TUCS

Yuliya Prokhorova

Rigorous Development of Safety-Critical Systems

Turku Centre *for* Computer Science

TUCS Dissertations
No 193, March 2015

Rigorous Development of Safety-Critical Systems

Yuliya Prokhorova

To be presented, with the permission of the Faculty of Science and Engineering of the Åbo Akademi University, for public criticism in Auditorium Gamma on March 5, 2015, at 12 noon.

Åbo Akademi University
Faculty of Science and Engineering
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

2015

Supervisors

Associate Professor Elena Troubitsyna
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

Adjunct Professor Linas Laibinis
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520 Turku
Finland

Reviewers

Professor Tim Kelly
Department of Computer Science
University of York
Deramore Lane, York YO10 5GH
United Kingdom

Senior Researcher Doctor Marco Bozzano
Fondazione Bruno Kessler
Via Sommarive 18, Povo, 38123 Trento
Italy

Opponent

Professor Tim Kelly
Department of Computer Science
University of York
Deramore Lane, York YO10 5GH
United Kingdom

ISBN 978-952-12-3174-2
ISSN 1239-1883

To my family

Моей семье посвящается

Abstract

Nowadays, computer-based systems tend to become more complex and control increasingly critical functions affecting different areas of human activities. Failures of such systems might result in loss of human lives as well as significant damage to the environment. Therefore, their safety needs to be ensured. However, the development of safety-critical systems is not a trivial exercise. Hence, to preclude design faults and guarantee the desired behaviour, different industrial standards prescribe the use of rigorous techniques for development and verification of such systems. The more critical the system is, the more rigorous approach should be undertaken.

To ensure safety of a critical computer-based system, satisfaction of the safety requirements imposed on this system should be demonstrated. This task involves a number of activities. In particular, a set of the safety requirements is usually derived by conducting various safety analysis techniques. Strong assurance that the system satisfies the safety requirements can be provided by formal methods, i.e., mathematically-based techniques. At the same time, the evidence that the system under consideration meets the imposed safety requirements might be demonstrated by constructing safety cases. However, the overall safety assurance process of critical computer-based systems remains insufficiently defined due to the following reasons. Firstly, there are semantic differences between safety requirements and formal models. Informally represented safety requirements should be translated into the underlying formal language to enable further verification. Secondly, the development of formal models of complex systems can be labour-intensive and time consuming. Thirdly, there are only a few well-defined methods for integration of formal verification results into safety cases.

This thesis proposes an integrated approach to the rigorous development and verification of safety-critical systems that (1) facilitates elicitation of safety requirements and their incorporation into formal models, (2) simplifies formal modelling and verification by proposing specification and refinement patterns, and (3) assists in the construction of safety cases from the artefacts generated by formal reasoning. Our chosen formal framework is Event-B. It allows us to tackle the complexity of safety-critical systems as well as to structure safety requirements by applying abstraction and stepwise refinement. The Rodin platform, a tool supporting Event-B, assists in automatic model transformations and proof-based verification of the desired system properties. The proposed approach has been validated by several case studies from different application domains.

Sammanfattning

Nuförtiden tenderar datorbaserade system att bli mer komplexa och kontrollera allt mer kritiska funktioner som påverkar olika områden av mänskliga aktiviteter. Brister i sådana system kan leda till förlust av människoliv eller stora skador på miljön. Detta innebär att vi måste försäkra oss om säkerheten hos dessa system, men att utveckla säkerhetskritiska system är ingen enkel uppgift. För att förhindra designfel och garantera önskvärt beteende föreskriver olika industriella standarder användning av rigorösa tekniker för utveckling och verifiering av sådana system. Ju mer kritiskt systemet är desto mer rigoröst tillvägagångssätt bör användas.

För att garantera säkerheten hos ett kritiskt datorbaserat system måste det visas att systemet uppfyller de säkerhetskrav det har ålagts. I denna uppgift ingår ett flertal förehavanden, men framför allt kan man genom att bedriva olika former av säkerhetsanalys erhålla en mängd säkerhetskrav. En stark försäkran om att systemet uppfyller sina säkerhetskrav kan fås via formella metoder, dvs. matematiskt baserade tekniker. Samtidigt kan bevis för att systemet uppfyller sina ålagda säkerhetskrav ges genom att framställa säkerhetsbevisningar (*safety cases*). Den övergripande processen för att garantera säkerhet för kritiska datorbaserade system förblir dock otillräckligt definierad på grund av följande orsaker. För det första finns det semantiska skillnader mellan säkerhetskrav och formella modeller. Informellt beskrivna säkerhetskrav bör översättas till det underläggande formella språket för att kunna verifieras. För det andra kan utveckling av formella modeller av komplexa system vara arbets- och tidskrävande. För det tredje finns det enbart ett fåtal väldefinierade metoder för att integrera formella verifikationsresultat i säkerhetsbevisningar.

I denna avhandling föreslås ett integrerat förhållningssätt till rigorös utveckling och verifiering av säkerhetskritiska system, vilket (1) underlättar insamling av säkerhetskrav och deras inkorporering i formella modeller, (2) förenklar formell modellering och verifiering genom att föreslå specifikations- och preciseringsmönster, och (3) bistår konstruktionen av säkerhetsbevisningar utgående från materialet skapat av formellt resonemang. Det formella ramverk vi har valt är Event-B. Detta låter oss hantera komplexiteten hos säkerhetskritiska system och likaså strukturera säkerhetskraven genom att utnyttja abstraktion och stegvis precisering. Rodin-plattformen, ett verktyg som stöder Event-B, hjälper till med automatiska modelltransformationer och bevisbaserad verifiering av systemets önskvärda egenskaper. Det föreslagna tillvägagångssättet har validerats genom ett flertal fallstudier från olika tillämpningsområden.

Acknowledgements

First of all, I would like to express my sincere gratitude to my both supervisors Associate Professor Elena Troubitsyna and Adjunct Professor Linas Laibinis for their guidance, expert advice, fruitful scientific discussions, and invaluable comments on this thesis. It has been an honour for me to work under their supervision. Furthermore, I would like to thank Professor Vyacheslav Kharchenko for guiding and encouraging me in the beginning of my PhD studies at National Aerospace University "KhAI".

I am very thankful to Professor Tim Kelly and Doctor Marco Bozzano for reviewing this thesis and for providing valuable comments that improved the quality of the final version of this dissertation. I am also very grateful to Professor Tim Kelly for kindly agreeing to act as an opponent at my doctoral defence.

Moreover, I would like to thank my external co-authors, Professor Alexander Romanovsky, Doctor Alexei Iliasov, Doctor Ilya Lopatkin, Doctor Timo Latvala, Doctor Kimmo Varpaaniemi, Doctor Dubravka Ilić, and Professor Vyacheslav Kharchenko, for their valuable contribution to this thesis.

I would like to express my gratitude to the members of the Distributed Systems Laboratory: Adjunct Professor Marina Waldén, Adjunct Professor Luigia Petre, Doctor Pontus Boström, Doctor Marta Olszewska, Doctor Mats Neovius, Doctor Maryam Kamali, Petter Sandvik, and Sergey Ostroumov. I am especially thankful to Professor Kaisa Sere for her sage advice and support.

Furthermore, I extend my sincere thanks to the members of the Embedded Systems Laboratory, which I joined in the final year of my PhD studies. In particular, I am thankful to Professor Johan Lilius, Doctor Sébastien Lafond, Doctor Leonidas Tsiopoulos, Doctor Johan Ersfolk, Doctor Anton Tarasyuk, Inna Pereverzeva, Wictor Lund, Simon Holmbacka, and Sudeep Kanur.

Moreover, I would like to acknowledge Professor Ion Petre for his encouragement and Professor Ivan Porres for all those interesting discussions we have had.

I am also thankful to the administrative personnel of TUCS, the Faculty of Science and Engineering and the Academic Office at Åbo Akademi University for their continuous assistance. In particular, I would like to thank Tomi Suovuo, Christel Engblom, Nina Hultholm, Tove Österroos, Susanne Ramstedt, Solveig Vaherkylä, Pia-Maria Kallio, and Outi Tuohi. I would also like to express my gratitude to the technical personnel: Marat, Karl, Joakim, and Niklas.

I would like to thank TUCS – Turku Centre for Computer Science and the Department of Information Technologies at Åbo Akademi University for the generous funding of my doctoral studies. I am also very grateful and honoured to receive research scholarships from Foundation of Nokia Corporation and Ulla Tuominen Foundation.

The final months of my work on the dissertation I spent working at Space Systems Finland Ltd. I wish to sincerely thank all my new colleagues for their concern and moral support at this stage.

Last but by not least, I would like to express my true thankfulness to my family and my friends for being there for me during these recent years. I wish to thank Magnus, Irum, Charmi and Faezeh for brining so many positive moments into my life.

I owe my special thanks to Petter Sandvik for being a friend and a great room-mate as well as for his help with the Swedish version of the abstract.

I would like to express my deep gratitude to my best friends, Marta and Miki Olszewski, for supporting, inspiring and guiding me from the very first days in Turku. I really appreciate the time spend with you.

I am especially grateful to my parents, Irina and Nikolay, as well as my grandmother, Galina, for their patience, understanding, support and unlimited love.

Finally, I would like to thank my dearest husband, Sergey. This thesis would not have been possible without your love, care and support.

Yuliya Prokhorova
Espoo, January 2015

List of original publications

I Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova, and Elena Troubitsyna. Patterns for Representing FMEA in Formal Specification of Control Systems, In Ankur Agarwal, Swapna Gokhale, Taghi M. Khoshgoftaar (Eds.), *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium (HASE 2011)*, pp. 146–151, IEEE Computer Society, 2011.

II Yuliya Prokhorova, Linas Laibinis, Elena Troubitsyna, Kimmo Varpaaniemi, and Timo Latvala. Deriving a Mode Logic Using Failure Modes and Effects Analysis, In *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 3, No. 4, pp. 305—328, Inderscience Publishers, 2012.

III Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System, In Anatoliy Gorbenko, Alexander Romanovsky, Vyacheslav Kharchenko (Eds.), *Proceedings of the 5th International Workshop on Software Engineering for Resilient Systems (SERENE 2013)*, Lecture Notes in Computer Science Vol. 8166, pp. 79-–93, Springer-Verlag Berlin Heidelberg, 2013.

IV Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ilić, and Timo Latvala. Formalisation of an Industrial Approach to Monitoring Critical Data, In Friedemann Bitsch, Jérémie Guiochet, Mohamed Kaâniche (Eds.), *Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013)*, Lecture Notes in Computer Science Vol. 8153, pp. 57-–69, Springer-Verlag Berlin Heidelberg, 2013.

V Yuliya Prokhorova and Elena Troubitsyna. Linking Modelling in Event-B with Safety Cases, In Paris Avgeriou (Ed.), *Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, Lecture Notes in Computer Science Vol. 7527, pp. 47-–62, Springer-Verlag Berlin Heidelberg, 2012.

VI Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Towards Rigorous Construction of Safety Cases. *TUCS Technical Report 1110*, TUCS, May 2014. (Shortened version is available as: Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Facilitating Construction of Safety Cases from Formal Models in Event-B, In *Information and Software Technology Journal*, Vol. 60, pp. 51–76, Elsevier, 2015.)

VII Yuliya Prokhorova and Elena Troubitsyna. A Survey of Safety-Oriented Model-Driven and Formal Development Approaches, In *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 2, pp. 93—118, Inderscience Publishers, 2013.

Contents

Part I

Research Summary

Chapter 1

Introduction

1.1 Motivation

Computer-based systems are found in different areas of human life. Examples of such systems are medical equipment and traffic control systems, car airbag and braking systems, nuclear reactor control and cooling systems, aerospace on-board systems, etc. In recent years, computer-based systems have become more complex and tend to control more and more critical functions. Failures of such systems might lead to severe consequences such as loss of human lives, damage to the environment or high economic losses. Therefore, it is important to achieve *dependability* of these systems, i.e., the ability of a system to deliver services that can be justifiably trusted [12, 13, 127].

Dependability is a multifaceted concept that has several attributes, e.g., *safety*, *reliability*, *availability*, etc. [12, 13]. Depending on the type of a system under consideration, the emphasis may be put on ensuring one or several of these attributes. In the context of this thesis, we focus on safety. *Safety* is the ability of a system to operate without causing catastrophic consequences for its users and environment [13, 127]. The critical systems where the main focus is put on safety are called *safety-critical systems* [87].

To properly design such systems, the developers should take into account the interplay between software and hardware as well as the environment in which the system will be expected to function. Complexity of modern safety-critical systems has stimulated the growth of *model-based development* techniques that can be applied at the early stages of *System Development Life Cycle (SDLC)*. The main goal of these techniques is the development and analysis of a *model* of a system rather than inspection of the final system implementation. A *model* is an abstraction allowing for representation or approximation of selected features of a system, process or concept. The model-based development involves construction of a system model, its analysis, verification, as well as generation of code and test cases.

The model-based development vary in the degree of rigour. On the one hand, there are model-based approaches that put emphasis on a graphical notation without formally defined semantics, e.g., *Unified Modelling Language (UML)* [108]. On the other hand, there are approaches that rely on formal specification languages, i.e., languages with rigorously defined semantics. As recommended by the standards [77, 78], the rigorous techniques such as *formal methods* (mathematically-based techniques) are applied to systems with a high level of criticality to prevent design faults and ensure the correct system behaviour.

Usually, the model-based development techniques allow us to capture the nominal system behaviour. However, to attain system safety, we need to guarantee that, even in the presence of failures of system components or some adverse changes in the environment, the system will not be put into a hazardous state. The required mechanisms for failure detection and fault tolerance further increase complexity of such systems. Hence, we need a scalable approach to rigorous system development that would consider the system in its entirety and would support handling of complexity, abstraction, and proof-based verification. In this thesis, we rely on the Event-B formalism [3] that satisfies these criteria. The development process in Event-B starts from abstract models and continues to concrete models via a chain of correctness preserving transformations called *refinements* [3, 18]. Refinement allows us to structure complex (safety) requirements by gradually introducing their representation in the system model as well as reason about system-level properties at different levels of abstraction.

To ensure safety, we need to demonstrate satisfaction of the safety requirements imposed on the system. A valuable source of safety requirements is safety analysis. Hence, it can be beneficial to connect formal development with safety analysis techniques such as *Fault Tree Analysis (FTA)*, *Failure Modes and Effects Analysis (FMEA)*, etc. However, due to the semantic differences between the results of safety analysis and formal models, establishing this connection remains challenging. Furthermore, certification of safety-critical systems requires submission of safety assurance documents, e.g., in the form of safety cases. A *safety case* justifies why a system is safe and whether the design adequately implements the imposed safety requirements [25, 45, 85]. The results of formal verification can be used to provide the required evidence for a system safety case.

To build a rigorous framework spanning from safety requirements and safety analysis to safety assurance, we need an integrated approach that incorporates safety requirements, including those given as the results of safety analysis, into formal models for verification, facilitates formal development and verification of safety-critical systems via pre-defined patterns, as well as establishes a link between the results of formal verification and safety cases. This thesis proposes such an approach. Next we state the research objectives of this thesis and outline the proposed solutions to achieve them.

1.2 Research Objectives

The overall objective of this thesis is to propose an approach to rigorous development and verification of safety-critical systems that integrates safety analysis into formal development and facilitates safety assurance. To achieve this objective, we address the following **Research Questions (RQs)**:

RQ1: *How to ensure a seamless integration of safety requirements into formal development?*

RQ2: *How to facilitate formal development and verification of safety-critical systems using specification and refinement patterns?*

RQ3: *How to support the safety assurance process with the artefacts generated by formal development and verification?*

Next we discuss each research question in detail. We first provide the motivation behind a research question and then justify the approaches taken in this thesis to address it. A more detailed description of the thesis contributions is presented in Chapter 3.

Integrating Safety Requirements into Formal Development. Usually, *System Development Life Cycle (SDLC)* starts from the *requirements phase* where the requirements are elicited, analysed and documented. There exist two commonly recognised types of requirements: *functional* and *non-functional*. The former define the dynamic behaviour of a system, while the latter impose restrictions on the quality of service delivered by the system. The examples of non-functional requirements for safety-critical systems are *safety*, *reliability*, *performance*, *security*, and *usability* [106].

Various dedicated techniques are used to elicit and analyse different kinds of requirements within *requirements engineering* [142]. They may include such activities as analysis of existing systems or documentation, prototyping, development of use cases and scenarios as well as hazard and risk analysis. As a result, system requirements, including safety requirements, can be represented either textually, graphically or in a tabular form.

In the development of safety-critical systems, safety analysis plays an important role. For instance, FMEA [58, 93, 127] defines a systematic framework to deal with safety requirements. A system under consideration is analysed component by component in order to obtain a set of failure modes as well as detection and recovery actions. However, the representation of the desired (safety) requirements in a formal system model is not a straightforward task. Despite the fact that the safety analysis techniques give us a certain structure of safety requirements, they are still mostly defined in the natural language. Therefore, additional research efforts are needed to translate informally given requirements into the underlying formal language for further verification. This leads to our first research question **RQ1**:

How to ensure a seamless integration of safety requirements into formal development?

Approach: In general, this question can be addressed in several ways. For example, the safety requirements structured according to a particular used safety analysis technique can be incorporated into a formal model by establishing the mapping between the structural blocks imposed by the technique and the formal model elements. Moreover, safety requirements informally expressed in the natural language can be translated into a formal language by defining the correspondence between natural language words or phrases and the respective elements of the formal language.

In this thesis, we firstly propose an approach to direct representation of safety analysis results, specifically FMEA results, within a formal model. We achieve it by defining a set of *patterns*, i.e., generic solutions for certain typical problems, and automate their application. These generic patterns play the role of an intermediary between the informal description of safety requirements derived from FMEA and their formal representation. A similar approach can be applied to different types of computer-based systems. In particular, since operational modes are a commonly used mechanism for structuring the system behaviour, we demonstrate how FMEA results can be used to derive the fault tolerance procedures for mode-rich systems, i.e., the systems whose dynamic behaviour is defined in terms of modes and transitions between these modes.

Secondly, we propose an approach to incorporating safety requirements represented in the natural language into a formal model by classifying them and defining a specific *mapping function* for each class. This function establishes a link between the given safety requirements and a set of the related formal model expressions constructed from formal model elements (e.g., axioms, variables, invariants, etc.). This allows us to give the requirements the formal semantics of the underlying formal language and also facilitate their traceability.

We validate the proposed approaches by several case studies, specifically, a sluice gate control system, an attitude and orbit control system, and a steam boiler control system.

Facilitating Formal Development and Verification. The requirements phase is followed by the *design phase* in SDLC. In the context of this thesis, this phase is associated with the formal development of a system by refinement. According to this approach, the system development starts from an abstract system model, which is then gradually elaborated by unfolding system implementation details in a number of model transformation steps. This approach allows us to deal with the system complexity and to develop the system correct-by-construction. Moreover, refinement supports an iterative development and feedback by verification. In this way, contradicting system requirements can be detected.

Despite clear benefits of using formal methods in SDLC as fault prevention techniques, development of formal models is a labour-intensive and time consuming task. The use of *formal development patterns*, i.e., generic reusable models devoted to formalise typical problems [5], may simplify this task and reduce the overall development time. To enhance formal development of safety-critical systems, in our second research question **RQ2**, we address the problem of derivation and reuse of patterns:

How to facilitate formal development and verification of safety-critical systems using specification and refinement patterns?

Approach: We follow the system approach [69] to model a computer-based system together with its environment. In addition to relying on stepwise refinement to deal with the system complexity, in this thesis we propose specification and refinement patterns that cover an abstract model of a system as well as specific model transformations expressed as generic refinement steps. The refinement steps introduce details of the nominal behaviour as well as the error detection and error handling procedures needed to improve safety of the modelled system. The proposed specification and refinement patterns have a number of generic parameters (constants, functions, etc.), which allows us to instantiate the patterns for a class of suitable systems.

Our patterns cover different types of computer system architectures, namely, *centralised* and *distributed*. A centralised system is a system that runs computations on a single computer, while a distributed system is a system where computations are allocated to several computers to be processed. In this thesis, we consider *control* and *monitoring systems* as examples of such centralised and distributed systems. A control system is a system in which a controller analyses the state of the environment by reading sensors and directly affects it by commanding actuators. Malfunctioning of the controller may lead to a hazardous situation. A monitoring system is a system that analyses a state of the environment and provides data to (human) operators. It does not directly command actuators. However, an operator can make a decision based on incorrect data, which may cause a hazard.

The proposed specification and refinement patterns have been validated by the following case studies: a sluice gate control system, a steam boiler control system, and a temperature monitoring system.

Assuring Safety. Exploitation of safety-critical systems is not possible without their prior *certification*. A *certificate* accommodates all the necessary information for an independent assessment of the required properties of a system [46]. Construction of a *safety case* for assuring safety of a system with a high level of criticality has become a strongly recommended part of the certification process [139]. A safety case is a structured argument providing justification that a system is acceptably safe for a certain application in a certain environment [25, 85]. To tackle the complexity of building a system safety case, *argument patterns* (or *safety case patterns*) have been proposed. Argument patterns are commonly used structures intended to assist in construction of a safety case [86]. A safety case aims at demonstrating (relying on the available evidence) that the system is safe. The evidence may comprise the results of analysis, testing, simulation as well as formal verification. Nonetheless, the use of formal verification results as the evidence in safety cases is currently limited. A possible reason for this is a small number of well-defined methods for using formal development and verification results in safety cases. Therefore, our third research question **RQ3** is

How to support the safety assurance process with the artefacts generated by formal development and verification?

Approach: To address this question, we propose an approach to rigorous construction of structured safety cases from formal models. It guides the developers starting from the informal representation of safety requirements to building the corresponding parts of safety cases via formal modelling and verification in Event-B and the accompanying toolsets. The proposed approach is supported by a set of argument patterns. These patterns are graphically represented by means of *Goal Structuring Notation (GSN)* [61, 85]. We base our argument patterns on our earlier defined classification of safety requirements and their mapping into the respective elements of formal models.

Additionally, in the scope of this approach, we propose the argument pattern for demonstrating that the formal system models themselves are well-defined. Specifically, one is needed to argue that all the models in the development do not contain logical inconsistencies and there are no infeasible mathematical definitions of model elements.

We validate the proposed approach and demonstrate instantiation of the argument patterns by series of small case studies as well as a larger case study – a steam boiler control system.

Overview of the General Approach. Figure 1.1 gives an overview of the approach proposed in this thesis. There are several steps associated with this approach: (1) elicitation and formalisation of informally defined safety requirements of critical systems, (2) formal development of such systems and formal verification of the associated safety properties, as well as (3) demonstration that the imposed safety requirements are met. These steps are

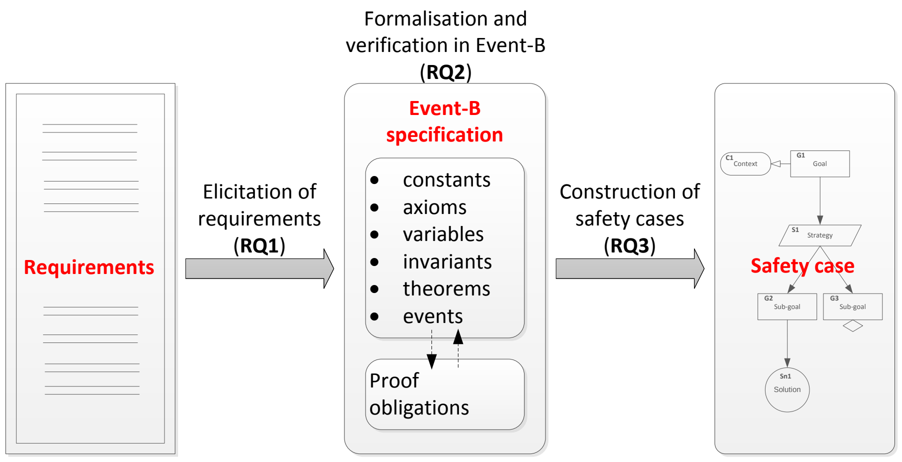


Figure 1.1: Overall picture of the approach proposed in the thesis

respectively reflected by the considered research questions **RQ1**–**RQ3**. The research questions complement each other and result in an integrated approach to the formal development and verification of safety-critical systems. Based on the obtained validation results, we can claim that the proposed approach has demonstrated good scalability, i.e., its capability to rigorously assure safety of systems with a high degree of complexity.

1.3 Organization of the Thesis

This thesis consists of two parts. In Part I, we overview the research presented in the thesis. In Part II, we include reprints of the original publications. The research summary is structured as follows. In Chapter 2, we overview the dependability concepts, briefly describe the safety analysis techniques utilised in the thesis, and outline rigorous approaches to development and verification of critical systems. Moreover, we introduce an adopted formalism, the Event-B method, and overview safety cases and their supporting graphical notation. Chapter 3 elaborates on the main contributions of this thesis by focusing on our proposed methods for integrating safety requirements into formal models of critical systems, approaches that aim at simplifying and guiding formal development and verification of these systems, as well as techniques towards utilising formal development and verification results in safety cases. In Chapter 4, we give a detailed description of the original publications included in the thesis. Chapter 5 contains an overview of related work in the field of formal development of safety-critical systems. Finally, in Chapter 6, we give the concluding remarks as well as discuss future research directions.

Chapter 2

Background

In this chapter, we outline the system dependability concept focusing on safety aspect. We also overview formal development and verification approaches and briefly introduce Event-B. Finally, we describe safety cases and their graphical notation.

2.1 Dependability and Safety: Definitions and Taxonomy

The concept of system dependability was proposed by Laprie [89] and enhanced by Avižienis et al. [12, 13]. They define *dependability* as the ability of a system to deliver services that can be justifiably trusted. This property comprises several attributes such as *safety*, *reliability*, *availability*, etc. Nowadays, dependability has been extended to address openness and inevitable changes (both internal and external) that modern systems have to cope with. The ability of a system to remain dependable despite changes is called *resilience* [90, 91]. In this thesis, we focus on safety aspects of critical computer-based systems. *Safety* is the ability of a system to operate without causing catastrophic consequences for its users and environment [13, 127].

The dependability taxonomy introduces threats and means for dependability. The threats are *faults*, *errors* and *failures*. A *fault* is a defect in a system. It can be either a software bug (a development fault), a hardware imperfection (a physical fault), or an incorrect user action (an interaction fault). A fault may cause an error. An *error* is a system state deviating from the correct system state and appearing at runtime. In its turn, an error may lead to a failure if it has not been detected and handled by a system. A *failure* is an event that occurs when a system deviates from its specification.

The means to deal with these threats are *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting* techniques [12, 13]. More specifically, the *fault prevention* and *fault removal* methods aim at avoiding introduction or reducing a number of faults in a system at the stage of its development, i.e., before the system is put into operation. Typically, they include various testing techniques as well as the methods for rigorous development, verification and validation such as formal methods and model checking.

The *fault tolerance* techniques allow a system to continue its intended operation in the presence of faults. There are two stages associated with fault tolerance – *error detection* and *system recovery* [13]. Error detection is a process of identifying the presence of errors in a system. Then, when errors have been detected, the system performs recovery by eliminating errors from the system state (i.e., *error handling*) and preventing faults from reappearing (i.e., *fault handling*). Typically, error handling is performed using the *rollback*, *rollforward* procedures (a system is put either in the previous or the next state without the detected errors), or *compensation* procedures (a system remains in the current state but errors are masked using, e.g., redundancy). In its turn, fault handling is achieved by, e.g., *isolation* (logical exclusion of faulty components) or *reconfiguration* (switching to spare components). Finally, the *fault forecasting* techniques aim at estimating the number of faults and impact of these faults on the system behaviour.

To assure that a critical system provides an acceptable level of safety, different safety analysis techniques can be conducted. Next, we overview the techniques we rely upon in this thesis.

2.2 Techniques for Safety Analysis

Safety analysis aims at identifying risks and hazards associated with a system, possible causes of these hazards, as well as the methods to cope with them. The results of safety analysis allow us to define the safety requirements to be imposed on a system.

There is a wide range of safety analysis techniques adopted in the model-based development process [84, 130]. Among them, Event Tree Analysis (ETA) [9], Failure Modes and Effects Analysis (FMEA) [58, 93, 127], Fault Tree Analysis (FTA) [127, 136], Functional Hazard Analysis (FHA) [140], HAZard and OPerability analysis (HAZOP) [127], Preliminary Hazard Analysis (PHA) [127], etc., should be mentioned. It is often the case that several techniques are combined in order to cover different types of safety requirements. For instance, according to the Society of Automotive Engineers (SAE) International Aerospace Recommended Practice (ARP) 4754A and 4761, FHA is advised to be conducted to identify hazards and classify them depending on their consequences. This classification typically produces specific kinds of safety requirements, e.g., about avoidance of catastrophic failure conditions. Safety analysis techniques such as FMEA and FTA can then be used to derive additional safety requirements.

FMEA allows us to analyse complex systems by proceeding hierarchically over their structure (components). It permits us to perform the analysis of subsystems as well as of individual system components. FMEA tables are very useful to describe system and component failures in detail. However, FMEA is typically limited to consideration of single failures at a time. This necessitates the use of complementary safety analysis techniques, e.g., FTA, to identify and analyse combinations of failures and generic (common cause) failures. The results of FMEA grouped together as single failure modes with the same failure effect correspond to basic events in FTA.

Combined application of a bottom-up safety analysis method, such as FMEA, and a top-down method, such as FTA, is recommended by the railway domain standard EN 50129 [54]. The goal is to assure that no single (hardware) component failure mode is hazardous.

In our work, we rely on FMEA and FTA to systematically derive the safety and fault tolerance requirements for critical computer-based systems. Therefore, in this section, we overview both FMEA and FTA.

Failure Modes and Effects Analysis (FMEA) [58, 93, 127] is an inductive, bottom-up safety analysis technique. For each item in a system, e.g., a system function or a component, it identifies possible failure modes, failure causes, local and system effects, as well as detection procedures and recovery actions. The corresponding information is typically collected in a tabular form.

FMEA is a technique which can be performed at different levels, e.g., system level (focusing on global functions) or component level (examining functions of each component separately). As a result, the content of a FMEA worksheet may vary. For example, among other things, it may take into account the probability of failure occurrence, severity, risk level, etc. The variation of FMEA that takes into account the criticality of a failure mode is called FMECA. The criticality is a relative measure of the consequences of a failure mode and the frequency of its occurrence [102]. Since the focus of our work is logical modelling and verification, we leave the quantitative system aspects aside. Therefore, we utilise only the core fields of a FMEA worksheet for our purposes. These fields and their descriptions are given in Figure 2.1.

However, FMEA does not allow us to analyse multiple and common cause failures and their influence on the overall safety of the system. Therefore, we need to conduct safety analysis of a critical system by relying on several techniques. For example, we can complement FMEA by FTA.

Fault Tree Analysis (FTA) [127, 136] is a deductive, graphical top-down method for safety analysis of both hardware and software systems. A fault tree starts with an event directly related to the identified hazard and works

Item (component)	Name of a component
Failure mode	Potential failure modes
Possible cause	The most probable causes associated with the assumed failure mode
Local effects	Caused changes in the component behaviour
System effects	Caused changes in the system behaviour
Detection	A description of the methods by which occurrence of the failure mode is detected
Remedial action	Actions to tolerate the failure

Figure 2.1: FMEA worksheet

backwards to identify the chain of events and their interplay that lead to the occurrence of the top event. FTA relies on boolean logic to express relations between events. The FTA results are represented as a tree structure (Figure 2.2). More details on the graphical representation of tree elements can be found in [59, 93, 127]. A set of basic events, the simultaneous occurrence of which would cause the occurrence of the top event, is called a *cut set*. A *minimal cut set* is the least set of basic events that lead to the top event. Examination of the obtained minimal cut sets allows us to identify the weak points in the design of a system as well as recognise single point failures, i.e., when the occurrence of a single basic event causes the immediate occurrence of the main hazard.

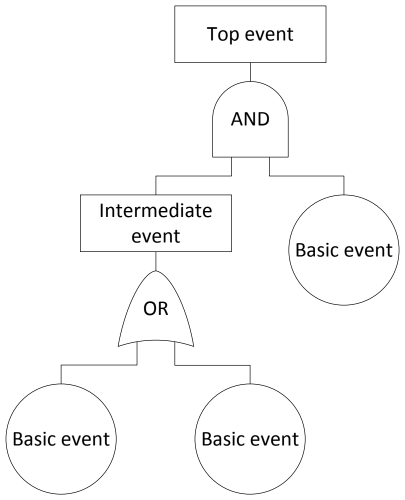


Figure 2.2: Fault tree structure

Analysis of the resulting fault trees determines whether a system is sufficiently safe, i.e., the intended safety criteria are met. If it is not the case, the system design needs to be improved and the system is analysed again. This process can be repeated several times until the safety criteria are met and we can conclude that the system is sufficiently safe.

The safety analysis techniques is a valuable source of safety requirements. To verify that the system fulfils these requirements, we need rigorous techniques such as formal methods. In the following two sections, we overview formal development and verification approaches as well as present the Event-B method adopted in this thesis.

2.3 Model-Based Development Using Formal Methods

Application of formal methods is an evolving direction in the model-based engineering [19, 27]. *Formal methods* are mathematically-based techniques for system modelling and verification. They rely on formal specification languages, i.e., languages with rigorously defined semantics, to create a model of a system. Then, the obtained formal model can be used for analysis and verification of system properties as well as code generation, static verification of existing code or generation of test cases. In this thesis, we employ a specific formal method (Event-B) to develop a system model which satisfies the safety requirements imposed on this system.

Abadi and Lamport [1] have distinguished two categories of formal specification methods: *state-based* and *action-based*. System models built using the state-based formalisms are viewed as state transition systems. It means that a modelled system formally represents an abstract machine that consists of a distinguishable set of states and a set of transitions between these states. A state typically corresponds to a vector of values assigned to the model variables. The examples of such formalisms are VDM [135], the Z notation [126], Action Systems [16, 17], the B Method [2], and Event-B [3], etc. System models in action-based formalisms can be seen as a sequence of actions where an action is defined as a state change. These formalisms include process algebraic methods such as Communicating Sequential Processes (CSP) [72], π-calculus [103], ambient calculus [124], etc. They are used for high-level specification of interactions, communications, and synchronisations between independent processes in cryptographic protocols, business processes and molecular biology.

Availability of tool support plays an important role in the choice of a formal method to be employed for the system formal development and verification. Typically, such tools not only perform syntactical checks of a system model but also assist in verifying the desired system properties by the means of *theorem proving* or *model checking*.

Theorem proving is a verification technique used to ensure that a model satisfies the desired system properties by proving the obtained proof obligations without model execution or simulation. As a result, the full model state space is explored with respect to the specified properties. There are two types of theorem provers: *general purpose* such as ACL2 [6], HOL [73], or Isabelle/HOL [105] and *specialised* (integrated with a specific formal development framework) such as the integrated theorem provers for Event-B and VDM.

Model checking [35] is a technique for verifying finite-state systems. Properties of such systems are expressed as temporal logic formulas defining constraints over the system dynamics. Similarly to theorem provers, model checkers can be either stand-alone, e.g., NuSMV [107], SPIN [74], Uppaal [22], etc., or provided as a part of a development framework, e.g., a plug-in for the Rodin platform enabling model checking in Event-B [113].

In our work, we have chosen the Event-B formalism [3] due to several reasons. First, it supports modelling of a system together with its environment. Consequently, it allows us to verify the system-level properties such as safety. Second, modern safety-critical systems are complex, hence their models are also complex and large in size. To address complexity, Event-B supports the stepwise refinement approach. It allows us to gradually introduce the details of system behaviour as well as verify system properties at different levels of abstraction. Third, the refinement steps preserve correctness leading to the correct-by-construction system development. Finally, Event-B has a mature tool support – the Rodin platform [56]. This tool allows for automatic model transformations, generation of proof obligations and has a user-friendly interface for discharging the generated proof obligations utilising both automatic and interactive provers. Additionally, the platform enables animation, model checking, and code generation with the help of the accompanying plug-ins.

Industrial applications of formal methods. During the last two decades, formal methods have been successfully applied to development and verification of systems from various domains. Experience in the practical use of formal methods is summarised in [26, 141]. For instance, Woodcock et al. [141] have surveyed a number of projects and concluded that the use of formal methods allows the developers to improve quality of the final products. The authors also confirm scalability of these methods for industrial applications.

The most famous examples from the railway domain are the application of the B Method by Alstom – Automatic Train Protection for the French railway company (SNCF) in 1993, Siemens Transportation Systems – Automatic Train Control for the driverless metro line 14 in Paris (RATP) in 1998, and ClearSy – Section Automatic Pilot for the light driverless shuttle for Paris-Roissy airport (ADP) in 2006 [36]. Moreover, Alstom and Siemens are currently reusing their B models to develop new products worldwide.

Another examples from avionics include the application of VDM to modelling and verification of a part of the UK air traffic management system, the Central Control Function (CCF) Display Information System, in 1996

[67] and more recent use of SCADE Suite [123] to develop the Airbus A380 control and display system [23, 53].

Finally, a success story on the use of formal methods for development and verification of a medical device, a cardiac pacemaker, is reported in [62]. The adopted formalism in this study is the Z notation.

2.4 Event-B Method

Event-B [3, 56] is a formal method for development and verification of dependable reactive and distributed systems. It has evolved from the Action Systems formalism [16, 17] and the B Method [2]. The associated Rodin platform [56], together with the accompanying plug-ins [122], provides a mature tool support for automated modelling and verification of critical computer-based systems.

An Event-B model can be defined by a tuple $(d, c, A, v, \Sigma, I, Init, E)$, where d represents sets, c stands for constants, v corresponds to a vector of model variables, Σ is a model state space determined by all possible values of the vector v. $A(d, c)$ is a conjunction of axioms, while $I(d, c, v)$ is a conjunction of invariants. *Init* is an non-empty set of model initial states such that $Init \subseteq \Sigma$. Finally, E is a set of model *events*.

Event-B uses the *abstract machine notation* [3], which consists of the static and dynamic parts called CONTEXT and MACHINE respectively. On the one hand, the sets (d) and constants (c) of a modelled system are defined in the CONTEXT components, where their properties are postulated as axioms (A). On the other hand, the model variables (v), invariants (I) and events (E), including the initialisation event (*Initialisation*), are introduced in the MACHINE components. The model variables are strongly typed by constraining predicates stated as system invariants. A MACHINE component can be connected to one or several CONTEXT components by a binding relation called *Sees*. A generalised representation of the Event-B components is shown in Figure 2.3.

Each event $e \in E$ may have the following form:

$$e \mathrel{\hat{=}} \textbf{any } lv \textbf{ where } g \textbf{ then } R \textbf{ end},$$

where lv is a list of local variables, the guard g is a conjunction of predicates defined over the model variables, and the action R is a parallel composition of assignments over the model variables.

The guard of an event defines when this event is enabled. In case when several events are enabled simultaneously, any of them can be chosen for execution non-deterministically. If there is no enabled event, the system deadlocks. There are two types of assignments over the variables: deterministic and non-deterministic. A deterministic assignment is expressed as $x := Expr(v)$, where x is a state variable and $Expr(v)$ is an expression over

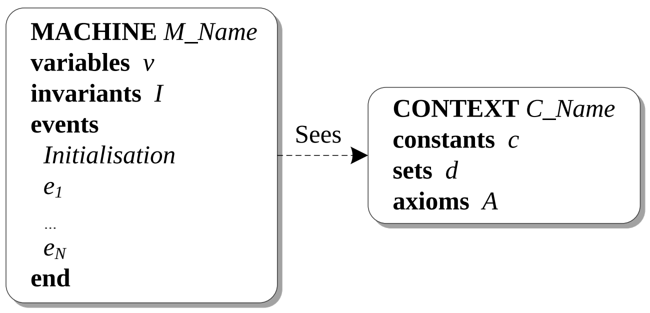


Figure 2.3: General representation of Event-B MACHINE and CONTEXT

the state variables v. A non-deterministic assignment can be denoted as $x :\in S$ or $x :\mid Q(v, x')$, where S is a set of values and $Q(v, x')$ is a predicate. As a result of a non-deterministic assignment, x gets any value from S or it obtains a value x' satisfying $Q(v, x')$.

The Event-B language can also be extended by different kinds of *attributes* attached to model events, guards, variables, etc. We will use Event-B attributes to contain formulas or expressions to be used by external tools or Rodin plug-ins, e.g., *Linear Temporal Logic (LTL)* formulas to be checked.

Semantically Event-B events are defined using *before-after predicates (BAs)* [3, 75, 101]. BAs declare the relationships between the system states before and after execution of events. Hence, the semantic definition of an event presented above can be given as the relation describing the corresponding state transformation from v to v', such that:

$$e(v, v') = g_e(v) \wedge I(v) \wedge BA_e(v, v'),$$

where g_e is the guard of the event e, BA_e is the before-after predicate of this event, and v, v' are the system states before and after event execution respectively.

Refinement in Event-B. The main development methodology of Event-B is top-down *refinement* (Figure 2.4). It is a process of transforming an abstract specification of a system (i.e., an abstract model) into a detailed system model via stepwise unfolding of implementation details (i.e., refinement steps) yet preserving its correctness. We say that a more detailed model *Refines* a more abstract one.

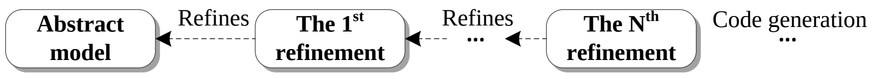


Figure 2.4: Event-B refinement

To preserve a validity of a refinement step, every possible execution of the refined model must correspond to some execution of the abstract model.

The refinement-based development allows us to reduce non-determinism of an abstract model as well as introduce new concrete variables and events into the model. This type of a refinement is known as a *superposition refinement.* Additionally, Event-B supports *data refinement.* It allows us to replace some abstract variables with their concrete counterparts. In this case, a part of the invariant of a refined model, called *gluing invariant*, formally defines the relationship between the abstract and concrete variables.

Furthermore, the most detailed model obtained during the refinement-based development can be used for code generation. Specifically, the Rodin platform [56], a tool supporting Event-B, permits for program code generation utilising a number of plug-ins. For example, EB2ALL [52], which is a set of translator plug-ins, allows for automatic generation of a target programming language code from an Event-B specification. In particular, EB2C allows for generation of C code, EB2C++ supports C++ code generation, using EB2J one can obtain Java code, and using EBC# – C# code.

The consistency (e.g., invariant preservation) and well-definedness of Event-B models as well as correctness of refinement steps are demonstrated by discharging the respective *proof obligations* (*POs*). The complete list of POs can be found in [3]. The Rodin platform [56] automatically generates the required POs and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. In general, the tool achieves a high level of automation (usually over 80%) in proving. Therefore, the Event-B method together with its mature tool support is attractive for both academic and industrial applications.

Event-B is highly suitable for safety assurance of highly critical systems due to its proof-based verification. However, to justifiably demonstrate that such systems can be safely operated, we utilise safety cases. Next, we describe in detail safety cases, their supporting graphical notation, as well as provide a small example illustrating the structural representation of a safety case.

2.5 Safety Cases

To assure dependability, security and safety of critical systems in a structured way, *assurance cases* have been proposed. An assurance case is *"a body of evidence organized into an argument demonstrating that some claim about a system holds, i.e., is assured"* [112]. The construction, review and acceptance of assurance cases are the key elements of safety assurance process for many industrial applications. Several standards request submission and evaluation of assurance cases, e.g., the automotive domain standard ISO 26262 [78], the railway domain standard EN 50128 [55], and the UK Defence Standard 00-56 [45]. We can distinguish three types of widely recognized assurance cases named accordingly to what property or attribute they allow to assure: *dependability cases (D-cases)* [40], *security cases* [112] and *safety cases* [25, 85].

