $SS_flow = F_3$

then

`SS_flow_Act` : $SS_flow := F_4$

`F_3_Act` : $ss_breach := TRUE$

`CA_Act` : $CA := FALSE$

end

Event *CollisionAvoidance* $\widehat{=}$

when

`CA_ExtPoint` : $SS_flow = F_4$

`CA_Pre_1` : $ss_breach = TRUE \land mission = On_Route$

`CA_Grd` : $CA = FALSE$

then

`CA_Act` : $CA := TRUE$

`CA_RejPoint` : $SS_flow := SS_Final$

`CA_Post_Act` : $ca_breach, intruder_m1, mission : | ca_breach' = FALSE \land intruder_m1' = FALSE \land mission'$
$\{CA_Plan, On_Route\}$

end

Event *CollisionAvoidance_FALSE* $\widehat{=}$

when

`CA_Grd` : $CA = FALSE$

`CA_ExtPoint` : $SS_flow = F_4$

`CA_Pre_1_Neg` : $\neg(ss_breach = TRUE \land mission = On_Route)$

then

`CA_Act` : $CA := TRUE$

end

Event $F_4 \widehat{=}$

when

SS_flow_Grd : $SS_flow = F_4$

CA_Grd : $CA = TRUE$

then

F_3_Act : $ca_breach := TRUE$

SS_flow_Act : $SS_flow := SS_Final$

end

END

An Event-B Specification of CollisionAvoidance_Flow

CONTEXT CollisionAvoidance_Flow

SETS

CA_FLOW

CONSTANTS

$CA_Initial, CA_Trigger, CA_1, CA_2, CA_3, CA_4, CA_5, CA_Final$

AXIOMS

CA_FLOW_Type : $partition(CA_FLOW, \{CA_Initial\}, \{CA_Trigger\}, \{CA_1\}, \{CA_2\}, \{CA_3\}, \{CA_4\}, \{CA_5\}, \{CA_Final\})$

END

An Event-B Specification of m2_CollisionAvoidance_Scenario

MACHINE m2_CollisionAvoidance_Scenario

REFINES m1_SafeSeparation_Scenario

SEES SafeSeparation_Static, SafeSeparation_Flow, CollisionAvoidance_Flow

VARIABLES

$SS, CA, CA_flow, intruder, separation, SS_flow, ss_risk, ss_plan, mission_m2,$

$ss_response, ss_breach, intruder_m1, ca_breach, ca_risk, ss_breach_m2,$

$ca_plan, ca_response, intruder_m2, ca_breach_m2, mission$

INVARIANTS

CA_flow_Type : $CA_flow \in CA_FLOW$

ca_plan_Type : $ca_plan \in BOOL$

ca_risk_Type : $ca_risk \in BOOL$

ca_response_Type : $ca_response \in RESPONSE$

ca_breach_m2_Type : $ca_breach_m2 \in BOOL$

intruder_threat_m2_Type : $intruder_m2 \in BOOL$

mission_m2_Type : $mission_m2 \in PLAN$

ss_breach_m2_Type : $ss_breach_m2 \in BOOL$

CA_Glue_Variables : $CA_flow = CA_Trigger \lor (CA_flow = CA_Final \land CA = TRUE \land SS_flow = SS_Final) \Rightarrow (ca_breach_m2 = ca_breach) \land (intruder_m2 = intruder_m1)$

CA_Glue_Flow : $CA_flow \in CA_FLOW \setminus \{CA_Initial, CA_Final\} \Rightarrow CA = FALSE \land SS_flow = F_4$

CA_Scenario_Pre : $CA_flow \in CA_FLOW \setminus \{CA_Initial\} \land CA = FALSE \land SS_flow = F_4 \Rightarrow ss_breach = TRUE \land mission = On_Route$

`CA_Scenario_Post` : $CA_flow = CA_Final \Rightarrow ca_breach_m2 = FALSE \wedge intruder_m2 = FALSE \wedge mission_m2 \in \{CA_Plan, On_Route\}$

`CA_Scenario_Inv` : $(separation = TRUE \Leftrightarrow (ss_breach_m2 = FALSE \wedge ca_breach_m2 = FALSE) \vee (ss_breach_m2 = TRUE \wedge ca_breach_m2 = FALSE))$
$\wedge$
$(separation = FALSE \Leftrightarrow ca_breach_m2 = TRUE)$
$\wedge$
$(separation = FALSE \Leftrightarrow ca_breach_m2 = TRUE)$
$\wedge$
$(\neg(ca_breach_m2 = TRUE \wedge ss_breach_m2 = FALSE))$

`CA_StepAssert_1` : $CA_flow = CA_5 \Rightarrow mission_m2 = CA_Plan$

EVENTS

Initialisation

extended

begin

`S_Init` : $SS := FALSE$
`intruder_Init` : $intruder := TRUE$
`separation_Init` : $separation := TRUE$
`CA_Init` : $CA := FALSE$
`SS_flow_Init` : $SS_flow := SS_Initial$
`ss_risk_Init` : $ss_risk := FALSE$
`ss_response_Init` : $ss_response := Idle$
`ss_plan_Init` : $ss_plan := FALSE$
`ss_breach_Init` : $ss_breach := FALSE$
`intruder_m1_Init` : $intruder_m1 := TRUE$
`ca_breach_Init` : $ca_breach := FALSE$
`mission_Init` : $mission := On_Route$
`CA_flow_Init` : $CA_flow := CA_Initial$
`ca_plan_Init` : $ca_plan := FALSE$
`ca_risk_Init` : $ca_risk := FALSE$
`ca_response_Init` : $ca_response := Idle$
`ca_breach_m2_Init` : $ca_breach_m2 := FALSE$
`intruder_m2_Init` : $intruder_m2 := TRUE$
`mission_m2_Init` : $mission_m2 := On_Route$
`ss_breach_m2_Init` : $ss_breach_m2 := FALSE$