In this thesis, we use *safety cases* as justification arguments for safety of critical computer-based systems. Therefore, here we give a detailed description of the safety cases and essential elements of their supporting graphical notation. A safety case is *"a structured argument, supported by a body of evidence that provides a convincing and valid case that a system is safe for a given application in a given operating environment"* [25, 45]. Essentially, a safety case includes the following elements [25]:

- *claim* – a statement defining a property of a system or a subsystem;
- *evidence* – facts, assumptions or sub-claims founding the basis of the safety argument;
- *argument* – a link between the evidence and the claim based on *inference* (transformation) rules.

To facilitate construction of safety cases, two main graphical notations have been proposed: *Claims, Arguments and Evidence (CAE)* notation [32] and *Goal Structuring Notation (GSN)* [61, 85]. In our work, we rely on the latter one due to its support for *argument patterns* (or *safety case patterns*), i.e., common structures capturing successful argument approaches that can be reused within a safety case [86].

GSN aims at graphical representation of the safety case elements and the relationships that declare a link between these elements. Figure 2.5 illustrates the elements of the notation. Typically, a safety case constructed using GSN consists of *goals*, *solutions* and *strategies* that correspond to *claims*, *evidence* and *arguments*. More specifically,

- *goals* are propositions in an argument that can be said to be true or false (e.g., claims of requirements to be met by a system);
- *solutions* contain references to the information obtained from analysis, testing or simulation of a system, i.e., the evidence used to demonstrate that goals have been met;
- *strategies* are reasoning steps describing how goals are decomposed and addressed by sub-goals.

In other words, a safety case constructed in GSN represents decomposition of goals into sub-goals following some strategies until the sub-goals can be supported by the direct evidence. It may also explicitly define relied *assumptions* and the *context* in which the goals and strategies are declared as well

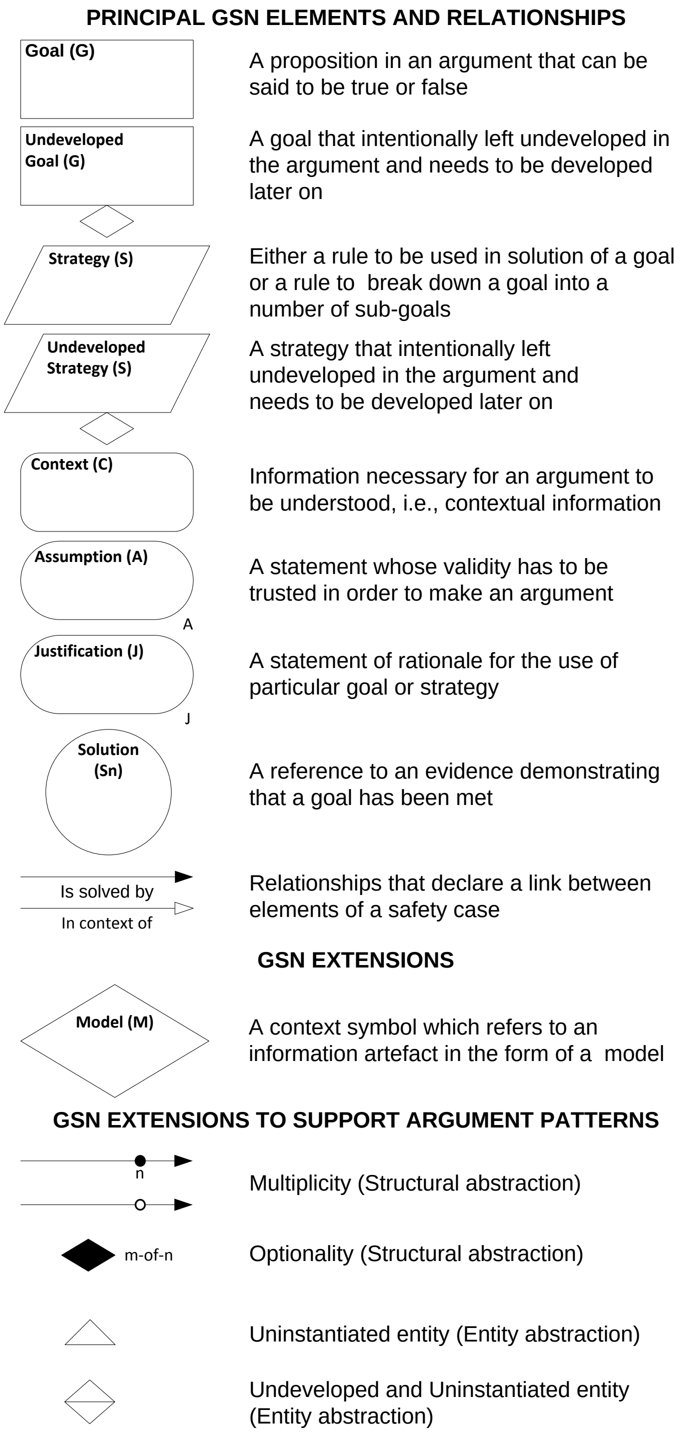

PRINCIPAL GSN ELEMENTS AND RELATIONSHIPS
Goal (G)
A proposition in an argument that can be said to be true or false
Undeveloped Goal (G)
A goal that intentionally left undeveloped in the argument and needs to be developed later on
Strategy (S)
Either a rule to be used in solution of a goal or a rule to break down a goal into a number of sub-goals
Undeveloped Strategy (S)
A strategy that intentionally left undeveloped in the argument and needs to be developed later on
Context (C)
Information necessary for an argument to be understood, i.e., contextual information
Assumption (A)
A
A statement whose validity has to be trusted in order to make an argument
Justification (J)
J
A statement of rationale for the use of particular goal or strategy
Solution (Sn)
A reference to an evidence demonstrating that a goal has been met
Is solved by
In context of
Relationships that declare a link between elements of a safety case
GSN EXTENSIONS
Model (M)
A context symbol which refers to an information artefact in the form of a model
GSN EXTENSIONS TO SUPPORT ARGUMENT PATTERNS
n
Multiplicity (Structural abstraction)
m-of-n
Optionality (Structural abstraction)
Uninstantiated entity (Entity abstraction)
Undeveloped and Uninstantiated entity (Entity abstraction)

Figure 2.5: Elements of GSN (detailed description is given in [20, 61, 85, 86])

as *justification* for the use of a particular goal or strategy. If the contextual information contains a model, a special GSN symbol called *model* can be used instead of a regular GSN context element.

There are two types of relationships present in a safety case:

- *"Is solved by"* – a type of relationships used between goals, strategies and solutions (evidential relationship);
- *"In context of"* – a type of relationships that links a goal to a context, a goal to an assumption, a goal to a justification, a strategy to a context, a strategy to an assumption, a strategy to a justification (contextual relationship).

The safety argument can be decomposed into two types of arguments (and the related evidence): a *direct* argument and a *backing* (confidence) argument [68, 119, 120]. The direct argument records the arguments and evidence used to establish direct claims of system safety. The backing argument justifies the sufficiency of confidence in the direct safety argument. The role of the backing argument is to explicitly address uncertainties present in the direct argument as well as to explain why there is sufficient confidence in this direct argument [68].

To permit construction of argument patterns, GSN has been extended by a number of elements for structural and entity abstraction [61, 85, 86]. In this thesis, we adopt the following *structural abstractions*: *multiplicity* and *optionality*. Multiplicity stands for generalised n-ary relationships between the GSN elements, while optionality corresponds to optional and alternative relationships between the GSN elements. Graphically, the former is depicted as a solid or a hollow ball on an arrow *"Is solved by"* (Figure 2.5), where the label n indicates the cardinality of a relationship, while a hollow ball means zero or one. The latter is represented as a solid diamond (Figure 2.5), where *m-of-n* denotes a possible number of alternatives. Additionally, the multiplicity and optionality relationships can be combined. Thereby, if the multiplicity symbol is placed in front of the optionality symbol, this corresponds to a multiplicity over all the options.

There exist two extensions to GSN used to represent *entity abstraction*: *uninstantiated entity* and *undeveloped and uninstantiated entity*. The former one indicates that the entity needs to be instantiated, i.e., at some later stage the "abstract" entity needs to be replaced with a more concrete instance. In Figure 2.5, the corresponding symbol is depicted as a hollow triangle. This annotation can be used with any GSN element. The latter one specifies that the entity requires both further development and instantiation. In Figure 2.5, it is shown as a hollow diamond with a line in the middle. This can be applied to GSN goals and strategies only.

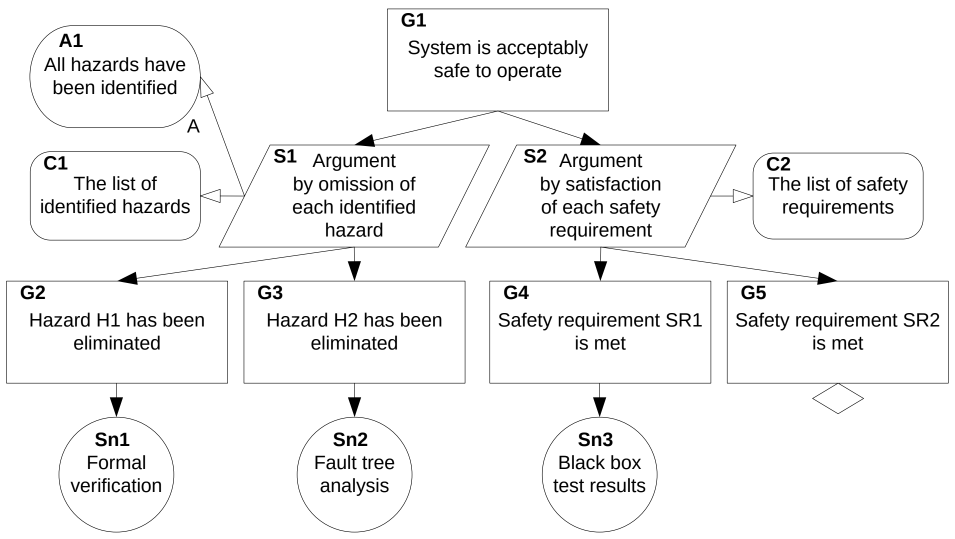

G1
System is acceptably safe to operate
A1
All hazards have been identified
A
C1
The list of identified hazards
S1 Argument by omission of each identified hazard
S2 Argument by satisfaction of each safety requirement
C2
The list of safety requirements
G2
Hazard H1 has been eliminated
G3
Hazard H2 has been eliminated
G4
Safety requirement SR1 is met
G5
Safety requirement SR2 is met
Sn1
Formal verification
Sn2
Fault tree analysis
Sn3
Black box test results

Figure 2.6: Structural representation of a safety case

Figure 2.6 illustrates a fragment of a safety case of a critical control system. This example is based on the safety case presented in [61, 66]. The top goal of the given safety case is *"**G1:** System is acceptably safe to operate"*. To provide the evidence that this goal holds, two subsequent strategies are applied: *"**S1:** Argument by omission of each identified hazard"* and *"**S2:** Argument by satisfaction of each safety requirement"*. They lead to obtaining a number of sub-goals, e.g., ***G2***, ***G3***, etc. The decomposition of sub-goals into even more detailed sub-goals can be continued until we derive a statement that can be directly supported by some evidence, e.g., formal verification results (***Sn1***), fault tree analysis results (***Sn2***), etc. In Figure 2.6, the sub-goal ***G5*** is left undeveloped meaning that one needs to further elaborate on it in order to support by the direct evidence.

Chapter 3

Safety-Driven Formal Development and Verification of Critical Systems

In this chapter, we explain how the research questions formulated in Section 1.2 are addressed in the thesis. Specifically, we present our approach to integrating safety analysis into rigorous development of safety-critical computer-based systems. We also discuss how formal development and verification of such systems can be facilitated by using pre-defined patterns. Finally, we describe our approach to constructing safety cases based on the results of formal verification.

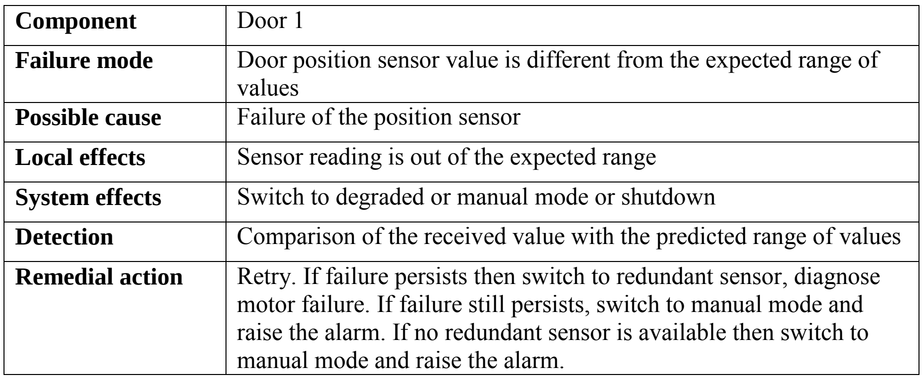
3.1 Incorporating Safety Requirements from Safety Analysis into Formal Development

Elicitation, i.e., extraction and identification, of safety requirements as well as their analysis is a complex task. Safety analysis usually associated with identifying hazards, their causes and measures to be taken to eliminate and mitigate them [93, 127]. In this thesis, we use FMEA and FTA to extract safety and fault tolerance requirements. To verify safety of critical computer-based systems, it should be demonstrated that the system adheres to the safety requirements. However, the task of faithful representation of safety requirements in a formal model for verification remains challenging. Therefore, to address our first research question – *How to ensure a seamless integration of safety requirements into formal development?*, we propose a couple of formally based techniques to facilitate this task. First, we show how to incorporate safety requirements derived from FMEA into formal system models. Second, for the safety requirements derived using different safety analysis techniques, e.g., FTA, and represented in the natural language, we introduce a requirements classification and establish a link between the classified requirements and the corresponding elements of formal models. In this section, we describe in detail the first part of the approach, while the proposed classification will be discussed in Section 3.3.

Incorporating FMEA Results into Formal Development. In general, safety-critical computer-based systems are heterogeneous systems that contain such basic building blocks as hardware and software components as well as various mechanisms to ensure their correct functioning and dependability. For example, control systems, a broad class of safety-critical systems, typically have the following building blocks: sensors, controllers, actuators, as well as the incorporated fault tolerance (e.g., error detection and error handling) mechanisms.

FMEA (e.g., Figure 3.1) allows us to analyse possible failure modes of the system components and define the required detection and recovery actions. The challenge is to find a proper way to incorporate the results of such analysis into formal models. Within Event-B, we represent system components (sensors and actuators) as well as the fault tolerance mechanisms using certain *model elements*, e.g., *constants*, *variables*, *invariants*, and *events*. Since the same types of components and the associated fault tolerance actions can be formalised in similar ways, we are able to identify generic patterns for their formalisation. We distinguish the following types of patterns:

1. *Component patterns:* allow for representation of system components such as sensors and actuators by creating new variables, invariants, and initialisation actions;
2. *Detection patterns:* represent generic mechanisms for error detection by specific events and condition checks (guards), e.g., checking for divergence between the actual and expected states of a component or sensor reading going beyond the working legitimate range;



Component	Door 1
Failure mode	Door position sensor value is different from the expected range of values
Possible cause	Failure of the position sensor
Local effects	Sensor reading is out of the expected range
System effects	Switch to degraded or manual mode or shutdown
Detection	Comparison of the received value with the predicted range of values
Remedial action	Retry. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Figure 3.1: Example of a FMEA table

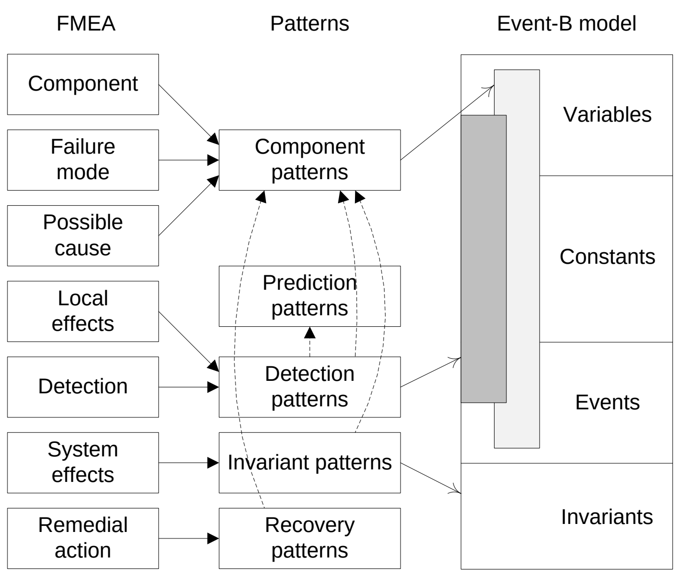


Figure 3.2: FMEA representation patterns

3. *Prediction patterns:* represent typical computations of the next possible states of components, e.g., computations based on the underlying physical system dynamics and evolution of processes;
4. *Recovery patterns:* include actions to tolerate various failures, e.g., representing multiple retries of required actions or computations, switching to spare components or safe shutdown;
5. *Invariant patterns:* represent safety and/or gluing invariants. The *safety invariants* define safety properties extracted from FMEA results, while the *gluing invariants* describe the correspondence between the states of refined and abstract models. These patterns are usually applied in combination with patterns of other types to define how a model transformation is related with the model invariant properties.

Each pattern describes how specific Event-B model elements are to be created to represent certain information from FMEA. Often the described patterns complement or rely on each other. Such interdependency and mapping to FMEA is schematically shown in Figure 3.2. For example, an application of a detection pattern can result in creating new constants and variables (reflected as a dark grey rectangle in Figure 3.2). Moreover, it may require an instantiation of a component pattern to create the elements it depends on (a light grey rectangle in Figure 3.2). To increase usability of the approach, automated instantiation of the proposed patterns have been implemented as a plug-in for the Rodin platform.

Let us now briefly illustrate an application of the proposed approach by a small example. The example is taken form the sluice gate case study [95].

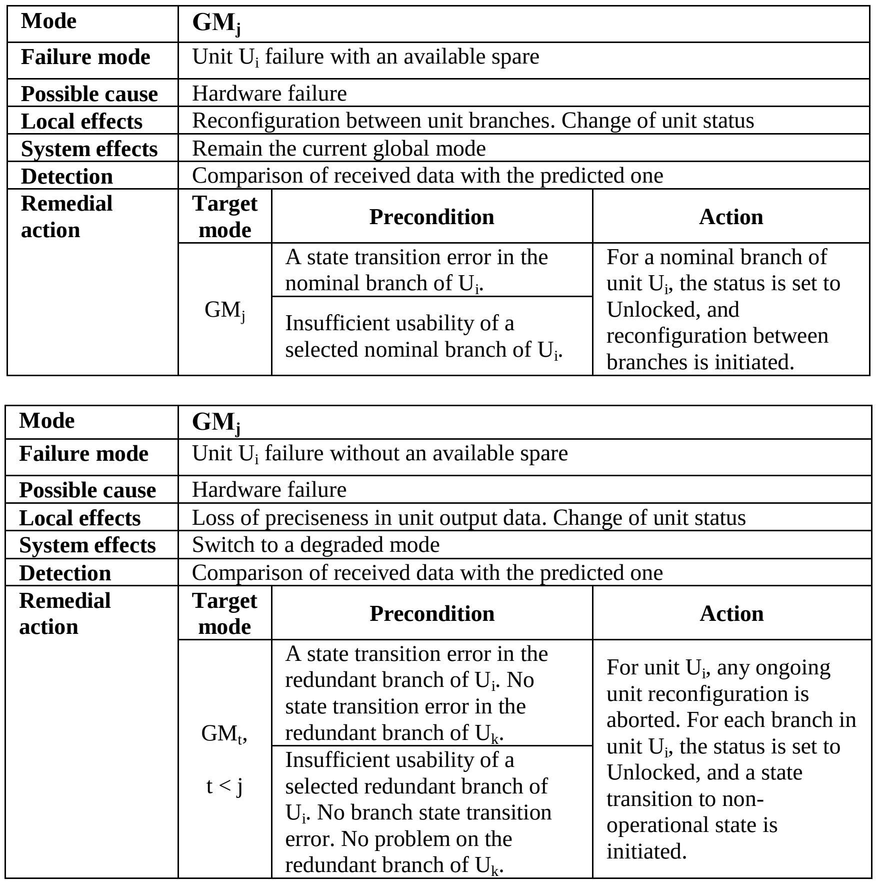
```
variables
    door1_position_sensor
    door1_fail
    door1_position_sensor_pred
    d1_exp_max
    d1_exp_min

invariants
    flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨
      door2_fail=TRUE ∨ pressure_fail=TRUE) // gluing
    flag≠CONT ⇒ (door1_fail=TRUE ⇔
      door1_position_sensor_abs=TRUE ∨
      door1_position_sensor_pred=TRUE ∨
      door1_opened_sensor_inconsistent=TRUE ∨
      door1_closed_sensor_inconsistent=TRUE) // gluing
    door1_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒
      Stop=TRUE // safety
    <other invariants>

<event INITIALISATION>

event Detection_Door1_checks
where
    flag = DET
    Stop = FALSE
then
    door1_position_sensor_pred ≔
      bool((door1_position_sensor < d1_exp_min ∨
      door1_position_sensor > d1_exp_max) ∧
      door1_sensor_disregard = FALSE)
    <other checks>
end
```

```
event RetryPosition
where
    flag = CONT
    door1_position_sensor_abs = TRUE ∨
    door1_position_sensor_pred = TRUE
    retry < 3
then
    door1_position_sensor_abs ≔ FALSE
    door1_position_sensor_pred ≔ FALSE
    door1_fail_masked ≔
      bool(door1_opened_sensor_inconsistent = TRUE ∨
      door1_closed_sensor_inconsistent = TRUE)
    retry ≔ retry + 1
end

<other recovery events>

event Prediction refines Prediction
where
    flag = PRED
    door1_fail = FALSE ∧ door2_fail = FALSE ∧
      pressure_fail = FALSE
    Stop = FALSE
then
    flag ≔ ENV
    d1_exp_min ≔
      min_door(door1_position↦door1_motor)
    d1_exp_max ≔
      max_door(door1_position↦door1_motor)
    <other predictions>
end
```

Figure 3.3: Event-B development of the sluice gate control system with example of *Retry recovery pattern*

The sluice system connects areas with dramatically different pressures by operating two doors (*Door 1* and *Door 2*). To guarantee safety, a door can be open only if the pressure in the locations it connects is equalised. Moreover, the doors should not be open simultaneously. Figure 3.1 shows the resulting FMEA table for one of the door components (*Door 1*). We demonstrate our formalisation based on a component with a value-type sensor used to determine the door position. The list of patterns to be instantiated for this table is as follows: (1) component patterns – *Value sensor pattern*, (2) detection patterns – *Expected range pattern*, (3) prediction patterns – *Range prediction pattern*, (4) recovery patterns – *Retry recovery pattern*, *Component redundancy recovery pattern*, and *Safe stop recovery pattern*, (5) invariant patterns – *Safety invariant pattern* and *Gluing invariant pattern*.

In Figure 3.3, we give an excerpt from the Event-B development of the sluice gate control system obtained by instantiating the listed above patterns. We highlight the model area affected by application of *Retry recovery pattern*. Let us also observe that, if two or more patterns affect the same variables, only the first pattern to be instantiated creates the required variables. This



Mode	GM_j		
Failure mode	Unit U_i failure with an available spare		
Possible cause	Hardware failure		
Local effects	Reconfiguration between unit branches. Change of unit status		
System effects	Remain the current global mode		
Detection	Comparison of received data with the predicted one		
Remedial action	Target mode	Precondition	Action
	GM_j	A state transition error in the nominal branch of U_i.	For a nominal branch of unit U_i, the status is set to Unlocked, and reconfiguration between branches is initiated.
		Insufficient usability of a selected nominal branch of U_i.	

Mode	GM_j		
Failure mode	Unit U_i failure without an available spare		
Possible cause	Hardware failure		
Local effects	Loss of preciseness in unit output data. Change of unit status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	Target mode	Precondition	Action
	GM_t, $t < j$	A state transition error in the redundant branch of U_i. No state transition error in the redundant branch of U_k.	For unit U_i, any ongoing unit reconfiguration is aborted. For each branch in unit U_i, the status is set to Unlocked, and a state transition to non-operational state is initiated.
		Insufficient usability of a selected redundant branch of U_i. No branch state transition error. No problem on the redundant branch of U_k.	

Figure 3.4: Modified FMEA table

process is not repeated by other patterns. The same rule applies to events, actions, guards, etc.

A similar approach can be applied to different types of systems. In this thesis, we have experimented with applying our FMEA-based approach to mode-rich safety-critical systems, i.e., the systems whose dynamic behaviour is defined in terms of operational modes and transitions between these modes. Here *modes* can be understood as mutually exclusive sets of the system behaviour [92]. Rules for transitioning between modes represent a *mode logic* of a system.

In the safety-critical systems, a part of the system mode logic is associated with specific system transitions as reactions on faults. Moreover, for reconfigurable systems, we cover two cases: (1) when a system reconfiguration is possible due to the availability of spare components; and (2) when a reconfiguration cannot be done due to the absence of spare components. Then, to implement fault tolerance, we need to define rollback procedures to the non-erroneous system states where the faulty components are not used.

```
event MM_Error_Handling refines MM_Error_Handling
  any m gp
  where
    m ∈ MODES
    error ≠ No_Error
    prev_targ ↦ m ∈ Mode_Order∼
    m ≠ next_targ
    next_targ ≠ OFF
    m ↦ gp ∈ GPS_mode
    gp ≠ GPS_next_targ
   <guards related to other AOCS units>
  then
    error ≔ No_Error
    prev_targ ≔ next_targ
    last_mode ≔ next_targ
    next_targ ≔ m
    GPS_Status ≔ fun_GPS_status(GPS_Reconfig ↦ GPS_next_targ ↦ gp)
    GPS_prev_targ ≔ GPS_next_targ
    GPS_last_mode ≔ GPS_next_targ
    GPS_next_targ ≔ gp
    GPS_Reconfig ≔ FALSE
    <actions related to other AOCS units>
 end
```

Figure 3.5: Excerpt from Event-B development of the attitude and orbit control system (AOCS) with example of *system rollback*

For this purpose, we analyse system modes rather than physical components. To allow for the safety analysis specific for mode-rich systems, we tailor a FMEA table by adding specific sub-fields into the field of remedial actions as shown in Figure 3.4. Specifically, the added sub-fields are *target mode*, *precondition*, and *action*. These sub-fields contain the information about a new target mode, the conditions under which a rollback to this mode should be started, and the actions to be taken, e.g., lower-level transitions or reconfiguration procedures. The safety analysis performed in such a way allows us to derive a *rollback scenario*.

The second example (Figure 3.5) is taken from the resulting Event-B specification of the AOCS system [116]. AOCS is a typical layered control system. The main function of the system is to control the attitude and the orbit of a satellite. Since the orientation of a satellite may change due to disturbances of the environment, the attitude needs to be continuously monitored and adjusted. The optimal attitude is required to support the needs of payload instruments and to fulfil the mission of the satellite [50]. At the top layer of AOCS is *Mode Manager (MM)*. It controls several *Unit Managers (UMs)*, which are responsible for a number of hardware units. AOCS has seven units. For brevity, let us consider only one of them, namely, *Global Positioning System (GPS)*.

We define the rollback procedures according to the results of the performed safety analysis represented in the form of the modified FMEA table (similar as in Figure 3.4). Let us assume that some error has occurred

($error \neq No\_Error$) but the reconfiguration is not possible, i.e., both the nominal and redundant branches failed. Then, the MM component performs a rollback to such a global mode where the failed component is not used, i.e., it is in the mode *Off*. To perform a rollback transition, all the AOCS units have to be set into appropriate modes according to the correspondence relations between the MM modes and each unit modes (e.g., *GPS_mode*). All the variables reflecting the current state of AOCS and all the units (such as *prev_targ*, *last_mode*, *next_targ*, etc.) have to be assigned new values. Once the resulting model is fully created according to the applied pattern(s), the essential mode consistency conditions are verified.

We believe that application of FMEA patterns helps the developers to elicit safety and fault tolerance requirements and also assists them in incorporating these requirements into formal Event-B models. Moreover, the developed plug-in for the Rodin platform facilitates the automatic instantiation of these patterns.

3.2 Facilitating Rigorous Development and Verification of Safety-Critical Systems

The use of *specification* and *refinement patterns* is an important technique in facilitating formal development. It is also know as *pattern-driven formal development* [76]. Graphically, the approach can be represented as shown in Figure 3.6. According to the pattern-driven formal development approach, the initial (abstract) model is created by instantiating a specification pattern, i.e., a parametrised specification that contains generic (abstract) types, constants and variables. Then, the obtained model can be refined by applying refinement patterns, i.e., patterns generalising certain typical model transformations reoccurring in a particular development method.

To instantiate the given specification and refinement patterns, the model parameters need to be replaced by concrete data structures or other model expressions (e.g., concrete variables, guards, etc.). To show applicability of patterns for the given concrete values, the model constraints defined for the parameters become theorems to be proved. Successful instantiation of these patterns allows for obtaining the proved essential system properties of a generic model (e.g., its invariants) for free, i.e., without any additional proof effort. Therefore, pattern-driven formal development supports reuse of both models and proofs.

Certain specification and refinement patterns can be defined for particular types of systems. To address the second research question – *How to facilitate formal development and verification of safety-critical systems using specification and refinement patterns?*, we consider two widespread types of safety-critical systems, namely *control* and *monitoring* systems, and conse-

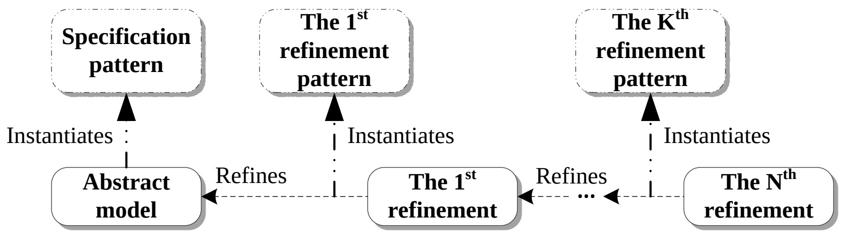


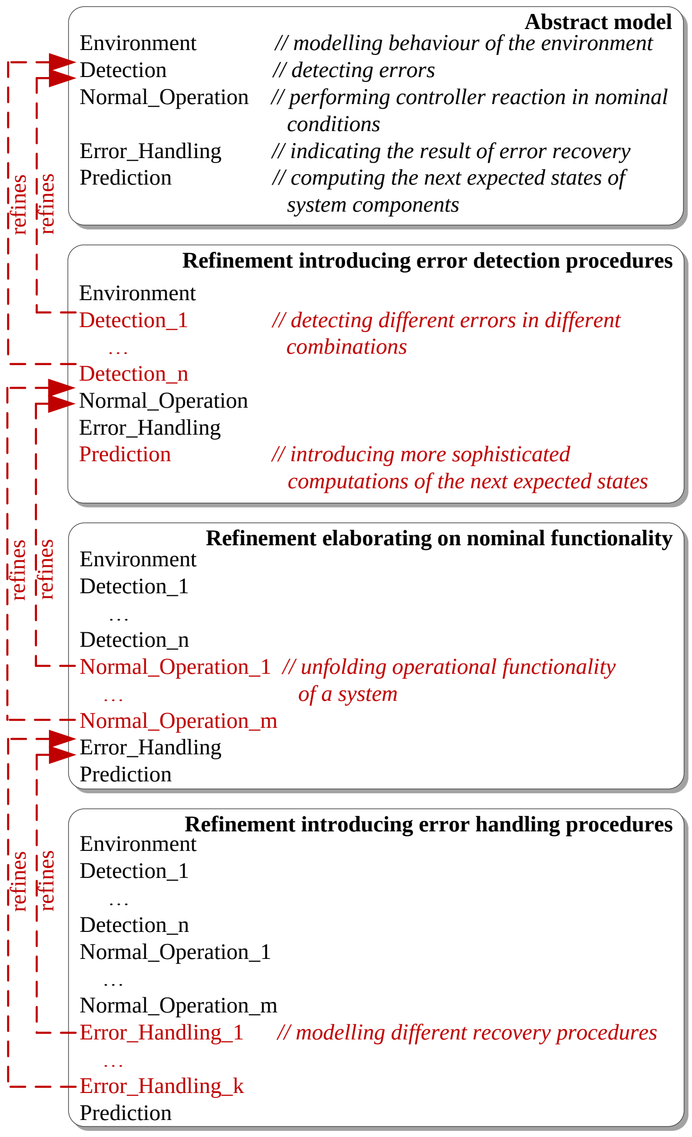
Figure 3.6: Pattern-driven formal development

quently propose a number of patterns to support the corresponding pattern-driven development of these systems.

Formal Development of Control Systems. *Control systems* are computer-based systems that observe the state of the environment by reading sensors and react on the detected changes by commanding actuators. Safety-critical control systems are in the heart of, e.g., nuclear reactor cooling systems and on-board aircraft systems. Typically, these systems are cyclic. Each cycle the controller reads sensors values, processes them and sends commands to the actuators. The mechanisms to detect and tolerate failures of a system and its components are identified during safety analysis (e.g., FMEA). They should be taken into account while developing the controller. Therefore, when a failure is detected, the controller attempts to tolerate it, for example, by managing a system reconfiguration to use spare components or by safe shutdown of the system.

To adequately represent the safety and fault tolerance properties while modelling the described above systems, we need to consider both the controller and the environment with which it interacts. We achieve this by relying on the system approach [69] to formal development and verification. As a result, we propose a number of specification and refinement patterns to support pattern-driven development of control systems. Our work has been also inspired by the work describing refinement of fault tolerant control systems in B [88]. We merge the ideas given in [88] and our FMEA patterns presented in Section 3.1. Specifically, the FMEA patterns can be used as separate steps in the proposed pattern-driven development. Therefore, our approach provides a useful support for formal development as well as facilitates traceability of the safety and fault tolerance requirements.

The proposed pattern-driven development of control systems consists of a specification pattern for the most abstract model as well as several refinement patterns for introducing details of the nominal system functionality, error detection and error handling procedures, as shown in Figure 3.7. According to our specification pattern, the *Abstract model* implements the cyclic behaviour of the modelled system. It has dedicated events for differ-



Abstract model
Environment // modelling behaviour of the environment
Detection // detecting errors
Normal_Operation // performing controller reaction in nominal conditions
Error_Handling // indicating the result of error recovery
Prediction // computing the next expected states of system components
Refinement introducing error detection procedures
Environment
Detection_1 // detecting different errors in different combinations
…
Detection_n
Normal_Operation
Error_Handling
Prediction // introducing more sophisticated computations of the next expected states
Refinement elaborating on nominal functionality
Environment
Detection_1
…
Detection_n
Normal_Operation_1 // unfolding operational functionality of a system
…
Normal_Operation_m
Error_Handling
Prediction
Refinement introducing error handling procedures
Environment
Detection_1
…
Detection_n
Normal_Operation_1
…
Normal_Operation_m
Error_Handling_1 // modelling different recovery procedures
…
Error_Handling_k
Prediction
refines
refines
refines
refines
refines
refines

Figure 3.7: Pattern-driven development of a control system

ent system phases: modelling the environment, error detection, the nominal operation of the system, possible error recovery, and error prediction (based on computations of the next expected states of system components to be used at the next cycle to detect possible failures of these components). The control flow between events is set by the variable *flag*. It represents a current phase of the control cycle, which can be either ENV, DET, $CONT$, or $PRED$. In the model invariant we declare the types of the variables and define the conditions when the system is operational or shut down.

The specified description of the system behaviour given in the initial model is still very abstract. In particular, the events designating the system phases are missing many essential details of their implementation. The refinement patterns allow us to elaborate on the system functionality by introducing concrete details such as system components and their possible failures, specific mechanisms to detect and tolerate these failures, as well as the corresponding invariant properties to be proved. Namely, the pattern *Refinement introducing error detection procedures* (Figure 3.7) introduces more sophisticated error detection procedures, while the pattern *Refinement elaborating on nominal functionality* allows for elaboration on the nominal functionality of the system and possibly introduces different operational modes. Finally, the refinement step based on application of the pattern *Refinement introducing error handling procedures* allows us to represent different error handling mechanisms. The correctness of model transformations is guaranteed by the discharged proof obligations.

We have validated the proposed approach by two case studies – a sluice gate control system [95] and a steam boiler control system [117]. This allowed us to obtain formal models of the considered systems and verify their essential functional and safety properties.

Formal Development of Monitoring Systems. *Data Monitoring Systems (DMSs)* constitute an important subset of safety-critical systems. They can be found in nuclear power plants, chemical, oil and gas industrial applications. DMSs allow for recording operations carried out by these systems in the corresponding log files, analysing performance and status of their components, as well as displaying the results of such analysis to human operators. DMSs also supply important information to human operators or other systems based on which the safety-critical actions are performed. Consequently, the operators must receive the correct, fresh and consistent information to be able to make adequate and timely decisions.

In this thesis, we propose pattern-driven development of monitoring systems to facilitate formal modelling of such systems and verifying their essential properties: *data freshness* and *data integrity*.

To increase fault tolerance, DMSs are typically built as distributed networks consisting of a number of modules, called *Data Processing Units (DPUs)*, connected to sensors and possibly several displays. We consider the displayed data to be *fresh*, if the difference between the local DPU time and the data timestamp is less than δ time units. In its turn, *data integrity* guarantees that the produced output data are always based on valid, i.e., correct and fresh, input data. In other words, it ensures that a failure of a sensor, a failure of a module, or data corruption (occurred while transmitting data through the network) does not influence the displayed output.

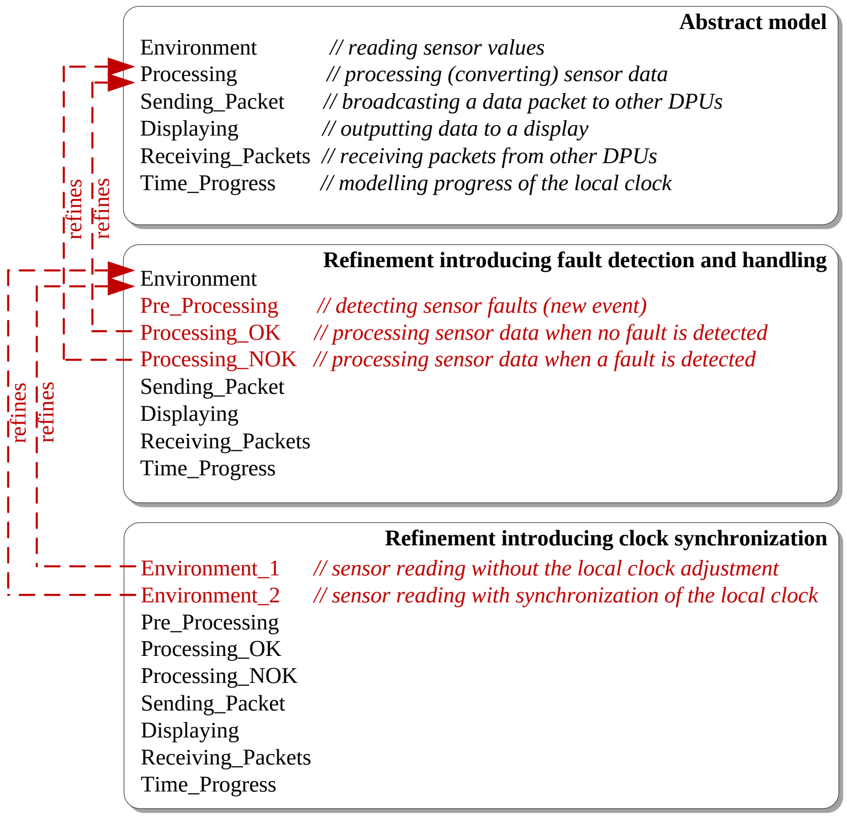

Abstract model
Environment // reading sensor values
Processing // processing (converting) sensor data
Sending_Packet // broadcasting a data packet to other DPUs
Displaying // outputting data to a display
Receiving_Packets // receiving packets from other DPUs
Time_Progress // modelling progress of the local clock
refines
refines
Refinement introducing fault detection and handling
Environment
Pre_Processing // detecting sensor faults (new event)
Processing_OK // processing sensor data when no fault is detected
Processing_NOK // processing sensor data when a fault is detected
Sending_Packet
Displaying
Receiving_Packets
Time_Progress
refines
refines
Refinement introducing clock synchronization
Environment_1 // sensor reading without the local clock adjustment
Environment_2 // sensor reading with synchronization of the local clock
Pre_Processing
Processing_OK
Processing_NOK
Sending_Packet
Displaying
Receiving_Packets
Time_Progress

Figure 3.8: Pattern-driven development of a monitoring system

Due to the highly asynchronous nature and independence of modules in the proposed architecture, it is possible to reason about the overall system by modelling a single DPU, thus dramatically reducing modelling and verification effort. The interactions with other modules present in the system are defined as a part of the environment specification. We show the proposed specification and refinement patterns for a single DPU in Figure 3.8.

In the *Abstract model* (our specification pattern), we define the essential functional behaviour of the DPU. Similarly to the behaviour of a control system, the DPU's behaviour is cyclic, yet with several differences. At each cycle, the DPU reads and processes sensor data, broadcasts the processed data to the other DPUs present in the system, possibly receives data from them, and finally produces the value, which is then displayed. We model these activities by the corresponding dedicated events. Progress of the local clock is modelled as a separate event. The interaction of the modelled DPU with other DPUs, is represented by asynchronous reception of data packets from these modules. Each data packet contains the identification number of the module that sent it, a timestamp and actual data. At this stage, we are able to formulate and verify the data freshness and correctness properties as the respective model invariants.

By instantiating the proposed pattern *Refinement introducing fault detection and handling*, we refine the abstract model to incorporate fault tolerance mechanisms into the development (Figure 3.8). Here we explicitly specify the effect of three types of failures: sensor failures, sensor data processing failures, and communication errors. Additionally, we extend a data packet structure by the following fields: a field containing the information about the status (absence or presence of a failure) of a DPU that sent the packet, and a field storing a checksum used to decide whether the packet was corrupted during the transmission or not. Now we can formulate and verify the data integrity property by defining the corresponding invariants.

The aim of another refinement pattern called *Refinement introducing clock synchronization* is to refine the mechanism of local clock adjustment. Every k cycles, the DPU receives the reference time signal and adjusts its local clock according to it. This prevents an unbounded local clock drift and allows the overall system guarantee the "global" data freshness. We again verify this by proving the corresponding theorems.

In general, the idea of pattern-driven development is applicable for wide range of different systems. The presented patterns for control and monitoring systems can be already used for a class of suitable systems by instantiating generic parameters (constants, functions, etc.). For instance, a number and types of sensors, DPUs, displays, etc. may vary depending on a particular system. The presented collection of patterns can be further extended by introducing separate patterns for different error detection and handling procedures of systems with different levels of redundancy, modelling fail-safe or fail-operational systems, etc.

3.3 Constructing Safety Cases from Artefacts of Formal Development and Verification

The described above approaches allow us to formally develop safety-critical systems, verify their safety-related properties and obtain formal proofs that these properties hold. To exploit the benefits of formal methods in the safety assurance process, we need to demonstrate how the obtained formal proofs of the required safety-related properties can be used as the evidence in safety cases. Therefore, to tackle our third research question – *How to support the safety assurance process with the artefacts generated by formal development and verification?*, we propose an approach to construction of safety cases based on the results of formal verification. To achieve this, we introduce a classification of safety requirements associated with specific ways these requirements can be mapped into formal system models in Event-B. Moreover, we propose a set of argument patterns based on the proposed classification to facilitate the construction of safety cases from the associated Event-B models. The proposed classification also contributes to addressing the first research question of this thesis.

Classification of Safety Requirements. Our classification of safety requirements addresses different aspects of the system behaviour. For instance, we may consider whether a requirement stipulates a property that needs to hold during the entire operation of a system or only under specific conditions. In general, we distinguish eight classes of safety requirements (*SRs*):

- *Class 1*: *SRs about global properties* represent properties that must hold in all system states;
- *Class 2*: *SRs about local properties* define properties that need to be true at specific system states;
- *Class 3*: *SRs about control flow* prescribe the required order of some system events/actions;
- *Class 4*: *SRs about the absence of system deadlock* prohibit an unexpected stop of a system, which may lead to a safety-related hazard;
- *Class 5*: *SRs about system termination* are the requirements related to a certain class of control systems where non-termination of the system in a specific situation may lead to a safety-related hazard;
- *Class 6*: *Hierarchical SRs* are the requirements that are hierarchically structured to deal with the system complexity, i.e., a more general requirement may be decomposed into several more detailed ones;
- *Class 7*: *SRs about temporal properties* represent properties related to reachability of specific system states;
- *Class 8*: *SRs about timing properties* prescribe the timing constraints to be imposed on a system.

Mapping into Formal Development. To map safety requirements of different classes into the formal development in Event-B, we define the function F_M:

$$F_M\text{: } SRs \rightarrow \mathcal{P}(MExpr),$$

where $\mathcal{P}(T)$ stands for a power set on elements of T and *MExpr* corresponds to a generalised type for all possible expressions that can be built from the model elements, i.e., *model expressions.* Here *model elements* are elements of Event-B models such as *axioms*, *variables*, *invariants*, *events*, and *attributes.* *MExpr* includes such model elements as trivial (basic) expressions. It also may include *state predicates* defining post-conditions and shutdown conditions, as well as *event control flow*, *Linear Temporal Logic (LTL)* and *Timed Computation Tree Logic (TCTL)* formulas.

Within Event-B models, most of the constructed expressions can be directly associated with specific proof obligations (theorems). The discharged proof obligations then may serve as the evidence in a constructed safety case. For the remaining expressions, we use external tools that can be bridged with Event-B to verify the corresponding requirements and thus construct the evidence for a safety case. In particular, we propose to use the Usecase/Flow [57] and ProB [113] plug-ins for the Rodin platform, as well as the external model checker for verification of real-time systems Uppaal [22]. As a result, the obtained verification results either in the form of the discharged proof obligations in Event-B or model checking results produced by ProB and Uppaal are used as solutions in safety cases.

One of the main goals of our classification was to facilitate the construction of safety cases from the associated formal models. Therefore, each class of safety requirements is affiliated with a separate argument pattern.

Argument Patterns. The proposed argument patterns generalise the safety arguments for justifying that the safety requirements imposed on a system hold. The patterns have been developed using the corresponding GSN extensions (Figure 2.5). Some parts of an argument pattern may remain the same for any instance, while others need to be further instantiated (they are labelled with a specific GSN symbol – a hollow triangle). The text highlighted by braces { } should be replaced by a concrete value.

The generic representation of an argument pattern is shown in Figure 3.9, a. Here, the goal **GX** is associated with a safety requirement *Requirement* of some class *Class {X}*, where X is a class number.

To obtain the evidence that a specific safety requirement is satisfied, different arguments can be used. Within the proposed approach, a requirement is verified in a formal Event-B model M referred to in the model element **MX.1**. For example, if a safety requirement is mapped into an Event-B model as a model invariant property, the corresponding theorem for each event in the model M is required to be proved. Therefore, the number of subsequent sub-goals depends on the number of events in the model. Correspondingly, the proofs of these theorems (i.e., discharged *Proof Obligations (POs)*) are attached as the evidence for the constructed safety case.

The formulated properties and theorems associated with a particular requirement can be automatically derived from the given formal model. Nonetheless, to increase clarity of a safety case, we propose to explicitly refer to any theorem or property whose verification result is used as a solution of the top goal (**CX.2** in Figure 3.9, a.).

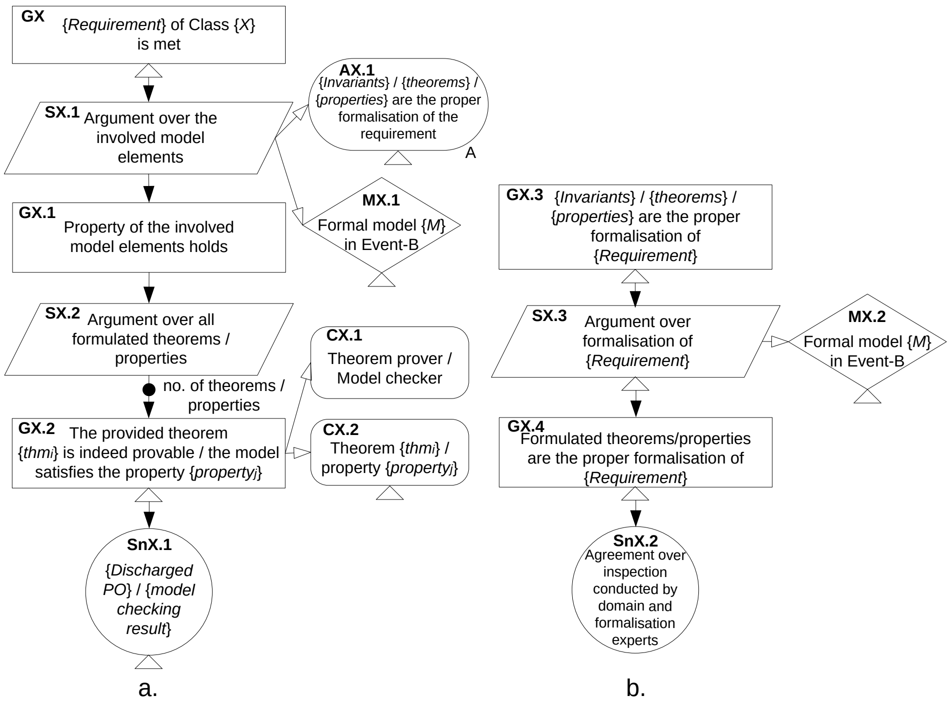

GX {Requirement} of Class {X} is met
AX.1 {Invariants} / {theorems} / {properties} are the proper formalisation of the requirement
A
SX.1 Argument over the involved model elements
GX.1 Property of the involved model elements holds
MX.1 Formal model {M} in Event-B
SX.2 Argument over all formulated theorems / properties
no. of theorems / properties
CX.1 Theorem prover / Model checker
GX.2 The provided theorem {thmi} is indeed provable / the model satisfies the property {propertyj}
CX.2 Theorem {thmi} / property {propertyj}
SnX.1 {Discharged PO} / {model checking result}
a.
GX.3 {Invariants} / {theorems} / {properties} are the proper formalisation of {Requirement}
SX.3 Argument over formalisation of {Requirement}
MX.2 Formal model {M} in Event-B
GX.4 Formulated theorems/properties are the proper formalisation of {Requirement}
SnX.2 Agreement over inspection conducted by domain and formalisation experts
b.

Figure 3.9: Generic argument pattern

In our argument patterns, we assume that the formulated model invariant, theorem or property represents the proper formalisation of the considered safety requirement (the assumption **AX.1**). Such an assumption can be substantiated by arguing over formalisation of the requirements as demonstrated in Figure 3.9, b. We rely on a joint inspection conducted by the involved domain and formalisation experts (**SnX.2**) as the evidence that the formulated theorems/properties are proper formalisations of the considered requirement. Such substantiation is applicable to all the classification-based argument patterns and their instances. This allows for reduction of a semantic gap in the mapping associating an informally specified safety requirement with the corresponding formal expression that is verified and connected to evidence.

While instances of the proposed argument patterns serve as inputs to the final *direct* safety argument, the argument and evidence proposed to substantiate the assumptions may also form a part of the *backing* (confidence) argument to support the direct argument and evidence.

Moreover, to fully rely on formal modelling and verification, we need to demonstrate well-definedness of each model in the formal development. The results of this demonstration also become a part of the backing argument. To verify this (e.g., that a partial function is applied within its domain), Event-B

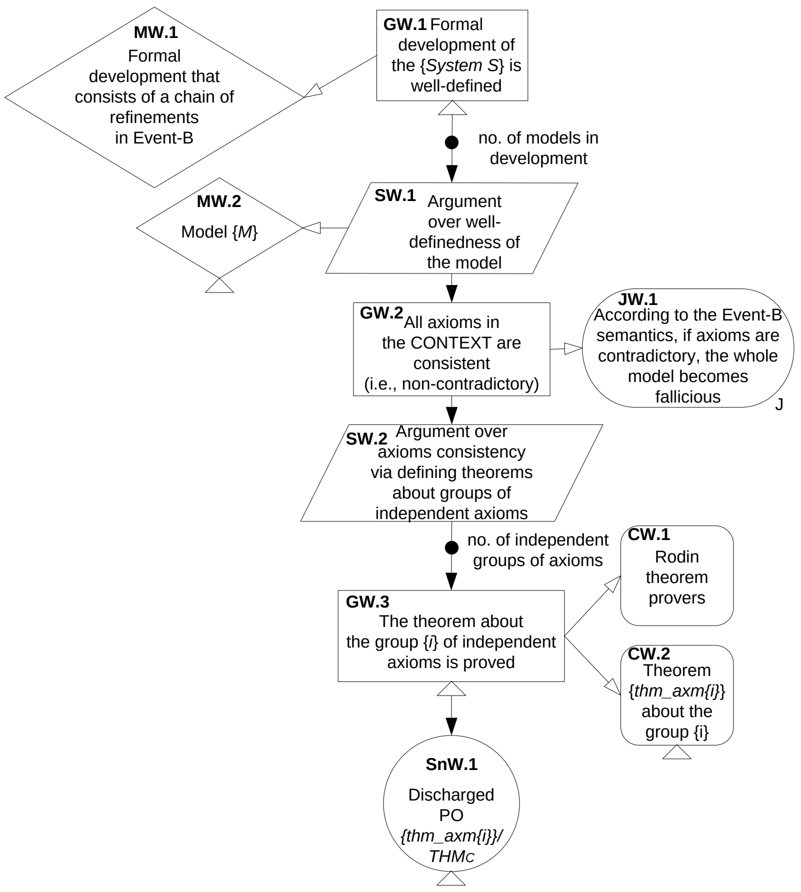

MW.1
Formal development that consists of a chain of refinements in Event-B
GW.1 Formal development of the {System S} is well-defined
no. of models in development
MW.2
Model {M}
SW.1 Argument over well-definedness of the model
GW.2 All axioms in the CONTEXT are consistent (i.e., non-contradictory)
JW.1
According to the Event-B semantics, if axioms are contradictory, the whole model becomes fallicious
J
SW.2 Argument over axioms consistency via defining theorems about groups of independent axioms
no. of independent groups of axioms
CW.1
Rodin theorem provers
GW.3
The theorem about the group {i} of independent axioms is proved
CW.2
Theorem {thm_axm{i}} about the group {i}
SnW.1
Discharged PO {thm_axm{i}}/ THMc

Figure 3.10: Argument pattern for well-definedness of the formal development

defines a number of proof obligations (*well-definedness (WD)* for theorems, invariants, guards, actions, etc. and *feasibility (FIS)* for events [3]), which are automatically generated by the Rodin platform. We assume here that all such proof obligations are discharged for models in question. However, if model axioms are inconsistent (i.e., contradictory), the whole model becomes fallacious and thus logically meaningless. Demonstrating that this is not the case is a responsibility of the developer. To handle this problem, we introduce a specific argument pattern shown in Figure 3.10. In Event-B, well-definedness of a CONTEXT can be ensured by proving axiom consistency (the goal **GW.2** in Figure 3.10).

The described above approach allows us to derive formal evidence (either proofs or model checking results) that the desired system properties stipulated by safety requirements are preserved. Moreover, the proposed set of classification-based argument patterns facilitate the construction of safety cases from Event-B models. The approach has been validated [114] by series of small case studies and a larger case study – a steam boiler control system.

To summarise, in this chapter, we elaborated on the approaches proposed in this thesis to address the raised research questions. First, we presented the approaches to formalisation of safety requirements obtained as the results of safety analysis and described in the natural language. Second, we proposed the formal specification and refinement patterns for safety-critical computer-based systems. Third, we presented an approach to rigorous construction of safety cases from formal models. In conjunction, the discussed approaches contribute to building an integrated approach to the safety-driven rigorous development of critical computer-based systems, as illustrated in Figure 1.1.

3.4 Discussion

We evaluate the approach proposed in this thesis by several case studies. The majority of the chosen case studies has been developed within European projects such as RODIN [121] and DEPLOY [51] together with the involved industrial partners. Both requirement classification and proposed rigorous mappings between requirements and the corresponding model elements have been strongly influenced by cooperation with our industrial partners, thus giving us a necessary reality check.

In this thesis, we rely on formal modelling techniques, including external tools that can be bridged together, that are scalable to analyse the entire system. Recently, our chosen formal framework, Event-B, has been actively used within large EU FP7 projects (RODIN [121], DEPLOY [51], ADVANCE [7]) to model and verify complex computer-based systems. Our classification of safety requirements as well as the developed formal specification and refinement patterns have shown to be expressive enough for a number of selected safety-critical systems from different domains.

Moreover, the obtained Event-B models can be also used as inputs for various model-based testing and static verification techniques, provided that the mapping between the corresponding model and code elements is given. There is a number of works dedicated to this topic, see, e.g., [96, 97, 100].

Despite the fact that a system safety case inevitably becomes larger and more complex for systems with a higher level of complexity, our approach does not prohibitively increase the resulting complexity of the constructed safety case. That is due to orthogonality of the classification of safety requirements on which it is based. Within the classification, each class is associated with an independent procedure for formalising a safety requirement of this class and constructing a fragment of the safety case demonstrating that this requirement has been met. This allows us to avoid the exponential growth of the added complexity while constructing safety cases for safety-critical systems.

The internal completeness of safety cases can be measured by a metric called the coverage of claims (goals) [49]. The coverage of claims (COV) is calculated as the proportion of the total number of developed claims (C_D) to the total number of claims (C). Specifically,

$$COV = \frac{C_D}{C}.$$

In [115], we provide these calculated metrics for the constructed fragments of safety cases for a number of the selected case studies. Even though the results are encouraging, building complete safety cases of complex computer-based systems in the industrial setting is still needed to more extensively evaluate the proposed approach. This remains one of our future goals.

Chapter 4

Summary of the Original Publications

In this chapter, we present a short summary of the original publications, which constitute the second part of the thesis.

Paper I: Patterns for Representing FMEA in Formal Specification of Control Systems

In this work, we explore the problem of integrating safety analysis, namely FMEA, into the formal development of safety-critical control systems. We propose a set of generic patterns that help the developers to trace the safety and fault tolerance requirements from FMEA to formal models. We identify five categories of patterns: component, detection, recovery, prediction and invariant. To ease instantiation of these patterns, we provide a tool support in the form of a plug-in for the Rodin platform.

We also demonstrate how formal models can be constructed to support seamless integration of safety requirements into formal modelling and verification of control computer-based systems. The proposed approach is inspired by the work [88]. We merge the ideas presented in [88] and our proposed FMEA patterns. This allows us to express formal development of safety-critical control systems as a generic, pattern-driven development process. We illustrate our approach by a case study – a sluice gate control system. The abstract model of this system is obtained by instantiating the abstract specification pattern for modelling a control system in Event-B. It describes the overall behaviour of the system as an interleaving between the events modelling the environment and the controller. Then, we refine this abstract specification to introduce the system components, error detection procedures as well as error masking and recovery actions systematically defined by FMEA. To achieve this, we instantiate the proposed FMEA patterns. The corresponding refinement steps can be done as instances of such refinement patterns as *Refinement introducing error detection procedures* and *Refinement introducing error handling procedures*. At the subsequent refinement step (an instance on the *Refinement elaborating on nominal functionality* pattern), we introduce a specific detailed specification of the nominal control logic.

Application of FMEA patterns helps the developers to elicit safety and fault tolerance requirements and assists in incorporating them into formal models, while application of specification and refinement patterns facilitates the formal development of safety-critical control systems.

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. The initial FMEA patterns were defined by Yuliya Prokhorova and Dr. Ilya Lopatkin under supervision of Associate Prof. Elena Troubitsyna, Dr. Alexei Iliasov and Prof. Alexander Romanovsky. Yuliya Prokhorova was also responsible for conducting safety analysis and contributed to the formal development of the sluice gate control system. The theoretical results were summarised by Associate Prof. Elena Troubitsyna, Dr. Ilya Lopatkin, Dr. Alexei Iliasov, Yuliya Prokhorova and Prof. Alexander Romanovsky.

Paper II: Deriving a Mode Logic Using Failure Modes and Effects Analysis

Structuring the system architecture and functionality in a layered fashion helps the developers to deal with the complexity of control systems. The dynamic behaviour of such systems is often defined in the terms of operational modes and transitions between these modes (a mode logic). The layered system structure also implies that the corresponding system components act as mode managers responsible for particular system layers. One part of the system mode logic prescribes the nominal system behaviour, while the other one determines mode transitions in the presence of faults, i.e., it is dedicated to improving fault tolerance of the system. However, derivation and verification of such a mode logic can be challenging.

In this paper, we propose a systematic approach to deriving the fault tolerance part of a mode logic. Similarly to Paper I, we utilise the FMEA technique. However, in Paper II, we propose to conduct FMEA of each operational mode to identify mode transitions required to implement fault tolerance. We achieve fault tolerance by two main means – system transitions to specific degraded modes and the system dynamic reconfiguration using redundant components. The former is usually performed by the top-layered mode manager, while the latter is done by lower-layered unit managers.

Formal development of such a layered control system in Event-B is done by stepwise refinement according to the following refinement strategy. Firstly, in the abstract specification and possibly several subsequent refinements, we develop the top level mode manager which implements the global mode logic. Secondly, at further refinement steps, we introduce the lower layer managers (unit managers) together with their mode logics. Finally, in the last refinement, we model the redundant branches (components) of each unit. The essential mode consistency conditions are verified in the refinement process.

Therefore, in this paper, we define a rigorous approach to designing layered mode-rich systems by refinement while ensuring their fault tolerance. We exemplify it by the development and verification of a mode-rich layered control system – an attitude and orbit control system (AOCS).

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. The informal specification and preliminary safety analysis of the AOCS were given by the industrial partners – Space Systems Finland, namely Dr. Kimmo Varpaaniemi and Dr. Timo Latvala. The formal development of the AOCS in Event-B was done by Yuliya Prokhorova and Adjunct Prof. Linas Laibinis. The theoretical results were summarised by Adjunct Prof. Linas Laibinis, Associate Prof. Elena Troubitsyna and Yuliya Prokhorova.

Paper III: A Case Study in Refinement-Based Modelling of a Resilient Control System

In this paper, we aim at validating our approach to formalising safety requirements as well as our approach to modelling control systems in Event-B. As an example, we consider a complex control system, namely, a steam boiler control system. To derive a list of safety requirements of this system, we use the FTA technique.

Formal development of the steam boiler system is performed by the stepwise refinement in Event-B and follows the described below strategy. The abstract model implements a basic control loop. It is an instance of the abstract specification pattern for modelling control systems in Event-B. At the first refinement, we introduce an abstract representation of the activities performed after the system is powered on and during its operation. At the second refinement step (combining instances of the *Refinement introducing error detection procedures* and *Refinement elaborating on nominal functionality* patterns), we incorporate a detailed representation of the conditions leading to a system failure. Moreover, we distinguish that the considered system has several operational modes and one non-operational mode to cover the system behaviour under different execution and fault conditions. At the third refinement step (an instance of the pattern *Refinement introducing error detection procedures*), we further elaborate on the physical behaviour of the steam boiler and refine the failure detection procedures. Finally, at the fourth refinement step, we explicitly define the system modes. At each refinement step, we define variables that stand for system components, e.g., sensors and actuators, necessary to be modelled at a particular level of abstraction, as well as represent the functional and safety requirements. As a result, the application of our previously proposed approaches allows us to obtain a formal model of the steam boiler control system and verify essential functional and safety properties.

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. Yuliya Prokhorova was responsible for conducting safety analysis. The formal development of the steam boiler control system in Event-B was done by Yuliya Prokhorova and Adjunct Prof. Linas Laibinis. The theoretical results were summarised by Associate Prof. Elena Troubitsyna, Adjunct Prof. Linas Laibinis and Yuliya Prokhorova.

Paper IV: Formalisation of an Industrial Approach to Monitoring Critical Data

Numerous safety-critical control systems contain monitoring subsystems. However, they are often not included into the safety kernel and hence are built from less reliable components. Nonetheless, these subsystems play an important indirect role in providing safety, i.e., based on the monitored data (human) operators or other subsystems should make adequate and timely decisions. In this paper, we take an example of a Data Monitoring System (DMS) suggested by our industrial partners, namely, a temperature monitoring system, and verify it. Specifically, we verify that the proposed architectural solution (a networked DMS) ensures data freshness and integrity despite unreliability of the system components.

The proposed formal specification in Event-B can be seen as a generic pattern for designing networked DMSs. Such systems are modular and the behaviour of their modules, namely Data Processing Units (DPUs), follows the same generic algorithm independently of other such units. Therefore, we can reason about the overall system by modelling a single DPU. The interactions of this module with the other modules of the system are represented as a part of the environment specification. The most abstract model (an instance of the abstract specification pattern for modelling monitoring systems in Event-B) allows us to define the essential functional behaviour of the DPU and prove the formulated data freshness properties. One of the refinement steps, which instantiates the proposed refinement pattern called *Refinement introducing fault detection and handling*, allows us to incorporate and verify the data integrity properties. Moreover, at another refinement step (an instance of the pattern *Refinement introducing clock synchronization*), we refine the mechanism of local clock adjustment and prove the corresponding theorems.

Due to the high level of abstraction, i.e., abstract data structures (constants and functions), arbitrary number of processing units, etc., the proposed patterns can be easily instantiated to verify data monitoring systems from different domains. As a result of our modelling and verification, we have received formally grounded assurance of dependability of the proposed industrial architecture of the temperature monitoring system.

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. The informal specification of the temperature monitoring system was given by the industrial partners – Space Systems Finland, namely Dr. Dubravka Ilić and Dr. Timo Latvala. The formal development of this system in Event-B was done by Yuliya Prokhorova under supervision of Adjunct Prof. Linas Laibinis. The theoretical results were summarised by Associate Prof. Elena Troubitsyna, Adjunct Prof. Linas Laibinis and Yuliya Prokhorova.

Paper V: Linking Modelling in Event-B with Safety Cases

In this paper, we give preliminary exploration of the problem associated with the use of formal verification results in safety cases. This paper discusses integration of two frameworks: formal modelling of a system in Event-B and constructing a structured safety case of a system using Goal Structuring Notation (GSN). We aim at contributing to the construction of a sufficient safety case and propose a fragment of it derived from a formal system specification. Firstly, we classify safety requirements and define how each class can be represented in a formal specification in Event-B. Secondly, we divide the safety case construction into two main parts: argumentation over the whole formal development and argumentation over safety requirements individually. Finally, we define a number of invariants and theorems to support the argumentation. The discharged proof obligations are used in the constructed safety case as the evidence that the given requirements hold. We illustrate the proposed approach by a case study – a sluice gate control system. The presented approach was later significantly improved and extended (see Paper VI).

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. The proposed fragments of safety cases were constructed by

Yuliya Prokhorova under supervision of Associate Prof. Elena Troubitsyna. The theoretical results were summarised by Associate Prof. Elena Troubitsyna and Yuliya Prokhorova.

Paper VI: Towards Rigorous Construction of Safety Cases

In this paper, we build up on the work presented in Paper V. We propose a methodology that covers two main processes: representation of formalised safety requirements in Event-B models, and derivation of safety cases from such Event-B models. To connect these processes, we utilise the classification of safety requirements introduced in Paper V. We modify and extend it to comprise the broader range of different safety requirements including, e.g., requirements about temporal and timing properties of a system. Firstly, the proposed classification is associated with particular ways the safety requirements can be represented in Event-B. We provide a formal mapping of different classes into Event-B model elements (i.e., axioms, variables, invariants, events, and attributes) and model expressions that can be build from model elements. Moreover, we show how the corresponding theorems for formal verification of safety requirements can be constructed out of these expressions. The described above mechanism allows us to derive formal evidence (either proofs or model checking results) that the desired system properties stipulated by safety requirements preserved by the modelled system. Secondly, we derive a set of classification-based argument patterns that allows us to facilitate the construction of fragments of safety cases from Event-B models. Additionally, we provide a separate argument pattern used to ensure that the formal development of a system in Event-B is well-defined. We validate the proposed methodology by series of small case studies as well as a larger case study – a steam boiler control system.

Author's contribution: This work was a natural continuation of Paper V. The proposed argument patterns were developed by Yuliya Prokhorova under supervision of Adjunct Prof. Linas Laibinis. The theoretical results were summarised by Adjunct Prof. Linas Laibinis, Yuliya Prokhorova and Associate Prof. Elena Troubitsyna.

Paper VII: A Survey of Safety-Oriented Model-Driven and Formal Development Approaches

In this paper, we overview the approaches that have been recently proposed to integrate safety analysis into model-driven and formal development of critical systems. We aim at guiding industry practitioners to identify the most suitable approaches to fulfil their design objectives as well as identifying promising research directions in the area. To perform a search for the relevant papers on the topic of our studies, we have used scientific paper databases such as IEEE Xplore, SpringerLink, ACM Digital Library, and Science Direct. Additionally, we have conducted a manual search of conference and workshop proceedings. We have reviewed over 200 papers and chosen 14 model-driven and 7 formal safety-oriented approaches for the detailed consideration. In this paper, we classify the approaches according to seven criteria such as the application domain, the modelling languages used, the covered phases of the development life cycle, the adopted safety analysis techniques, the nature of safety analysis (qualitative or quantitative), and the availability of automated tool support. We also define what are the inputs and outputs of each approach to facilitate a selection of a suitable technique. The results reveal that there is no single unified model-based approach that would facilitate achieving different levels of safety (i.e., SILs), while addressing various domain aspects and covering all the stages of system development process. We have also identified that there is the lack of mature tool support that would allow the safety engineers to adopt the proposed approaches in the industrial practice.

Author's contribution: This work was initiated by Associate Prof. Elena Troubitsyna. The literature analysis was performed by Yuliya Prokhorova. The theoretical results were summarised by Associate Prof. Elena Troubitsyna and Yuliya Prokhorova.

Chapter 5

Related Work

Model-based development approaches have been actively used in various domains. However, it is infeasible to overview all the available techniques. Therefore, in this chapter, we consider only the recent and most prominent approaches to model-based development and verification of safety-critical systems. Firstly, we describe the approaches that aim at extracting, identifying and structuring system requirements as well as consequently formalising them. Secondly, we overview the methods for safety-oriented model-based development of critical systems. Finally, we consider the approaches to demonstrating safety assurance and constructing the corresponding model-oriented safety cases.

5.1 Approaches to Requirements Elicitation and Formalisation

Various approaches can be undertaken to extract and structure both functional and non-functional requirements from system descriptions in the natural language, see [64, 80, 133]. Since the functional safety standards such as IEC 61508 [77] highly recommend the use of formal methods in the safety-critical domains, a vast amount of research has been dedicated to the requirements formalisation, e.g., [39, 70, 137]. Here we limit ourselves only to those approaches that have been linked with the Event-B formalism employed within the thesis. We can distinguish two categories among the considered approaches: (1) the approaches addressing requirements extraction and identification from the results of safety analysis for further formalisation and verification, and (2) the approaches aiming at achieving requirements traceability within formal models.

Formalising Safety Analysis Results in Event-B. Let us now take a closer look at the first category of the approaches to requirements elicitation, namely, the approaches to requirements extraction and identification from the results of safety analysis as well as their consequent formalisation in Event-B and similar formalisms, e.g., Action Systems and the B Method.

In the development of safety-critical computer-based systems, the use of safety analysis techniques such as FHA, FTA and FMEA at the early stages of the system development life cycle allows for obtaining lists of hazards and safety requirements. Moreover, the information collected during these analyses includes description of the failure detection and recovery procedures to be performed by the system in order to avoid the occurrence of severe hazards.

For example, the work [33] aims at investigating how the AltaRica [8, 110] and Event-B formalisms can be associated to perform the operational safety analysis and validate system safety. The AltaRica language has been developed to formally specify the behaviour of systems under the presence of faults and to allow for safety assessment [24]. It utilises FHA to derive safety requirements and permits, for example, generation of fault trees. However, the main goal of the work presented in [33] is not to translate an AltaRica model into an Event-B model but rather to maintain both models. On the one hand, an AltaRica model is used to analyse and assess the non-functional system aspects. On the other hand, an Event-B model is used mostly to specify the functional behaviour of a system. In our approach, both the system functional behaviour and its behaviour in the presence of faults are captured by the Event-B formalism. However, our approach does not support quantitative safety assessment. Therefore, AltaRica can be used to complement our approach.

In [125, 131], the authors propose approaches to integrating FTA and FMEA into formal system development within the Action Systems formalism that inspired Event-B. Specifically, [125] proposes to formalise fault tree events as predicates over state variables. These predicates then appear as guards of the actions that specify the reaction of the system on specific fault occurrences. The gates of a fault tree define logical operations (e.g., conjunction or disjunction) between the predicates in the action guards. A representation of FMEA results in the Action Systems formalism is described in [131]. This is achieved by using statecharts to structure the information about the failure modes and remedial actions obtained by FMEA. Finally, an approach to using statecharts as a middle ground between safety analysis and formal system specifications in the B Method is described in [132]. Specifically, statecharts assist in deriving the safety invariants to be verified from the given FTA results. These invariants define the safety conditions to be maintained to avoid the corresponding hazards.

In this thesis, we rely on the ideas put forward in [131, 132] to define a general approach to integrating results of FMEA into the formal Event-B specification and refinement process. However, we do not use statecharts as an intermediary but rather integrate FMEA into the formal development directly. In our work, in contrast to [131, 132], we define more strict guidelines how the results of FMEA can be mapped into the corresponding Event-B elements (variables, invariants, etc.). We propose a set of FMEA representation patterns, instantiation of which aims at facilitating the formal system development of safety-critical systems. The pattern application automatically transforms a model to represent the results of FMEA in a consistent way. The available automatic tool support for the Event-B modelling as well as the developed plug-in for FMEA pattern instantiation ensure better scalability of our approach.

In this thesis, we focus on a specific type of requirements – safety requirements. However, there are approaches dealing with the requirements analysis in general. The aim of these approaches is to extract requirements from the natural language and represent them in formal models. Next, we overview some of them.

The ProR Approach. The ProR approach proposed by Jastram et al. [81, 82] is an approach that establishes traceability between requirements and system modelling in Event-B. It is based on the *WRSPM* reference model [64]. The WRSPM model applies formal analysis to the requirements engineering process and is based on using two basic notations for describing artefacts and phenomena. Artefacts depict constraints on the system state space and state transitions, while phenomena represent the state space and state transitions of the system and its domain. Phenomena are split into two disjoint sets. One set contains the phenomena controlled by the system, while the second one consists of the phenomena controlled by the environment. There are five artefacts in WRSPM: domain knowledge or world (W), requirements (R), specifications (S), program (P), and programming platform or machine (M). In contrast to WRSPM, where R stands for all requirements, the authors of the ProR approach differentiate between the functional and non-functional requirements by introducing the artefact N, corresponding to the non-functional requirements, and keeping R for the functional ones. Moreover, they introduce a new artefact to represent design decisions (D).

The ProR approach provides traceability between informally stated system requirements and a specification in Event-B by formalising phenomena for the given artefacts as variables, events and constants, while formalising the artefacts themselves as either invariants or events. The approach is supported by a plug-in for the Rodin platform called ProR [118].

ProR is a generic approach, while, in this thesis, we propose an approach specific for safety-critical systems. This allows us to better support incorporation of safety and fault tolerance requirements as well as guide formal development and verification of such systems by providing a set of specification and refinement patterns for introducing specific safety and fault tolerance related mechanisms.

Linking Problem Frames and Event-B. Another approach to requirements analysis and structuring is called *Problem Frames* [80]. It allows for describing problems rather than solutions to these problems. Semantically, Problem Frames can be defined as the reference model that follows the principle of separation between the system (the machine) and its environment (the problem domains). The model is based on a number of artefacts (i.e., the domain knowledge, the requirements and the specification) and a vocabulary (phenomena) to describe the system, the environment and the interface between them. The Problem Frames approach is supported by the graphical problem diagrams. There are three types of elements in these diagrams: machine, problem world, and requirements.

Loesch et al. [94] have proposed an approach that permits for translating the requirements structured using Problem Frames into formal specifications in Event-B. The authors propose to model each problem diagram as an Event-B MACHINE and its associated CONTEXT. The refinement principle of Event-B is utilised to represent an elaboration of an abstract diagram into a more concrete one. Decomposition of a problem into subproblems is performed by the shared variables decomposition in Event-B [4]. Each phenomenon can be modelled in Event-B as a constant or a variable. Evolution of phenomena is modelled as data refinement in Event-B. Finally, the requirements stated in problem diagrams are formalised in Event-B by events and invariants.

The approach proposed in this thesis also facilitates traceability of requirements. However, in contrast to the described above two approaches (the ProR approach and the linking Problem Frames and Event-B approach), we widen instruments for representation of requirements. More specifically, requirements are represented by linking them with various expressions of model elements and then associating them with specific theorems, serving as the evidence that these requirements hold.

Combining KAOS with Event-B. In the area of the requirements engineering, the principle of *goal-oriented requirements engineering* [133] has recently received a significant attention. According to this principle, a *goal* is defined as an objective that the system under consideration should achieve. It covers both functional and non-functional aspects. Among the examples of the goal-oriented requirements engineering methods are *NFR* [34], *i*/Tropos* [60], *Goal-Based Requirements Analysis Method (GBRAM)* [10], and the method called *KAOS* [41, 134], where the abbreviation stands for *"Knowledge Acquisition in autOmated Specification"*.

KAOS is a semi-formal method that allows for capturing requirements as goals, constraints, objects, operations, actions, and agents [41]. It includes the following stages:

- identification and refinement of goals until the specific constraints imposed on agents are derived,
- identification of objects and actions from goals,
- derivation of requirements on the objects and actions in order to meet the constraints,
- allocation of these objects, actions and constraints to the agents that constitute a system.

The method supports such models as a *goal model* to represent the goals of a system and its environment, an *object model* to describe the system objects, an *agent responsibility model* to assign the goals to specific agents, and an *operational model* to represent the behaviour of agents. Moreover, KAOS is supported by the LTL notation to formalise specification of goals.

Several approaches have been proposed to combine KAOS with Event-B [15, 98, 99, 111]. To systematically derive Event-B models from KAOS goal models, the authors of [98, 99] formalise the KAOS refinement patterns used to generate the KAOS goal hierarchy. A KAOS goal can be represented as a postcondition of an event in Event-B, where an action of this event stands for the achievement of the goal. A decomposition of goals into subgoals is reflected as event refinement and guaranteed by the corresponding proof obligations. In contrast, the approach to linking KAOS and Event-B proposed by Aziz et al. [15] as well as the approach presented by Ponsard and Devroey [111] do not focus on the level of individual events but deal with transformation of a KAOS object model into Event-B in general. For example, the authors of [111] obtain an abstract Event-B MACHINE that meets the given KAOS requirements. Then, this MACHINE can be refined according to the KAOS agent responsibility model.

The KAOS approach decomposes requirements hierarchically. A similar mechanism is used within our approach to represent hierarchically structured safety requirements. However, we also focus on other classes of safety requirements (those safety requirements that are not hierarchically structured) and provide their corresponding mapping into Event-B model expressions.

5.2 Approaches to Model-Based Development and Verification of Critical Systems

In this section, we briefly overview several most recent and prominent approaches to safety-oriented model-based development and verification of critical computer-based systems. A more detailed analysis of such approaches can be found in our survey (Paper VII) given in the second part of this thesis.

Correctness, Modelling and Performance of Aerospace Systems (COMPASS). Bozzano et al. [28, 29, 30] propose a methodology to system-software co-engineering of safety-critical embedded systems, specifically aerospace systems. The approach is based on the AADL modelling framework and supports the safety and dependability analysis of AADL specifications by means of the toolset called COMPASS. To model the heterogeneous systems that include software and hardware components and their interactions, the authors extend the formal semantics of AADL and propose the System-Level Integrated Modelling (SLIM) language. A SLIM specification includes descriptions of both nominal system behaviour and its error behaviour as well as fault injections, i.e., descriptions how the error behaviour influences the nominal behaviour. The approach supports such hazard analysis techniques as FMEA and FTA. Moreover, the toolset assists in symbolic and probabilistic model checking. It takes as an input a SLIM specification as well as specific patterns for property specifications and generates as an output, in particular, the traces resulting from a simulation of the SLIM specification. The COMPASS toolset can also generate (probabilistic) fault trees and FMEA tables.

The COMPASS methodology is comprehensive and well founded. However, verification of essential properties of safety-critical systems is performed by model checking. This might lead to the state explosion problem. In this thesis, we propose approaches to verification of safety-critical systems based on theorem proving, which allows us to avoid the above mentioned problem.

MeDISIS Methodology. The MeDISIS methodology [38, 42, 43, 44] combines multiple technologies to improve efficiency of the safety and reliability analyses. It includes the following steps: (1) the use of FMEA to study the behaviour of a system in the presence of faults; (2) construction of a model integrating the functional system behaviour and the system behaviour in the presence of faults with Altarica Data Flow; (3) analysis and quantification of the non-functional behaviour and the impact on requirements and timing constraints with AADL. This can be also considered as a part of a SysML-centred model-based system engineering process that incorporates safety and dependability relevant activities into the system development life cycle [44].

The application of the MeDISIS methodology to the development of the embedded controller of an aircraft system is presented in [38]. The paper shows how to perform a reliability analysis using AltaRica Data Flow and timing analysis using AADL as well as how to finalise the specification stage with Matlab/Simulink. As a result, the obtained Simulink model, incorporating the system behaviour in the presence of faults allows the developers to get information about error propagation at the early stages of the design process. Even though we do not consider the quantitative system aspects, the formal models obtained by applying our approach also represent both nominal behaviour and the behaviour of the system in the presence of faults. Within these models, we are able to verify essential safety-related properties of critical computer-based systems and justify by proofs that the imposed safety requirements hold.

Integrating Formal Models and Safety Analysis (ForMoSA). The ForMoSA approach [109] combines the aspects of fault tolerance, functional correctness and quantitative analysis to address common problems of safety-critical systems and ensure a high level of safety. ForMoSA consists of three different parts: (1) building a formal specification; (2) qualitative analysis; (3) quantitative analysis. The first part includes building a functional model, a failure-sensitive specification, an independent traditional FTA, a functional error model, as well as checking correct integration of the failure modes into the functional model. The result of this step is a functional error model that defines not only functionality but also possible component failures and errors. The second part integrates the traditional FTA and formal methods by verifying functional correctness of the functional error model and performing the formal FTA (i.e., a technique that combines the rigorous proof concepts of formal methods into the reasoning process of FTA). The formal FTA results in a set of temporal logic formulas. They have to be proved correct for the functional error model. Then, the discharged proof obligations show that the fault tree is complete. The result of this part is a precise description of the qualitative relationship between component failures and hazards. The final, third part is based on applying the quantitative FTA, risk analysis, as well as mathematical optimisation. It results in a suggestion of the optimal configuration of the system.

The approach proposed in this thesis focuses on the logical representation of faults. We model the system architecture and design and verify that they comply with the given safety requirements. Moreover, we use FTA to only derive a list of system safety requirements, which currently do not involve quantitative characteristics or probabilities. However, it is possible to complement our approach by combining it with the work presented in [128]. This is one of the directions of our future work.

5.3 Approaches to Model-Oriented Safety Case Construction

Recently, model-based development of safety-critical computer-based systems has been linked to construction of safety cases. The research issues related to model-based safety cases, e.g., the concerns about justification of model appropriateness for a safety case, are outlined in [31]. Below we give several examples of existing approaches to model-based construction of safety cases.

In [14, 83], the authors propose safety case patterns for constructing arguments over the correctness of implementations developed from timed automata models. An instantiation of these patterns is shown on the implementation software of a patient controlled analgesic infusion pump [14] and a cardiac pacemaker [83].

Another example is the work conducted by Basir et al. [21]. The authors propose an approach to the construction of a hierarchical safety case for the generated program code according to the hierarchical structure of a system within model-based development. To demonstrate safety assurance, the authors perform formal proof-based verification of the annotated code. The obtained formal proofs serve as the evidence that the generated code conforms to the requirements imposed on the system under consideration. The approach provides independent assurance of both code and system model. It is validated using the results of the formal verification of safety requirements for the code generated from a Simulink model of a navigation system.

To reduce the costs of safety-critical systems development in a specific domain, industrial practitioners often adopt a product-line approach [37]. Habli [65] proposes a model-based approach to assuring product-line safety, e.g., in the automotive domain. The author defines a safety metamodel to capture the dependencies between the safety case construction, the assessment of safety, and the development aspects of a product line. The authors of [138] propose domain specific safety case patterns to construct a safety case for a cruise control system from automotive domain-specific models.

In this thesis, we contribute to a set of safety case patterns and describe in detail their instantiation process for different classes of safety requirements. Our argument patterns facilitate construction of safety cases where safety requirements are verified formally and the corresponding formal-based evidence is derived to represent justification for safety assurance.

Currently, all the described above safety case patterns lack a formal semantics to facilitate automated instantiation, composition and manipulation. In [48], the authors give a formal definition for safety case patterns, define a formal semantics for these patterns, and propose a generic data model as well as an algorithm for pattern instantiation. Moreover, recent developments in the area of safety cases include introduction of querying GSN safety case argument structures, which are used to provide argument structure views [47]. Specifically, GSN nodes are semantically enriched by metadata, given as a set of attributes. Then, the Argument Query Language is used to query safety case argument structures and create views. The methodology proposed in [47] can be useful to address stakeholders' concerns about system safety, and facilitate allocating and displaying the information of interest. We consider the works described in [47, 48] as complementary to our approach.

Chapter 6

Conclusions and Future Work

In this chapter, we summarise the main contributions of this thesis, discuss the limitations of the proposed approach as well as highlight future research directions.

6.1 Research Conclusions

Formal methods play an important role in development and verification of modern critical computer-based systems. Their application is strongly recommended by various standards for systems with a high level of criticality. Nonetheless, these methods are mostly used to guarantee the functional correctness of systems in question. While being an area of active research, integrating formal methods into engineering of safety-critical systems still remains a challenging task.

This thesis proposes an integrated approach to rigorous development of critical computer-based systems taking into account their safety aspects. We have made contributions to the processes of requirements elicitation, formal modelling and verification, as well as facilitation of system certification by showing how to incorporate safety requirements into formal models in Event-B and use the verification results of these formal models as the evidence for construction of safety cases.

To be able to represent and verify safety requirements formally in Event-B, we have proposed an approach to their structuring and precise mapping into the model elements either by integrating the FMEA technique with formal development or by classifying safety requirements and defining a specific mapping function for each class.

This gives us a starting point for formal development of safety-critical systems. However, formal development can be complex and time-consuming. Therefore, to support this task and promote the use of this technique in industrial applications, we have proposed a number of formal specification and refinement patterns for two types of safety-critical systems, namely, control and monitoring systems. Correctness of these patterns has been formally verified. Due to genericity, the patterns can be instantiated for a broad range of critical computer-based systems, e.g., by assigning concrete values to constants and instantiating abstract functions.

Finally, to be able to facilitate the safety assurance process of formally developed safety-critical systems, we have established a link between Event-B and safety cases. This link is defined via the safety requirements classification-based argument patterns, where strategies of goal decomposition rely on formal reasoning in Event-B. The set of proposed argument patterns allows us to contribute to the construction of safety case fragments assuring that system safety requirements are met as well as that system models in Event-B are well-defined.