end

Event *SafeSeparation_Initial* $\widehat{=}$
extends *SafeSeparation_Initial*

when

`SS_Grd` : $SS = FALSE$
`SS_flow_Grd` : $SS_flow = SS_Initial$
`SS_Pre_1` : $intruder = TRUE$

then

`SS_flow_Act` : $SS_flow := SS_Trigger$
`intruder_m1_Equal` : $intruder_m1 := intruder$

end

Event *SafeSeparation_Trigger* $\widehat{=}$
extends *SafeSeparation_Trigger*

when

`SS_flow_Grd` : $SS_flow = SS_Trigger$

`SS_Trig_1` : $intruder_m1 = TRUE$

then

`SS_flow_Act` : $SS_flow := SS_1$

end

Event *SS_1* $\widehat{=}$
extends *SS_1*

when

`SS_flow_Grd` : $SS_flow = SS_1$

then

`SS_flow_Act` : $SS_flow := SS_2$

`SS_1_Act` : $ss_risk := TRUE$

end

Event *SS_2* $\widehat{=}$
extends *SS_2*

when

`SS_flow_Grd` : $SS_flow = SS_2$

then

`SS_flow_Act` : $SS_flow := SS_3$

`SS_2_Act` : $ss_plan := TRUE$

end

Event *SS_3* $\widehat{=}$
extends *SS_3*

when

`SS_flow_Grd` : $SS_flow = SS_3$

then

`SS_flow_Act` : $SS_flow := SS_4$

`SS_3_Act` : $ss_response := Accept$

end

Event *SS_4* $\widehat{=}$
extends *SS_4*

when

`SS_flow_Grd` : $SS_flow = SS_4$

then

`SS_flow_Act` : $SS_flow := SS_5$

`SS_4_Act` : $mission := SS_Plan$

end

Event *SS_5* $\widehat{=}$
extends *SS_5*

when

`SS_flow_Grd` : $SS_flow = SS_5$

then

`SS_flow_Act` : $SS_flow := SS_Final$

`SS_5_Act` : $intruder_m1 := FALSE$

end

Event *SafeSeparation_Final* $\widehat{=}$
extends *SafeSeparation_Final*

when

`SS_Grd` : $SS = FALSE$

`SS_flow_Grd` : $SS_flow = SS_Final$

then

`SS_Act` : $SS := TRUE$

intruder_equal : $intruder := intruder_m1$

end

Event *SafeSeperationFailure_Trigger* $\widehat{=}$
extends *SafeSeperationFailure_Trigger*

when

SS_flow_Grd : $SS_flow = SS_2$

F_Trig_1 : $ss_risk = TRUE$

then

SS_flow_Act : $SS_flow := F_1$

end

Event *F_1* $\widehat{=}$
extends *F_1*

when

SS_flow_Grd : $SS_flow = F_1$

then

SS_flow_Act : $SS_flow := F_2$

F_1_Act : $ss_plan := FALSE$

end

Event *F_2* $\widehat{=}$
extends *F_2*

when

SS_flow_Grd : $SS_flow = F_2$

then

SS_flow_Act : $SS_flow := F_3$

F_2_Act : $mission := On_Route$

end

Event *F_3* $\widehat{=}$
extends *F_3*

when

SS_flow_Grd : $SS_flow = F_3$

then

SS_flow_Act : $SS_flow := F_4$

F_3_Act : $ss_breach := TRUE$

CA_Act : $CA := FALSE$

end

Event *CollisionAvoidance_Initial* $\widehat{=}$

when

CA_flow_Grd : $CA_flow = CA_Initial$

CA_Pre_1 : $ss_breach = TRUE \land mission = On_Route$

CA_Grd : $CA = FALSE$

CA_ExtPoint : $SS_flow = F_4$

then

CA_flow_Act : $CA_flow := CA_Trigger$

ss_breach_m2_Equal : $ss_breach_m2 := ss_breach$

ca_zone_m2_Equal : $ca_breach_m2 := ca_breach$

intruder_threat_m2_Equal : $intruder_m2 := intruder_m1$

mission_m2_Equal : $mission_m2 := mission$

end

Event *CollisionAvoidance_Trigger* $\widehat{=}$

when

CA_flow_Grd : $CA_flow = CA_Trigger$

`CA_Trig_1` : $ss_breach_m2 = TRUE \land mission_m2 = On_Route$

then

`CA_flow_Act` : $CA_flow := CA_1$

end

Event *CA_1* $\widehat{=}$

when

`CA_flow_Grd` : $CA_flow = CA_1$

then

`CA_flow_Act` : $CA_flow := CA_2$

`CA_1_Act` : $ca_risk := TRUE$

end

Event *CA_2* $\widehat{=}$

when

`CA_flow_Grd` : $CA_flow = CA_2$

then

`CA_flow_Act` : $CA_flow := CA_3$

`CA_2_Act` : $ca_plan := TRUE$

end

Event *CA_3* $\widehat{=}$

when

`CA_flow_Grd` : $CA_flow = CA_3$

then

`CA_flow_Act` : $CA_flow := CA_4$

`CA_2_Act` : $ca_response := Accept$

end

Event *CA_4* $\widehat{=}$

when

`CA_flow_Grd` : $CA_flow = CA_4$

then

`CA_flow_Act` : $CA_flow := CA_5$

`CA_4_Act` : $mission_m2 := CA_Plan$

end

Event *CA_5* $\widehat{=}$

when

`CA_flow_Grd` : $CA_flow = CA_5$

then

`CA_flow_Act` : $CA_flow := CA_Final$

`CA_5_Act` : $intruder_m2 := FALSE$

end

Event *CollisionAvoidance_Final* $\widehat{=}$

refines *CollisionAvoidance*

when

`CA_flow_Grd` : $CA_flow = CA_Final$

`CA_Grd` : $CA = FALSE$

`CA_ExtPoint` : $SS_flow = F_4$

then

`CA_Act` : $CA := TRUE$

`CA_RejPoint` : $SS_flow := SS_Final$

`intruder_m1_Equal` : $intruder_m1 := intruder_m2$

`ca_breach_Equal` : $ca_breach := ca_breach_m2$

`mission_Equal` : $mission := mission_m2$

end

Event *CollisionAvoidance_FALSE* $\widehat{=}$
extends *CollisionAvoidance_FALSE*

when

`CA_Grd` : $CA = FALSE$

`CA_ExtPoint` : $SS_flow = F_4$

`CA_Pre_1_Neg` : $\neg(ss_breach = TRUE \wedge mission = On_Route)$

then

`CA_Act` : $CA := TRUE$

end

Event *F_4* $\widehat{=}$
extends *F_4*

when

`SS_flow_Grd` : $SS_flow = F_4$

`CA_Grd` : $CA = TRUE$

then

`F_3_Act` : $ca_breach := TRUE$

`SS_flow_Act` : $SS_flow := SS_Final$

end

END

Appendix C

Case Studies: State Charts

C.1 UC1: Water Tank System

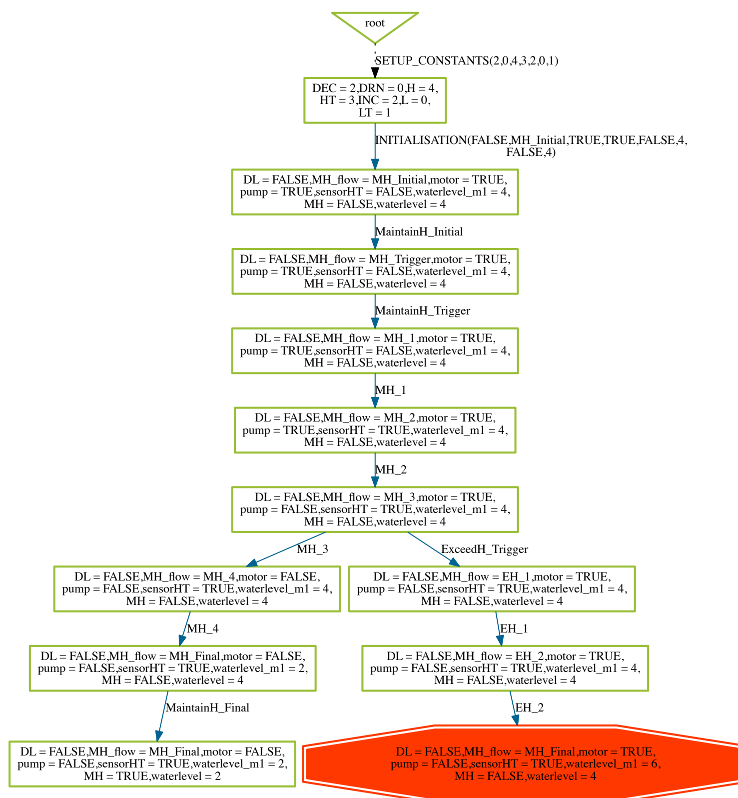


Figure C.1: MaintainH_Scenario_m1*: Deviation from* ExceedH *accident case without prevention from* DrainToL *extension use case.*