We believe that our integrated approach can facilitate rigorous development of safety-critical systems and increase the interest in the application of formal techniques for industrial practitioners.

Nevertheless, the approach presented in the thesis has several limitations. The first and the main one is the lack of automation. At the moment, only instantiation of component-oriented FMEA patterns has been automated. However, neither the FMEA patterns used to derive the fault tolerance part of a system mode logic nor the proposed specification and refinement patterns have a tool support. Automated pattern instantiation would significantly facilitate their reuse for a class of suitable systems. The second problem also related to the lack of automation is manual construction of safety cases. This also decreases the overall impact of the proposed approach. The third issue is related to the completeness of the proposed classification. Despite the fact that the proposed classification of safety requirements covers a wide range of different safety requirements, it could be further extended if needed. Consequently, the set of argument patterns could be extended as well.

6.2 Future Work

The limitations of the proposed approach discussed above lead to the following possible future research directions. Firstly, a library of automated FMEA patterns and a set of formal specification and refinement patterns could be extended to cover more application domains. Secondly, automated construction of safety case fragments and their subsequent integration with the safety cases built using the existing tools, e.g., [11, 63, 79], would result in more efficient and scalable application of the proposed approach. Therefore, one of the future research directions could be dedicated to development of a plug-in for the Rodin platform, the supporting toolset for Event-B. Such a tool would allow one to automatically construct a fragment of a safety case and convert it into a format compatible with other tools for safety case construction.

We also foresee another direction of our future work. In this thesis, we have not taken into account the quantitative aspects of safety (e.g., failure rates). However, integration of our approach with quantitative assessment of safety-critical systems would be beneficial. There already exist several works that aim at addressing this problem in Event-B, e.g., [71, 104, 128, 129]. They can serve as a basis for extending our approach with probabilistic reasoning for representing and verifying requirements involving quantitative characteristics or constraints.

Bibliography

[1] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[4] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, January 2007.

[5] J.-R. Abrial and T.S. Hoang. Using Design Patterns in Formal Methods: An Event-B Approach. In J.S. Fitzgerald, A.E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing (ICTAC'08)*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer Berlin Heidelberg, 2008.

[6] A Computational Logic for Applicative Common Lisp (ACL2). http://www.cs.utexas.edu/∼moore/best-ideas/acl2/.

[7] ADVANCE – Advanced Design and Verification Environment for Cyber-physical System Engineering. http://www.advance-ict.eu/, 2011–2014.

[8] Success Story: Siemens RailCom and Model Driven Architecture. http://altarica.fr/.

[9] J.D. Andrews and S.J. Dunnett. Event-Tree Analysis Using Binary Decision Diagrams. *IEEE Transactions on Reliability*, 49(2):230–239, 2000.

[10] A. Anton. *Goal Identification and Refinement in the Specification of Software-Based Information Systems.* PhD Thesis, Georgia Institute of Technology, Atlanta, GA, USA, June 1997.

[11] The Adelard Assurance and Safety Case Environment (ASCE). http://www.adelard.com/asce/v4.1/download.html.

[12] A. Avižienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. In *Processing of the 3rd Information Survivability Workshop*, pages 7–12, 2000.

[13] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January–March 2004.

[14] A. Ayoub, B.G. Kim, I. Lee, and O. Sokolsky. A Safety Case Pattern for Model-Based Development Approach. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*, pages 141–146, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] B. Aziz, A. Arenas, J. Bicarregui, C. Ponsard, and P. Massonet. From Goal-Oriented Requirements to Event-B Specifications. In *Proceedings of the 1st Nasa Formal Method Symposium (NFM'09)*, California, April 2009.

[16] R.J. Back and K. Sere. Stepwise Refinement of Action Systems. *Mathematics of Program Construction*, 375:115–138, 1989.

[17] R.J. Back and K. Sere. From Action Aystems to Modular Systems. *FME'94: Industrial Benefit of Formal Methods*, 873:1–25, 1994.

[18] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[19] L.M. Barroca and J.A. McDermid. Formal Methods: Use and Relevance for the Development of Safety Critical Systems. *The Computer Journal*, 35:579–599, 1992.

[20] N. Basir. *Safety Cases for the Formal Verification of Automatically Generated Code.* Doctoral thesis, University of Southampton, 2010.

[21] N. Basir, E. Denney, and B. Fischer. Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In E. Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 68–81. Springer Berlin Heidelberg, 2010.

[22] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[23] G. Berry. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In S. Leue and P. Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin Heidelberg, 2008.

[24] P. Bieber, C. Bougnol, C. Castel, J.-P. Christophe K., Heckmann, S. Metge, and C. Seguin. Safety Assessment with AltaRica. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information*, pages 505–510. Springer US, 2004.

[25] P. Bishop and R. Bloomfield. A Methodology for Safety Case Development. In *Safety-Critical Systems Symposium, Birmingham, UK*. Springer-Verlag, 1998.

[26] J. Bowen and M. Hinchey. The Use of Industrial-Strength Formal Methods. In *Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 332–337, August 1997.

[27] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[28] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and X. Olive. Formal Verification and Validation of AADL Models. In *Proceedings of Embedded Real Time Software and Systems Conference (ERTS'10)*, pages 1–9, 2010.

[29] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Proceedings of International Conference on Computer Safety, Reliability and Security (SAFECOMP'09)*, pages 173–186, 2009.

[30] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *The Computer Journal*, 54(5):754–775, 2010.

[31] P. Braun, J. Philipps, B. Schätz, and S. Wagner. Model-Based Safety-Cases for Software-Intensive Systems. *Electronic Notes in Theoretical Computer Science*, 238(4):71–77, September 2009.

[32] Claims, Arguments and Evidence (CAE).
http://www.adelard.com/asce/choosing-asce/cae.html.

[33] J.-C. Chaudemar, E. Bensana, C. Castel, and C. Seguin. AltaRica and Event-B Models for Operational Safety Analysis: Unmanned Aerial Vehicle Case Study. In *Proceeding of the Workshop on Integration of Model-Based Formal Methods and Tools*, pages 15–19, Düsseldorf, Germany, February 2009.

[34] L. Chung and J.C.S. Prado Leite. On Non-Functional Requirements in Software Engineering. In A.T. Borgida, V.K. Chaudhri, P. Giorgini, and E.S. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin Heidelberg, 2009.

[35] E. Clarke. 25 Years of Model Checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[36] Industrial Use of the B Method.
http://www.methode-b.com/documentation_b/ClearSy-Industrial_Use_of_B.pdf.

[37] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[38] R. Cressent, V. Idasiak, F. Kratz, and P. David. Mastering Safety and Reliability in a Model Based Process. In *Proceedings of Reliability and Maintainability Symposium (RAMS'11)*, pages 1–6, 2011.

[39] J. Crow and B. Di Vito. Formalizing Space Shuttle Software Requirements: Four Case Studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[40] Dependability Engineering for Open Systems. D-Case.
http://www.dependable-os.net/osddeos/en/concept.html.

[41] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-driven Requirements Engineering. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 612–613, New York, NY, USA, 1997. ACM.

[42] P. David, V. Idasiak, and F. Kratz. Towards a Better Interaction Between Design and Dependability Analysis: FMEA Derived from UML/SysML Models. In *Proceedings of ESREL'08 and the 17th SRA-EUROPE annual conference*, 2008.

[43] P. David, V. Idasiak, and F. Kratz. Automating the synthesis of AltaRica Data-Flow models from SysML. In *Reliability, Risk, and Safety,*

Three Volume Set: Theory and Applications; Proceedings of European Safety and Reliability Conference (ESREL'09), pages 105–112, 2009.

[44] P. David, V. Idasiak, and F. Kratz. Reliability Study of Complex Physical Systems Using SysML. *Reliability Engineering and System Safety*, 95(4):431–450, 2010.

[45] UK Ministry of Defence. 00-56 Safety Management Requirements for Defence Systems, 2007.

[46] E. Denney and B. Fischer. Software Certification and Software Certificate Management Systems (Position Paper). In *Proceedings of ASE Workshop on Software Certificate Management (SCM'05)*, pages 1–5, Long Beach, CA, November 2005.

[47] E. Denney, D. Naylor, and G. Pai. Querying Safety Cases. In A. Bondavalli and F. Di Giandomenico, editors, *Computer Safety, Reliability, and Security (SAFECOMP'14)*, volume 8666 of *Lecture Notes in Computer Science*, pages 294–309. Springer International Publishing, 2014.

[48] E. Denney and G. Pai. A Formal Basis for Safety Case Patterns. In F. Bitsch, J. Guiochet, and M. Kaâniche, editors, *Computer Safety, Reliability, and Security (SAFECOMP'13)*, volume 8153 of *Lecture Notes in Computer Science*, pages 21–32. Springer Berlin Heidelberg, 2013.

[49] E. Denney, G. Pai, and J. Pohl. Automating the Generation of Heterogeneous Aviation Safety Cases. NASA Contractor Report NASA/CR-2011-215983, 2011.

[50] Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). IST FP7 IP Project. http://www.deploy-project.eu.

[51] DEPLOY – Industrial deployment of system engineering methods providing high dependability and productivity. http://www.deploy-project.eu/index.html, 2008–2012.

[52] EB2ALL - The Event-B to C, C++, Java and C# Code Generator. http://eb2all.loria.fr/.

[53] ESTEREL - Critical Systems and Software Development Solutions. Success Stories. http://www.esterel-technologies.com/success-stories/airbus-a380/.

[54] European Committee for Electrotechnical Standardization (CENELEC). EN 50129 Railway applications – Communication, signalling

and processing systems. Safety related electronic systems for signalling. February 2003.

[55] European Committee for Electrotechnical Standardization (CENELEC). EN 50128 Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. June 2011.

[56] Event-B and the Rodin Platform. http://www.event-b.org/.

[57] The Flow plug-in. http://iliasov.org/usecase/.

[58] FMEA Info Centre. http://www.fmeainfocentre.com/.

[59] Fault Tree Analysis. http://www.fault-tree.net.

[60] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[61] Goal Structuring Notation Working Group. Goal Structuring Notation Standard. http://www.goalstructuringnotation.info/, November 2011.

[62] A. Gomes and M. Oliveira. Formal Development of a Cardiac Pacemaker: From Specification to Code. In J. Davies, L. Silva, and A. Simao, editors, *Formal Methods: Foundations and Applications*, volume 6527 of *Lecture Notes in Computer Science*, pages 210–225. Springer Berlin Heidelberg, 2011.

[63] GSN Editor. http://www.dependablecomputing.com/tools/gsn-editor/index.html.

[64] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May 2000.

[65] I. Habli. *Model-Based Assurance of Safety-Critical Product Lines.* Doctoral thesis, University of York, 2009.

[66] I. Habli and T. Kelly. A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. *Electronic Notes in Theoretical Computer Science*, 238(4):27–39, September 2009.

[67] A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13(2):66–76, March 1996.

[68] R. Hawkins, T. Kelly, J. Knight, and P. Graydon. A New Approach to Creating Clear Safety Arguments. In *Advances in Systems Safety*, pages 3–23. Springer London, 2011.

[69] I.J. Hayes, M.A. Jackson, and C.B. Jones. Determining the Specification of a Control System from That of Its Environment. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods (FME 2003)*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2003.

[70] C. Heitmeyer. Formal Methods for Specifying Validating, and Verifying Requirements. *Journal of Universal Computer Science*, pages 607–618, 2007.

[71] T.S. Hoang, Z. Jin, K. Robinson, A. McIver, and C. Morgan. Development via Refinement in Probabilistic B – Foundation and Case Study. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 355–373. Springer Berlin Heidelberg, 2005.

[72] C.A.R. Hoare. Communicating Sequential Processes. *ACM Communications*, 21(8):666–677, August 1978.

[73] The HOL System. http://www.cl.cam.ac.uk/research/hvg/HOL/.

[74] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[75] A. Iliasov. Use Case Scenarios as Verification Conditions: Event-B/Flow Approach. In *Proceedings of the 3rd International Workshop on Software Engineering for Resilient Systems (SERENE'11)*, pages 9–23, Berlin, Heidelberg, 2011. Springer-Verlag.

[76] A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky. Patterns for Refinement Automation. In F.S. de Boer, M.M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 70–88. Springer Berlin Heidelberg, 2010.

[77] International Electrotechnical Commission. IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. April 2010.

[78] International Organization for Standardization. ISO 26262 Road Vehicles Functional Safety. November 2011.

[79] ISCaDE (Integrated Safety Case Development Environment). http://www.iscade.co.uk/.

[80] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[81] M Jastram. *The ProR Approach: Traceability of Requirements and System Descriptions.* Phd thesis, University of Dusseldorf, 2012.

[82] M. Jastram, S. Hallerstede, M. Leuschel, and Jr Russo, A.G. An Approach of Requirements Tracing in Formal Refinement. In GaryT. Leavens, Peter O'Hearn, and SriramK. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin Heidelberg, 2010.

[83] E. Jee, I. Lee, and O. Sokolsky. Assurance Cases in Model-Driven Development of the Pacemaker Software. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 2010.

[84] A. Joshi, M. Whalen, and M.P.E. Heimdahl. Model-Based Safety Analysis: Final Report. Technical report, NASA, 2005.

[85] T.P. Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases.* Doctoral thesis, University of York, September 1998.

[86] T.P. Kelly and J.A. McDermid. Safety Case Construction and Reuse Using Patterns. In P. Daniel, editor, *Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP'97)*, pages 55–69. Springer-Verlag London, 1997.

[87] J.C. Knight. Safety Critical Systems: Challenges and Directions. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE'02)*, pages 547–550, May 2002.

[88] L. Laibinis and E. Troubitsyna. Refinement of Fault Tolerant Control Systems in B. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Computer Safety, Reliability, and Security*, volume 3219 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2004.

[89] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[90] J.-C. Laprie. From Dependability to Resilience. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages G8—-G9, 2008.

[91] N. Leveson, N. Dulac, D. Zipkin, J. Cutcher-Gershenfeld, J. Carroll, and B. Barrett. *Resilience Engineering: Concepts and Precepts*, chapter Engineering Resilience into Safety-Critical Systems. Aldershot, Ashgate, UK, September 2006.

[92] N. Leveson, L. Pinnel, S. Sandys, S. Koga, and J. Reese. Analyzing Software Specifications for Mode Confusion Potential. In *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, 1997.

[93] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[94] F. Loesch, R. Gmehlich, K. Grau, C. Jones, and M. Mazzara. Report on Pilot Deployment in Automotive Sector. DEPLOY Deliverable D19, Project DEPLOY Grant Agreement 214158, January 2010.

[95] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Report 1003, 2011.

[96] Q.A. Malik, L. Laibinis, D. Truscan, and J. Lilius. Requirement-Driven Scenario-Based Testing Using Formal Stepwise Development. *International Journal On Advances in Software*, 3(1):147–160, 2010.

[97] Q.A. Malik, J. Lilius, and L. Laibinis. Model-Based Testing Using Scenarios and Event-B Refinements. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 177–195. Springer Berlin Heidelberg, 2009.

[98] A. Matoussi, F. Gervais, and R. Laleau. An Event-B Formalization of KAOS Goal Refinement Patterns. Technical Report TR–LACL–2010–1, January 2010.

[99] A. Matoussi, R. Laleau, and D. Petit. Bridging the Gap Between KAOS Requirements Models and B Specifications. Technical Report TR-LACL-2009-5, September 2009.

[100] D. Méry and R. Monahan. Transforming Event-B Models into Verified C# Implementations. In A. Lisitsa and A. Nemytykh, editors, *International Workshop on Verification and Program Transformation (VPT'13)*, volume 16 of *EPiC Series*, pages 57–73, 2013.

[101] C. Metayer, J.-R. Abrial, and L. Voisin. Event-B Language. Rigorous Open Development Environment for Complex Systems (RODIN) Deliverable 3.2. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, May 2005.

[102] Military Standard. MIL-STD-1629A Procedures for performing a Failure Mode, Effects, and Criticality Analysis. November 1980.

[103] R. Milner. The Polyadic π-Calculus: a Tutorial. In F. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series*, pages 203–246. Springer Berlin Heidelberg, 1993.

[104] C. Morgan, T.S. Hoang, and J.-R. Abrial. The Challenge of Probabilistic Event-B – Extended Abstract –. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Formal Specification and Development in Z and B (ZB'05)*, volume 3455 of *Lecture Notes in Computer Science*, pages 162–171. Springer Berlin Heidelberg, 2005.

[105] T. Nipkow, M. Wenzel, and L. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[106] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pages 35–46, New York, NY, USA, 2000. ACM.

[107] NuSMV: a New Symbolic Model Checker. http://nusmv.fbk.eu/.

[108] OMG UML. Unified Modeling Language. http://www.uml.org/.

[109] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif. Combining Formal Methods and Safety Analysis – The ForMoSA Approach. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 474–493. Springer Berlin Heidelberg, 2004.

[110] G. Point and A. Rauzy. AltaRica: Constraint Automata as a Description Language. *Journal Européen des Systèmes Automatisés*, 33(8–9):1033–1052, 1999.

[111] C. Ponsard and X. Devroey. Generating High-Level Event-B System Models from KAOS Requirements Models. In *Proceedings of INFormatique des ORganisations et Systémes d'Information et de Décision (INFORSID'11)*, Lille, France, May 2011.

[112] B.R. Poreddy and S. Corns. Arguing Security of Generic Avionic Mission Control Computer System (MCC) using Assurance Cases. *Procedia Computer Science*, 6:499–504, 2011.

[113] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ ProB/index.php5/Main_Page.

[114] Y. Prokhorova, L. Laibinis, and E. Troubitsyna. Towards Rigorous Construction of Safety Cases. TUCS Technical Report 1110, 2014.

[115] Y. Prokhorova, L. Laibinis, and E. Troubitsyna. Facilitating Construction of Safety Cases from Formal Models in Event-B. *Information and Software Technology*, 60:51–76, April 2015.

[116] Y. Prokhorova, L. Laibinis, E. Troubitsyna, K. Varpaaniemi, and T. Latvala. Deriving a mode logic using failure modes and effects analysis. *International Journal of Critical Computer-Based Systems*, 3(4):305—328, 2012.

[117] Y. Prokhorova, E. Troubitsyna, and L. Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System. TUCS Technical Report 1086, 2013.

[118] ProR. Requirements Engineering Platform. http://www.eclipse.org/rmf/pror/.

[119] F. Redmill and T. Anderson, editors. *Developments in Risk-based Approaches to Safety: Proceedings of the Fourteenth Safety-citical Systems Symposium, Bristol, UK, 7-9 February 2006.* Springer London Ltd, 2006.

[120] W. Ridderhof, H.-G. Gross, and H. Doerr. Establishing Evidence for Safety Cases in Automotive Systems – A Case Study. In F. Saglietti and N. Oster, editors, *Computer Safety, Reliability, and Security (SAFECOMP'07)*, volume 4680 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2007.

[121] RODIN – Rigorous Open Development Environment for Complex Systems. http://rodin.cs.ncl.ac.uk/, 2004–2007.

[122] Rodin Plug-ins. http://wiki.event-b.org/index.php/Rodin_Plug-ins.

[123] SCADE Suite. Control and Logic Application Development. http://www.esterel-technologies.com/products/scade-suite/.

[124] I. Scagnetto and M. Miculan. Ambient Calculus and its Logic in the Calculus of Inductive Constructions. *Electronic Notes in Theoretical Computer Science*, 70(2):76–95, 2002.

[125] K. Sere and E. Troubitsyna. Safety Analysis in Formal Specification. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II (FM'99)*, pages 1564–1583, London, UK, 1999. Springer-Verlag.

[126] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[127] N. Storey. *Safety-Critical Computer Systems.* Addison-Wesley, Harlow, UK, 1996.

[128] A. Tarasyuk. *Formal Development and Quantitative Verification of Dependable Systems.* PhD thesis, Turku Centre for Computer Science, 2013.

[129] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Integrating Stochastic Reasoning into Event-B Development. *Formal Aspects of Computing*, pages 1–25, 2014.

[130] A.C. Tribble, D.L. Lempia, and S.P. Miller. Software Safety Analysis of a Flight Guidance System. In *Proceedings of the 21st Digital Avionics Systems Conference*, volume 2, 2002.

[131] E. Troubitsyna. Integrating Safety Analysis into Formal Specification of Dependable Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, April 2003.

[132] E. Troubitsyna. Elicitation and Specification of Safety Requirements. In *Proceedings of the 3rd International Conference on Systems (ICONS'08)*, pages 202–207, April 2008.

[133] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–262, 2001.

[134] A. van Lamsweerde and E. Letier. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *Lecture Notes in Computer Science*, pages 325–340. Springer Berlin Heidelberg, 2004.

[135] VDM. The Vienna Development Method. http://www.vdmportal.org.

[136] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback. Fault Tree Handbook with Aerospace Applications. NASA Technical Report, 2002.

[137] S.A. Vilkomir, J.P. Bowen, and A.K. Ghose. Formalization and assessment of regulatory requirements for safety-critical software. *Innovations in Systems and Software Engineering*, 2(3-4):165–178, 2006.

[138] S. Wagner, B. Schatz, Stefan Puchner, and P. Kock. A Case Study on Safety Cases in the Automotive Domain: Modules, Patterns, and Models. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pages 269–278, November 2010.

[139] A. Wassyng, T. Maibaum, M. Lawford, and H. Bherer. Software Certification: Is There a Case against Safety Cases? In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Lecture Notes in Computer Science*, pages 206–227. Springer Berlin Heidelberg, 2011.

[140] P.J. Wilkinson and T.P. Kelly. Functional Hazard Analysis for Highly Integrated Aerospace Systems. In *Certification of Ground/Air Systems Seminar*, pages 4/1–4/6, 1998.

[141] J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19:1–19:36, October 2009.

[142] D. Zowghi and C. Coulin. Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In A. Aurum and C. Wohlin, editors, *Engineering and Managing Software Requirements*, pages 19–46. Springer Berlin Heidelberg, 2005.

Complete List of Original Publications Related to the Thesis

1. Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Facilitating Construction of Safety Cases from Formal Models in Event-B, In *Information and Software Technology Journal*, Vol. 60, pp. 51–76, Elsevier, 2015.

2. Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Towards Rigorous Construction of Safety Cases. *TUCS Technical Report 1110*, TUCS, May 2014.

3. Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System, In Anatoliy Gorbenko, Alexander Romanovsky, Vyacheslav Kharchenko (Eds.), *Proceedings of the 5th International Workshop on Software Engineering for Resilient Systems (SERENE 2013)*, Lecture Notes in Computer Science Vol. 8166, pp. 79–93, Springer-Verlag Berlin Heidelberg, 2013.

4. Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ilić, and Timo Latvala. Formalisation of an Industrial Approach to Monitoring Critical Data, In Friedemann Bitsch, Jérémie Guiochet, Mohamed Kaâniche (Eds.), *Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013)*, Lecture Notes in Computer Science Vol. 8153, pp. 57–69, Springer-Verlag Berlin Heidelberg, 2013.

5. Yuliya Prokhorova and Elena Troubitsyna. A Survey of Safety-Oriented Model-Driven and Formal Development Approaches, In *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 2, pp. 93–118, Inderscience Publishers, 2013.

6. Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis, Supporting Formal Modelling in Event-B with Safety Cases. In: Michael Butler, Stefan Hallerstede, Marina Waldén (Eds.), *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes 18, pp. 8–11, TUCS, 2013.

7. Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis, A Case Study in Refinement-Based Modelling of a Resilient Control System. *TUCS Technical Report 1086*, TUCS, June 2013.

8. Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ilić, and Timo Latvala, Formalisation of an Industrial Approach to Monitoring Critical Data. *TUCS Technical Report 1070*, TUCS, March 2013.

9. Yuliya Prokhorova, Linas Laibinis, Elena Troubitsyna, Kimmo Varpaaniemi, and Timo Latvala. Deriving a Mode Logic Using Failure Modes and Effects Analysis, In *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 3, No. 4, pp. 305–328, Inderscience Publishers, 2012.

10. Yuliya Prokhorova and Elena Troubitsyna. Linking Modelling in Event-B with Safety Cases, In Paris Avgeriou (Ed.), *Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, Lecture Notes in Computer Science Vol. 7527, pp. 47-–62, Springer-Verlag Berlin Heidelberg, 2012.

11. Yuliya Prokhorova, Linas Laibinis, Elena Troubitsyna, Kimmo Varpaaniemi, and Timo Latvala, Derivation and Formal Verification of a Mode Logic for Layered Control Systems. In: Tran Dan Thu, Karl Leung (Eds.), *Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC 2011)*, pp. 49–56, IEEE Computer Society, 2011.

12. Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova, and Elena Troubitsyna. Patterns for Representing FMEA in Formal Specification of Control Systems, In Ankur Agarwal, Swapna Gokhale, Taghi M. Khoshgoftaar (Eds.), *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium (HASE 2011)*, pp. 146–151, IEEE Computer Society, 2011.

13. Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, and Vyacheslav Kharchenko, Development of Safety-Critical Control Systems in Event-B Using FMEA. In: Luigia Petre, Kaisa Sere, Elena Troubitsyna (Eds.), *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, pp. 75–91, IGI Global, 2011.

14. Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Kimmo Varpaaniemi, and Timo Latvala, Deriving Mode Logic for Fault-Tolerant Control Systems. In: Naveed Ahmed, Daniele Quercia, Christian D. Jensen (Eds.), *Workshop Proceedings of the 5th IFIP WG 11.11 International Conference on Trust Management (IFIPTM 2011)*, pp. 309–323, Technical University of Denmark, 2011.

15. Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova, and Elena Troubitsyna, Patterns for Representing FMEA in Formal Specification of Control Systems. *TECHNICAL REPORT SERIES CS-TR-1261*, Newcastle University, 2011.

16. Ilya Lopatkin, Yuliya Prokhorova, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky, Patterns for Representing FMEA in Formal Specification of Control Systems. *TUCS Technical Report 1003*, TUCS, March 2011.

17. Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis, Integrating FMEA into Event-B Development of Safety-Critical Control Systems. *TUCS Technical Report 986*, TUCS, October 2010.

Part II

Original Publications

Paper I

Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova, and Elena Troubitsyna

Originally published in: Ankur Agarwal, Swapna Gokhale, Taghi M. Khoshgoftaar (Eds.), *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium (HASE 2011)*, pp. 146–151, IEEE Computer Society, 2011.

Extended version is available as:

Ilya Lopatkin, Yuliya Prokhorova, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems, *TUCS Technical Report 1003*, TUCS, 2011.

©2011 IEEE. Reprinted with kind permission of IEEE.

2011 IEEE 13th International Symposium on High-Assurance Systems Engineering

Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
{Ilya.Lopatkin, Alexei.Iliasov, Alexander.Romanovsky}@ncl.ac.uk

Yuliya Prokhorova, Elena Troubitsyna
Turku Centre for Computer Science
Department of Information Technologies
Åbo Akademi University
Turku, Finland
{Yuliya.Prokhorova, Elena.Troubitsyna}@abo.fi

***Abstract* — Failure Modes and Effects analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides engineers with valuable information about failure modes of system components as well as procedures for error detection and recovery. In this paper we propose an approach that facilitates representation of FMEA results in formal Event-B specifications of control systems. We define a number of patterns for representing requirements derived from FMEA in formal system model specified in Event-B. The patterns help the developers to trace the requirements from safety analysis to formal specification. Moreover, they allow them to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice control system.**

Keywords - formal specification; Event-B; FMEA; patterns; safety; control systems

I. Introduction

A. Motivation and Overview of an Approach

Formal modelling and verification are valuable for ensuring system dependability. However, often formal development process is perceived as being too complex to be deployed in the industrial engineering process. Hence, there is a clear need for methods that facilitate adopting of formal modelling techniques and increase productivity of their use.

Reliance on patterns – the generic solutions for certain typical problems – facilitates system engineering. Indeed, it allows the developers to document the best practices and reuse previous knowledge.

In this paper we propose an approach to automating formal system development by refinement. We connect formal modelling and refinement with Failure Modes and Effects Analysis (FMEA) via a set of patterns.

FMEA is a widely-used inductive technique for safety analysis [5,13,16]. We define a set of patterns formalising the requirements derived from FMEA and automate their integration into the formal specification. Our formal modelling framework is Event-B – a state-based formalism for formal system development by refinement and proof-based verification [1]. Currently, the framework is actively used by several industrial partners of EU FP7 project Deploy [2] for developing dependable systems from various domains.

The approach proposed in this paper allows us to automate the formal development process via two main steps: *choice of suitable patterns that generically define FMEA result*, and *instantiation of chosen patterns with model-specific information.* We illustrate this process with excerpts from the automated development of a sluice gate system [7].

Our approach allows the developers to verify (by proofs) that safety invariants are preserved in spite of identified component failures. Hence we believe that it provides a useful support for formal development and improves traceability of safety requirements.

B. Related Work

Over the last few years integration of the safety analysis techniques into formal system modelling has attracted a significant research attention. There are a number of approaches that aim at direct integration of the safety analysis techniques into formal system development. For instance, the work of Ortmeier et al. [15] focuses on using statecharts to formally represent the system behaviour. It aims at combining the results of FMEA and FTA to model the system behaviour and reason about component failures as well as overall system safety. Our approach is different – we aim at automating the formal system development with the set of patterns instantiated by FMEA results. The application of instantiated patterns automatically transforms a model to represent the results of FMEA in a coherent and complete way. The available automatic tool support for the Event-B modelling as well as for plug-in instantiation and application ensures better scalability of our approach.

In our previous work, we have proposed an approach to integrating safety analysis into formal system development within Action Systems [18]. Since Event-B incorporates the ideas of Action Systems into the B Method, the current work is a natural extension of our previous results.

The research conducted by Troubitsyna [19] aims at demonstrating how to use statecharts as a middle ground between safety analysis and formal system specifications in the B Method. This work has inspired our idea of deriving Event-B patterns.

Patterns defined for formal system development by Hoang et al. [17] focus on describing model manipulations only and do not provide the insight on how to derive a formal model from a textual requirements description that has a negative impact on requirements traceability.

Another strand of research aims at defining general guidelines for ensuring dependability of software-intensive systems. For example, Hatebur and Heisel [6] have derived patterns for representing dependability requirements and ensuring their traceability in the system development. In our approach we rely on specific safety analysis techniques rather than on the requirements analysis in general to derive guidelines for modelling dependable systems.

II. Modelling Control Systems in Event-B

A. Event-B Overview

Event-B [1] is a specialisation of the B Method aimed at facilitating modelling of parallel, distributed and reactive systems [9]. The Rodin Platform provides an automated support for modelling and verification in Event-B [4].

In Event-B system models are defined using the Abstract Machine Notation. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. The machine is uniquely identified by its name. The state variables of the machine are declared in the **VARIABLES** clause and initialized in the *INITIALISATION* event. The variables are strongly typed by constraining predicates of invariants given in the **INVARIANTS** clause. Usually the invariants also define the properties of the system that should be preserved during system execution. The data types and constants of the model are defined in a separate component called **CONTEXT**. The behaviour of the system is defined by a number of atomic events specified in the **EVENTS** clause. An event is defined as follows:

$$E = \textbf{ANY}\ lv\ \textbf{WHERE}\ g\ \textbf{THEN}\ S\ \textbf{END}$$

where lv is a list of new local variables, the guard g is a conjunction of predicates defined over the state variables, and the action S is an assignment to the state variables.

The guard defines when the event is enabled. If several events are enabled simultaneously then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a composition of variable assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := E(v)$, where x is a state variable and $E(v)$ is an expression over the state variables v. The non-deterministic assignment can be denoted as $x :\in S$ or $x :| Q(v, x')$, where S is a set of values and $Q(v, x')$ is a predicate. As a result of the non-deterministic assignment, x gets any value from S or it obtains such a value x' that $Q(v, x')$ is satisfied.

The main development methodology of Event-B is *refinement.* Refinement formalises model-driven development and allows us to develop systems correct-by-construction. Each refinement transforms the abstract specification to gradually introduce implementation details. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.

Next we describe specification and refinement of control systems in Event-B. It follows the specification pattern proposed earlier [11].

B. Modelling Control Systems

The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. In our specification the sensors and actuators are represented by the corresponding state variables. We follow the systems approach, i.e., model the controller together with its environment – plant. This allows us to explicitly state the assumptions about environment behaviour. At each cycle the plant assigns the variables modelling the sensor readings. They depend on the physical processes of the plant and the current state of the actuators. In its turn, the controller reads the variables modelling sensors and assigns the variables modelling the actuators. We assume the controller reaction takes negligible amount of time and hence the controller can react properly on changes of the plant state.

In this paper, we focus on modelling failsafe control systems. A system is failsafe if it can be put into a safe but non-operational state to preclude an occurrence of a hazard.

The general specification pattern **Abs_M** for modelling a failsafe control system in Event-B is presented in [14]. It represents the overall behaviour of the system as an interleaving between the events modelling the plant and the controller. The behaviour of the controller has the following stages: *Detection; Control (Normal Operation* or *Error Handling); Prediction.* The variable *flag* of type **PHASE:**{ENV, DET, CONT, PRED} models the current stage.

In the model invariant we declare the types of the variables and define the operational conditions. The system is operational if it has not failed. However, it must be stopped at the end of the current cycle if a failure occurred.

The events **Environment, Normal_Operation** and **Prediction** abstractly model environment behaviour, controller reaction and computation of the next expected states of system components correspondingly. The event **Detection** non-deterministically models the outcome of the error detection. A result of error recovery is abstractly modelled by the event **Error_Handling**.

In the next section we demonstrate how to arrive at a detailed specification of a control system by refinement in Event-B. We use the sluice gate control system to exemplify the refinement process.

III. Refinement of Control Systems in Event-B

A. The Sluice Gate Control System

The general specification pattern **Abs_M** given in [14] defines the initial abstract specification for any typical control system. The sluice gate control system shown in Fig. 1 is among them. The system is a sluice connecting areas with dramatically different pressures [7]. The purpose of the system is to adjust the pressure in the sluice area. The sluice gate system consists of two doors - *door1* and *door2* that can be operated independently of each other and *a pressure chamber pump* that changes the pressure in the sluice area. To guarantee safety, a door may be opened only if the pressure in the locations it connects is equalized. Moreover, at most one door can be opened at any moment and the pressure chamber pump can only be switched on when both doors are closed.

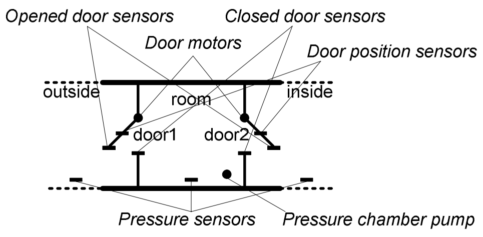


Figure 1. Sluice gate system.

The sluice gate system is equipped with the sensors and actuators shown in Fig.1. The system has physical redundancy - the door position sensors have spares; and information redundancy - when the doors are fully opened or closed, the door position sensor readings should match the readings of the switch sensors.

B. *Introducing Error Detection and Recovery by Refinement*

At the first refinement step we aim at introducing models of system components, error detection procedures as well as error masking and recovery actions.

To systematically define failure modes, detection and recovery procedures, for each component, we conduct Failure Modes and Effects Analysis. FMEA [5,13,16] is a well-known inductive safety analysis technique. For each system component it defines its possible failure modes, local and system effects of a failure as well as detection and recovery procedures. Fig. 2 shows an excerpt from FMEA of the *Door1* component of our sluice system.

The *Door1* component is composed of several hardware units. Their failures correspond to the failure modes of the *Door1* component. Next we discuss how to specify error detection and recovery for the failure mode described in the FMEA table in Fig. 2.

Component	Door1
Failure mode	Door position sensor value is different from the door closed sensor value
Possible cause	Failure of position sensor or closed sensor
Local effects	Sensor readings are not equal in corresponding states
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the values received from position and closed sensors
Remedial action	Retry. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Figure 2. FMEA table

In the refined specification we introduce the variables representing the units of *Door1*: door position sensor - *door1_position_sensor*, motor - *door1_motor* and door opened and closed sensors - *door1_opened_sensor, door1_closed_sensor.* In the event **Environment** we introduce the actions that change the values of *door1_position_sensor, door1_closed_sensor* and *door1_opened_sensor.* The event **Normal_Operation** defines the action that non-deterministically changes the value of *door1_motor.*

We refine the event **Detection** by splitting it into a group of events responsible for the detection of each failure mode of all system components. We introduce the variable *door1_fail* to designate a failure of the door component. This failure is assigned TRUE when any failure mode of *Door1* component is detected. The event **Detection_door1_checks** included in this group contains the actual checks for the value ranges and consistency. The variables *d1_exp_min* and *d1_exp_max* are the new variables introduced to model the next expected sensor readings. These variables are updated in the **Prediction** event. The event **Detection_Door1** combines the results of the checks of the status of the *door1* component.

```
event Detection_Door1_checks
  where
   flag = DET ∧ Stop = FALSE
  then
   door1_position_sensor_pred ≔ bool((door1_position_sensor <
       d1_exp_min ∨ door1_position_sensor > d1_exp_max) ∧
       door1_sensor_disregard=FALSE)
   door1_closed_sensor_inconsistent ≔
       bool(¬(door1_closed_sensor=TRUE ⇔
          (door1_position=0 ∨ door1_sensor_disregard=TRUE)))
   <other checks>
end
```

The failure of the component *Door1* is detected if any check of the error detection events for any of its failure modes finds a discrepancy between a fault free and the observed states. In a similar manner, the system failure is detected if a failure of any of the system components – *Door1*, *Door2* or *PressurePump* is detected, as specified in the event **Detection_Fault**.

```
event Detection_Door1
 where
  flag = DET ∧ Stop = FALSE
 then door1_fail ≔ bool( door1_position_sensor_pred=TRUE ∨
            door1_closed_sensor_inconsistent=TRUE ∨
            <other check statuses>)
end
event Detection_Fault refines Detection
 where
  flag = DET ∧ Stop = FALSE
  door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail = TRUE
 with Failure' Failure'=TRUE
 then flag ≔ CONT
end
```

Observe that by performing FMEA of each system component we obtain a systematic textual description of all procedures required to detect component errors and perform

Component	Door1
Failure mode	Door position sensor value is different from the expected range of values
Possible cause	Failure of the position sensor
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the received value with the predicted range of values
Remedial action	The same as for Fig. 2

Figure 3. FMEA table for "out of predicted range" failure mode of a positioning sensor

their recovery. We gradually (by refinement) introduce the specification of these requirements into the system model.

While analysing the refined specification, it is easy to note that there are several typical specification solutions called patterns that represent certain groups of requirements. This observation prompts the idea of creating an automated tool support that would automatically transform a specification by applying the patterns chosen and instantiated by the developer. In the next section we describe the essence of such a tool.

IV. Patterns and Tool for Representing Results of FMEA in Event-B

A. *Patterns for Representing FMEA Results*

Our approach aims at structuring and formalising FMEA results via a set of generic patterns. These patterns serve as a middle hand between informal requirements description and their formal Event-B model.

While deriving the patterns we assume that the abstract system specification adheres to the generic pattern given in [14]. Moreover, we also assume that components can be represented by the corresponding state variables. Our patterns establish a correspondence between the results of FMEA and the Event-B terms.

We distinguish four groups of patterns: detection, recovery, prediction and invariants. The detection patterns reflect such generic mechanisms for error detection as discrepancy between the actual and expected component state, sensor reading outside of the feasible range etc. The recovery patterns include retry of actions or computations, switch to redundant components and safe shutdown. The prediction patterns represent the typical solutions for computing estimated states of components, e.g., using the underlying physical system dynamics or timing constraints. Finally, the invariant patterns are usually used in combination with other types of patterns to postulate how a model transformation affects the model invariant. This type contains safety and gluing invariant patterns. The safety invariant patterns define how safety conditions can be introduced into the model. The gluing invariant patterns depict the correspondence between the states of refined and abstract model.

A *pattern* is a model transformation that upon instantiation adds or modifies certain elements of Event-B model. By *elements* we mean the terms of Event-B mathematical language such as variables, constants, invariants, events, guards etc. A pattern can add or modify several elements at once. Moreover, it can be composed of several other patterns.

To illustrate how to match FMEA results with the proposed patterns, let us consider FMEA of a door1 position sensor shown in Fig. 3.

To simplify illustration, the patterns are shown in a declarative form. The identifiers shown in brackets should be substituted by those given by a user during the pattern instantiation (see next sections).

Our sensor is a value type sensor. Hence we can apply the *Value sensor pattern* to introduce the model of the sensor into our specification:

```
variables [sensor]_value
invariants
  [sensor]_value : NAT
events
  event INITIALISATION
  then
     [sensor]_value := 0
  end
end
```

An application of the value sensor pattern leads to creating a new variable, its typing invariant, and an initialisation action. To model identified detection of the failure mode, we use the *Expected range pattern*:

```
variables
   [component]_[sensor]_[error], [component]_fail, [sensor]_exp_min,
   [sensor]_exp_max
invariants
   [component]_[sensor]_[error] : BOOL
   [component]_fail : BOOL
   [sensor]_exp_min : NAT
   [sensor]_exp_max : NAT
events
  event Detection_[component]_checks
  where flag = DET ∧ Stop = FALSE then
  [component]_[sensor]_[error] ≔ bool(
  [sensor]_value<[sensor]_exp_min∨[sensor]_value>[sensor]_exp_max)
  <other checks>
   end
  event Detection_[component]
  where flag = DET ∧ Stop = FALSE then
  [component]_fail ≔ bool([component]_[sensor]_[error] ∨
  <other check statuses>)
   end
 end
```

This pattern adds the detection events and the variables required to model error detection: expected minimal and maximal values. The pattern ensures that the detection checks added previously by other patterns are preserved (this is informally shown in the angle brackets). The expected range of values used by this pattern must be assigned by some event in the previous control cycle. To ensure that such assignment exists in the model, the *Expected range pattern* instantiates the *Range prediction pattern*. An application of this pattern results in a non-deterministic specification of prediction. It can be further refined to take into account the specific functionality of the system under development.

Let us observe that the *Expected range pattern* and *Range prediction pattern* affect the same variables. To avoid conflicts and inconsistencies, only the first pattern to be instantiated actually creates the required variables. The same rule applies to events, actions, guards etc.

To establish refinement between the model created using patterns and the abstract model, we use the *Gluing invariant pattern*, which links the sensor failure with the component failure:

```
variables
   [component]_fail
invariants
   flag≠DET ⇒ (Failure=TRUE ⇔ [component]_fail=TRUE ∨
                    <other component failures>)
   flag≠CONT ⇒ ([component]_fail=TRUE ⇔
                    [component]_[sensor]_[error]=TRUE ∨
                    <other sensor errors>)
```

In our example, the remedial action can be divided into three actions. The first action retries reading the sensor for a specified number of times (*Retry recovery pattern*). The second action deactivates the faulty component and activates its spare (*Component redundancy recovery pattern*). The third action is enabled when the spare component has also failed. It switches the system from the operational state to the non-operational one (*Safe stop recovery pattern*). The system effect can be represented as a safety property (*Safety invariant pattern*). We omit showing all the patterns due to the lack of space.

As shown in the example, each FMEA field is mapped to one or more patterns. The patterns have interdependencies. Moreover, they are composable. For instance, the *recovery patterns* reference the variables set by the sensor, and thus depend on the results of the *Value sensor pattern*. The *Expected value detection pattern* needs to instantiate the *Range prediction pattern* to rely on the values predicted at the previous control cycle. Each pattern creates Event-B elements specific to the pattern, and requires elements created by other patterns. Such interdependency and mapping to FMEA is schematically shown in Fig. 4.