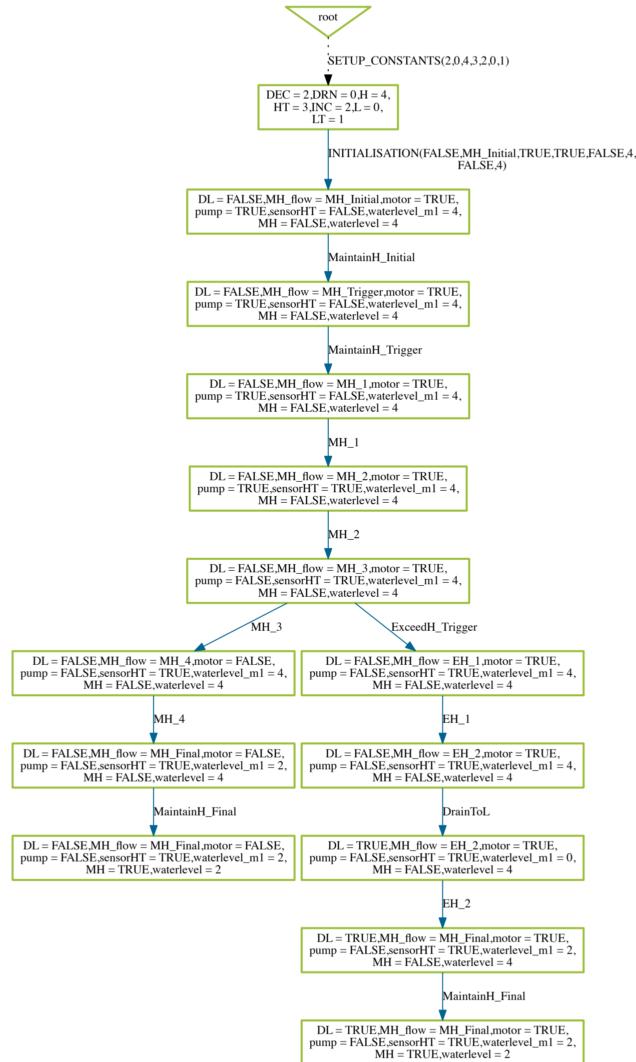


Figure C.2: MaintainH_Scenario_m1*: Deviation from* ExceedH *accident case with prevention from* DrainToL *extension use case.*

C.2 UC2: Train Door Control System

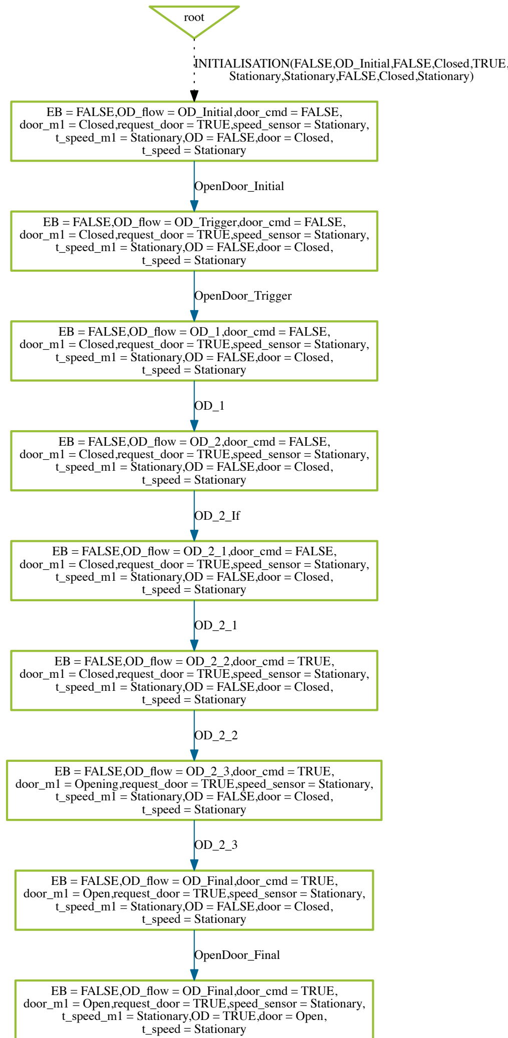


Figure C.3: OpenDoor_m1*: TDCS issues door open command when train speed is stationary.*

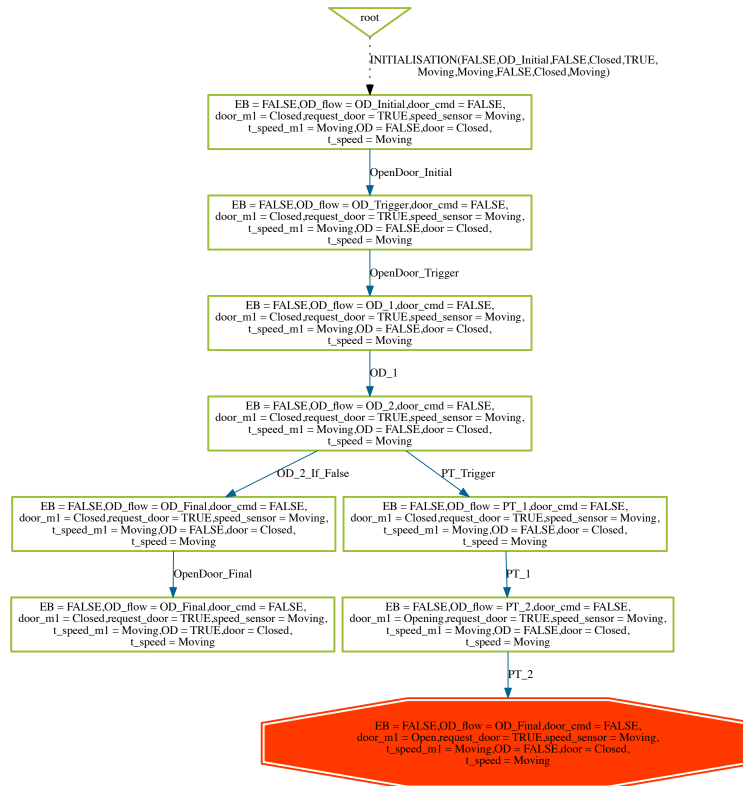


Figure C.4: OpenDoor_m1*: Deviation from* PassengerFallsOffMovingTrain *accident case.*

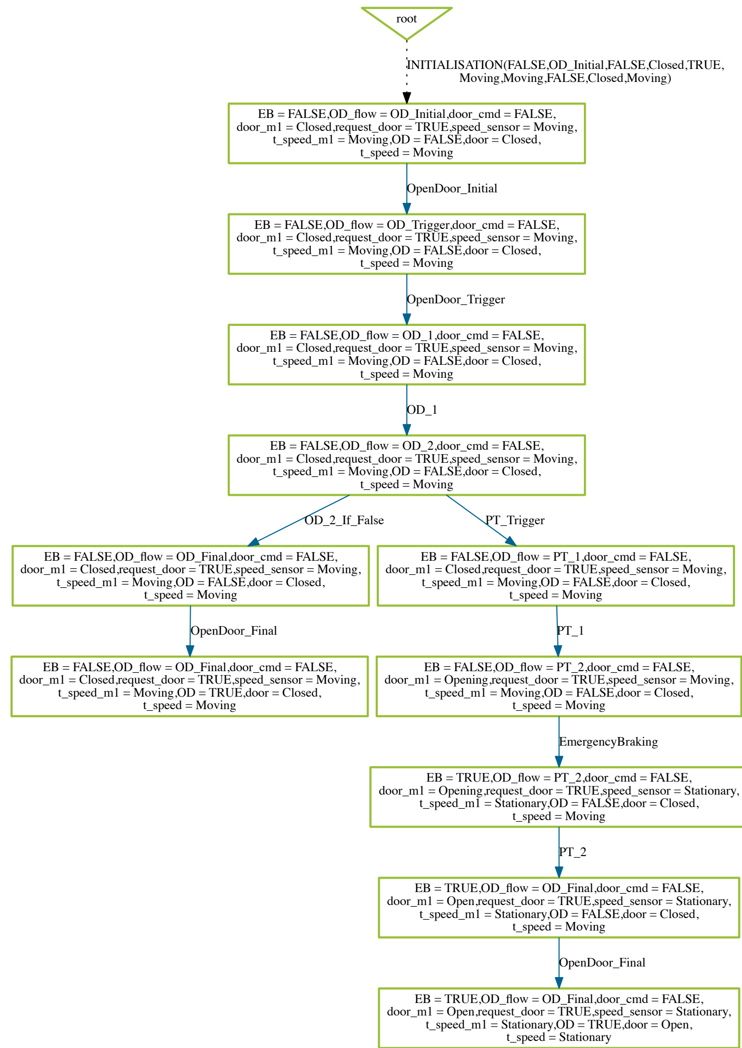


Figure C.5: OpenDoor_m1*: Deviation from* PassengerFallsOffMovingTrain *accident case with prevention from* EmergencyBraking *extension use case.*

C.3 UC3: Automated Teller Machine

Figure C.6: State chart for Withdraw *use case with extesnion* BlockCard.

C.4 UC4: Sense and Avoid

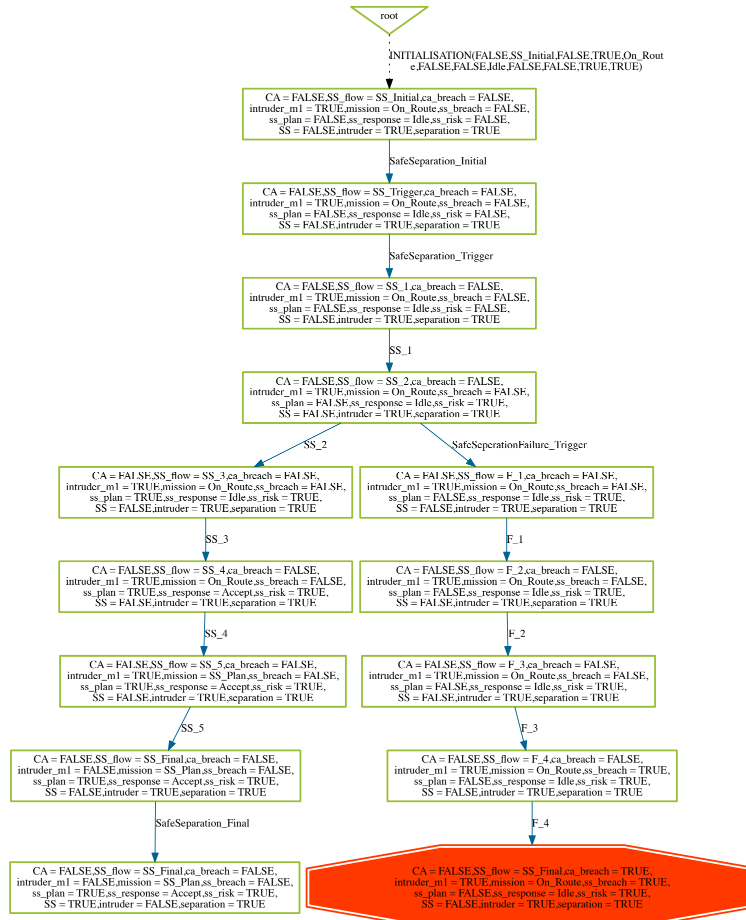


Figure C.7: SafeSeparation *with deviation from accident case* CollisionWithIntruderAircraft.