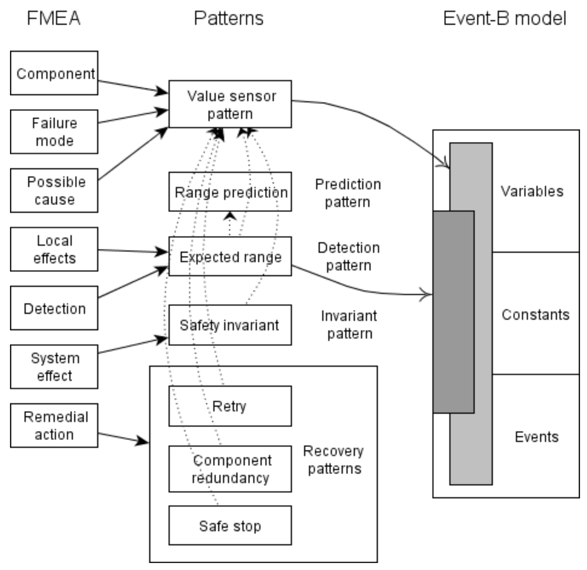


Figure 4. FMEA representation patterns

Let us note that the *Expected range pattern* creates new constants and variables (dark grey rectangle, variable [sensor]_exp_min from the example) and instantiates the *Value sensor pattern* to create the elements it depends on (light grey rectangle, variable [sensor]_value from the example).

B. Automation of Patterns Implementation

The automation of the pattern instantiation is implemented as a tool plug-in for the Rodin Platform [4]. Technically, each pattern is a program written in a simplified Eclipse Object Language (EOL). It is a general purpose programming language in the family of languages of the Epsilon framework [10] which operates on EMF [3] objects. It is a natural choice for automating model transformations since Event-B is interoperable with EMF.

The tool extends the application of EOL to Event-B models: it adds simple user interface features for instantiation, extends the Epsilon user input facility with discovery of the Event-B elements, and provides a library of Event-B and FMEA-specific transformations.

To apply a pattern, a user chooses a target model and a pattern to instantiate. A pattern application may require user input: variable names or types, references to existing elements of the model etc. The input is performed through a series of simple dialogs.

The requested input comprises the applicability conditions of the pattern. In many cases it is known that instantiation of a pattern depends primarily on the results of a more basic pattern. In those cases the former directly instantiates the latter and reuses the user input. Also more generally, if several patterns require the same unit of user input then the composition of such patterns will ask for such input only once. Typically, a single pattern instantiation requires up to 3-4 inputs.

If a pattern only requires user input and creates new elements then its imperative form is close to declarative as shown in the example below:

```
var flag: Variable=
chooseOrCreateVariable("Phase variable");
createTypingInvariant(flag, "PHASE");
var failure: Variable =
chooseOrCreateVariable("Failure variable");
createTypingInvariant(failure, "BOOL");
newEvent("Detection")
.addGuard("phase_grd", flag.name+" = DET")
.addGuard("failure_grd", failure.name+"=FALSE")
.addAction("phase_act", flag.name+":=CONT")
.addAction("failure_act",failure.name+"::BOOL");
```

Here the tool will ask the user to select two variables (or create new ones). It will create typing invariants and a new model event with several guards and actions. We have used the tool to automate the first refinement of the sluice gate control system. The complete specification can be found in [14].

C. Ensuring Safety by Refinement

In the second refinement step we introduce the detailed specification of the normal control logic. This refinement step leads to refining the event **Normal_Operation** into a group of events that model the actual control algorithm. These events model opening and closing the doors as well as activation of the pressure chamber pump.

Refinement of the normal control operation results in restricting non-determinism. This allows us to formulate safety invariants that our system guarantees:

$failure = FALSE \land door1\_position = door1\_position \Rightarrow door1\_position = 0$

$failure = FALSE \land (door1\_position > 0 \lor door1\_motor=\mathbf{MOTOR\_OPEN}) \Rightarrow pressure\_value = \mathbf{PRESSURE\_OUTSIDE}$

$failure = FALSE \land (door2\_position > 0 \lor door2\_motor=\mathbf{MOTOR\_OPEN}) \Rightarrow pressure\_value = \mathbf{PRESSURE\_INSIDE}$

$failure = FALSE \land pressure\_value \neq \mathbf{PRESSURE\_INSIDE} \land pressure\_value \neq \mathbf{PRESSURE\_OUTSIDE} \Rightarrow door1\_position=0 \land door2\_position=0$

$failure = FALSE \land pump\neq\mathbf{PUMP\_OFF} \Rightarrow (door1\_position=0 \land door2\_position=0)$

These invariants formally define the safety requirements informally described in subsection III.A. While verifying the correctness of this refinement step, we formally ensure (by proofs) that safety is preserved while the system is operational.

At the consequent refinement steps we introduce the error recovery procedures. This allows us to distinguish between criticality of failures and ensure that if a non-critical failure occurs then the system can still remain operational.

V. CONCLUSIONS

In this paper we have made two main technical contributions. Firstly, we derived a set of generic patterns for elicitation and structuring of safety and fault tolerance requirements from FMEA. Secondly, we created an automatic tool support that enables interactive pattern instantiation and automatic model transformation to capture these requirements in formal system development. Our methodology facilitates requirements elicitation as well as supports traceability of safety and fault tolerance requirements within the formal development process.

Our approach enables *guided* formal development process. It supports the reuse of knowledge obtained during formal system development and verification. For instance, while deriving the patterns we have analysed and generalised our previous work on specifying various control systems [8,11,12].

We believe that the proposed approach and tool support provide a valuable support for formal modelling that is traditionally perceived as too cumbersome for engineers. Firstly, we define a generic specification structure. Secondly, we automate specification of a large part of modelling decisions. We believe that our work can potentially enhance productivity of system development and improve completeness of formal models.

As a future work we are planning to create a library of domain-specific patterns and automate their application. This would result in achieving even greater degree of development automation and knowledge reuse.

ACKNOWLEDGMENT

The work reported in this paper is supported by FP7 ICT DEPLOY.

REFERENCES

[1] J.-R. Abrial, "Modeling in Event-B: system and software engineering", Cambridge University Press, 2010.

[2] Deploy project, www.deploy-project.eu.

[3] Eclipse GMT – Generative Modeling Technology, http://www.eclipse.org/gmt.

[4] Event-B and the Rodin Platform, http://www.event-b.org/, 2010.

[5] FMEA Info Centre, http://www.fmeainfocentre.com/.

[6] D. Hatebur and M. Heisel, "A foundation for requirements analysis of dependable software", Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP), Springer, 2009, pp. 311-325.

[7] I. Lopatkin, A. Iliasov, A. Romanovsky, "On Fault Tolerance Reuse during Refinement". In Proc. Of the 2nd International Workshop on Software Engineering for Resilient Systems (SERENE), April 13-16, 2010.

[8] D. Ilic and E. Troubitsyna, "Formal development of software for tolerating transient faults". In Proc. of the 11th IEEE Pacific Rim International Symposium on Dependable Computing, IEEE Computer Society, Changsha, China, December 2005.

[9] ClearSy, Safety critical systems engineering, http://www.clearsy.com/.

[10] D. S. Kolovos, "Extensible platform for specification of integrated languages for model management (Epsilon)", Official web-site: http://www.cs.york.ac.uk/~dkolovos/epsilon.

[11] L. Laibinis and E. Troubitsyna, "Refinement of fault tolerant control systems in B", SAFECOMP 2004, Springer, Potsdam, Germany, 2004.

[12] L. Laibinis and E. Troubitsyna, "Fault tolerance in a layered architecture: a general specification pattern in B", In Proc. of International Conference on Software Engineering and Formal Methods SEFM'2004, IEEE Computer Society Press, pp.346-355, Beijing, China, September 2004.

[13] N.G. Leveson, "Safeware: system safety and computers", Addison-Wesley, 1995.

[14] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov and A. Romanovsky, "Patterns for Representing FMEA in Formal Specification of Control Systems", Technical Report 1003, TUCS, March 2011.

[15] F. Ortmeier, M. Guedemann and W. Reif, "Formal failure models", Proceedings of the IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07), Elsevier, 2007.

[16] N. Storey, "Safety-critical computer systems", Addison-Wesley, 1996.

[17] Thai Son Hoang, A, Furst and J.-R. Abrial, "Event-B patterns and their tool support", SEFM 2009, IEEE Computer Press, 2009, pp. 210-219.

[18] E. Troubitsyna, "Elicitation and specification of safety requirements", Proceedings of the Third International Conference on Systems (ICONS 2008), 2008, pp. 202-207.

[19] E. Troubitsyna, "Integrating safety analysis into formal specification of dependable systems", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), 2003, p. 215b.

Paper II

Deriving a Mode Logic Using Failure Modes and Effects Analysis

Yuliya Prokhorova, Linas Laibinis, Elena Troubitsyna, Kimmo Varpaaniemi, and Timo Latvala

Originally published in: *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 3, No. 4, pp. 305—328, Inderscience Publishers, 2012.

Paper III

A Case Study in Refinement-Based Modelling of a Resilient Control System

Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis

Originally published in: Anatoliy Gorbenko, Alexander Romanovsky, Vyacheslav Kharchenko (Eds.), *Proceedings of the 5th International Workshop on Software Engineering for Resilient Systems (SERENE 2013)*, Lecture Notes in Computer Science Vol. 8166, pp. 79-–93, Springer-Verlag Berlin Heidelberg, 2013.

Extended version is available as:

Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System, *TUCS Technical Report 1086*, TUCS, 2013.

Paper IV

Formalisation of an Industrial Approach to Monitoring Critical Data

Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ilić, and Timo Latvala

Originally published in: Friedemann Bitsch, Jérémie Guiochet, Mohamed Kaâniche (Eds.), *Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013)*, Lecture Notes in Computer Science Vol. 8153, pp. 57-–69, Springer-Verlag Berlin Heidelberg, 2013.

Extended version is available as:

Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Dubravka Ilić, and Timo Latvala. Formalisation of an Industrial Approach to Monitoring Critical Data, *TUCS Technical Report 1070*, TUCS, 2013.

Paper V

Linking Modelling in Event-B with Safety Cases

Yuliya Prokhorova and Elena Troubitsyna

Originally published in: Paris Avgeriou (Ed.), *Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, Lecture Notes in Computer Science Vol. 7527, pp. 47-–62, Springer-Verlag Berlin Heidelberg, 2012.

Paper VI

Towards Rigorous Construction of Safety Cases

Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna

Originally published in: *TUCS Technical Report 1110*, TUCS, May 2014.

Shortened version is available as:

Yuliya Prokhorova, Linas Laibinis, and Elena Troubitsyna. Facilitating Construction of Safety Cases from Formal Models in Event-B, In *Information and Software Technology Journal*, Vol. 60, pp. 51–76, Elsevier, 2015.

Towards Rigorous Construction of Safety Cases

Yuliya Prokhorova
TUCS – Turku Centre for Computer Science,
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
yuliya.prokhorova@abo.fi
Linas Laibinis
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linas.laibinis@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
elena.troubitsyna@abo.fi

TUCS Technical Report
No 1110, May 2014

Abstract

Certification of safety-critical software systems requires submission of safety assurance documents, e.g., in the form of safety cases. A safety case is a justification argument used to show that a system is safe for a particular application in a particular environment. Different argumentation strategies are applied to determine the evidence for a safety case. They allow us to support a safety case with such evidence as results of hazard analysis, testing, simulation, etc. On the other hand, application of formal methods for development and verification of critical software systems is highly recommended for their certification. In this paper, we propose a methodology that combines these two activities. Firstly, it allows us to map the given system safety requirements into elements of the formal model to be constructed, which is then used for verification of these requirements. Secondly, it guides the construction of a safety case demonstrating that the safety requirements are indeed met. Consequently, the argumentation used in such a safety case allows us to support the safety case with formal proofs and model checking results as the safety evidence. Moreover, we propose a set of argument patterns that aim at facilitating the construction of (a part of) a safety case from a formal model. In this work, we utilise the Event-B formalism due to its scalability and mature tool support. We illustrate the proposed methodology by numerous small examples as well as validate it by a larger case study – a steam boiler control system.

Keywords: safety-critical software systems, safety requirements, formal development, formal verification, Event-B, safety cases, argument patterns.

TUCS Laboratory
Embedded Systems Laboratory

1 Introduction

Safety-critical software systems are subject to certification. More and more standards in different domains require construction of *safety cases* as a part of the safety assurance process of such systems, e.g., ISO 26262 [40], EN 50128 [25], and UK Defence Standard [19]. Safety cases are justification arguments for safety. They justify why a system is safe and whether the design adequately incorporates the safety requirements defined in a system requirement specification to comply with the safety standards. To facilitate the construction of safety cases, two main graphical notations have been proposed: *Claims, Arguments and Evidence (CAE)* notation [17] and *Goal Structuring Notation (GSN)* [44]. In our work, we rely on the latter one due to its support for *argument patterns*, i.e., common structures capturing successful argument approaches that can be reused within a safety case [45]. To demonstrate the compliance with the safety standards, different types of evidence can be used [51]. Among them are results of hazard analysis, testing, simulation, formal verification, manual inspection, etc.

At the same time, the use of *formal methods* is highly recommended for certification of safety-critical software systems [36]. Safety cases constructed using formal methods give us extra assurance that the desired safety requirements are satisfied. There are several works dedicated to show how formal proofs can contribute to a safety case, e.g., [8–10, 21, 43]. For instance, such approaches as [8, 10] apply formal methods to ensure that different types of safety properties of critical systems hold while focusing on particular blocks of software system implementation (C code). The authors of [21] propose a generic approach to automatic transformation of the formal verification output into a software safety assurance case. Similarly to [8, 10], a formalised safety requirement in [21] is verified to hold at a specific location (a specific line number for code, a file, etc.).

In our work, we deal with formal system models rather than the code. A high level of abstraction allows us to cope with complexity of systems yet ensuring the desired safety properties. We rely on formal modelling techniques, including external tools that can be used together, that are scalable to analyse the entire system. Our chosen formal framework is Event-B [4] – a state-based formal method for system level modelling and verification. Event-B aims at facilitating modelling of parallel, distributed and reactive systems. Scalability in Event-B can be achieved via abstraction, proof and decomposition. Moreover, this formalism has strictly defined semantics and mature tool support – the Rodin platform [26] accompanied by various plug-ins, including the ones for program code generation, e.g, C, Java, etc. This allows us to model and verify a wide range of different safety-related properties stipulated by the given system safety requirements. Those requirements may include the safety requirements about global and local system properties, the absence of system deadlocks, temporal and timing properties, etc.

In this paper, we significantly extend and exemplify with a large case study our approach to linking modelling in Event-B with safety cases presented in [54]. More specifically, we further elaborate on the classification of safety requirements and define how each class can be treated formally to allow for verification of the given safety requirements, i.e., we define *mapping* of the classified safety requirements into the corresponding elements of Event-B.

The Event-B semantics then allows us to associate them with particular theorems (proof obligations) to be proved when verifying the system. The employed formal framework assists the developers in automatic generation of the respective proof obligations. This allows us to use the obtained proofs as the evidence in safety cases, demonstrating that the given safety requirements have been met. Finally, to facilitate the construction of safety cases, we define a set of argument patterns where the argumentation and goal decomposition in safety cases are based on the results obtained from the associated formal reasoning.

Therefore, the overall contribution of this paper is a developed methodology that covers two main processes: (1) integration of formalised safety requirements into formal models of software systems, and (2) construction of structured safety cases [1] from such formal models.

The remainder of the paper is organised as follows. In Section 2, we briefly introduce our modelling framework – Event-B, its refinement-based approach to modelling software systems as well as the Event-B verification capabilities based on theorem proving. Additionally, we overview the notion of safety cases and their supporting graphical notation. In Section 3, we describe our methodology and provide the proposed classification of safety requirements. We elaborate on the proposed methodology in Section 4, where we define a set of argument patterns and their verification support. In Section 5, we illustrate application of the proposed patterns on a larger case study – a steam boiler control system. In Section 6, we overview the related work. Finally, in Section 7, we give concluding remarks as well as discuss our future work.

2 Preliminaries

In this section, we briefly outline the Event-B formalism that we use to derive models of safety-critical systems. In addition, we briefly describe the notion of safety cases and their supporting notation that we will rely on in this paper.

2.1 Overview of Event-B

Event-B language. Event-B [4, 26] is a state-based formal method for system level modelling and verification. It is a variation of the B Method [2]. Automated support for modelling and verification in Event-B is provided by the Rodin platform [26].

Formally, an Event-B model is defined by a tuple $(d, c, A, v, \Sigma, I, Init, E)$, where d stands for sets (data types), c are constants, v is a vector of model variables, Σ corresponds to a model state space defined by all possible values of the vector v. $A(d, c)$ is a conjunction of axioms defining properties of model data structures, while $I(d, c, v)$ is a conjunction of invariants defining model properties to be preserved. $Init$ is an non-empty set of model initial states, $Init \subseteq \Sigma$. Finally, E is a set of model *events* where each event e is a relation of the form $e \subseteq \Sigma \times \Sigma$.

The sets and constants of the model are stated in a separate component called CONTEXT, where their properties are postulated as axioms. The model variables, invariants and events,

[1]From now on, by safety cases we mean structured safety cases.

including initialisation event, are introduced in the component called MACHINE. The model variables are strongly typed by the constraining predicates in terms of invariants.

In general, an event e has the following form:

$$e \mathrel{\hat{=}} \textbf{any}\ lv\ \textbf{where}\ g\ \textbf{then}\ R\ \textbf{end},$$

where lv is a list of local variables, the guard g is a conjunction of predicates defined over the model variables, and the action R is a parallel composition of assignments over the variables.

The event guard defines when an event is enabled. If several events are enabled simultaneously then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. In general, the action of an event is a composition of assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := Expr(v)$, where x is a state variable and $Expr(v)$ is an expression over the state variables v. The non-deterministic assignment can be denoted as $x :\in S$ or $x :\mid Q(v, x')$, where S is a set of values and $Q(v, x')$ is a predicate. As a result of the non-deterministic assignment, *x* gets any value from S or it obtains a value x' such that $Q(v, x')$ is satisfied.

The Event-B language can also be extended by different kinds of attributes attached to model events, guards, variables, etc. We will use Event-B attributes to contain formulas or expressions to be used by external tools or Rodin plug-ins, e.g., *Linear Temporal Logic (LTL)* formulas to be checked.

Event-B semantics. The semantics of Event-B events is defined using before-after predicates [50]. A *before-after predicate (BA)* describes a relationship between the system states before and after execution of an event. Hence, the definition of an event presented above can be given as the relation describing the corresponding state transformation from v to v', such that:

$$e(v, v') = g_e(v) \wedge I(v) \wedge BA_e(v, v'),$$

where g_e is the guard of the event e, BA_e is the before-after predicate of this event, and v, v' are the system states before and after event execution respectively.

Sometimes, we need to explicitly reason about possible model states before or after some particular event. For this purpose, we introduce two sets – *before*(e) and *after*(e). Specifically, *before*(e) represents a set of all possible pre-states defined by the guard of the event *e*, while *after*(e) is a set of all possible post-states of the event *e*, i.e., *before*$(e) \subseteq \Sigma$ and *after*$(e) \subseteq \Sigma$ denote the domain and range of the relation e [37]:

$$\textit{before}(e) = \{v \in \Sigma \mid I(v) \wedge g_e(v)\},$$
$$\textit{after}(e) = \{v' \in \Sigma \mid I(v') \wedge (\exists v \in \Sigma \cdot I(v) \wedge g_e(v) \wedge BA_e(v, v'))\}.$$

To verify correctness of an Event-B model, we generate a number of *proof obligations* (*POs*). More precisely, for an initial (i.e., abstract) model, we prove that its initialisation and all events preserve the invariant:

$$A(d, c),\ I(d, c, v),\ g_e(d, c, v),\ BA_e(d, c, v, v') \vdash I(d, c, v'). \tag{INV}$$

Since the initialisation event has no initial state and guard, its proof obligation is simpler:

$$A(d,c),\ BA_{Init}(d,c,v')\ \vdash\ I(d,c,v'). \tag{INIT}$$

On the other hand, we verify event feasibility. Formally, for each event e of the model, its feasibility means that, whenever the event is enabled, its before-after predicate is well-defined, i.e., there exists some reachable after-state:

$$A(d,c),\ I(d,c,v),\ g_e(d,c,v)\ \vdash\ \exists\, v' \cdot BA_e(d,c,v,v'). \tag{FIS}$$

Refinement in Event-B. Event-B employs a top-down refinement-based approach to formal development of a system. The development starts from an abstract specification of the system (i.e., an abstract machine) and continues with stepwise unfolding of system properties by introducing new variables and events into the model (i.e., refinements). This type of a refinement is known as a *superposition refinement*. Moreover, Event-B formal development supports *data refinement* allowing us to replace some abstract variables with their concrete counterparts. In this case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants is called a *gluing invariant*.

To verify correctness of a refinement step, one needs to discharge a number of POs for a refined model. For brevity, here we show only essential ones. The full list of POs can be found in [4].

Let us introduce a shorthand $H(d,c,v,w)$ that stands for the hypotheses $A(d,c)$, $I(d,c,v)$ and $I'(d,c,v,w)$, where I and I' are respectively the abstract and the refined invariants, while v, w are respectively the abstract and concrete variables.

When refining an event, its guard can only be strengthened:

$$H(d,c,v,w),\ g'_e(d,c,w)\ \vdash\ g_e(d,c,v), \tag{GRD}$$

where g_e, g'_e are respectively the abstract and concrete guards of the event e.

The *simulation* proof obligation (SIM) requires to show that the action (i.e., the modelled state transition) of a refined event is not contradictory to its abstract version:

$$H(d,c,v,w),\ g'_e(d,c,w),\ BA'_e(d,c,w,w')\ \vdash\ \exists v'.BA_e(d,c,v,v') \wedge I'(d,c,v',w'), \tag{SIM}$$

where BA_e, BA'_e are respectively the abstract and concrete before-after predicates of the same event e, w and w' are the concrete variable values before and after this event execution.

All the described above proof obligations are automatically generated by the Rodin platform [26] that supports Event-B. Additionally, the tool attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation in proving (usually over 80% of POs are proved automatically).

Verification via theorem proving. Additionally, the Event-B formalism allows the developers to formulate theorems either in the model CONTEXT or MACHINE components. In the first case, theorems are logical statements about model static data structures that are provable (derivable) from the model axioms given in the CONTEXT component. In the latter case, these are logical statements about model dynamic properties that follow from the given formal definitions of the model events and invariants.

The *theorem* proof obligation (THM) indicates that this is a theorem proposed by the developers. Depending whether a theorem is defined in the CONTEXT or MACHINE components, it has a slightly different form. To highlight this difference, we use indexes C and M in this paper. The first variant of a proof obligation is defined for a theorem $T(d, c)$ in the CONTEXT component:

$$A(d, c) \vdash T(d, c). \tag{THM$_C$}$$

The second variant is defined for a theorem $T(d, c, v)$ in the MACHINE component:

$$A(d, c), I(d, c, v) \vdash T(d, c, v). \tag{THM$_M$}$$

2.2 Safety cases

A *safety case* is *"a structured argument, supported by a body of evidence that provides a convincing and valid case that a system is safe for a given application in a given operating environment"* [13, 19]. The construction, review and acceptance of safety cases are the valuable steps in safety assurance process of critical software systems. Several standards, e.g., ISO 26262 [40] for the automotive domain, EN 50128 [25] for the railway domain, and the UK Defence Standard [19], prescribe production and evaluation of safety (or more generally assurance) cases for certification of such critical systems [31].

In general, safety cases can be documented either textually or graphically. However, a growing number of industrial companies working with safety-critical systems adopt a graphical notation, namely *Goal Structuring Notation (GSN)* proposed by Kelly [44], in order to present safety arguments within safety cases [30]. GSN aims at graphical representation of safety case elements as well as the relationships that exist between these elements. The principal building blocks of the GSN notation are shown in Figure 1. Essentially, a safety case constructed using GSN consists of *goals*, *strategies* and *solutions*. Here *goals* are propositions in an argument that can be said to be true or false (e.g., claims of requirements to be met by a system). *Solutions* contain the information extracted from analysis, testing or simulation of a system (i.e., evidence) to show that the goals have been met. Finally, *strategies* are reasoning steps describing how goals are decomposed and addressed by sub-goals.

Thus, a safety case constructed in the GSN notation presents decomposition of the given safety case goals into sub-goals until they can be supported by the direct evidence (a solution). It also explicitly defines the argument strategies, relied assumptions, the context in which goals are declared, as well as justification for the use of a particular goal or strategy. If the contextual information contains a model, a special GSN symbol called *model* can be used instead of a regular GSN context element.

The elements of a safety case can be in two types of relationships: *"Is solved by"* and *"In context of"*. The former is used between goals, strategies and solutions, while the latter

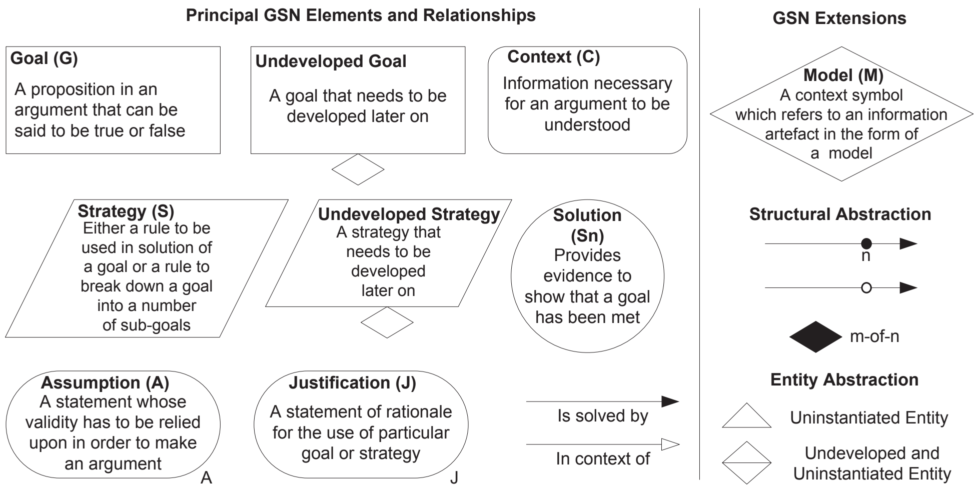

Principal GSN Elements and Relationships
Goal (G)
A proposition in an argument that can be said to be true or false
Undeveloped Goal
A goal that needs to be developed later on
Context (C)
Information necessary for an argument to be understood
Strategy (S)
Either a rule to be used in solution of a goal or a rule to break down a goal into a number of sub-goals
Undeveloped Strategy
A strategy that needs to be developed later on
Solution (Sn)
Provides evidence to show that a goal has been met
Assumption (A)
A statement whose validity has to be relied upon in order to make an argument
A
Justification (J)
A statement of rationale for the use of particular goal or strategy
J
Is solved by
In context of
GSN Extensions
Model (M)
A context symbol which refers to an information artefact in the form of a model
Structural Abstraction
n
m-of-n
Entity Abstraction
Uninstantiated Entity
Undeveloped and Uninstantiated Entity

Figure 1: Elements of GSN (detailed description is given in [7, 28, 44, 45])

links a goal to a context, a goal to an assumption, a goal to a justification, a strategy to a context, a strategy to an assumption, a strategy to a justification.

To allow for construction of argument patterns, GSN has been extended to represent generalised elements [44, 45]. We utilise the following elements from the extended GSN for structural abstraction of our argument patterns: *multiplicity* and *optionality*. Multiplicity is a generalised n-ary relationship between the GSN elements, while optionality stands for optional and alternative relationship between the GSN elements. Graphically, the former is represented as a solid ball or a hollow ball on an arrow *"Is solved by"* shown in Figure 1, where the label *n* indicates the cardinality of a relationship, while a hollow ball means zero or one. The latter is depicted as a solid diamond in Figure 1, where *m-of-n* denotes a possible number of alternatives. The multiplicity and the optionality relationships can be combined. If a multiplicity symbol is placed in front of the optionality symbol, this stands for a multiplicity over all the options.

There are two extensions for entity abstraction in GSN: (1) *uninstantiated entity*, and (2) *undeveloped and uninstantiated entity*. The former one specifies that the entity requires to be instantiated, i.e., the "abstract" entity needs to be replaced with a more concrete instance later on. In Figure 1, the corresponding annotation is depicted as a hollow triangle. It can be used with any GSN element. The latter one indicates that the entity needs both further development and instantiation. In Figure 1, it is shown as a hollow diamond with a line in the middle. This annotation can be applied to GSN goals and strategies only.

3 Methodology

In this section, we describe our methodology that aims at establishing a link between formal verification of safety requirements in Event-B and the construction of safety cases.

3.1 General methodology

In this work, we contribute to the process of development, verification and certification of software systems by showing how to proceed from the given safety requirements to safety cases via formal modelling and verification in Event-B (Figure 2). We distinguish two main processes: (1) representation of formalised safety requirements in Event-B models, and (2) derivation of safety cases from the associated Event-B specifications. Let us point out that these activities are tightly connected to each other. Accuracy of the safety requirements formalisation influences whether we are able to construct a safety case sufficient to demonstrate safety of a system. This dependence is highlighted in Figure 2 as a dashed line. If a formal specification is not good enough, we need to return and improve it.

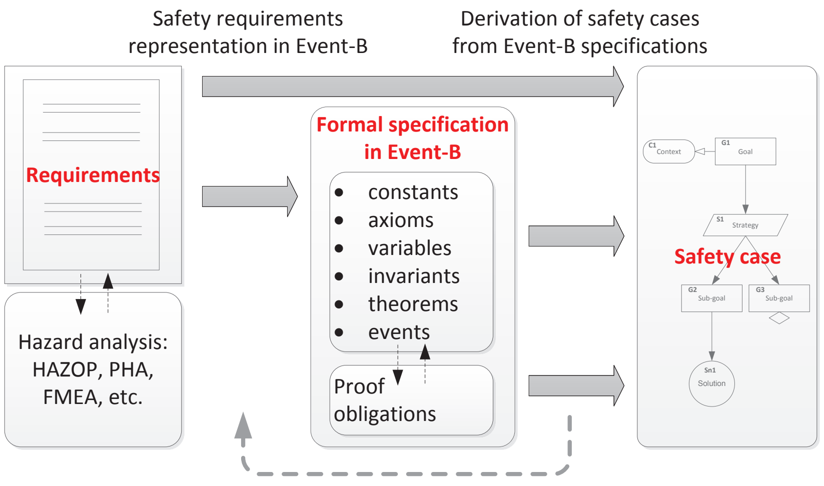

Safety requirements representation in Event-B
Derivation of safety cases from Event-B specifications
Requirements
Hazard analysis: HAZOP, PHA, FMEA, etc.
Formal specification in Event-B
constants
axioms
variables
invariants
theorems
events
Proof obligations
C1 Context
G1 Goal
S1 Strategy
Safety case
G2 Sub-goal
G3 Sub-goal
Sn1 Solution

Figure 2: High-level representation of the overall approach

We connect these two processes via classification of safety requirements. On the one hand, we propose a specific classification associated with particular ways these requirements can be represented in Event-B. On the other hand, we propose a set of classification-based argument patterns to facilitate the construction of safety cases from the associated Event-B models. The classification includes separate classes for safety requirements about global and local system properties, the absence of system deadlock, temporal and timing properties, etc. We are going to present this classification in detail in Section 3.2.

In this paper, we leave out of the scope the process of elicitation of system safety requirements. We assume that the given list of these requirements is completed beforehand by applying well-known hazard analysis techniques such as HAZard and OPerability (HAZOP) analysis, Preliminary Hazard Analysis (PHA), Failure Modes and Effects Analysis (FMEA), etc.

Incorporating safety requirements into formal models. Each class of safety requirements can be treated differently in an Event-B specification (model). In other words, various model expressions based on model elements, e.g., axioms, variables, invariants, events, etc., can be used to formalise a considered safety requirement. Consequently, the argument strategies and resulting evidence in a safety case built based on such a formal model may also vary. Using the defined classification, we provide the reader with the precise guidelines on how to map safety requirements of some class into a particular subset of model elements. Moreover, we define how to construct from these model elements a specific theorem to be verified. Later on, we will show how the verification results (e.g., discharged proof obligations and model checking results) can be used as the evidence in the associated safety cases.

Our methodology allows us to cope with two cases: (1) when a formal Event-B specification of the system under consideration has been already developed, and (2) when it is performed simultaneously with the safety case construction. In the first case, we assume that adequate models are constructed and linked with the classification we propose. In the second case, the formal development is guided by our proposed classification and methodology. Consequently, both ways allow us to contribute towards obtaining adequate safety cases.

Constructing safety cases from formal models. Model-based development in general and development using formal methods in particular typically require additional argumentation about model correctness and well-definedness [6]. In this paper, we address this challenge and provide the corresponding argument pattern as shown in Section 4.1.

Having a well-defined classification of safety requirements benefits both stages of the proposed methodology, i.e., while incorporating safety requirements into formal models and while deriving safety cases from such formal models. To simplify the task of linking the formalised safety requirements with the safety case to be constructed, we propose a set of classification-based argument patterns (Sections 4.2-4.9). The patterns have been developed using the corresponding GSN extensions (Figure 1). Some parts of an argument pattern may remain the same for any instance, while others need to be further instantiated (they are labelled with a specific GSN symbol – a hollow triangle). The text highlighted by braces { } should be replaced by a concrete value.

The generic representation of a classification-based argument pattern is given in Figure 3. Here, a safety requirement *Requirement* of some class *Class* $\{X\}$ is reflected in the goal **GX**, where X is a class number (see the next section for the reference). According to the proposed approach, the requirement is verified within a formal model M in Event-B (the model element **MX.1**).

In order to obtain the evidence that a specific safety requirement is met, different construction techniques might be undertaken. The choice of a particular technique influences the argumentation strategies to be used in each pattern. For example, if a safety requirement can be associated with a model invariant property, the corresponding theorem for each event in the model M is required to be proved. Correspondingly, the proofs of these theorems are attached as the evidence for the constructed safety case.

The formulated properties and theorems associated with a particular requirement can be automatically derived from the given formal model. Nonetheless, to increase clarity of a safety case, any theorem or property whose verification result is provided as a solution of the top goal needs to be referred to in the GSN context element (**CX.2** in Figure 3).

To bridge a semantic gap in the mapping associating an informally specified safety re-

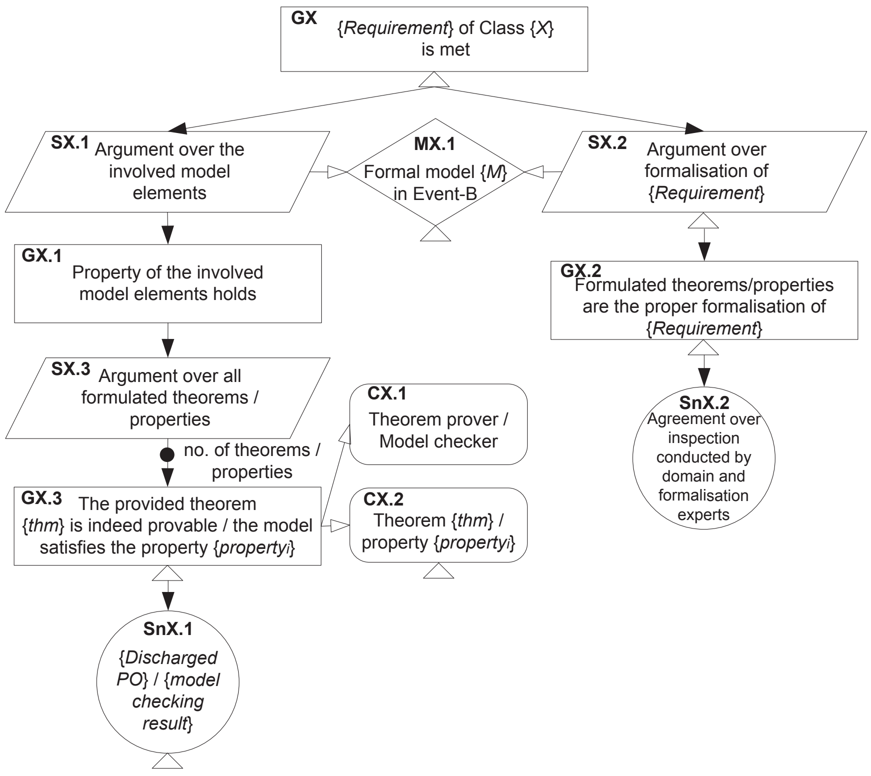

GX
{Requirement} of Class {X} is met
SX.1 Argument over the involved model elements
MX.1 Formal model {M} in Event-B
SX.2 Argument over formalisation of {Requirement}
GX.1 Property of the involved model elements holds
GX.2 Formulated theorems/properties are the proper formalisation of {Requirement}
SX.3 Argument over all formulated theorems / properties
CX.1 Theorem prover / Model checker
SnX.2 Agreement over inspection conducted by domain and formalisation experts
no. of theorems / properties
GX.3 The provided theorem {thm} is indeed provable / the model satisfies the property {propertyi}
CX.2 Theorem {thm} / property {propertyi}
SnX.1 {Discharged PO} / {model checking result}

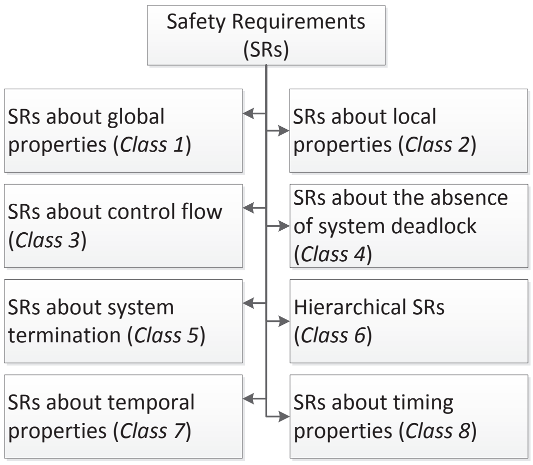
Figure 3: Generic argument pattern

quirement with the corresponding formal expression that is verified and connected to evidence, we need to argue over a correct formalisation of the requirement (**SX.2** in Figure 3). We rely on a joint inspection conducted by domain and formalisation experts (**SnX.2**) as the evidence that the formulated theorems/properties are proper formalisations of the requirement.

Generating code. Additionally, the most detailed (concrete) specification obtained during the refinement-based development can be used for code generation. The Rodin platform, Event-B tool support, allows for program code generation utilising a number of plug-ins. One of these plug-ins, EB2ALL [24], automatically generates a target programming language code from an Event-B formal specification. In particular, EB2C allows for generation of C code, EB2C++ supports C++ code generation, using EB2J one can obtain Java code, and using EBC# – C# code. The alternative solution is to use the constructed formal specification for verification of an already existing implementation code. Then, if the code has successfully passed such a verification, the existing safety case derived from the formal specification implies the code safety for the verified safety properties. Nonetheless, in both cases a safety case based on formal analysis cannot be used solely. It requires additional argumentation, for example, over the correctness of the code generation process itself [30, 43].

3.2 Requirements classification and its mapping into Event-B elements

To classify safety requirements, we have firstly adopted the taxonomy proposed by Bitsch [15] as presented in our previous work [54]. However, the Bitsch's approach uses *Computational Tree Logic (CTL)* to specify the requirements and relies on model checking as a formal verification technique. The differences between the semantics of CTL and Event-B significantly restrict the use of the Bitsch's classification in the Event-B framework. As a result, we extensively modified the original classification. In this paper, we propose the following classification of safety requirements, as shown in Figure 4.



Safety Requirements (SRs)
SRs about global properties (*Class 1*)
SRs about local properties (*Class 2*)
SRs about control flow (*Class 3*)
SRs about the absence of system deadlock (*Class 4*)
SRs about system termination (*Class 5*)
Hierarchical SRs (*Class 6*)
SRs about temporal properties (*Class 7*)
SRs about timing properties (*Class 8*)

Figure 4: Classification of safety requirements

We divide safety requirements (*SRs*) into eight classes:

- *Class 1*: *SRs about global properties* are the requirements stipulating the system safety properties that must be always maintained by the modelled system;
- *Class 2*: *SRs about local properties* are the requirements that reflect the necessity of some property to be true at a specific system state;
- *Class 3*: *SRs about control flow* are the requirements that define the necessary flow (order) in occurrences of some system events;
- *Class 4*: *SRs about the absence of system deadlock* are the requirements related to a certain class of control systems where an unexpected stop of the system may lead to a safety-related hazard;
- *Class 5*: *SRs about system termination* are the requirements related to a certain class of control systems where non-termination of the system in a specific situation may lead to a safety-related hazard;
- *Class 6*: *Hierarchical SRs* are the requirements that are hierarchically structured to deal with the complexity of the system, i.e., a more general requirement may be decomposed into several more detailed ones;

- *Class 7*: *SRs about temporal properties* are the requirements that describe the properties related to reachability of specific system states;
- *Class 8*: *SRs about timing properties* are the requirements that establish timing constraints of a system, for example, of a safety-critical real-time system where the response time is crucial.

The given classes of SRs are represented differently in a formal model. For instance, SRs of *Class 1* are modelled as invariants in the MACHINE component, while SRs of *Class 2* are modelled by defining a theorem about the required post-state of a specific Event-B model event. However, in some cases requirements of *Class 2* can be also formalised as requirements of *Class 1* by defining implicative invariants, i.e., invariants that hold in specific system states. The SRs about control flow *(Class 3)* can be expressed as event-flow properties (e.g., by using Event-B extension – the graphical Usecase/Flow language [37]). The SRs about the absence of system deadlock *(Class 4)* are represented as deadlock freedom conditions, while the SRs of *Class 5* are modelled as shutdown conditions. In both cases, these conditions are turned into specific model theorems to be proved. The class of hierarchical SRs *(Class 6)* is expressed within Even-B based on refinement between the corresponding Event-B models. Finally, the associated ProB tool for the Rodin platform [52] allows us to support the SRs of *Class 7* by model checking.

Let us note however that the representation of timing properties *(Class 8)* in the Event-B framework is a challenging task. There are several works dedicated to address this issue [12, 18, 39, 58]. In this paper, we adopt the approach that establishes a link between timing constraints defined in Event-B and verification of real-time properties in Uppaal [39].

Formally, the described above relationships can be defined as a function F_M mapping safety requirements (*SRs*) into a set of the related model expressions:

$$\mathit{SRs} \rightarrow \mathcal{P}(\mathit{MExpr}),$$

where $\mathcal{P}(T)$ corresponds to a power set on elements of T and *MExpr* stands for a generalised type for all possible expressions that can be built from the model elements, i.e., *model expressions*. Here *model elements* are elements of Event-B models such as *axioms*, *variables*, *invariants*, *events*, and *attributes*. *MExpr* includes such model elements as trivial (basic) expressions. Among other possible expressions of this type are *state predicates* defining post-conditions and shutdown conditions, *event control flow* expressions as well as *Linear Temporal Logic (LTL)* and *Timed Computation Tree Logic (TCTL)* formulas based on elements of the associated Event-B model.

The defined strict mapping allows us to trace the safety requirements given in an informal manner into formal specifications in Event-B as well as into the accompanying means for verification, i.e., the Flow and ProB plug-ins and Uppaal. In Figure 5, we illustrate the steps of evidence construction in our proposed approach. Firstly, we map a safety requirement into a set of model expressions. Secondly, we construct a specific theorem or a set of theorems out of these model expressions, thus essentially defining the semantics of the formalised requirement. Finally, we prove each theorem using the theorem provers of Event-B or perform model checking using, e.g., Event-B extension ProB. As a result, we obtain either a discharged proof obligation or a result of model checking. We include such results into the fragment of a safety case corresponding to the considered safety requirement as the evidence that this requirement holds. Table 1 illustrates the correspondence between safety requirements of different classes, model expressions and constructed theorems.

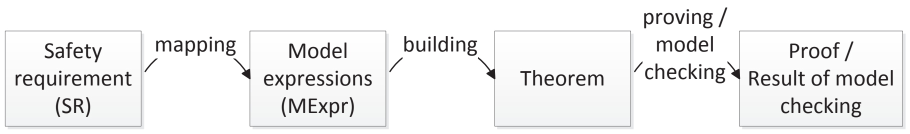


Figure 5: Steps of evidence construction

Table 1: Formalisation of safety requirements

Safety requirement	Model element expressions	Theorem
SR of Cl. 1	invariants	group of theorems for each event $Event_k/safety_i/INV$
SR of Cl. 2	event, state predicate	theorem about a specific post-state of an event thm_ap/THM
SR of Cl. 3	pairs of events, event control flow	group of theorems about enabling relationships between events, e.g., $Event_i/Event_j/FENA$
SR of Cl. 4	all events	theorem about the deadlock freedom thm_dlf/THM
SR of Cl. 5	state predicate, all events	theorem about a shutdown condition thm_shd/THM
SR of Cl. 6	abstract event, concrete event(s)	theorem about guard strengthening $Event'_k/grd/GRD$, theorem about action simulation $Event'_k/act/SIM$
SR of Cl. 7	LTL formula	*LTL* $property_i$
SR of Cl. 8	TCTL formula	*TCTL* $property_j$

As soon as all safety requirements are assigned to their respective classes and their mapping into Event-B elements is performed, we can construct the part of a safety case corresponding to assurance of these requirements. We utilise GSN to graphically represent such a safety case.

4 Argument patterns

In this section, we present the argument patterns corresponding to each of the introduced classes. In addition, to obtain an adequate safety case, we need to demonstrate well-definedness of the formal models we rely on. Therefore, we start this section by presenting a specific argument pattern to address this issue.

4.1 Argumentation that formal development of a system is well-defined

We propose the argument pattern shown in Figure 6 in order to provide evidence that the proposed formal development of a system is well-defined. To verify this (e.g., that a partial function is applied within its domain), Event-B defines a number of proof obligations (*well-definedness (WD)* for theorems, invariants, guards, actions, etc. and *feasibility (FIS)* for events [4]), which are automatically generated by the Rodin platform. We assume here that all such proof obligations are discharged for models in question. However, if model axioms are inconsistent (i.e., contradictory), the whole model becomes fallacious and thus logically meaningless. Demonstrating that this is not the case is a responsibility of the developer. To handle this problem, we introduce a specific argument pattern shown in Figure 6. In Event-B, well-definedness of a CONTEXT can be ensured by proving axiom consistency (the goal **G1.2** in Figure 6).

We propose to construct a theorem showing axiom consistency and prove it. However,

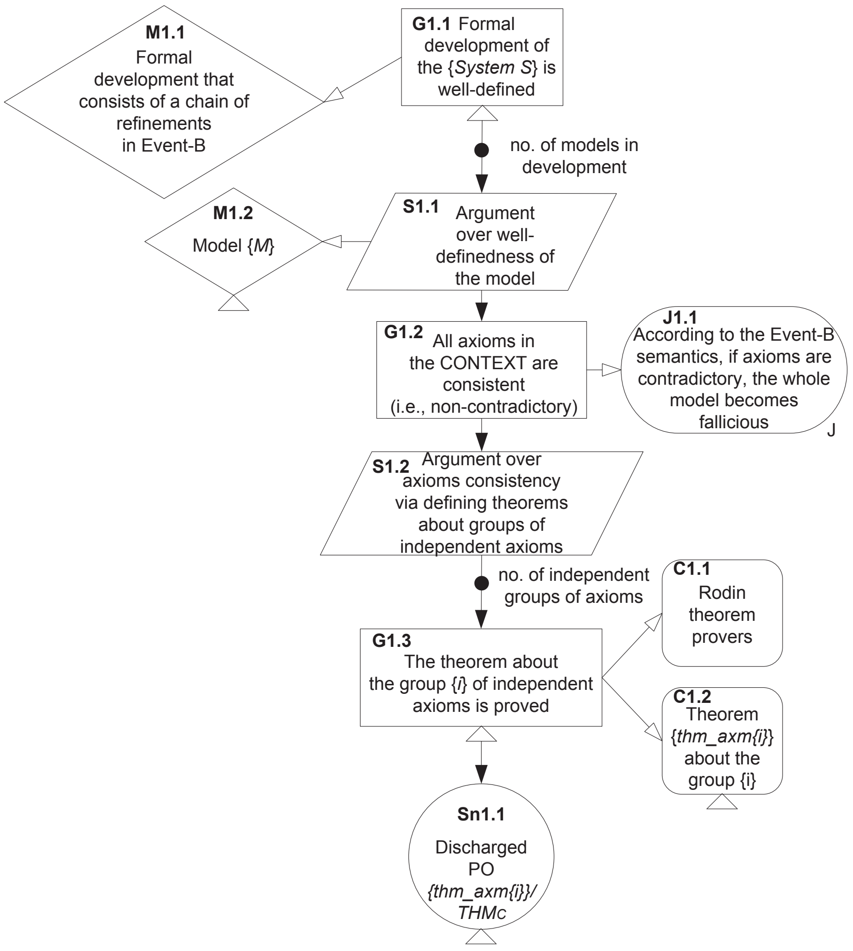

M1.1
Formal development that consists of a chain of refinements in Event-B
G1.1 Formal development of the {System S} is well-defined
no. of models in development
M1.2
Model {M}
S1.1 Argument over well-definedness of the model
G1.2 All axioms in the CONTEXT are consistent (i.e., non-contradictory)
J1.1
According to the Event-B semantics, if axioms are contradictory, the whole model becomes fallicious
J
S1.2 Argument over axioms consistency via defining theorems about groups of independent axioms
no. of independent groups of axioms
C1.1
Rodin theorem provers
G1.3
The theorem about the group {i} of independent axioms is proved
C1.2
Theorem {thm_axm{i}} about the group {i}
Sn1.1
Discharged PO {thm_axm{i}}/ THMc

Figure 6: Argument pattern for well-definedness of formal development

such a theorem could be very large in size. Thus, for simplicity, we suggest to divide axioms into groups, where each group consists of axioms that use shared constants and sets. In other words, each group of axioms is independent from each other. Consequently, we define theorems for all groups of independent axioms (the strategy **S1.2**) as shown below:

$$\textit{thm\_axm}\{i\}\text{:}\quad A(d,c) \vdash \exists\, d,c \cdot A_1(d,c) \wedge \ldots \wedge A_N(d,c),$$

where i stands for *i-th* group of axioms such that $i \in 1 \,..\, K$ and K is the number of independent groups of axioms. The number of axioms in a group is represented by N. The generated proof obligation (**Sn1.1**) shown in Figure 6 is an instance of the (THM_C) proof obligation given in Section 2.1.

In order to instantiate this pattern for each model in the development,

- a formal development that consists of a chain of refinements in Event-B should be defined in a GSN model element;
- a formal model *M*, for which a particular fragment of the safety case is constructed, should be referred to in a GSN model element;
- theorems about the defined groups of independent axioms should be formulated using the Event-B formal language and referred to in GSN context elements;
- the proof obligations of the type THM_C discharged by the Rodin platform should be included as solutions of the goal ensuring consistency of model axioms.

The instantiation example for this fragment of the safety case can be found in Section 5.3.1.

In the remaining part of Section 4, we introduce the argument patterns that correspond to each class of safety requirements proposed in Section 3.2. It is not necessarily the case that the final safety case of the modelled system will include SRs of all the classes. Moreover, it is very common for the Event-B practitioners to limit the requirements model representation to invariants, theorems and operation refinement [41, 49, 59]. However, to achieve a strong safety case, the developers need to provide the evidence that all the safety requirements listed in the requirements document hold. The proposed argument patterns cover a broader range of safety requirements, including also those that specify temporal and timing properties which cannot be formalised in Event-B directly.

4.2 Argument pattern for SRs about global properties (Class 1)

In this section, we propose an argument pattern for the safety requirements stipulating global safety properties, i.e., properties that must be maintained by the modelled system during its operation (Figure 7).

We assume that there is a model *M*, which is a part of the formal development of a system in Event-B, where a safety requirement of *Class 1* is verified to hold (**M2.1.1**). In addition, we assume that the model invariant $I(d,c,v)$ contains the conjuncts $\textit{safety}_1$, ..., $\textit{safety}_N$, where *N* is the number of safety invariants, which together represent a proper formalisation of the considered safety requirement (**A2.1.1**) [2]. Then, for each safety invariant $\textit{safety}_i$, $i \in$

[2]Such an assumption can be substantiated by arguing over formalisation of the requirements as demonstrated in Figure 3 (the strategy **SX.2**). It is applicable to all the classification-based argument patterns and their instances.

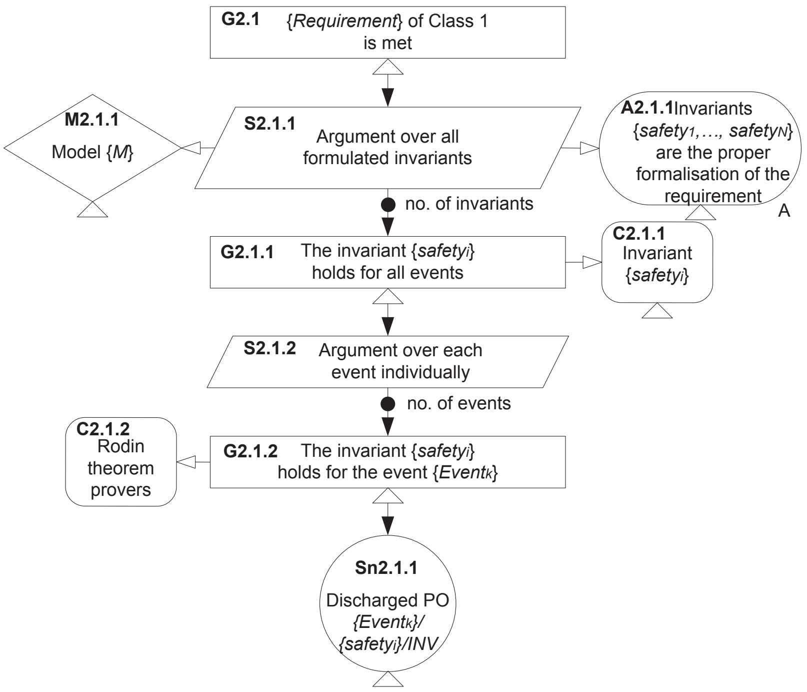

G2.1 {Requirement} of Class 1 is met
M2.1.1 Model {M}
S2.1.1 Argument over all formulated invariants
A2.1.1 Invariants {safety1,…, safetyN} are the proper formalisation of the requirement
A
no. of invariants
G2.1.1 The invariant {safetyi} holds for all events
C2.1.1 Invariant {safetyi}
S2.1.2 Argument over each event individually
no. of events
C2.1.2 Rodin theorem provers
G2.1.2 The invariant {safetyi} holds for the event {Eventk}
Sn2.1.1 Discharged PO {Eventk}/ {safetyi}/INV

Figure 7: Argument pattern for safety requirements of Class 1

1..N, the event $Event_k$, $k \in$ *1..K*, where *K* is the number of all model events, represents some event for which this invariant must hold.

We build the evidence for the safety case in the way illustrated in Figure 5. Thus, for each model expression, in this case, an invariant, formalising the safety requirement, we construct a separate fragment of the safety case. Then, a separate theorem is defined for each event where a particular invariant should hold. In other words, we define a group of theorems, one per each event. The number of model events influences the number of branches into which the goal **G2.1.1** is split. In a special case when the set of variables referred in an invariant is mutually exclusive with the set of variables modified by an event, such an event can be excluded from the list of events because the theorem generated for such an event is trivially true.

According to our approach, the generic mapping function F_M is of the form *SRs* $\rightarrow$ *$\mathcal{P}$(MExpr)*. In general case, for each requirement of this class the function returns a set of invariants $\{$*safety*$_1$, ..., *safety*$_N\}$ that can be represented as a conjunction. Due to this fact, each such an invariant can be verified independently. The theorem for verification that the safety invariant *safety*$_i$ (denoted by $I(d,c,v')$) holds for the event *Event*$_k$ is as follows:

$$A(d,c),\ I(d,c,v),\ g_{Event_k}(d,c,v),\ BA_{Event_k}(d,c,v,v') \vdash I(d,c,v'). \qquad \text{(INV)}$$

The Rodin platform allows us to prove this theorem using the integrated theorem provers and explicitly support the safety case to be built with the discharged proof obligations of the type INV for each event where the safety invariant has been verified to hold.

The key elements of the pattern to be instantiated are as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model invariants should be provided, while the invariants $safety_1$, ..., $safety_N$ formalising the requirement from this mapping should be referred to in GSN context elements;
- the proof obligations of the type INV discharged by the Rodin platform should be included in the safety case as the respective solutions.

Let us consider instantiation of the proposed pattern by an example – a sluice gate control system [46]. The system is a sluice connecting areas with dramatically different pressures. The purpose of the system is to adjust the pressure in the sluice area and operate two doors (*door1* and *door2*) connecting outside and inside areas with the sluice area. To guarantee safety, a door may be opened only if the pressure in the locations it connects is equalized, namely

SR-cl1-ex1: *When the door1 is open, the pressure in the sluice area is equal to the pressure outside*;
SR-cl1-ex2: *When the door2 is open, the pressure in the sluice area is equal to the pressure inside*.

These safety requirements are formalised in the Event-B model (available from Appendix C, Refinement 2 of [46]) as the invariants **inv_cl1_ex1** and **inv_cl1_ex2** such that

$$\textbf{SR-cl1-ex1} \mapsto \{\textbf{inv\_cl1\_ex1}\},$$
$$\textbf{SR-cl1-ex2} \mapsto \{\textbf{inv\_cl1\_ex2}\},$$

where:

$$\textbf{inv\_cl1\_ex1:}\ \textit{failure} = \textit{FALSE} \wedge (\textit{door1\_position} > 0 \vee \textit{door1\_motor} = \textit{MOTOR\_OPEN}) \Rightarrow \textit{pressure\_value} = \textit{PRESSURE\_OUTSIDE},$$
$$\textbf{inv\_cl1\_ex2:}\ \textit{failure} = \textit{FALSE} \wedge (\textit{door2\_position} > 0 \vee \textit{door2\_motor} = \textit{MOTOR\_OPEN}) \Rightarrow \textit{pressure\_value} = \textit{PRESSURE\_INSIDE}.$$

The expressions *doorX_position* > 0 and *doorX_motor* = *MOTOR_OPEN* indicate that the corresponding door *X* (where X = 1 or X = 2) is open. The variable *door1* models a door that connects the sluice area with the outside area and the variable *door2* models a door that connects the sluice area with inside area. The variable *pressure_value* stands for the pressure in the sluice area.

Then, according to the proposed approach, we show that these invariants hold for all events in the model. Due to the space limit, we give only an excerpt of the safety case that corresponds to the safety requirement **SR-cl1-ex1** (Figure 8). The associated invariant affects a number of model events, including such as *pressure_high* (changing pressure to high) and *closed2* (closing the door 2). From now on, we will hide a part of the safety case by three dots to avoid unnecessary big figures in the paper. We assume that the given part of the safety case is clear and can be easily repeated for the hidden items.

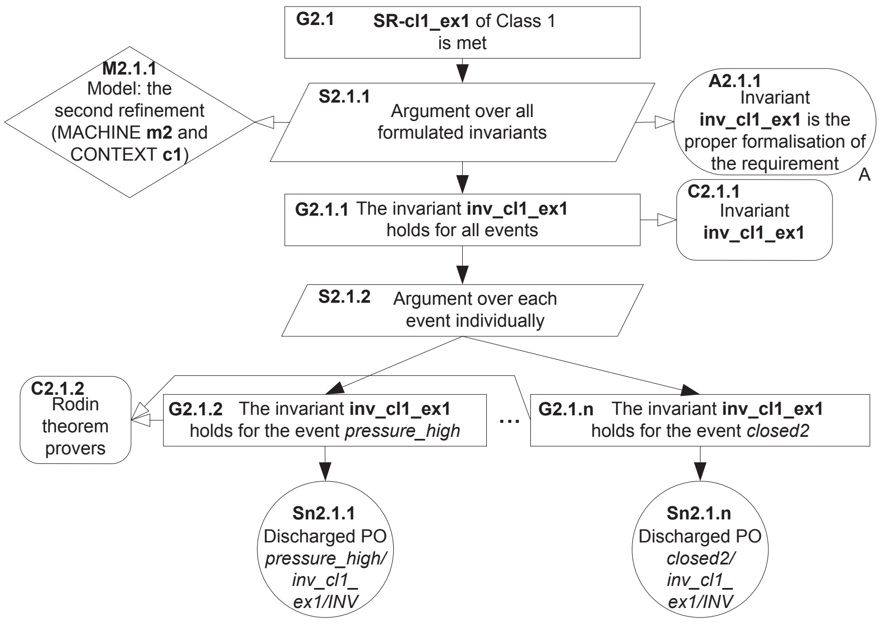

G2.1 SR-cl1_ex1 of Class 1 is met
M2.1.1 Model: the second refinement (MACHINE m2 and CONTEXT c1)
S2.1.1 Argument over all formulated invariants
A2.1.1 Invariant inv_cl1_ex1 is the proper formalisation of the requirement
A
G2.1.1 The invariant inv_cl1_ex1 holds for all events
C2.1.1 Invariant inv_cl1_ex1
S2.1.2 Argument over each event individually
C2.1.2 Rodin theorem provers
G2.1.2 The invariant inv_cl1_ex1 holds for the event pressure_high
...
G2.1.n The invariant inv_cl1_ex1 holds for the event closed2
Sn2.1.1 Discharged PO pressure_high/ inv_cl1_ ex1/INV
Sn2.1.n Discharged PO closed2/ inv_cl1_ ex1/INV

Figure 8: The pattern instantiation example

4.3 Argument pattern for SRs about local properties (Class 2)

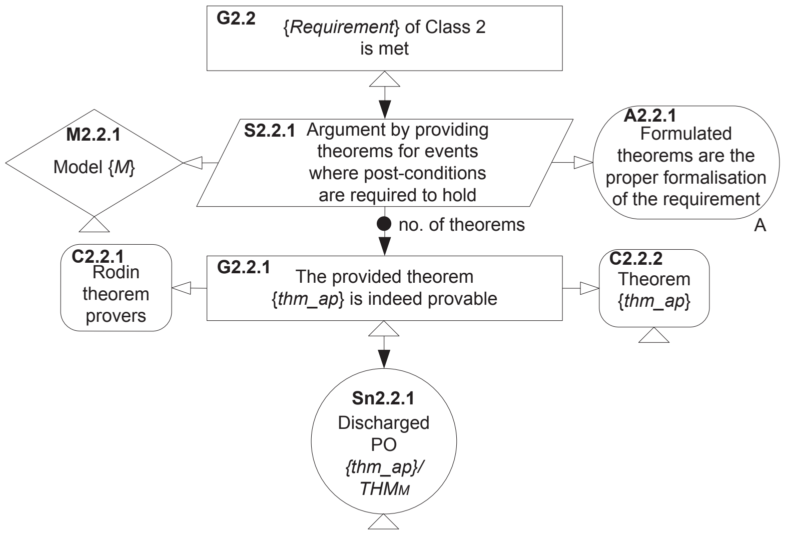
Safety requirements of *Class 2* describe local properties, i.e., the properties that need to be true at specific system states. For example, in case of a control system relying on the notion of operational modes, a safety requirement of *Class 2* may define a (safety) mode which the system enters after the execution of some transition. In terms of Event-B, the particular system states we are interested in are usually associated with some desired post-states of specific model events.

Figure 9 shows the argument pattern for justification of a safety requirement of *Class 2*. As for *Class 1*, the key argumentation strategy here (**S2.2.1**) is defined by the steps of evidence construction illustrated in Figure 5. However, in contrast to the invariant theorems established and proved for each event in the model, the theorem formalising the safety requirement of *Class 2* is formulated and proved only once for the whole model *M*.

As mentioned above, local properties are usually expressed in Event-B in terms of post-states of specific model events. This suggests the mapping function F_M for *Class 2* to be of the form:

$$\textit{Requirement} \mapsto \{(e_1, q_1),\ ...,\ (e_K, q_S)\},$$

where the events e_1,..., e_K and the state predicates q_1,..., q_S are model expressions based on which the corresponding theorems are constructed. The number of such theorems reflects the number of branches of a safety case for the goal **G2.2** (Figure 9). Specifically, we can verify a safety requirement of *Class 2* by proving the following theorem for each pair (e_i,q_j), where $i \in 1..K$ and $j \in 1..S$:



G2.2
{Requirement} of Class 2 is met
M2.2.1
Model {M}
S2.2.1 Argument by providing theorems for events where post-conditions are required to hold
A2.2.1
Formulated theorems are the proper formalisation of the requirement
A
no. of theorems
C2.2.1
Rodin theorem provers
G2.2.1
The provided theorem {thm_ap} is indeed provable
C2.2.2
Theorem {thm_ap}
Sn2.2.1
Discharged PO {thm_ap}/ THMM

Figure 9: Argument pattern for safety requirements of Class 2

$$\mathit{thm\_ap}: \quad A(d,c),\ I(d,c,v)\ \vdash\ \forall v' \cdot v' \in \mathit{after}(e_i)\ \Rightarrow\ q_j(v').$$

Here $\mathit{after}(e_i)$ is the set of all possible post-states of the event e_i as defined in Section 2.1.

This argument pattern (Figure 9) can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and event-post-condition pairs should be supplied, while the theorems *thm_ap* obtained from this mapping should be referred to in GSN context elements;
- the proof obligations of the type THM$_M$ discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.2**) holds.

In order to demonstrate application of this pattern, let us introduce another case study – Attitude and Orbit Control System (AOCS) [53]. The AOCS is a typical layered control system. The main function of this system is to control the attitude and the orbit of a satellite. Since the orientation of a satellite may change due to disturbances of the environment, the attitude needs to be continuously monitored and adjusted. At the top layer of the system there is a *mode manager (MM)*. The transitions between modes can be performed either to fulfil the predefined mission of the satellite (forward transitions) or to perform error recovery (backward transitions). Correspondingly, the MM component might be in either *stable*, *increasing* (i.e., in forward transition) or *decreasing* (i.e., in backward transition) state. As an example, let us consider the safety requirement

SR-cl2: *When a mode transition is completed, the state of the MM shall be stable.*

To verify this property on model events and variables, we need to prove that the corresponding condition q, namely

$$last\_mode = prev\_target \wedge next\_target = prev\_target,$$

holds after the execution of the event *Mode_Reached*. Here $prev\_target$ is the previous mode that a component was in transition to, $last\_mode$ is the last successfully reached mode, and $next\_target$ is the target mode that a component is currently in transition to. The event is enabled only when there is no critical error in the system, i.e., when the condition *error* $=$ *No_Error* holds.

We represent the mapping of the shown safety requirement on Event-B as F_M such that **SR-cl2** $\mapsto \{($*Mode_Reached*, $q)\}$. According to *thm_ap* and the definition of *after(e)* given in Section 2.1, we can construct the theorem to be verified as follows:

$$\begin{array}{ll}\textbf{thm\_cl2\_ex:} & \forall last\_mode',\ prev\_targ',\ next\_targ' \cdot \\ & (\exists\, next\_targ, error, prev\_targ \cdot \\ & (next\_targ \neq prev\_targ \wedge error = \mathit{No\_Error}) \wedge \\ & (last\_mode' = next\_targ \wedge prev\_targ' = next\_targ \wedge \\ & next\_targ' = next\_targ)) \\ & \Rightarrow \\ & last\_mode' = next\_targ \wedge prev\_targ' = next\_targ.\end{array}$$

Here, for simplicity, we omit showing types of the involved variables. The corresponding instance of the argument pattern is illustrated in Figure 10.

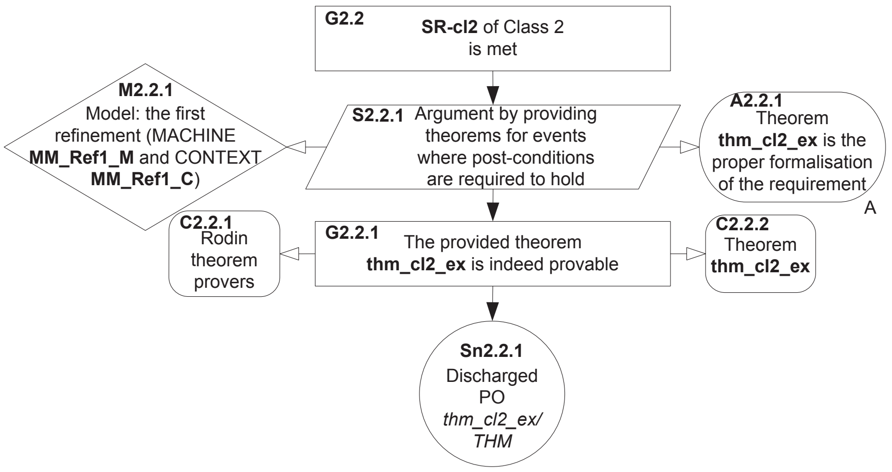

G2.2
SR-cl2 of Class 2 is met
M2.2.1
Model: the first refinement (MACHINE MM_Ref1_M and CONTEXT MM_Ref1_C)
S2.2.1 Argument by providing theorems for events where post-conditions are required to hold
A2.2.1
Theorem thm_cl2_ex is the proper formalisation of the requirement
A
C2.2.1
Rodin theorem provers
G2.2.1
The provided theorem thm_cl2_ex is indeed provable
C2.2.2
Theorem thm_cl2_ex
Sn2.2.1
Discharged PO thm_cl2_ex/ THM

Figure 10: The pattern instantiation example

4.4 Argument pattern for SRs about control flow (Class 3)

In this section, we propose an argument pattern for the requirements that define the flow in occurrences of some system events, i.e., safety requirements about control flow. For instance, this class may include certain requirements that define fault-tolerance procedures. Since fault detection, isolation and recovery actions are strictly ordered, we also need to preserve this order in a formal model of the system.

Formally, the ordering between system events can be expressed as a particular relationship amongst possible pre- and post-states of the corresponding model events. We consider three types of relationships proposed by Iliasov [37]: enabling (**ena**), disabling (**dis**) and possibly enabling (**fis**). In detail, enabling relationship between two events means that, when one event occurs, it is always true that the other one may occur next (i.e., the set of pre-states of the second event is included in the set of post-states of the first event). An event disables another event if the guard of the second event is always false after the first event occurs (i.e., the set of pre-states of the second event is excluded from the set of post-states of the first event). Finally, an event possibly enables another event if, after its occurrence, the guard of the second event is potentially enabled (i.e., there is a non-empty intersection of the set of pre-states of the second event with the set of post-states of the first event).

Let e_m and e_n be some events. Then, according to the usecase/flow approach [37], the proof obligations that support the relationships between these events can be defined as follows:

$$\begin{aligned} e_m \ \mathbf{ena}\ e_n \ &\Leftrightarrow \mathit{after}(e_m) \subseteq \mathit{before}(e_n) \\ &\Leftrightarrow \forall v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \Rightarrow \ g_{e_n}(v'), \end{aligned} \tag{FENA}$$

$$\begin{aligned} e_m \ \mathbf{dis}\ e_n \ &\Leftrightarrow \mathit{after}(e_m) \cap \mathit{before}(e_n) = \varnothing \\ &\Leftrightarrow \forall v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \Rightarrow \ \neg g_{e_n}(v'), \end{aligned} \tag{FDIS}$$

$$\begin{aligned} e_m \ \mathbf{fis}\ e_n \ &\Leftrightarrow \mathit{after}(e_m) \cap \mathit{before}(e_n) \neq \varnothing \\ &\Leftrightarrow \exists v, v' \cdot I(v) \ \wedge \ g_{e_m}(v) \ \wedge \ BA_{e_m}(v, v') \ \wedge \ g_{e_n}(v'). \end{aligned} \tag{FFIS}$$

The flow approach and its supporting plug-in for the Rodin platform, called Usecase/Flow plug-in [27], allows us to derive these proof obligations automatically.

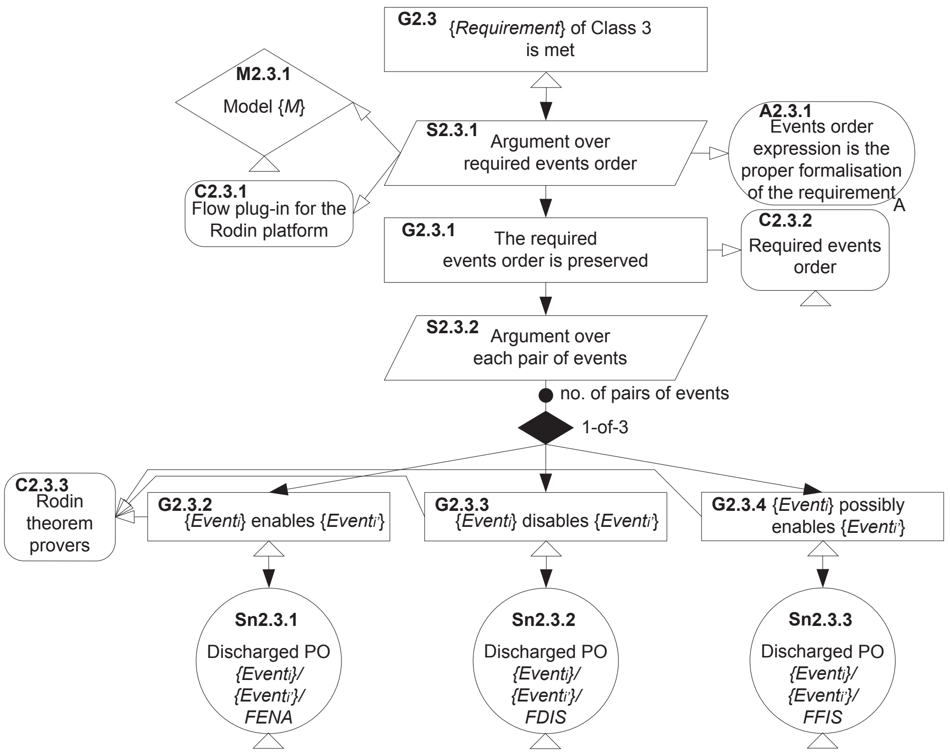
The argument pattern shown in Figure 11 pertains to the required events order (**C2.3.2**) which is proved to be preserved by the respective events of a model *M*. As explained above, each event $Event_{i'}$ can be either enabled (**ena**), or disabled (**dis**), or possibly enabled (**fis**) by some other event $Event_i$. This suggests that the mapping function F_M is of the form:

$$\begin{aligned} \mathit{Requirement} \mapsto \{&(Event_i,\ \mathbf{ena},\ Event_{i'}), \\ &(Event_j,\ \mathbf{dis},\ Event_{j'}), \\ &(Event_k,\ \mathbf{fis},\ Event_{k'}),\ ...\ \}. \end{aligned}$$

The corresponding theorem is constructed according to the definition of either (FENA), or (FDIS), or (FFIS). Then, the discharged proof obligations for each such a pair of events are provided as the evidence in a safety case, e.g., **Sn2.3.1** in Figure 11.

The instantiation of the proposed argument pattern can be achieved by preserving a number of the following steps:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;



G2.3 {Requirement} of Class 3 is met
M2.3.1 Model {M}
S2.3.1 Argument over required events order
A2.3.1 Events order expression is the proper formalisation of the requirement
A
C2.3.1 Flow plug-in for the Rodin platform
G2.3.1 The required events order is preserved
C2.3.2 Required events order
S2.3.2 Argument over each pair of events
no. of pairs of events
1-of-3
C2.3.3 Rodin theorem provers
G2.3.2 {Eventi} enables {Eventi'}
G2.3.3 {Eventi} disables {Eventi'}
G2.3.4 {Eventi} possibly enables {Eventi'}
Sn2.3.1 Discharged PO {Eventi}/ {Eventi'}/ FENA
Sn2.3.2 Discharged PO {Eventi}/ {Eventi'}/ FDIS
Sn2.3.3 Discharged PO {Eventi}/ {Eventi'}/ FFIS

Figure 11: Argument pattern for safety requirements of Class 3

- the concrete mapping between the requirement and the corresponding pairs of events and relationships between them should be provided, while the required events order based on this mapping should be referred to in a GSN context element;
- a separate goal for each pair should be introduced in the safety case;
- each goal that claims the enabling relationship between events should be supported by the proof obligation of the type FENA in a GSN solution element;
- each goal that claims the disabling relationship between events should be supported by the proof obligation of the type FDIS in a GSN solution element;
- each goal that claims the possibly enabling relationship between events should be supported by the proof obligation of the type FFIS in a GSN solution element.

In the already introduced case study AOCS (Section 4.3), there is a set of requirements regulating the order of actions to take place in the system control flow. These requirements define the desired rules of transitions between modes, e.g.,

SR-cl3: *The system shall perform its (normal or failure handling) operation only when there are no currently running transitions between modes at any level.*

This means that once a transition is initiated either by the high-level mode manager or lower level managers, it has to be completed before system operation continues.

As an example, we consider a formalisation of the requirement **SR-cl3** at the most abstract level, i.e., the MACHINE **MM_Abs_M** and the CONTEXT **MM_Abs_C**, where the essential behaviour of the high-level mode manager is introduced.

The required events order (**C2.3.2**) is depicted by the usecase/flow diagram in Figure 12. This flow diagram can be seen as a use case scenario specification attached to the MACHINE **MM_Abs_M**. The presented flow diagram is drawn in the graphical editor for the Usecase/Flow plug-in for the Rodin platform. While defining the desired relationships between events using this editor, the corresponding proof obligations are generated automatically by the Rodin platform.

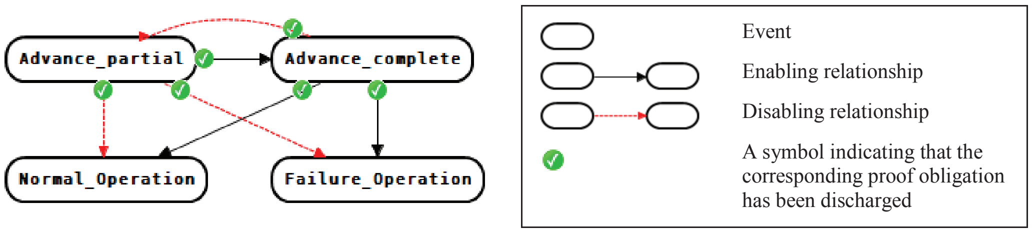


Figure 12: The partial flow diagram of the abstract machine of AOCS

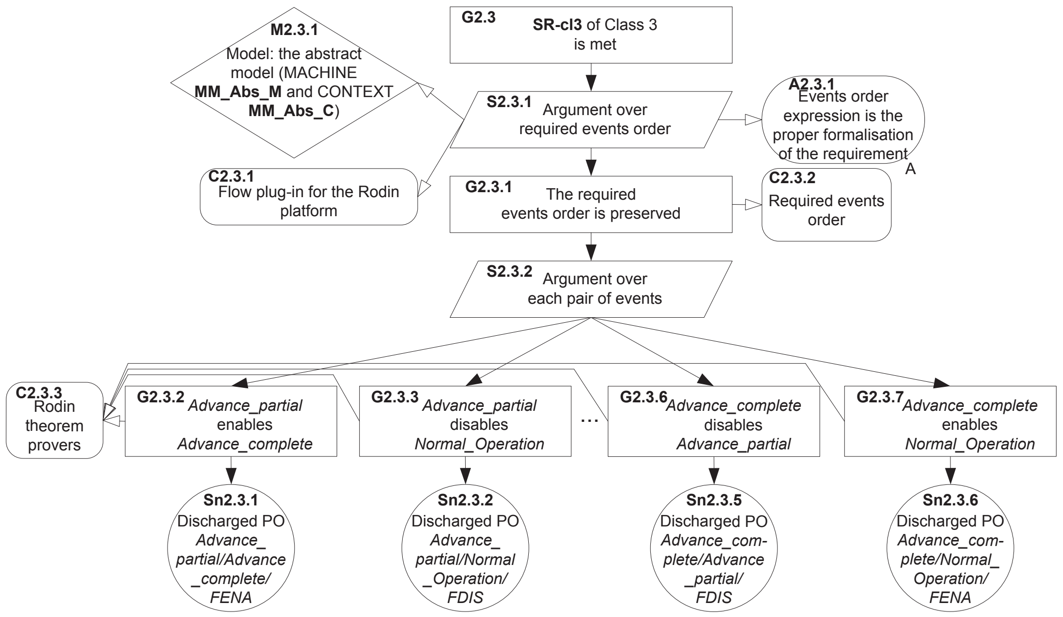
In terms of the usecase/flow approach, the requirement **SR-cl3** states that the event *Advance_partial* enables the event *Advance_complete* and disables operation events *Normal_Operation* and *Failure_Operation*. In its turn, the event *Advance_complete* disables the event *Advance_partial* and enables system (normal or failure handling) operation events. Then, the mapping function F_M is instantiated as follows:

$$\begin{aligned}\textbf{SR-cl3} \mapsto \{&(\textit{Advance\_partial}, \ \textbf{ena}, \ \textit{Advance\_complete}),\\ &(\textit{Advance\_partial}, \ \textbf{dis}, \ \textit{Normal\_Operation}),\\ &(\textit{Advance\_partial}, \ \textbf{dis}, \ \textit{Failure\_Operation}),\\ &(\textit{Advance\_complete}, \ \textbf{dis}, \ \textit{Advance\_partial}),\\ &(\textit{Advance\_complete}, \ \textbf{ena}, \ \textit{Normal\_Operation}),\\ &(\textit{Advance\_complete}, \ \textbf{ena}, \ \textit{Failure\_Operation})\}.\end{aligned}$$

The instance of the argument pattern for the safety requirement **SR-cl3** is shown in Figure 13.

4.5 Argument pattern for SRs about the absence of system deadlock (Class 4)

In this section, we propose an argument pattern for the safety requirements stipulating the absence of the unexpected stop of the system (Figure 14). We formalise requirements of *Class 4* within an Event-B model *M* as the deadlock freedom theorem. Similarly to the SRs of *Class 2*, this theorem has to be proved only once for the whole model *M*. The theorem is reflected in the argument strategy that is used to develop the main goal of the pattern (**S2.4.1** in Figure 14).



M2.3.1
Model: the abstract model (MACHINE MM_Abs_M and CONTEXT MM_Abs_C)
G2.3
SR-cl3 of Class 3 is met
S2.3.1
Argument over required events order
A2.3.1
Events order expression is the proper formalisation of the requirement
A
C2.3.1
Flow plug-in for the Rodin platform
G2.3.1
The required events order is preserved
C2.3.2
Required events order
S2.3.2
Argument over each pair of events
C2.3.3
Rodin theorem provers
G2.3.2
Advance_partial enables Advance_complete
G2.3.3
Advance_partial disables Normal_Operation
...
G2.3.6
Advance_complete disables Advance_partial
G2.3.7
Advance_complete enables Normal_Operation
Sn2.3.1
Discharged PO Advance_partial/Advance_complete/FENA
Sn2.3.2
Discharged PO Advance_partial/Normal_Operation/FDIS
Sn2.3.5
Discharged PO Advance_complete/Advance_partial/FDIS
Sn2.3.6
Discharged PO Advance_complete/Normal_Operation/FENA

Figure 13: The pattern instantiation example

Formally, the deadlock freedom theorem is formulated as the disjunction of guards of all model events $g_1(d, c, v) \vee ... \vee g_K(d, c, v)$, where K is the total number of model events:

$$thm\_dlf: \quad A(d, c),\ I(d, c, v)\ \vdash\ g_1(d, c, v) \vee ... \vee g_K(d, c, v).$$

The corresponding mapping function F_M for this argument pattern is defined as *Requirement* $\mapsto \{event_1,\ ...\ ,\ event_K\}$. Then, the instance of the (THM$_M$) proof obligation given in Section 2.1 provides the evidence for the safety case (**Sn2.4.1** in Figure 14).

The argument pattern presented in Figure 14 can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model events should be supplied, while the theorem *thm_dlf* formalising the requirement from this mapping should be referred to in a GSN context element;
- the proof obligation of the type THM$_M$ discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.4** in Figure 14) holds.

We illustrate the instantiation of this argument pattern by a simple example presented by Abrial in Chapter 2 of [4]. The considered system performs controlling cars on a bridge. The bridge connects the mainland with an island. Cars can always either enter the compound or leave it. Therefore, the absence of the system deadlock should be guaranteed, i.e.,

SR-cl4: *Once started, the system should work for ever.*

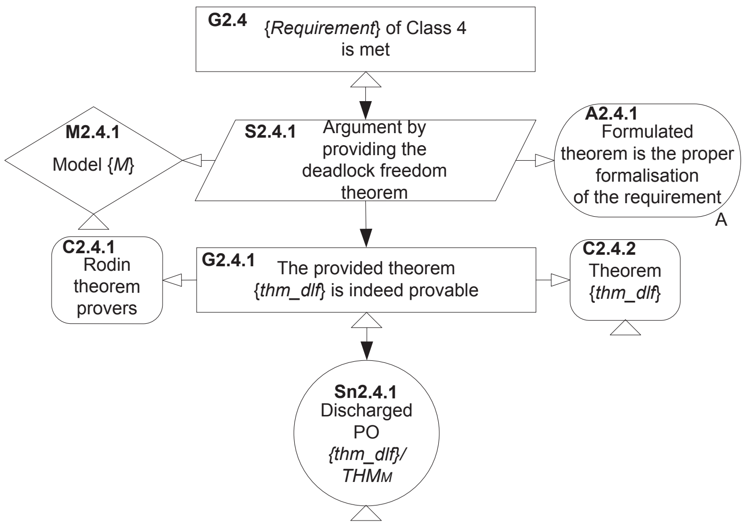

G2.4
{Requirement} of Class 4 is met
M2.4.1
Model {M}
S2.4.1
Argument by providing the deadlock freedom theorem
A2.4.1
Formulated theorem is the proper formalisation of the requirement
A
C2.4.1
Rodin theorem provers
G2.4.1
The provided theorem {thm_dlf} is indeed provable
C2.4.2
Theorem {thm_dlf}
Sn2.4.1
Discharged PO {thm_dlf}/ THMM

Figure 14: Argument pattern for safety requirements of Class 4

The semantics of Event-B allows us to chose the most abstract specification to argue over the deadlock freedom of a system. According to the notion of the *relative deadlock freedom*, which is a part of the Event-B semantics, new deadlocks cannot be introduced in a refinement step [3]. As a consequence, once the model is proved to be deadlock free, no new refinement step can introduce a deadlock.

The abstract model of the system has three events: *Initialisation*, *ML_out* and *ML_in*. Thus, the concrete mapping function F_M is as follows:

$$\textbf{SR-cl4} \mapsto \{\textit{Initialisation}, \textit{ML\_out}, \textit{ML\_in}\}.$$

Here *ML_out* models leaving the mainland, while *ML_in* models entering the mainland. The former event has the guard $n < d$, where n is a number of cars on the bridge and d is a maximum number of cars that can enter the bridge. The latter event is guarded by the condition $n > 0$, which allows this event to be enabled only when some car is on the island or the bridge. Therefore, the corresponding deadlock freedom theorem **thm_cl4_ex** can be defined as follows:

$$\textbf{thm\_cl4\_ex:} \quad n > 0 \lor n < d.$$

The event *Initialisation* does not have a guard and therefore is not reflected in the theorem. The instantiated fragment of the safety case for this example is shown in Figure 15.

The details on the considered formal development in Event-B (Controlling cars on a bridge) as well as the derived proof obligation of the deadlock freedom can be found in [3,4].

[3]This may be enforced by the corresponding generated theorem to be proved for the respective model.