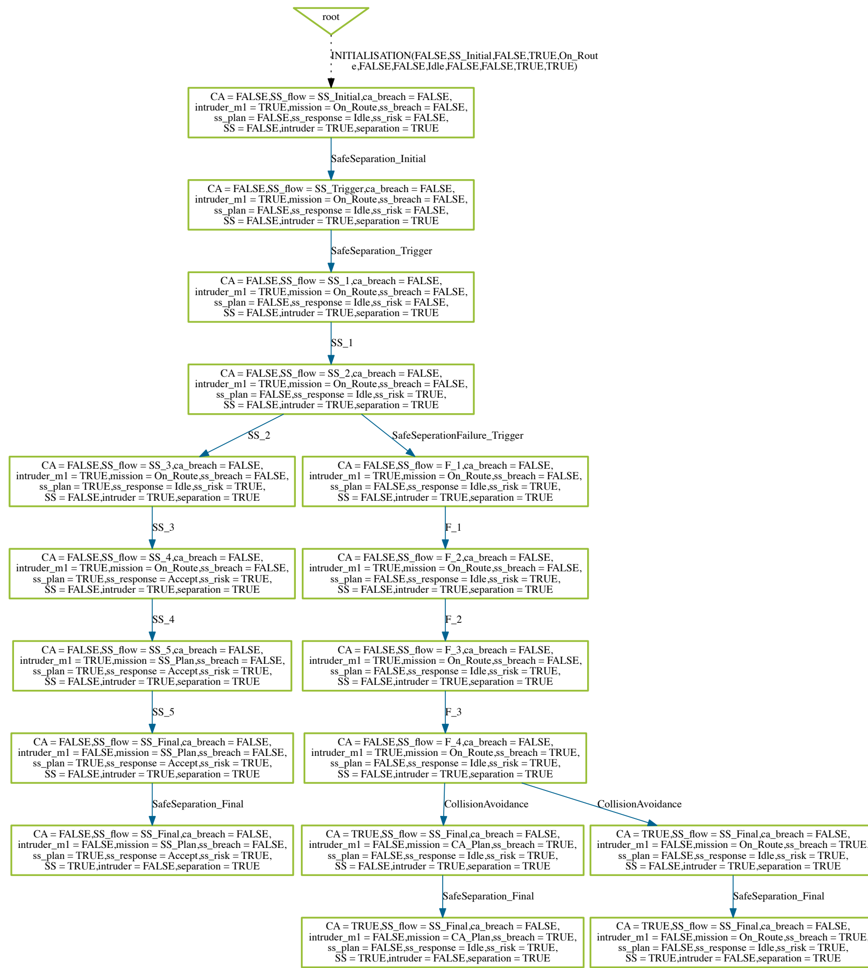


Figure C.8: SafeSeparation *with deviation from accident case* CollisionWithIntruderAircraft *and prevention from extension use case* CollisionAvoidance.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[2] Jean-Raymond Abrial. From Z to B and then Event-B: Assigning Proofs to Meaningful Programs. In *Integrated Formal Methods*, pages 1–15. Springer, 2013.

[3] Jean-Raymond Abrial, Jean-Raymond Abrial, and A Hoare. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[5] Steve Adolph, Alistair Cockburn, and Paul Bramble. *Patterns for Effective Use Cases.* Addison-Wesley Longman Publishing, 2002.

[6] Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Applied Formal Methods—FM-Trends 98*, pages 168–183. Springer, 1999.

[7] Karen Allenby and Tim Kelly. Deriving Safety Requirements using Scenarios. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 228–235. IEEE, 2001.

[8] Annie Antón and Colin Potts. The Use of Goals to Surface Requirements for Evolving Systems. In *Proceedings of the 20th international conference on Software engineering*, pages 157–166. IEEE Computer Society, 1998.

[9] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design.* Pearson Education, 2005.

[10] SAE ARP4754. Certification Considerations for Highly-Integrated or Complex Aircraft Systems. *SAE, Warrendale, PA*, 1996.

[11] Ralph-Johan Back, Luigia Petre, and Iván Porres Paltor. Analysing UML Use Cases as Contracts. In *UML'99—The Unified Modeling Language*, pages 518–533. Springer, 1999.

[12] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer Science & Business Media, 2012.

[13] Ralph-JR Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

[14] Stephen Barrett, Daniel Sinnig, Patrice Chalin, and Greg Butler. Merging of Use Case Models: Semantic Foundations. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 182–189. IEEE, 2009.

[15] Kamila Bartsch, Mike Robey, Jim Ivins, and Chiou Peng Lam. Consistency Checking between Use Case Scenarios and UML Sequence Diagrams. In *Software Engineering: IASTED International Conference Proceedings*, 2004.

[16] Thomas E Bell and Thomas A Thayer. Software Requirements: Are They Really a Problem? In *Proceedings of the 2nd international conference on Software engineering*, pages 61–68. IEEE Computer Society Press, 1976.

[17] Inquiry Board. ARIANE 5 Flight 501 Failure, Report by the Inquiry Board. *Paris, July*, 19, 1996.

[18] Jet Propulsion Laboratory (US). Special Review Board and John Casani. *Report on The Loss of the Mars Polar Lander and Deep Space 2 Missions*. Jet Propulsion Laboratory, California Institute of Technology, 2000.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language*. Addison-Wesley, 1997.

[20] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[21] Michael Butler. *Using Event-B Refinement to Verify a Control Strategy.* ECS, University of Southampton, 2009.

[22] Ana Cavalcanti and Jim Woodcock. ZRC–A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, 1998.

[23] Patrice Chalin, Joseph R Kiniry, Gary T Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal methods for components and objects*, pages 342–363. Springer, 2006.

[24] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5. Springer Science & Business Media, 2012.

[25] Alistair Cockburn. Structuring Use Cases with Goals. 1997.

[26] Alistair Cockburn. Writing Effective Use Cases. *preparation for Addison-Wesley Longman*, 1999.

[27] Alistair Cockburn. Writing Effective Use Cases, The Crystal Collection for Software Professionals, 2000.

[28] Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 612–613. ACM, 1997.

[29] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Number 47. Cambridge University Press, 1998.