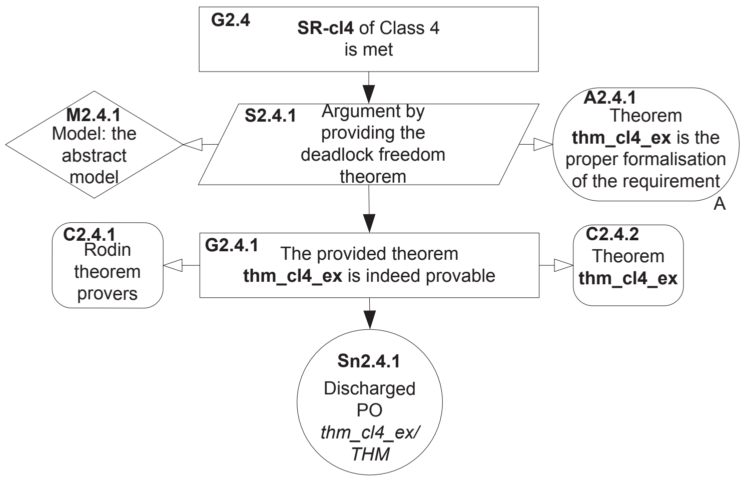

G2.4
SR-cl4 of Class 4 is met
M2.4.1
Model: the abstract model
S2.4.1
Argument by providing the deadlock freedom theorem
A2.4.1
Theorem **thm_cl4_ex** is the proper formalisation of the requirement
A
C2.4.1
Rodin theorem provers
G2.4.1
The provided theorem **thm_cl4_ex** is indeed provable
C2.4.2
Theorem **thm_cl4_ex**
Sn2.4.1
Discharged PO *thm_cl4_ex/ THM*

Figure 15: The pattern instantiation example

4.6 Argument pattern for SRs about system termination (Class 5)

In contrast to *Class 4*, *Class 5* contains the safety requirements stipulating the system termination in particular cases. For instance, it corresponds to failsafe systems (i.e., systems which need to be put into a safe but non-operational state to prevent an occurrence of a hazard). Despite the fact that the argument pattern is quite similar to the one about the absence of system deadlock, this class of safety requirements can be considered as essentially opposite to the previous one. Here the requirements define the conditions when the system must terminate. More specifically, the system is required to have a deadlock either (1) in a specific state of the model *M*, i.e., after the execution of some event e_i (where $i \in 1 \,..\, K$ and *K* is the total number of model events), or (2) once a shutdown condition (*shutdown_cond*) is satisfied:

$$\begin{array}{ll}(1) & \mathit{after}(e_i) \cap \mathit{before}(E) = \varnothing,\\ (2) & \mathit{shutdown\_cond} \cap \mathit{before}(E) = \varnothing,\end{array}$$

where *shutdown_cond* is a predicate formalising a condition when the system terminates and *before(E)* is defined as a union of pre-states of all the model events:

$$\mathit{before}(E) = \bigcup_{e \in E} \mathit{before}(e).$$

Correspondingly, the mapping function F_M for *Class 5* can be either of the form

$$\begin{array}{ll}(1) & \mathit{Requirement} \mapsto \{e_i, e_1, ..., e_K\}, \text{ or}\\ (2) & \mathit{Requirement} \mapsto \{\mathit{state\ predicate}, e_1, ..., e_K\},\end{array}$$

where *state predicate* is a formally defined shutdown condition.

Then, for the first case, the theorem about a shutdown condition has the following form:

$$\mathit{thm\_shd}: \quad A(d,c),\ I(d,c,v)\ \vdash\ \mathit{after}(e_i) \Rightarrow \neg\mathit{before}(E),$$

while, for the second case, it is defined as:

$$\textit{thm\_shd}: \quad A(d,c),\ I(d,c,v)\ \vdash\ \textit{shutdown\_cond} \Rightarrow \neg \textit{before}(E).$$

The argument pattern presented in Figure 16 can be instantiated as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding model events (and state predicates) should be provided, while the theorem *thm_shd* formalising the requirement from this mapping should be referred to in a GSN context element;
- the proof obligation of the type THM$_M$ discharged by the Rodin platform should be included in the safety case as the evidence supporting that the top-level claim (i.e., **G2.5**) holds.

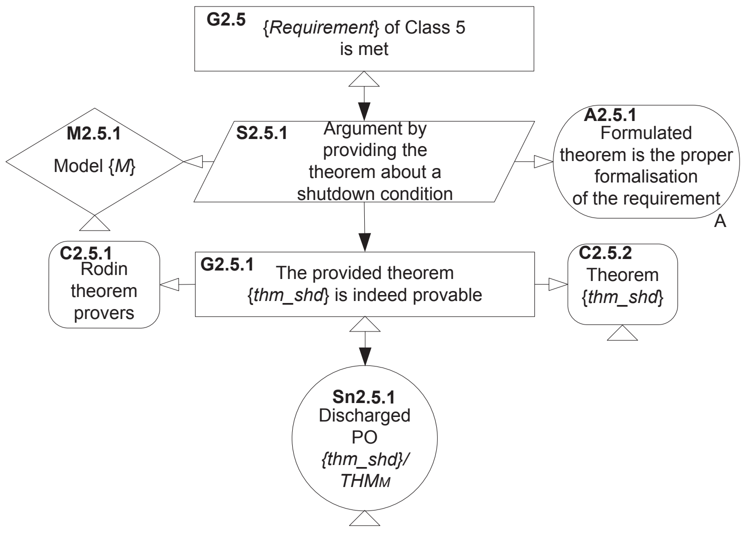

G2.5 {Requirement} of Class 5 is met
M2.5.1 Model {M}
S2.5.1 Argument by providing the theorem about a shutdown condition
A2.5.1 Formulated theorem is the proper formalisation of the requirement
A
C2.5.1 Rodin theorem provers
G2.5.1 The provided theorem {thm_shd} is indeed provable
C2.5.2 Theorem {thm_shd}
Sn2.5.1 Discharged PO {thm_shd}/ THMM

Figure 16: Argument pattern for safety requirements of Class 5

To show an example of the pattern instantiation, let us consider the sluice gate control system [46] described in detail in Section 4.2. This system is a failsafe system. To handle critical failures, it is required to raise an alarm and terminate:

SR-cl5: *When a critical failure is detected, an alarm shall be raised and the system shall be stopped.*

Thus, we need to assure that our model also terminates after the execution of the event which sets the alarm on (i.e., the event *SafeStop* in the model). This suggests the concrete instance of the mapping function F_M to be of the form:

$$\textbf{SR-cl5} \mapsto \{\textit{SafeStop}, \textit{Environment}, \textit{Detection\_door1}, ..., \textit{close2}, \textit{closed2}\}.$$

Then, the corresponding theorem **thm_cl5_ex**, which formalises the safety requirement **SR-cl5**, can be formulated as follows:

$$\begin{aligned}
\textbf{thm\_cl5\_ex:}\ \ & \forall flag', Stop' \cdot \\
& (\exists\, flag,\ door1\_fail,\ door2\_fail,\ pressure\_fail,\ Stop \cdot \\
& flag = CONT \land (door1\_fail = TRUE \lor \\
& door2\_fail = TRUE\ \lor\ pressure\_fail = TRUE) \land \\
& Stop = FALSE\ \land\ flag' = PRED \land Stop' = TRUE) \\
& \Rightarrow \\
& \neg(before(Environment) \lor before(Detection\_door1) \lor ... \lor \\
& before(close2) \lor before(closed2)),
\end{aligned}$$

where the variable *flag* indicates the current phase of the sluice gate controller, while the variables *door1_fail*, *door2_fail* and *pressure_fail* stand for failures of the system components (the doors and the pressure pump respectively). The variable *Stop* models an alarm and a signal to stop the physical operation of the system components. Finally, *Environment*, *Detection_door1*, ..., *closed2* are model events. The corresponding instance of the argument pattern is given in Figure 17.

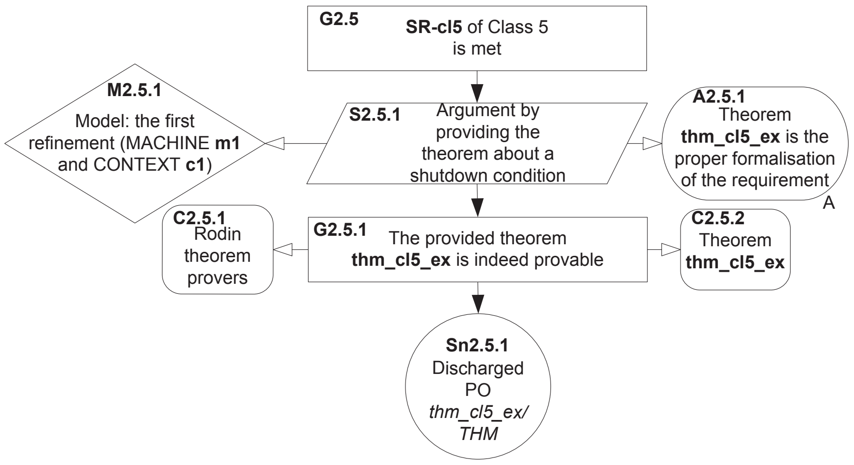

G2.5
SR-cl5 of Class 5 is met
M2.5.1
Model: the first refinement (MACHINE m1 and CONTEXT c1)
S2.5.1
Argument by providing the theorem about a shutdown condition
A2.5.1
Theorem thm_cl5_ex is the proper formalisation of the requirement
A
C2.5.1
Rodin theorem provers
G2.5.1
The provided theorem thm_cl5_ex is indeed provable
C2.5.2
Theorem thm_cl5_ex
Sn2.5.1
Discharged PO thm_cl5_ex/ THM

Figure 17: The pattern instantiation example

4.7 Argument pattern for Hierarchical SRs (Class 6)

Sometimes a whole requirements document or some particular requirements (either functional or safety) of a system may be structured in a hierarchical way. For example, a general safety requirement may stipulate actions to be taken in the case of a system failure, while more specific safety requirements elaborate on the general requirement by defining how the failures of different system components may contribute to such a failure of the system as well as regulate the actions to mitigate these failures. Often, the numbering of requirements may indicate such intended hierarchical relationships. A more general requirement can be numbered *REQ X*, while its more specific versions – *REQ X.1*, *REQ X.2*, etc. In our classification, we call such requirements *Hierarchical SRs*.

The class of *Hierarchical SRs* (*Class 6*) differs from the previously described classes since it involves several, possibly quite different yet hierarchically linked requirements. To create the corresponding argument patterns for such cases, we apply a *composite* approach. This means that the involved individual requirements (a more general requirement and its more detailed counterparts) can be shown to hold separately in different models of the system development in Event-B, by instantiating suitable argument patterns from the described classes 1-5. Moreover, to ensure the consistency of their hierarchical link, an additional fragment in a safety case is needed. This fragment illustrates that the formalisation of the involved requirements is consistent, even if it is done in separate models of the Event-B formal development. To address the class of hierarchical requirements, in this section we propose an argument pattern that facilitates the task of construction of such an additional fragment of a safety case.

Since the main property of the employed refinement approach is the preservation of consistency between the models, it is sufficient for us to show that the involved models are valid refinements of one another. In Event-B, to guarantee consistency of model transformations, we need to show that the concrete events refine their abstract versions by discharging the corresponding proof obligations to verify guard strengthening (GRD) and action simulation (SIM), as given in Section 2.1. This procedure may involve the whole set of the refined events. However, to simplify the construction of the corresponding fragment of a safety case, we limit the number of events by choosing only those events that are affected by the requirements under consideration. To achieve this, we rely on the given mappings for higher-level and lower-level requirements, returning the sets of the involved model expressions $Req_h \Rightarrow \{Expr_1, ..., Expr_N\}$ and $Req_l \Rightarrow \{Expr_1, ..., Expr_P\}$. Making a step further, we can always obtain the set of affected model events:

$$Req_h \Rightarrow \{Expr_1, ..., Expr_N\} \Rightarrow \{Event_{h_1}, ..., Event_{h_K}\},$$
$$Req_l \Rightarrow \{Expr_1, ..., Expr_P\} \Rightarrow \{Event'_{l_1}, ..., Event'_{l_L}\}.$$

As a result, we attach proofs only for those events from $\{\text{Event}'_{l_1}$, ..., $\text{Event}'_{l_L}\}$ that refine some events from $\{\text{Event}_{h_1}$, ..., $\text{Event}_{h_K}\}$.

Each higher-level requirement may be linked with a set of more detailed requirements in the requirements document. Nevertheless, to simplify the task, let us consider the case where there is only one such a lower-level requirement. If there are more than one such a requirement, one could reiterate the proposed approach by building a separate fragment of a safety case for each pair of linked requirements.

In Figure 18, *Higher-level req.* stands for some higher-level requirement, while *Lower-level req.* is a requirement that is a more detailed version of the higher-level one. The higher-level requirement is mapped onto a formal model M_{abs} and the lower-level requirement is mapped onto a formal model M_{concr} (where M_{concr} is a refinement of M_{abs}) using one of the mapping functions defined for the classes 1-5.

Following the procedure described above, we can associate *Higher-level req.* with the set

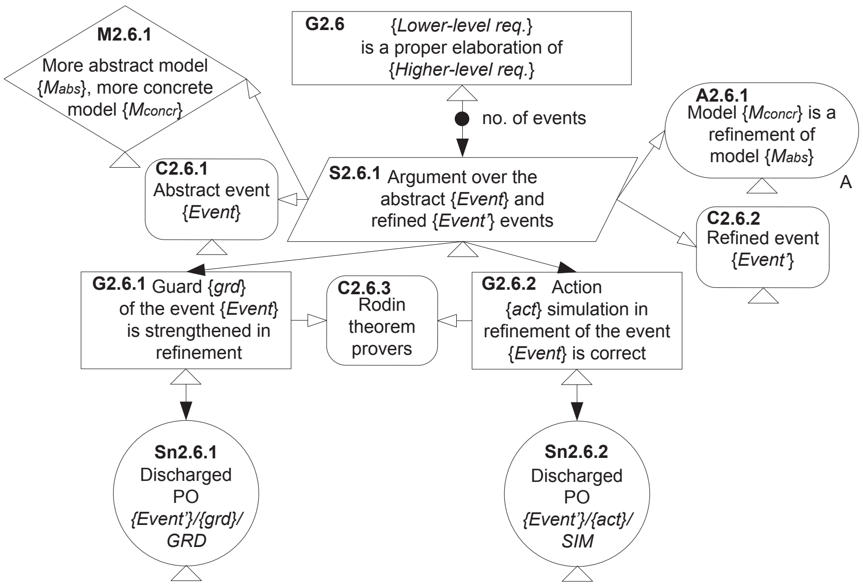

M2.6.1
More abstract model {Mabs}, more concrete model {Mconcr}
G2.6 {Lower-level req.} is a proper elaboration of {Higher-level req.}
no. of events
A2.6.1
Model {Mconcr} is a refinement of model {Mabs}
A
C2.6.1
Abstract event {Event}
S2.6.1 Argument over the abstract {Event} and refined {Event'} events
C2.6.2
Refined event {Event'}
G2.6.1 Guard {grd} of the event {Event} is strengthened in refinement
C2.6.3
Rodin theorem provers
G2.6.2 Action {act} simulation in refinement of the event {Event} is correct
Sn2.6.1
Discharged PO {Event'}/{grd}/ GRD
Sn2.6.2
Discharged PO {Event'}/{act}/ SIM

Figure 18: Argument pattern for safety requirements of Class 6

of affected events $\{Event_{h_1}, ..., Event_{h_K}\}$. Similarly, *Lower-level req.* is associated with its own set of affected events $\{Event'_{l_1}, ..., Event'_{l_L}\}$.

For each pair of events $Event$ and $Event'$ from the obtained sets, the following two generated proof obligations (GRD) and (SIM) are needed to be proved to establish correctness of model refinement (Section 2.1):

$$\begin{aligned}
&H(d,c,v,w),\ g'_{Event'}(d,c,w)\ \vdash\ g_{Event}(d,c,v),\\
&H(d,c,v,w),\ g'_{Event'}(d,c,w),\ BA'_{Event'}(d,c,w,w')\ \vdash\\
&\qquad\qquad\qquad \exists v'.BA_{Event}(d,c,v,v')\ \land\ I'(d,c,v',w').
\end{aligned}$$

The established proofs of the types GRD and SIM serve as solutions in our pattern, **Sn2.6.1** and **Sn2.6.2** in Figure 18 respectively.

The instantiation of the pattern proceeds as shown below:

- requirements *Higher-level req.* and *Lower-level req.* should be replaced with specific requirements;
- a more abstract formal model M_{abs} and a more concrete formal model M_{concr} should be referred to in a GSN model element;
- the pairs of the associated events of the respective abstract and concrete system models should be referred to in GSN context elements;
- the proof obligations of the types GRD and SIM discharged by the Rodin platform should be included in the safety case as solutions.

Moreover, there can be several hierarchical levels of requirements specification. To cope with this case, we propose to instantiate patterns for each such a level separately.

To illustrate the construction of a safety case fragment for this class of requirements, we refer to the sluice gate control system [46] described in Sections 4.2 and 4.6. Some safety requirements of this system are hierarchically structured. Thus, there is a more generic safety requirement **SR-cl6-higher-level**:

SR-cl6-higher-level: *The system shall be able to handle a critical failure by either initiating a shutdown or a recovery procedure*

stipulating that some actions should take place in order to tolerate any critical failure. However, it does not define the precise procedures associated with this failure handling. In contrast, there is a more detailed counterpart **SR-cl6-lower-level** of the requirement **SR-cl6-higher-level** (it was presented in the previous section as the requirement **SR-cl5**). It regulates precisely that an alarm should be raised and the system should stop its operation (the system should terminate):

SR-cl6-lower-level: *When a critical failure is detected, an alarm shall be raised and the system shall be stopped.*

These safety requirements are shown to hold in different models of the system development. The requirement **SR-cl6-higher-level** is formalised as two invariants at the most abstract level of the formal specification in Event-B, the MACHINE **m0**, while the requirement **SR-cl6-lower-level** is formalised as a theorem in the MACHINE **m1**. Note that the MACHINE **m1** is the refinement of the MACHINE **m0**.

The instance of the mapping function F_M for the requirement **SR-cl6-higher-level** is as follows:

$$\textbf{SR-cl6-higher-level} \mapsto \{\textbf{inv\_1\_cl6}, \textbf{inv\_2\_cl6}\},$$

where:

$$\textbf{inv\_1\_cl6:}\ \ \mathit{Failure} = \mathit{FALSE} \ \Rightarrow\ \mathit{Stop} = \mathit{FALSE},$$
$$\textbf{inv\_2\_cl6:}\ \ \mathit{Failure} = \mathit{TRUE} \ \wedge\ \mathit{flag} \neq \mathit{CONT} \ \Rightarrow\ \mathit{Stop} = \mathit{TRUE}.$$

The handling of critical failures is non-deterministically modelled in the event *ErrorHandling* of the abstract model (Figure 19). The local variable *res* is of the type *BOOL* and can be either *TRUE* or *FALSE*. It means that, if a successful error handling procedure that does not lead to the system termination has been performed, both variables standing for a critical failure (*Failure*) and for the system shutdown (*Stop*) are assigned the values *FALSE* and the system continues its operation. Otherwise, they are assigned the values *TRUE* leading to the system termination.

The fragment of a safety case for the safety requirement **SR-cl6-higher-level** can be constructed preserving the instructions determined in Section 4.2, while the fragment of a safety case for the requirement **SR-cl6-lower-level** can be found in Section 4.6.

Now let us focus on ensuring the hierarchical link between these requirements by instantiating the argument pattern for *Class 6*. Following the proposed approach, we define a set of the affected model events for the higher-level safety requirement: {*Environment*, *Detection*, *ErrorHandling*, *Prediction*, *NormalOperation*}, and for the lower-level safety requirement: {*Environment*, *Detection_NoFault*, *Detection_Fault*, *SafeStop*, *Prediction*, *NormalSkip*}. For

```
// Event in the MACHINE m0
 event ErrorHandling
  any res
  where
   @grd1 flag = CONT
   @grd2 Failure = TRUE
   @grd3 Stop = FALSE
   @grd4 res ∈ BOOL
  then
   @act1 flag ≔ PRED
   @act2 Stop ≔ res
   @act3 Failure ≔ res
 end
```

```
// Event in the refined MACHINE m1
 event SafeStop
        refines ErrorHandling
  where
   @grd1 flag = CONT
   @grd2 door1_fail = TRUE ∨
            door2_fail = TRUE ∨
            pressure_fail = TRUE
   @grd3 Stop = FALSE
  with
   @res res = TRUE
  then
   @act1 flag ≔ PRED
   @act2 Stop ≔  TRUE
 end
```

Figure 19: Events *ErrorHandling* and *SafeStop*

simplicity, here we consider only one pair of events *ErrorHandling* and *SafeStop* shown in Figure 19.

In the Event-B development of the sluice gate system, the non-determinism modelled by the local variable *res* is eliminated via introduction of a specific situation leading to the system shutdown. All other fault tolerance procedures are left out of the scope of the presented development.

Additionally to the introduction of the deterministic procedures for error handling, the variable *Failure* is data refined in the first refinement **m1**. Now, the system failure may occur either if the component *door1* fails (*door1_fail* = *TRUE*), or *door2* fails (*door2_fail* = *TRUE*), or the pressure pump fails (*pressure_fail* = *TRUE*). This relationship between the old abstract variable and new concrete ones is defined by the corresponding gluing invariant.

The corresponding instance of the argument pattern is presented in Figure 20. To ensure that the requirement **SR-cl6-lower-level** is a proper elaboration of the requirement **SR-cl6-higher-level** (the goal **G2.6** in Figure 20), we argue over the abstract event *ErrorHandling* and the refined event *SafeStop*. We show that the guard **grd2** is strengthened in the refinement (the discharged proof obligation (GRD)) and the action **act2** is not contradictory to the abstract version (SIM). The corresponding proof obligations are shown in Figure 21.

4.8 Argument pattern for SRs about temporal properties (Class 7)

So far, we have considered the argument patterns of safety requirements classes where the evidence that the top goal of the pattern holds is constructed based on the proof obligations generated by the Rodin platform. Not all types of safety requirements can be formally demonstrated in this way, however. In particular, the Event-B framework lacks direct support of temporal system properties such as reachability, liveness, existence, etc. Nevertheless, the Rodin platform has an accompanying plug-in, called ProB [47], which allows for model checking of temporal properties.

Therefore, in this section, we propose an argument pattern for the class of safety require-

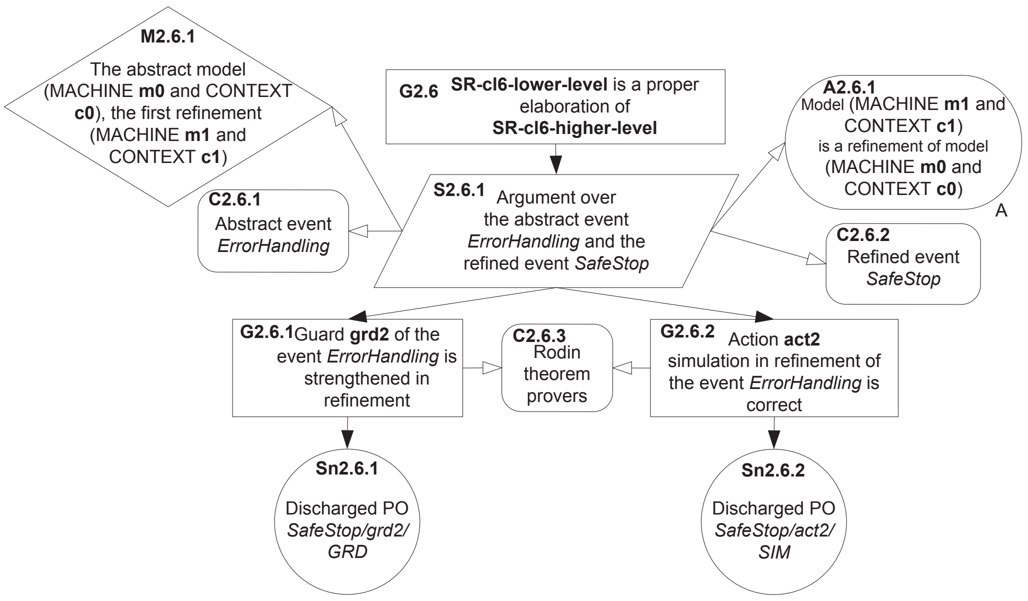

M2.6.1
The abstract model (MACHINE **m0** and CONTEXT **c0**), the first refinement (MACHINE **m1** and CONTEXT **c1**)
G2.6 **SR-cl6-lower-level** is a proper elaboration of **SR-cl6-higher-level**
A2.6.1
Model (MACHINE **m1** and CONTEXT **c1**) is a refinement of model (MACHINE **m0** and CONTEXT **c0**)
A
C2.6.1
Abstract event *ErrorHandling*
S2.6.1 Argument over the abstract event *ErrorHandling* and the refined event *SafeStop*
C2.6.2
Refined event *SafeStop*
G2.6.1 Guard **grd2** of the event *ErrorHandling* is strengthened in refinement
C2.6.3
Rodin theorem provers
G2.6.2 Action **act2** simulation in refinement of the event *ErrorHandling* is correct
Sn2.6.1
Discharged PO *SafeStop/grd2/GRD*
Sn2.6.2
Discharged PO *SafeStop/act2/SIM*

Figure 20: The pattern instantiation example

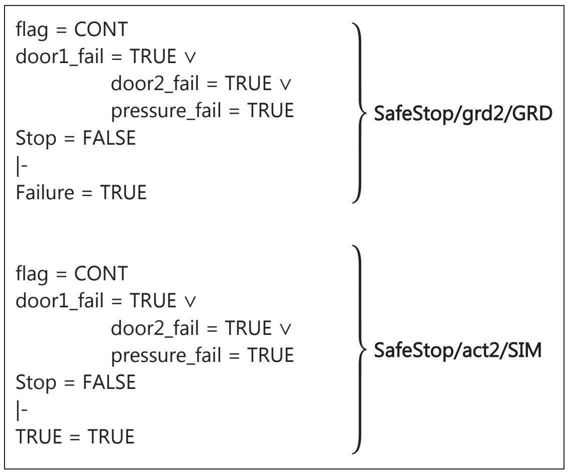

```
flag = CONT                                  ⎫
door1_fail = TRUE ∨                          ⎪
            door2_fail = TRUE ∨              ⎪
            pressure_fail = TRUE             ⎬ SafeStop/grd2/GRD
Stop = FALSE                                 ⎪
|-                                           ⎪
Failure = TRUE                               ⎭

flag = CONT                                  ⎫
door1_fail = TRUE ∨                          ⎪
            door2_fail = TRUE ∨              ⎪
            pressure_fail = TRUE             ⎬ SafeStop/act2/SIM
Stop = FALSE                                 ⎪
|-                                           ⎪
TRUE = TRUE                                  ⎭
```

Figure 21: The proof obligations of the types GRD and SIM

ments that can be expressed as temporal properties. The pattern is graphically shown in Figure 22. Here $property_i$ stands for some temporal property to be verified, for $i \in 1 .. N$, where N is the number of temporal properties of the system.

The property to be verified should be formulated as an LTL formula in the *LTL Model Checking* wizard of the ProB plug-in for some particular model *M*. This suggests the mapping function F_M for *Class 7* to be of the form

$$Requirement \mapsto \{LTL\,formula\}.$$

Each such a temporal property should be well-defined according to restrictions imposed on LTL in ProB. The tool can generate three possible results: (1) the given LTL formula is true for all valid paths (no counter-example has been found, all nodes have been visited); (2) there is a path that does not satisfy the formula (a counter-example has been found and it is shown in a separate view); (3) no counter-example has been found, but the temporal property in question cannot be guaranteed because the state space was not fully explored.

To instantiate this pattern, one needs to proceed as follows:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal model *M* should be referred to in a GSN model element;
- the concrete mapping between the requirement and the corresponding LTL formula should be supplied, while each temporal property $property_i$ from this mapping should be referred to in a GSN context element;
- model checking results on an instantiated property that have been generated by ProB should be included as the evidence that this property is satisfied.

There are several alternative ways to reason over temporal properties in Event-B [5, 29, 34, 48]. The most recent of them is that of Hoang and Abrial [34]. The authors propose a set of proof rules for reasoning about such temporal properties as liveness properties (existence, progress and persistence). The main drawback of this approach is that, even though it does not require extensions of the proving support of the Rodin platform, it necessitates extension of the Event-B language by special clauses (annotations) corresponding to different types of temporal properties. Alternatively, in the cases when a temporal property can be expressed as a condition on the system control flow, the usecase/flow approach [37] described in Section 4.4 can be used.

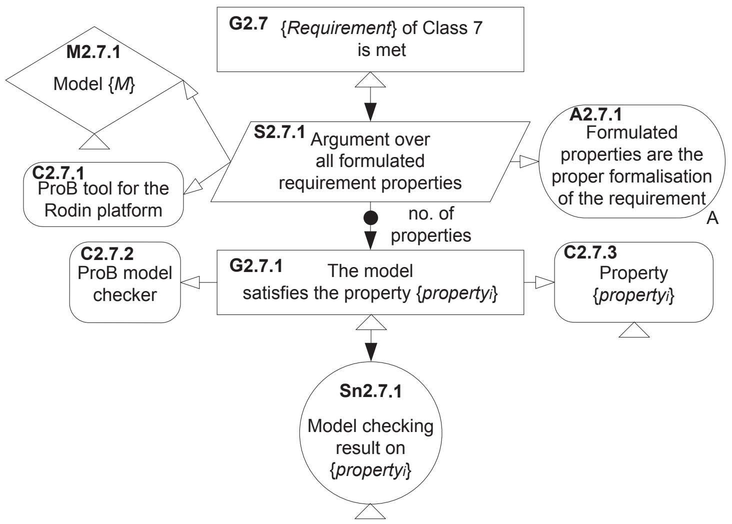

G2.7 {Requirement} of Class 7 is met
M2.7.1 Model {M}
S2.7.1 Argument over all formulated requirement properties
A2.7.1 Formulated properties are the proper formalisation of the requirement
A
C2.7.1 ProB tool for the Rodin platform
no. of properties
C2.7.2 ProB model checker
G2.7.1 The model satisfies the property {propertyi}
C2.7.3 Property {propertyi}
Sn2.7.1 Model checking result on {propertyi}

Figure 22: Argument pattern for safety requirements of Class 7

To exemplify the instantiation of the argument pattern for safety requirements of *Class 7*, we consider a distributed monitoring system – Temperature Monitoring System (TMS). The full system formal specification in Event-B is presented in [56]. In brief, the TMS consists of three data processing units (DPUs) connected to operator displays in the control room. At each cycle, the system performs readings of the temperature sensors, distributes preprocessed data among DPUs where they are analysed (processed), and finally displays the output to the operator. The system model is developed in such a way that it allows for ensuring integrity of the temperature data as well as its freshness.

A safety requirement about a temporal property of this system, which we consider here, is as follows:

SR-cl7: *Each cycle the system shall display fresh and correct data.*

We leave out of the scope of this paper the mechanism of ensuring data freshness, correctness and integrity, while focusing on the fact of displaying data at each cycle. In the given Event-B specification, a new cycle starts when the event *Environment* is executed. To verify that the system will eventually display the data to the operator (i.e., the corresponding event *Displaying* will be enabled), we formulate an LTL formula for the abstract model of the system (**temp_pr_ex**). Then, the instance of the mapping function F_M is defined as

$$\textbf{SR-cl7} \mapsto \{\textbf{temp\_pr\_ex}\},$$

where

$$\textbf{temp\_pr\_ex:}\ \Box\ (\mathit{after}(\mathit{Environment})\ \rightarrow\ \Diamond\ \mathit{before}(\mathit{Displaying})).$$

Here $\Box$ is an operator *"always"* and $\Diamond$ stands for *"eventually"*.

The formula **temp_pr_ex** has the following representation in ProB:

$$\begin{array}{l}
G\ (\{\forall\ main\_phase',\ temp\_sensor',\ curr\_time' \cdot \\
\quad main\_phase' \in \mathit{MAIN\_PHASES}\ \wedge\ temp\_sensor' \in \mathbb{N}\ \wedge\ curr\_time' \in \mathbb{N} \wedge \\
\quad (\exists\ main\_phase,\ sync\_t\ \cdot\ main\_phase \in \mathit{MAIN\_PHASES}\ \wedge\ sync\_t \in \mathbb{N} \wedge \\
\quad main\_phase' = PROC\ \wedge\ temp\_sensor' \in \mathbb{N}\ \wedge\ curr\_time' = sync\_t)\} \\
\quad \Rightarrow \\
\quad F\ \{\exists\ ss, \mathit{TEMP\_SET} \cdot main\_phase = \mathit{DISP} \wedge packet\_sent\_flag = \mathit{TRUE} \wedge \\
\quad \mathit{TEMP\_SET}\ \subseteq\ \mathbb{N}\ \wedge\ time\_progressed = \mathit{TRUE} \wedge \\
\quad ss = \{x\ \mapsto y \mid \exists\ i\ \cdot\ i \in dom(timestamp)\ \wedge\ x = timestamp(i) \wedge \\
\quad y = temperature(i)\}[curr\_time - Fresh\_Delta\ \mathinner{\ldotp\ldotp}\ curr\_time] \wedge \\
\quad (ss \neq \varnothing \Rightarrow \mathit{TEMP\_SET} = ss) \wedge (ss = \varnothing \Rightarrow \mathit{TEMP\_SET} = \{\mathit{ERR\_VAL}\})\}),
\end{array}$$

where G stands for the temporal operator *"globally"* and F is the temporal operator *"finally"*. These operators correspond to the standard LTL constructs *"always"* and *"eventually"* respectively. For the detailed explanation of the used variables, constants and language constructs, see [56].

In this case, the result of the model checking of this property in ProB is *"no counter-example has been found, all nodes have been visited"*. Figure 23 illustrates the corresponding instance of the argument pattern.

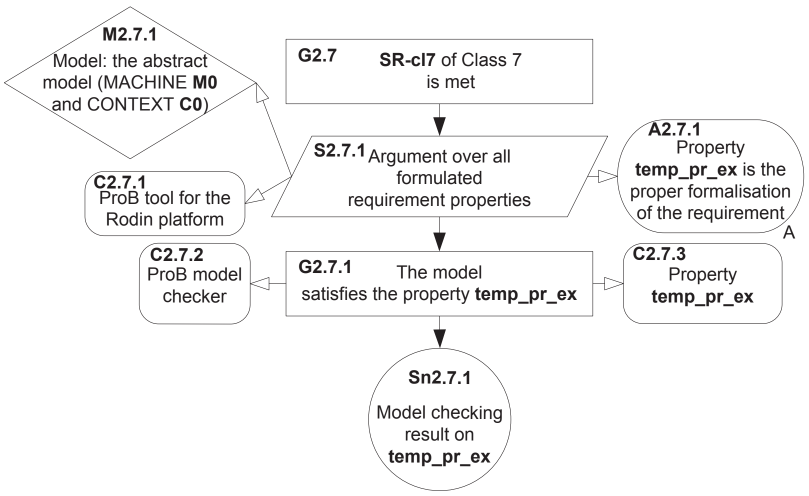

M2.7.1
Model: the abstract model (MACHINE **M0** and CONTEXT **C0**)
G2.7
SR-cl7 of Class 7 is met
S2.7.1 Argument over all formulated requirement properties
A2.7.1
Property **temp_pr_ex** is the proper formalisation of the requirement
A
C2.7.1
ProB tool for the Rodin platform
C2.7.2
ProB model checker
G2.7.1
The model satisfies the property **temp_pr_ex**
C2.7.3
Property **temp_pr_ex**
Sn2.7.1
Model checking result on **temp_pr_ex**

Figure 23: The pattern instantiation example

4.9 Argument pattern for SRs about timing properties (Class 8)

Another class of safety requirements that requires to be treated in a different way is *Class 8* containing timing properties of the considered system. As we have already mentioned, the representation of timing properties in Event-B has not been explicitly defined yet. Nonetheless, the majority of safety-critical systems rely on timing constraints for critical functions. Obviously, the preservation of such requirements must be verified. To address this, we propose to bridge Event-B modelling with model checking of timing properties in Uppaal.

Figure 24 shows our argument pattern for the safety requirements about timing properties. In our pattern, $property_j$ stands for some timing property to be verified, for $j \in 1 \;..\; N$, where N is the number of timing properties.

Following the approach proposed by Iliasov et al. [39], we rely on the Uppaal toolset for obtaining model checking results that further can be used as the evidence in a safety case. The timing property in question can be formulated using the TCTL language. A timed automata model (an input model of Uppaal) is obtained from a process-based view extracted from an Event-B model as well as additionally introduced clocks and timing constraints. The generic mapping function F_M for this class is then of the form $Requirement \mapsto \{TCTL\,formula\}$.

Uppaal uses a subset of TCTL to specify properties to be checked [11]. The results of the property verification can be of three types: (1) a trace is found, i.e., a property is satisfied (user can then import the trace into the simulator); (2) a property is violated; (3) the verification is inconclusive with the approximation used.

We propose the following steps in order to instantiate this pattern:

- a requirement *Requirement* should be replaced with a particular safety requirement;
- a formal development that consists of a chain of refinements in Event-B and the corresponding Uppaal model should be referred to in GSN model elements;
- the concrete mapping between the requirement and the corresponding TCTL formula

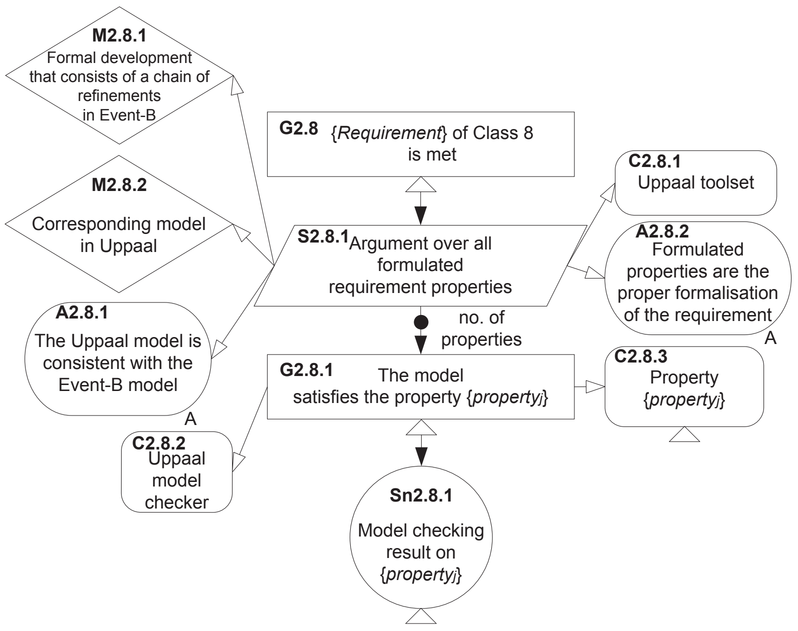

M2.8.1
Formal development that consists of a chain of refinements in Event-B
G2.8 {Requirement} of Class 8 is met
C2.8.1 Uppaal toolset
M2.8.2
Corresponding model in Uppaal
S2.8.1 Argument over all formulated requirement properties
A2.8.2 Formulated properties are the proper formalisation of the requirement
A
A2.8.1 The Uppaal model is consistent with the Event-B model
A
no. of properties
G2.8.1 The model satisfies the property {property_j}
C2.8.3 Property {property_j}
C2.8.2 Uppaal model checker
Sn2.8.1 Model checking result on {property_j}

Figure 24: Argument pattern for safety requirements of Class 8

should be provided, while each timing property *property*$_j$ from this mapping should be referred to in a GSN context element;

- model checking results on an instantiated property that have been generated by Uppaal should be included as the evidence that this property is satisfied.

We adopt a case study considered in [38, 39] in order to show the instantiation of the proposed argument pattern for a safety requirement of *Class 8*. The case study is the Data Processing Unit (DPU) – a module of Mercury Planetary Orbiter of the BepiColombo Mission. The DPU consists of the core software and software of two scientific instruments. The core software communicates with the BepiColombo spacecraft via interfaces, which are used to receive telecommands (TCs) from the spacecraft and transmit science and housekeeping telemetry data (TMs) back to it. In this paper, we omit showing the detailed specification of the DPU in Event-B as well as we do not explain how the corresponding Uppaal model was obtained. We rather illustrate how the verified liveness and time-bounded reachability properties of the system can be reflected in the resulting safety case (Figure 25).

The DPU is required to eventually return a TM for any received TC and must respond within a predefined time bound:

SR-cl8: *The DPU shall eventually return a TM for any received TC and shall respond no later than the maximal response time.*

We consider two timing properties associated with this requirement, i.e.,

time_pr_ex1: $(new\_tc == id) \;\rightarrow\; (last\_tm == id)$,
time_pr_ex2: $A[](last\_tm == id$ && *Obs1.stop*$)$ *imply* $($*Obs1_c* $< upper\_bound)$,

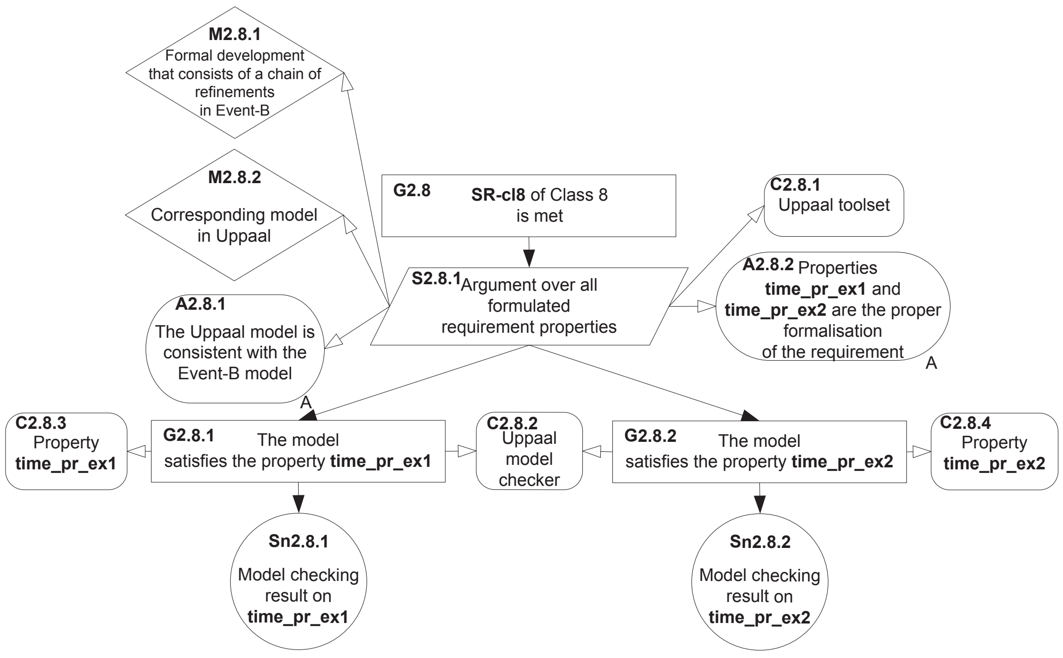

M2.8.1
Formal development that consists of a chain of refinements in Event-B
M2.8.2
Corresponding model in Uppaal
G2.8
SR-cl8 of Class 8 is met
C2.8.1
Uppaal toolset
S2.8.1
Argument over all formulated requirement properties
A2.8.2
Properties time_pr_ex1 and time_pr_ex2 are the proper formalisation of the requirement
A
A2.8.1
The Uppaal model is consistent with the Event-B model
A
C2.8.3
Property time_pr_ex1
G2.8.1
The model satisfies the property time_pr_ex1
C2.8.2
Uppaal model checker
G2.8.2
The model satisfies the property time_pr_ex2
C2.8.4
Property time_pr_ex2
Sn2.8.1
Model checking result on time_pr_ex1
Sn2.8.2
Model checking result on time_pr_ex2

Figure 25: The pattern instantiation example

such that the concrete instance of the generic mapping function F_M is as follows:

$$\textbf{SR-cl8} \mapsto \{\textbf{time\_pr\_ex1}, \textbf{time\_pr\_ex2}\}.$$

The symbol $\rightarrow$ stands for the TCTL *"leads-to"* operator, and id is some TC identification number. $A[]$ stands for *"Always, for any execution path"* and *Obs1* is a special observer process that starts the clock *Obs1_c*, whenever a TC command with id is received, and stops it, once the corresponding TM is returned. The variable *upper_bound* corresponds to the maximal response time. The corresponding instance of the argument pattern is given in Figure 25.