[30] Homayoon Dezfuli and Michael Stamatelatos. A Paradigm Shift in System Safety Processes at NASA. In *7th National Space Systems Engineering & Risk Management Symposium*, 2008.

[31] Merlin Dorfman. System and Software Requirements Engineering. In *IEEE Computer Society Press Tutorial.* Citeseer, 1990.

[32] Eugène Dürr and Jan van Katwijk. VDM++, A Formal Specification Language for Object-Oriented Designs. In *CompEuro'92.'Computer Systems and Software Engineering', Proceedings.*, pages 214–219. IEEE, 1992.

[33] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using Lightweight Formal Methods for Requirements Modeling. *Software Engineering, IEEE Transactions on*, 24(1):4–14, 1998.

[34] IDE Eclipse. Eclipse Foundation, 2007.

[35] Robert J Ellison, David A Fisher, Richard C Linger, Howard F Lipson, and Thomas Longstaff. Survivable Network Systems: An Emerging Discipline. Technical report, DTIC Document, 1997.

[36] Albert Endres. An Analysis of Errors and Their Causes in System Programs. In *ACM Sigplan Notices*, volume 10, pages 327–336. ACM, 1975.

[37] Clifton A Ericson. Event Tree Analysis. *Hazard Analysis Techniques for System Safety*, pages 223–234, 2005.

[38] Clifton A Ericson and Clifton Ll. Fault Tree Analysis. In *System Safety Conference, Orlando, Florida*, pages 1–9, 1999.

[39] EE Euler, Steven D Jolly, and HH Curtis. The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved. In *Proceedings of Guidance and Control 2001*, pages 01–074, 2001.

[40] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In *NASA formal methods*, pages 328–342. Springer, 2011.

[41] Stephen Fickas and B Robert Helm. Knowledge Representation and Reasoning in the Design of Composite Systems. *Software Engineering, IEEE Transactions on*, 18(6):470–482, 1992.

[42] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML profile for framework architectures*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[43] Brooks FP Jr. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, (4):10–19, 1987.

[44] F Ronald Frola and CO Miller. System Safety in Aircraft Acquisition. Technical report, DTIC Document, 1984.

[45] Rainer Gmehlich, Katrin Grau, Stefan Hallerstede, Michael Leuschel, Felix Lösch, and Daniel Plagge. On Fitting a Formal Method into Practice. In *Formal Methods and Software Engineering*, pages 195–210. Springer, 2011.

[46] Sol Greenspan, John Mylopoulos, and Alex Borgida. On Formal Requirements Modeling Languages: RML Revisited. In *Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press, 1994.

[47] Wolfgang Grieskamp, Markus Lepper, Wolfram Schulte, and Nikolai Tillmann. Testable Use Cases in the Abstract State Machine Language. In *apaqs*, page 0167. IEEE, 2001.

[48] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[49] Russ Hurlbut. A Survey of Approaches for Describing and Formalizing Use Cases. *Expertech, Ltd*, 1997.

[50] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[51] Daniel Jackson and Jeannete Wing. Lightweight Formal Methods. *Computer*, (4):21–22, 1996.

[52] Michael Jackson. Requirements and Specifications: A Lexicon of Software Practice, Principles and Prejudices. *Addison Wesley, Wokingham*, 1995.

[53] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[54] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.

[55] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*, volume 1. Addison-wesley Reading, 1999.

[56] Matthew S Jaffe and Nancy G Leveson. Completeness, Robustness, and Safety in Real-Time Software Requirements Specification. In *Proceedings of the 11th International Conference on Software Engineering*, pages 302–311. ACM, 1989.

[57] Matthew S Jaffe, Nancy G Leveson, Mats PE Heimdahl, and Bonnie E Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *Software Engineering, IEEE Transactions on*, 17(3):241–258, 1991.

[58] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryldo G Russo Jr. An Approach of Requirements Tracing in Formal Refinement. In *Verified Software: Theories, Tools, Experiments*, pages 97–111. Springer, 2010.

[59] Cliff Jones. Formal Methods Light. *ACM Computing Surveys (CSUR)*, 28(4es):121, 1996.

[60] Cliff B Jones. *Development Methods for Computer Programs Including a Notion of Interference*. Oxford University Computing Laboratory, 1981.

[61] Cliff B Jones. *Systematic software development using VDM*, volume 2. Citeseer, 1990.

[62] Cliff B Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, (2):26–49, 2003.

[63] Cliff B Jones and Ken G Pierce. *Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification*. Springer, 2008.

[64] Trevor A Kletz. Human Problems with Computer Control. *Plant/Operations Progress*, 1(4):209–211, 1982.

[65] Radosław Klimek and Piotr Szwed. Formal Analysis of Use Case Diagrams. *Computer Science*, 11:115–131, 2010.

[66] Warren Kuffel. Extra time saves money. *Computer Language, December*, 1990.

[67] Kevin Lano. Z++, An Object-Orientated Extension to Z. In *Z User Workshop, Oxford 1990*, pages 151–172. Springer, 1991.

[68] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus uml: an open source toolset for mda. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4. Citeseer, 2009.

[69] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative Integrating Tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.

[70] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In *FME 2002: Formal Methods—Getting IT Right*, pages 21–40. Springer, 2002.

[71] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, pages 855–874. Springer, 2003.

[72] Nancy Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety.* Mit Press, 2011.

[73] Nancy Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety.* Mit Press, 2011.

[74] Nancy G Leveson. *Safeware: System Safety and Computers.* ACM, 1995.

[75] Nancy G Leveson and Clark S Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[76] Lun-Cheng Lin, Bashar Nuseibeh, Daniel Ince, Michael Jackson, and Jonathan Moffett. Analysing Security Threats and Vulnerabilities using Abuse Frames. *ETAPS-04*, 2003.

[77] Robyn R Lutz. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 126–133. IEEE, 1993.

[78] John McDermott and Chris Fox. Using Abuse Case Models for Security Requirements Analysis. In *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*, pages 55–64. IEEE, 1999.

[79] Christophe Metayer, Jean-Raymond Abrial, and Laurent Voisin. Event-B Language. *RODIN Project Deliverable D*, 7, 2005.

[80] Robin Milner. *Communication and Concurrency*, volume 84. Prentice hall New York etc., 1989.

[81] Gary Mogyorodi. Requirements-Based Testing: An Overview. In *Technology of Object-Oriented Languages, International Conference on*, pages 0286–0286. IEEE Computer Society, 2001.

[82] Rajiv Murali, Andrew Ireland, and Gudmund Grov. A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios. In *NASA Formal Methods*, pages 263–278. Springer, 2015.

[83] Peter Naur, Brian Randell, Friedrich Ludwig Bauer, NATO Science Committee, et al. *Software Engineering: Report on a Conference Sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968.* Scientific Affairs Division, NATO, 1969.

[84] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic Test Generation: A Use Case Driven Approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006.

[85] Maria Nelson, Torsten Nelson, Paulo Alencar, and Don Cowan. Exploring Problem-Frame Concerns using Formal Analysis. In *Proceedings of the 1st International Workshop on Applications and Advances of Problem Frames*, pages 61–68. Citeseer, 2004.

[86] Bashar Nuseibeh and Steve Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[87] Gunnar Overgaard and Karin Palmkvist. A Formal Approach to Use Cases and their Relationships. In *The Unified Modeling Language.UML'98: Beyond the Notation*, pages 406–418. Springer, 1999.

[88] David Lorge Parnas and Paul C Clements. A Rational Design Process: How and Why to Fake It. *Software Engineering, IEEE Transactions on*, (2):251–257, 1986.

[89] James L Peterson. Petri Net Yheory and the Modeling of Systems. 1981.

[90] Amir Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*. Springer, 1986.

[91] Klaus Pohl and Peter Haumer. Modelling Contextual Information about Scenarios. In *Proceedings of the Third International Workshop on Requirements Engineering: Foundations of Software Quality REFSQ*, volume 97, pages 187–204, 1997.

[92] Chittoor V Ramamoorthy and Hon Hing So. *Software Requirements and Specifications: Status and Perspectives*. Electronics Research Laboratory, College of Engineering, University of California, 1978.

[93] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-wesley, 2012.

[94] Gruia-Catalin Roman. A Taxonomy of Current Issues in Requirements Engineering. *Computer*, 18(4):14–23, 1985.

[95] Alexander Romanovsky and Martyn Thomas. *Industrial Deployment of System Engineering Methods*. Springer, 2013.

[96] Aryldo G Russo Jr and Thiago de Sousa. Starting B Specifications from Use Cases. In *Abstract State Machines (ASM), Alloy, B and Z Conference*, 2010.

[97] Robert Seater and Daniel Jackson. Requirement Progression in Problem Frames Ppplied to a Proton Therapy System. In *Requirements Engineering, 14th IEEE International Conference*, pages 169–178. IEEE, 2006.

[98] Wuwei Shen and Shaoying Liu. Formalization, Testing and Execution of a Use Case Diagram. In *Formal Methods and Software Engineering*, pages 68–85. Springer, 2003.

[99] Guttorm Sindre and Andreas L Opdahl. Eliciting Security Requirements with Misuse Cases. *Requirements engineering*, 10(1):34–44, 2005.

[100] Colin Snook and Michael Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[101] Colin Snook and Rachel Harrison. Practitioners' Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *Information and Software Technology*, 43(4):275–283, 2001.

[102] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., 1997.

[103] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[104] Neil R Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[105] Thein Than Tun, Michael Jackson, Robin Laney, Bashar Nuseibeh, and Yijun Yu. Are Your Lights Off? Using Problem Frames to Diagnose System Failures. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 343–348. IEEE, 2009.

[106] Axel Van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the 22nd international conference on Software engineering*, pages 5–19. ACM, 2000.

[107] Axel Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[108] Axel Van Lamsweerde and Emmanuel Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.

[109] Axel Van Lamsweerde and Laurent Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *Software Engineering, IEEE Transactions on*, 24(12):1089–1114, 1998.

[110] Jos B Warmer and Anneke G Kleppe. The Object Constraint Language: Precise Modeling with UML. 1998.

[111] Jon Whittle. Precise Specification of Use Case Scenarios. In *Fundamental Approaches to Software Engineering*, pages 170–184. Springer, 2007.

[112] Niklaus Wirth. Extended Backus-Naur Form (EBNF). *ISO/IEC*, 14977:2996, 1996.

[113] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., 1996.

[114] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[115] Eric Yu and John Mylopoulos. Why Goal-Oriented Requirements Engineering. In *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, volume 15, 1998.

[116] Eric SK Yu. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[117] Pamela Zave. Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys (CSUR)*, 29(4):315–321, 1997.

[118] Jinqiang Zhao and Zhenhua Duan. Verification of Use Case with Petri Nets in Requirement Analysis. In *Computational Science and Its Applications–ICCSA 2009*, pages 29–42. Springer, 2009.