4.10 Summary of the proposed argument patterns

To facilitate the construction of safety cases, we have defined a set of argument patterns graphically represented using GSN. The argumentation and goal decomposition in these patterns were influenced by the formal reasoning in Event-B.

However, since the development utilising formal methods typically require additional reasoning about model correctness and well-definedness, we firstly proposed an argument pattern for assuring well-definedness of the system development in Event-B. Secondly, we proposed a number of argument patterns for assuring safety requirements of a system. We associated these argument patterns with the classification of safety requirements presented in Section 3.2. Therefore, we distinguished eight classification-based argument patterns. Despite the fact that the proposed classification of safety requirements covers a wide range of different safety requirements, the classification and subsequently the set of argument patterns can be further extended if needed.

Unfortunately, at the meantime not all the introduced classes of safety requirements can be formally demonstrated utilising Event-B solely. Therefore, among the proposed argument patterns there are several patterns where the evidence was constructed using accompanying toolsets – the Usecase/Flow and ProB plug-ins for the Rodin platform, as well as the external model checker for verification of real-time systems Uppaal.

In this section, we exemplified the instantiation of the proposed argument patterns for assuring safety requirements on several case studies. Among them are the sluice gate control system [46], the attitude and orbit control system [53], the system for controlling cars on a bridge [4], the temperature monitoring system [56], and the data processing unit of Mercury planetary orbiter of the BepiColombo mission [38, 39].

The instantiation of the proposed argument patterns is a trivial task. Nonetheless, the application of the overall approach requires basic knowledge of principles of safety case construction as well as a certain level of expertise in formal modelling. Therefore, experience in formal modelling and verification using state-based formalisms would be beneficial for safety and software engineers.

Currently, the proposed approach is restricted by the lack of the tool support. Indeed, manual construction of safety cases especially of large-scale safety-critical systems may be error-prone. We believe that the well-defined steps of evidence construction and the detailed guidelines on the pattern instantiation given in this paper will contribute to the development of the corresponding plug-in for the Rodin platform.

5 Case study – steam boiler

In this section, we demonstrate our proposed methodology (based on argument patterns) for building safety cases on a bigger case study. The considered case study is a steam boiler control system. It is a well-known safety-critical system widely used in industrial applications. Due to the large number of safety requirements of different types imposed on it, this system is highly suitable for demonstration of our methodology.

5.1 System description

The steam boiler (Figure 26) is a safety-critical control system that produces steam and adjusts the quantity of water in the steam boiler chamber to maintain it within the predefined safety boundaries [1]. The situations when the water level is too low or too high might result in loss of personnel life, significant equipment damage (the steam boiler itself or the turbine placed in front of it), or damage to the environment.

The system consists of the following units: a chamber, a pump, a valve, a sensor to measure the quantity of water in the chamber, a sensor to measure the quantity of steam

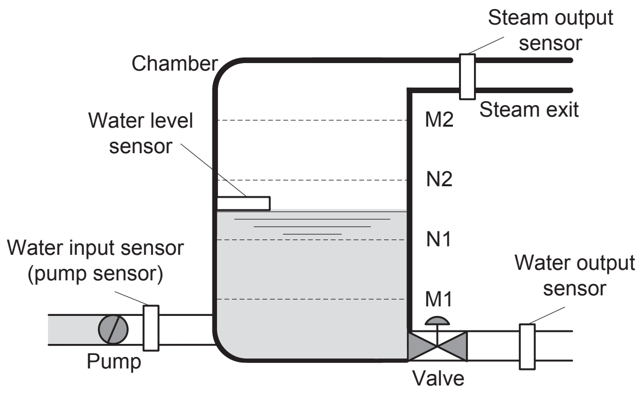


Figure 26: Steam boiler

Table 2: Parameters of the steam boiler

Label	Description	Unit
C	the total capacity of the steam boiler chamber	litre
P	the maximal capacity of the pump	litre/sec
W	the maximal quantity of steam produced	litre/sec
M1	the minimal quantity of water, i.e., the lower safety boundary	litre
M2	the maximal quantity of water, i.e., the upper safety boundary	litre
N1	the minimal normal quantity of water to be maintained during regular operation	litre
N2	the maximal normal quantity of water to be maintained during regular operation	litre

which comes out of the steam boiler chamber, a sensor to measure water input through the pump, and a sensor to measure water output through the valve. The essential system parameters are given in Table 2.

The considered system has several modes. After being powered on, the system enters the **Initialisation** mode. At each control cycle, the system reads sensors and performs failure detection. Then, depending on the detection result, the system may enter either one of its operational modes or the non-operational mode. There are three operational modes in the system: **Normal**, **Degraded**, **Rescue**. In the **Normal mode**, the system attempts to maintain the water level in the chamber between the normal boundaries N1 and N2 (such that N1 $<$ N2) providing that no failures of the system units have occurred. In the **Degraded mode**, the system tries to maintain the water level within the normal boundaries despite failures of some physical non-critical units. In the **Rescue mode**, the system attempts to maintain the normal water level in the presence of a failure of the critical unit – the water level sensor. If failures of the system units and the water level sensor occur simultaneously or the water level is outside of the predefined safety boundaries M1 and M2 (such that M1 $<$ M2), the system enters the non-operational mode **Emergency_Stop**.

In our development, we consider the following failures of the system and its units. The failure of the steam boiler control system is detected if either the water level in the chamber is outside of the safety boundaries (i.e., if it is lower than M1 or higher than M2) or the combination of a water level sensor failure and a failure of any other system unit (the pump or the steam output sensor) is detected. The water level sensor is considered as failed if it returns a value which is outside of the nominal sensor range or the estimated range predicted in the last cycle. Analogously, a steam output sensor failure is detected. The pump fails if it does not change its state when required.

A water level sensor failure by itself does not lead to a system failure. The steam boiler contains the information redundancy, i.e., the controller is able to estimate the water level in the chamber based on the amount of water produced by the pump and the amount of the released steam. Similarly, the controller is able to maintain the acceptable level of efficiency based on the water level sensor readings if either the pump or the steam output sensor fail. The detailed description of the system, its functional and safety requirements as well as the models of our formal development in Event-B can be found in [55].

5.2 Brief overview of the development

Our Event-B development of the steam boiler case study consists of an abstract specification and its four refinements [55]. The abstract model (MACHINE **M0**) implements a basic control loop. The first refinement (MACHINE **M1**) introduces an abstract representation of the activities performed after the system is powered on and during system operation (under both nominal and failure conditions). The second refinement (MACHINE **M2**) introduces a detailed representation of the conditions leading to a system failure. The third refinement (MACHINE **M3**) models the physical environment of the system as well as elaborates on more advanced failure detection procedures. Finally, the fourth refinement (MACHINE **M4**) introduces a representation of the required execution modes. Each MACHINE has the associated CONTEXT where the necessary data structures are introduced and their properties are postulated as axioms.

Let us now give a short overview of the basic model elements (i.e., constants, variables and events). The parameters of the steam boiler system presented in Table 2 are defined as constants in one of the CONTEXT components. Moreover, several abstract functions are defined there to formalise, for example, the critical water level (*WL_critical*).

The dynamic behaviour of the system is modelled in the corresponding MACHINE components. Some essential variables and events are listed below:

- The variables modelling the steam boiler actuators – the pump and the valve:
  - *pump_ctrl*: the value of this variable equals to *ON* if the pump is switched on, and *OFF* otherwise;
  - *valve_ctrl*: the value of this variable equals to *OPEN* if the valve is open, and *CLOSED* otherwise.
- The variables representing the amount of water passing through the pump and the valve:
  - *pump* stands for the amount of water incoming into the chamber through the pump;

  - *water_output* models the amount of water coming out of the chamber through the valve.

- The variables representing the water level in the chamber:
  - *water_level* models the latest water level sensor readings;
  - *min_water_level* and *max_water_level* represent the estimated interval for the *sensed* water level.
- The variables representing the amount of the steam coming out of the chamber:
  - *steam_output* models the latest steam output sensor readings;
  - *min_steam_output* and *max_steam_output* represent the estimated interval for the *sensed* amount of steam.
- The variables representing failures of the system and its components:
  - *failure* is an abstract boolean variable modelling occurrence of a system failure;
  - *wl_sensor_failure* represents a failure of the water level sensor;
  - *pump_failure* models a failure of the pump actuator;
  - *so_sensor_failure* represents a failure of the steam output sensor.
- The variables modelling phases of the control cycle and the system modes:
  - *phase*: the value of this variable can be equal either to *ENV*, *DET*, *CONT*, *PRED* corresponding to the current controller stage (i.e., reading environment, detecting system failures, performing routing control, or predicting the system state in the next cycle);
  - *preop_flag* is a flag which indicates whether the system is in the pre-operational stage or not;
  - *mode* models the current mode of the system, i.e., *Initialisation*, *Normal*, *Degraded*, *Rescue*, or *Emergency_Stop*.
- The variable *stop* abstractly models system shutdown and raising an alarm.
- Essential events of the modelled system:
  - *Environment*, modelling the behaviour of the environment;
  - *Detection*, representing detection of errors;
  - *PreOperational1* and *PreOperational2*, modelling the initial system procedures to establish the amount of water in the chamber within the safety boundaries;
  - *Operational*, performing controller actions under the nominal conditions;
  - *EmergencyStop*, modelling error handling;
  - *Prediction*, computing the next estimated states of the system.

In the refinement process, such events as *Detection* and *Operational* are split into a number of more concrete events modelling detection of failures of different system components as well as different system operational modes.

5.3 Application of the proposed approach

In this section, we follow our proposed approach to constructing a safety case of a system from its formal model in Event-B. More specifically, firstly we show that our formal development of the steam boiler control system is well-defined by instantiating the corresponding argument pattern (introduced in Section 4.1). Secondly, we apply the classification-based argument patterns (presented in Sections 4.2 – 4.9) to construct the corresponding fragments of the safety case related to specific safety requirements of the considered system.

The steam boiler control system is a complex system, which has a rich functionality and adheres to a large number of safety requirements. The accomplished formal development of this system as well as its safety case are also complex and large in size. Therefore, we will not show the system in its entirety but rather demonstrate application of our methodology on selected system fragments.

5.3.1 Instantiation of the argument pattern for well-definedness of the development

Due to a significant size of the system safety case, here we show only a part of the instantiated pattern for demonstrating well-definedness of a formal development (Section 4.1). Figure 27 presents the resulting fragment of the safety case concerning the first refinement model (MACHINE **M1** and the associated CONTEXT **C1**).

Let us remind that, to apply the well-definedness argument pattern, we have to formally demonstrate axiom consistency in the CONTEXT **C1**. To argue over axiom consistency, we define two groups of axioms. The first group includes axioms defining generic parameters of the system, e.g., the constants associated with the criticality of the water level, which is based on the pre-defined safety boundaries. The second group consists of the axioms defining the abstract function *Stable* needed to model the failure stability property. Here stability means that, once a failure occurred, the value of the variable representing this failure remains unchanged until the whole system is restarted. These groups are independent because they refer to distinct Event-B constants and sets. The corresponding theorems **thm_axm1** and **thm_axm2** are shown below. The first theorem verifies that the parameters of the steam boiler are introduced in the model correctly:

$$\begin{aligned}
\textbf{thm\_axm1:}\ & \exists\, \mathit{N1, N2, M1, M2, C, WL\_critical} \cdot \mathit{N1} \in \mathbb{N}1 \ \wedge\ \mathit{N2} \in \mathbb{N}1 \wedge \\
& \mathit{M1} \in \mathbb{N}1 \wedge \mathit{M2} \in \mathbb{N}1 \wedge \mathit{C} \in \mathbb{N}1 \wedge \mathit{WL\_critical} \in \mathbb{N} \times \mathbb{N} \rightarrow BOOL \wedge \\
& 0 < \mathit{M1} \ \wedge\ \mathit{M1} < \mathit{N1} \ \wedge\ \mathit{N1} < \mathit{N2} \ \wedge\ \mathit{N2} < \mathit{M2} \ \wedge\ \mathit{M2} < C \wedge \\
& (\forall\, x, y \cdot x \in \mathbb{N} \ \wedge\ y \in \mathbb{N} \wedge \\
& ((x < \mathit{M1} \vee y > \mathit{M2}) \Leftrightarrow WL\_critical(x \mapsto y) = \mathit{TRUE})) \wedge \\
& (\forall\, x, y \cdot x \in \mathbb{N} \ \wedge\ y \in \mathbb{N} \wedge \\
& ((x \geq \mathit{M1} \ \wedge\ y \leq \mathit{M2}) \Leftrightarrow WL\_critical(x \mapsto y) = \mathit{FALSE})).
\end{aligned}$$

The second theorem verifies that the group of axioms introduced to define a function about the failure stability is consistent:

$$\begin{aligned}
\textbf{thm\_axm2:}\ & \exists\, Stable \ \cdot\ Stable \in BOOL \times BOOL \rightarrow BOOL \wedge \\
& (\forall\, x, y \cdot x \in BOOL \ \wedge\ y \in BOOL \ \Rightarrow \\
& (Stable(x \mapsto y) = \mathit{TRUE} \ \Leftrightarrow\ (x = \mathit{TRUE} \ \Rightarrow\ y = \mathit{TRUE}))).
\end{aligned}$$

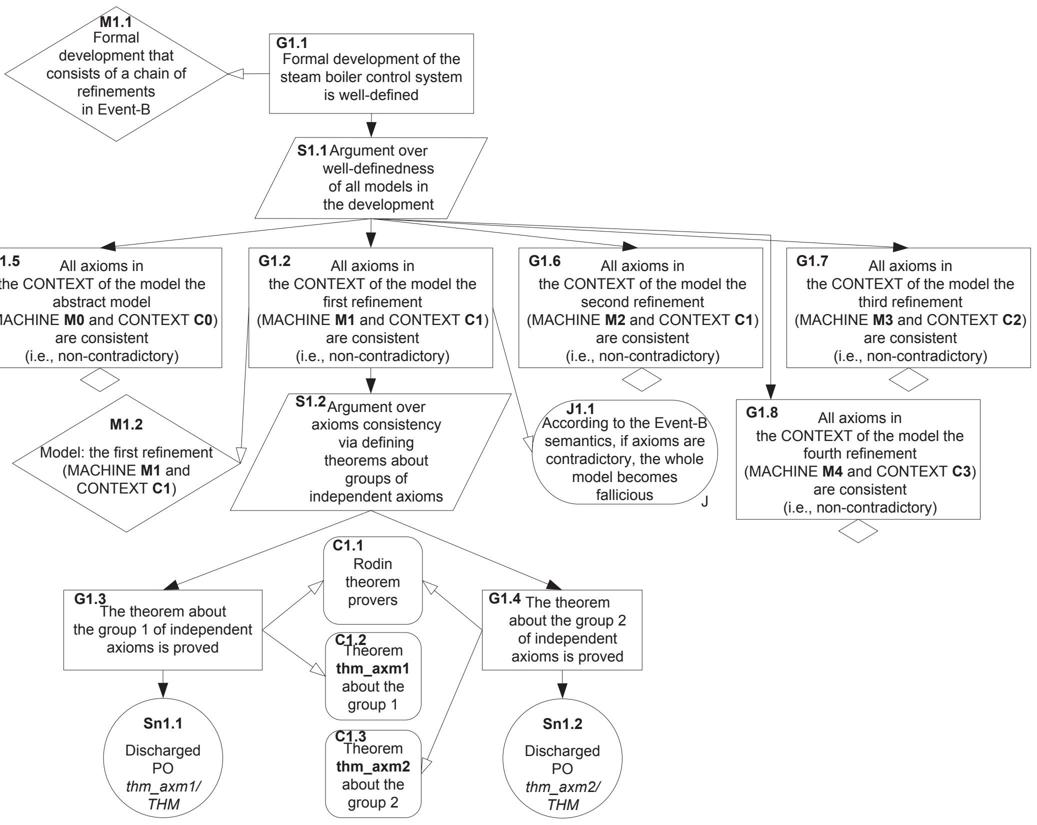

M1.1
Formal development that consists of a chain of refinements in Event-B
G1.1
Formal development of the steam boiler control system is well-defined
S1.1 Argument over well-definedness of all models in the development
G1.5
All axioms in the CONTEXT of the model the abstract model (MACHINE M0 and CONTEXT C0) are consistent (i.e., non-contradictory)
G1.2
All axioms in the CONTEXT of the model the first refinement (MACHINE M1 and CONTEXT C1) are consistent (i.e., non-contradictory)
G1.6
All axioms in the CONTEXT of the model the second refinement (MACHINE M2 and CONTEXT C1) are consistent (i.e., non-contradictory)
G1.7
All axioms in the CONTEXT of the model the third refinement (MACHINE M3 and CONTEXT C2) are consistent (i.e., non-contradictory)
M1.2
Model: the first refinement (MACHINE M1 and CONTEXT C1)
S1.2 Argument over axioms consistency via defining theorems about groups of independent axioms
J1.1
According to the Event-B semantics, if axioms are contradictory, the whole model becomes fallicious
J
G1.8
All axioms in the CONTEXT of the model the fourth refinement (MACHINE M4 and CONTEXT C3) are consistent (i.e., non-contradictory)
C1.1
Rodin theorem provers
G1.3 The theorem about the group 1 of independent axioms is proved
G1.4 The theorem about the group 2 of independent axioms is proved
C1.2
Theorem thm_axm1 about the group 1
C1.3
Theorem thm_axm2 about the group 2
Sn1.1
Discharged PO thm_axm1/THM
Sn1.2
Discharged PO thm_axm2/THM

Figure 27: A fragment of the safety case corresponding to well-definedness of the development

The obtained proofs of these theorems are included in the safety case as the solutions **Sn1.1** and **Sn1.2** correspondingly.

5.3.2 Instantiation of the argument pattern for Class 1

The steam boiler control system has a large number of safety requirements [55]. Among them there are several requirements that can be classified as SRs belonging to *Class 1*. Let us demonstrate the instantiation of the corresponding argument pattern by the example of one such a safety requirement:

SR-02 : *During the system operation the water level shall not exceed the predefined safety boundaries.*

We formalise it as the invariant **inv1.2** at the first refinement step of the Event-B development (MACHINE **M1**):

$$\textbf{inv1.2:}\ failure = \textit{FALSE} \wedge phase \neq ENV \wedge phase \neq DET \Rightarrow$$
$$min\_water\_level \geq M1 \wedge max\_water\_level \leq M2,$$

where the variable *failure* represents a system failure, the variable *phase* models the stages of the steam boiler controller behaviour (i.e., the stages of its control loop), and finally the variables *min_water_level* and *max_water_level* represent the estimated interval for the sensed water level.

The mapping function F_M for this case is

$$\textbf{SR-02} \mapsto \{\textbf{inv1.2}\},$$

which is a concrete instance of its general form *Requirement* $\mapsto$ {*safety*$_1$, ..., *safety*$_N$} for *Class 1* given in Section 4.2.

To provide evidence that this safety requirement is met by the system, we instantiate the argument pattern for *Class 1* as shown in Figure 28.

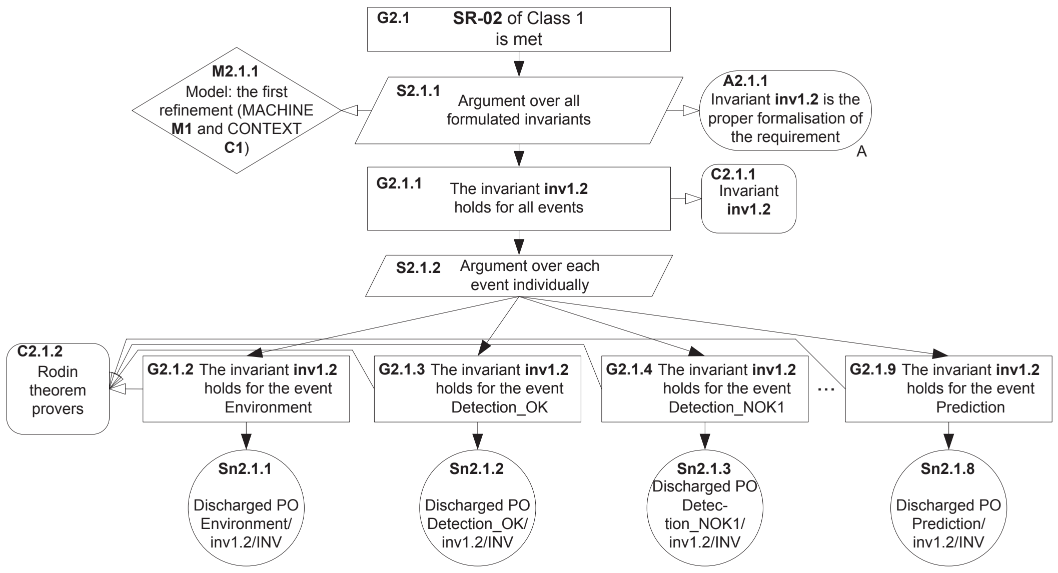

G2.1 SR-02 of Class 1 is met
M2.1.1 Model: the first refinement (MACHINE M1 and CONTEXT C1)
S2.1.1 Argument over all formulated invariants
A2.1.1 Invariant inv1.2 is the proper formalisation of the requirement
A
G2.1.1 The invariant inv1.2 holds for all events
C2.1.1 Invariant inv1.2
S2.1.2 Argument over each event individually
C2.1.2 Rodin theorem provers
G2.1.2 The invariant inv1.2 holds for the event Environment
G2.1.3 The invariant inv1.2 holds for the event Detection_OK
G2.1.4 The invariant inv1.2 holds for the event Detection_NOK1
...
G2.1.9 The invariant inv1.2 holds for the event Prediction
Sn2.1.1 Discharged PO Environment/ inv1.2/INV
Sn2.1.2 Discharged PO Detection_OK/ inv1.2/INV
Sn2.1.3 Discharged PO Detec- tion_NOK1/ inv1.2/INV
Sn2.1.8 Discharged PO Prediction/ inv1.2/INV

Figure 28: A fragment of the safety case corresponding to assurance of **SR-02**

The list of affected model events where this invariant must hold is the following: *Environment*, *Detection_OK*, *Detection_NOK1*, *Detection_NOK2*, *PreOperational1*, *PreOperational2*, *Operational*, *Prediction*. To support the claim that **inv1.2** holds for all these events, we attach the discharged proof obligations as the evidence. For brevity, we present only the supporting evidence **Sn2.1.2** of the goal **G2.1.3** as shown in Figure 29. This discharged proof obligation ensures that **inv1.2** holds for the event *Detection_OK* modelling detection of no failures.

$$\left.\begin{array}{l}\text{failure = FALSE} \wedge \text{phase} \neq \text{ENV} \wedge \text{phase} \neq \text{DET} \Rightarrow\\ \text{min\_water\_level} \geq \text{M1} \wedge \text{max\_water\_level} \leq \text{M2}\end{array}\right\}\ \text{I(d, c, v)}$$

$$\left.\begin{array}{l}\text{phase = DET}\\ \text{failure = FALSE}\\ \text{stop = FALSE}\\ \text{min\_water\_level} \geq \text{M1} \wedge \text{max\_water\_level} \leq \text{M2}\end{array}\right\}\ \text{g}_e\text{(d, c, v)}$$

$$\left.\text{phase' = CONT}\right\}\ \text{BA}_e\text{(d, c, v, v')}$$

$$\text{|-}$$

$$\left.\begin{array}{l}\text{failure=FALSE} \wedge\\ \text{CONT}\neq\text{ENV} \wedge\\ \text{CONT}\neq\text{DET} \Rightarrow\\ \text{min\_water\_level}\geq\text{M1} \wedge \text{max\_water\_level}\leq\text{M2}\end{array}\right\}\ \text{I(d, c, v')}$$

Figure 29: The proof obligation of the type INV for the event *Detection_OK* in **M1**

5.3.3 Instantiation of the argument pattern for Class 2

Since the steam boiler system is a failsafe system (i.e., it has to be put into a safe but non-operational state to prevent an occurrence of a hazard), whenever a system failure occurs, the system should be stopped. However, we abstractly model such failsafe procedures by assuming that, when the corresponding flag *stop* is raised thus indicating a system failure, the system is shut down and an alarm is activated. This condition is defined by the safety requirement **SR-01**:

SR-01 : *When a system failure is detected, the steam boiler control system shall be shut down and an alarm shall be activated.*

The stipulated property does not rely on a detailed representation of the steam boiler system and therefore can be incorporated at early stages of the development in Event-B, e.g., at the first refinement step (MACHINE **M1**). Since the property needs to be true at a specific state of the model, we classify this safety requirement as a SR belonging to *Class 2* and formalise it as the following theorem:

$$\begin{array}{ll}\textbf{thm1.1:} & \forall stop' \cdot stop' \in BOOL \wedge\\ & (\exists phase, stop \cdot phase \in \mathit{PHASE} \ \wedge \ stop \in BOOL \wedge\\ & phase = CONT \ \wedge \ stop = \mathit{FALSE} \ \wedge \ stop' = \mathit{TRUE})\\ & \Rightarrow\\ & stop' = \mathit{TRUE},\end{array}$$

where $stop' = \mathit{TRUE}$ is a predicate defining the required post-condition of the event *EmergencyStop*.

The corresponding instance of the mapping function F_M for this class of safety requirements in this case is

$$\textbf{SR-01} \mapsto \{(\mathit{EmergencyStop}, stop' = \mathit{TRUE})\}.$$

The instantiated fragment of the safety case is presented in Figure 30. The proof obligation (*thm1.1/THM*) serves as the evidence that this requirement holds.

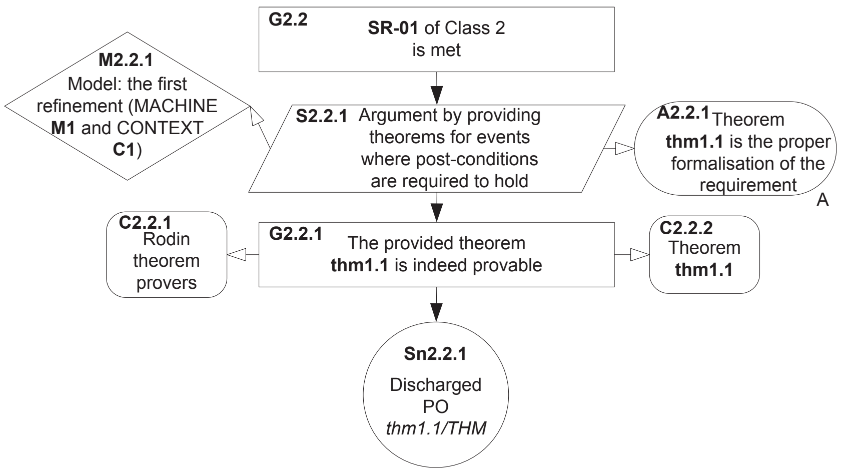

G2.2
SR-01 of Class 2 is met
M2.2.1
Model: the first refinement (MACHINE **M1** and CONTEXT **C1**)
S2.2.1 Argument by providing theorems for events where post-conditions are required to hold
A2.2.1 Theorem **thm1.1** is the proper formalisation of the requirement
A
C2.2.1 Rodin theorem provers
G2.2.1 The provided theorem **thm1.1** is indeed provable
C2.2.2 Theorem **thm1.1**
Sn2.2.1
Discharged PO *thm1.1/THM*

Figure 30: A fragment of the safety case corresponding to assurance of **SR-01**

5.3.4 Instantiation of the argument pattern for Class 3

Another safety-related property of the system under consideration is its cyclic behaviour. At each cycle the controller reads the sensors, performs computations and sets the actuators accordingly. Thus, the described control flow needs to be preserved by the Event-B model of the system as well. The safety requirement **SR-12** reflects the desired order in the control flow, associated with the corresponding order of the events in the Event-B model.

SR-12 : *The system shall operate cyclically. Each cycle it shall read the sensors, detect failures, perform either normal or degraded or rescue operation, or, in case of a critical system failure, stop the system, as well as compute the next values of variables to be used for failure detection at the next cycle if no critical system failure is detected.*

For the sake of simplicity, here we consider only the abstract model of the system (MACHINE **M0**). The refinement-based semantics of Event-B allows us to abstract away from detailed representation of the operational modes of the system (i.e., normal, degraded and rescue), ensuring nevertheless that the control flow properties proved at this step will be preserved by more detailed models.

We represent the required events order (**C2.3.2**) using the flow diagram (Figure 31). The generic mapping function *Requirement* $\mapsto \{(event_i, relationship, event_j)\}$ for *Class 3* can be instantiated in this case as

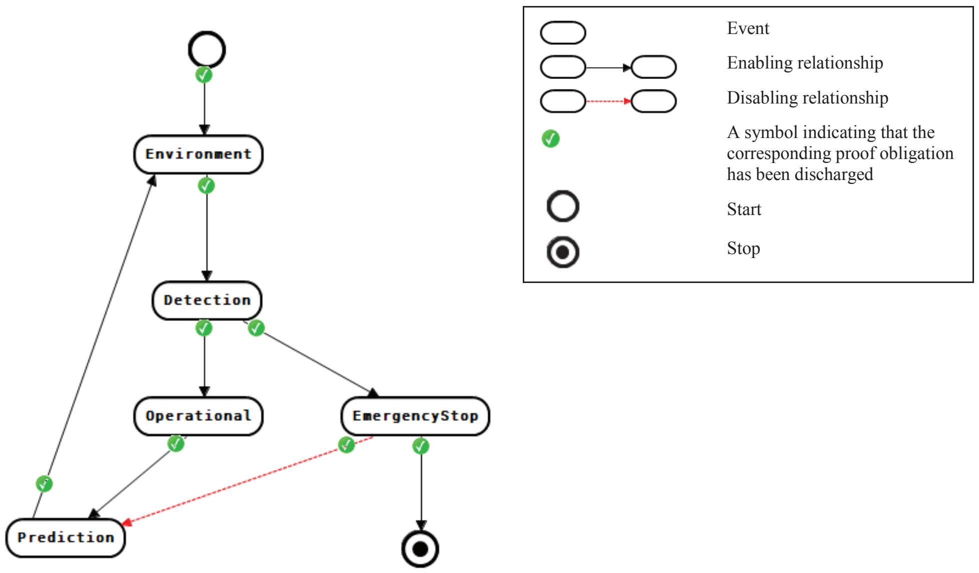


Figure 31: The flow diagram of the abstract MACHINE **M0**

$$\begin{aligned}\textbf{SR-12} \mapsto \{&(\mathit{Environment},\ \textbf{ena},\ \mathit{Detection}), (\mathit{Detection},\ \textbf{ena},\ \mathit{Operational}),\\ &(\mathit{Detection},\ \textbf{ena},\ \mathit{EmergencyStop}), (\mathit{Operational},\ \textbf{ena},\ \mathit{Prediction}),\\ &(\mathit{Prediction},\ \textbf{ena},\ \mathit{Environment}), (\mathit{EmergencyStop},\ \textbf{dis},\ \mathit{Prediction})\}.\end{aligned}$$

The instance of the pattern that ensures the order of the events in the MACHINE **M0** is presented in Figure 32. Due to the lack of space, we show only two proof obligations (Figure 33) discharged to support this branch of the safety case – *Environment/Detection/FENA* (**Sn2.3.1**) and *EmergencyStop/Prediction/ FDIS* (**Sn2.3.6**).

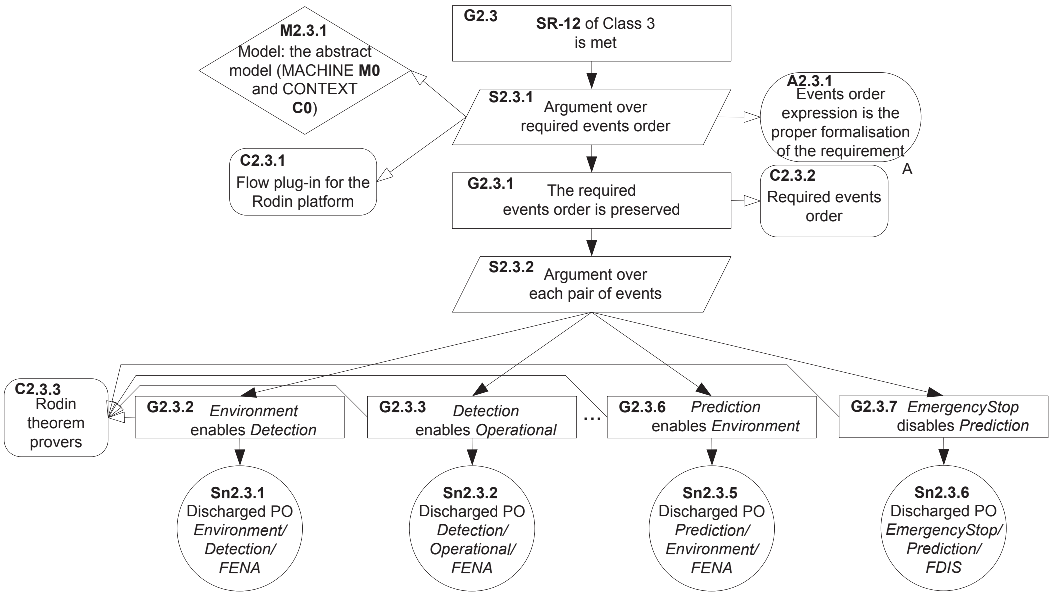
5.3.5 Instantiation of the argument pattern for Class 4

As we have already mentioned in Section 5.3.3, the steam boiler control system is a failsafe system. This means that it does contain a deadlock and therefore we do not need to construct the system safety case based on the argumentation defined by the pattern for *SRs about the absence of system deadlock* (*Class 4*). Quite opposite, we need to ensure that when the required sutdown condition is satisfied, the system terminates. Thus, we instantiate the pattern for *Class 5* instead.

5.3.6 Instantiation of the argument pattern for Class 5

Let us consider again the safety requirement **SR-01** given in Section 5.3.3:

SR-01 : *When a system failure is detected, the steam boiler control system shall be shut down and an alarm shall be activated.*



G2.3
SR-12 of Class 3 is met
M2.3.1
Model: the abstract model (MACHINE M0 and CONTEXT C0)
S2.3.1
Argument over required events order
A2.3.1
Events order expression is the proper formalisation of the requirement
A
C2.3.1
Flow plug-in for the Rodin platform
G2.3.1
The required events order is preserved
C2.3.2
Required events order
S2.3.2
Argument over each pair of events
C2.3.3
Rodin theorem provers
G2.3.2 Environment enables Detection
G2.3.3 Detection enables Operational
...
G2.3.6 Prediction enables Environment
G2.3.7 EmergencyStop disables Prediction
Sn2.3.1
Discharged PO Environment/ Detection/ FENA
Sn2.3.2
Discharged PO Detection/ Operational/ FENA
Sn2.3.5
Discharged PO Prediction/ Environment/ FENA
Sn2.3.6
Discharged PO EmergencyStop/ Prediction/ FDIS

Figure 32: A fragment of the safety case corresponding to assurance of **SR-12**

phase ∈ PHASE
failure ∈ BOOL
stop ∈ BOOL
failure = FALSE ⇒ stop = FALSE
failure = TRUE ∧ phase ≠ CONT ⇒ stop=TRUE
phase=ENV
stop=FALSE
|-
DET=DET ∧
failure=FALSE ∧
stop=FALSE

} **Environment/Detection/FENA**

failure ∈ BOOL
stop ∈ BOOL
failure = FALSE ⇒ stop = FALSE
failure = TRUE ∧ phase ≠ CONT ⇒ stop = TRUE
phase = CONT
failure = TRUE
stop = FALSE
|-
¬ (phase = PRED ∧ TRUE = FALSE)

} **EmergencyStop/Prediction/FDIS**

Figure 33: The proof obligations of the types FENA and FDIS

The corresponding model theorem **thm1.1** (see Section 5.3.3) guarantees that the system variables are updated accordingly to prepare for a system shutdown, e.g., the stop flag is raised. However, it does not ensure that the system indeed terminates, i.e., there are no enabled system events anymore. This should be done separately. Therefore, this safety requirement can be classified as a requirement belonging to both *Class 2* and *Class 5*. To show that our system definitely meets this requirement, we instantiate the argument pattern for *Class 5* as well (Figure 34).

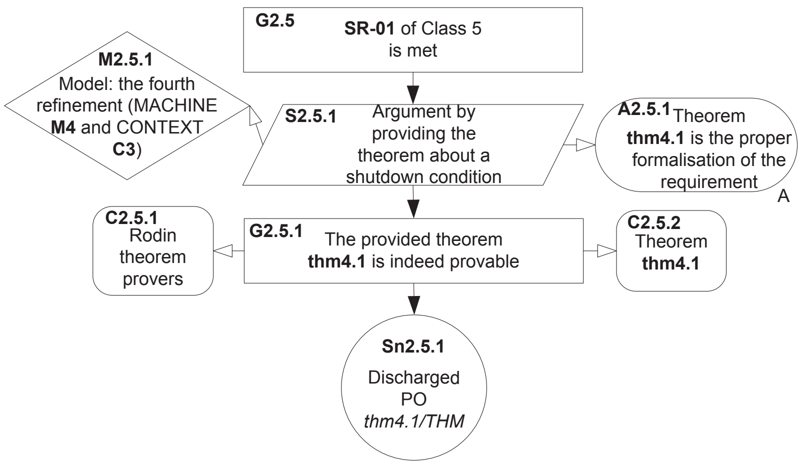

G2.5
SR-01 of Class 5 is met
M2.5.1
Model: the fourth refinement (MACHINE M4 and CONTEXT C3)
S2.5.1
Argument by providing the theorem about a shutdown condition
A2.5.1
Theorem thm4.1 is the proper formalisation of the requirement
A
C2.5.1
Rodin theorem provers
G2.5.1
The provided theorem thm4.1 is indeed provable
C2.5.2
Theorem thm4.1
Sn2.5.1
Discharged PO thm4.1/THM

Figure 34: A fragment of the safety case corresponding to assurance of **SR-01**

In this case, the corresponding instance of the generic mapping function F_M for *Class 5* is

$$\textbf{SR-01} \mapsto \{\mathit{stop} = \mathit{TRUE}, \mathit{Environment}, \mathit{Detection\_OK\_no\_F}, ..., \\ \mathit{EmergencyStop}, \mathit{Prediction}\},$$

where *stop* = *TRUE* stands for the required shutdown condition.

Then, the corresponding theorem **thm4.1** is formulated as follows:

$$\textbf{thm4.1:}\ \mathit{stop} = \mathit{TRUE} \Rightarrow \neg(\mathit{before}(\mathit{Environment}) \vee \mathit{before}(\mathit{Detection\_OK\_no\_F}) \\ \vee\ ..\ \vee \mathit{before}(\mathit{EmergencyStop}) \vee \mathit{before}(\mathit{Prediction})),$$

which in turn can be rewritten (by expanding the definition of *before(e)* described in detail in Section 2.1) as:

$$\begin{aligned}\textbf{thm4.1:}\ & \mathit{stop} = \mathit{TRUE} \Rightarrow \neg((\mathit{stop} = \mathit{FALSE} \wedge \mathit{phase} = ENV \wedge ..\,) \vee \\ & (\mathit{stop} = \mathit{FALSE} \wedge \mathit{phase} = DET \wedge ..\,) \vee\ ..\ \vee \\ & (\mathit{stop} = \mathit{FALSE} \wedge \mathit{phase} = CONT \wedge ..\,) \vee \\ & (\mathit{stop} = \mathit{FALSE} \wedge \mathit{phase} = PRED \wedge ..\,)); \\ \\ & \mathit{stop} = \mathit{TRUE} \Rightarrow \neg(\mathit{stop} = \mathit{FALSE} \wedge ((\mathit{phase} = ENV \wedge ..\,) \vee \\ & (\mathit{phase} = DET \wedge ..\,) \vee\ ..\ \vee \\ & (\mathit{phase} = CONT \wedge ..\,)\ \vee\ (\mathit{phase} = PRED \wedge ..\,)));\end{aligned}$$

$$\begin{aligned}&stop = \mathit{TRUE} \Rightarrow \neg stop = \mathit{FALSE} \lor \neg((phase = ENV \land ..\,) \lor \\ &(phase = DET \land ..\,) \lor\ ..\lor \\ &(phase = CONT \land ..\,)\ \lor\ (phase = PRED \land ..\,));\end{aligned}$$

$$\begin{aligned}&stop = \mathit{TRUE} \Rightarrow stop = \mathit{TRUE} \lor \neg((phase = ENV \land ..\,) \lor \\ &(phase = DET \land ..\,) \lor\ ..\lor \\ &(phase = CONT \land ..\,)\ \lor\ (phase = PRED \land ..\,)).\end{aligned}$$

The discharged proof obligation (*thm4.1/THM*) provides the evidence for validity of the claim **G2.5** (see Figure 34).

5.3.7 Instantiation of the argument pattern for Class 6

We demonstrate an application of the argument pattern for *Class 6* on a pair of hierarchically linked requirements for the steam boiler system.

The requirement **R-09-higher-level** describes general behaviour of the pump actuator in the operational system phase, which concerns safety of the system only implicitly:

R-09-higher-level : *In the operational phase of the system execution, the pump actuator can be switched on or off (based on the water level estimations), or stay in the same mode*,

while its more detailed counterpart (**SR-09-lower-level**) does this explicitly. It stipulates the behaviour of the system and the pump actuator in the presence of a pump actuator failure:

SR-09-lower-level : *When the pump actuator fails, it shall stay in its current mode.*

In our formal development, these requirements are also introduced gradually at different refinement steps. More specifically, the first one is formalised at the first refinement step (MACHINE **M1**), while the second one is incorporated at the second refinement step (MACHINE **M2**).

We consider both requirements as requirements belonging to *Class 2*. Therefore, their verification is done by proving the corresponding theorems about post-states of specific events. Here we assume that the corresponding separate fragments of the safety case have been constructed using the argument pattern for *Class 2* to guarantee that the requirements **R-09-higher-level** and **SR-09-lower-level** hold. However, in this section we leave out these fragments of the safety case while focusing on ensuring the hierarchical consistency between these requirements. In other words, we focus on application of the argument pattern for *Class 6*.

The correctness of the hierarchical link between the requirements **R-09-higher-level** and **SR-09-lower-level** is guaranteed via operation refinement of the affected events belonging to MACHINE **M1** and MACHINE **M2** correspondingly. In this particular case, these are the abstract event *Operational* in **M1** and its refinement – the event *Degraded_Operational* in **M2**. The events are presented in Figure 35.

```
// Event in the MACHINE M1
event Operational refines Operational
  where
    @grd1 phase = CONT
    @grd2 failure = FALSE
    @grd3 stop = FALSE
    @grd4 preop_flag = FALSE
    @grd5 min_water_level ≥ M1 ∧
            max_water_level ≤ M2
  then
    @act1 phase ≔ PRED
    @act2 pump_ctrl :| pump_ctrl' ∈ PUMP_MODE ∧
          (pump_ctrl' = pump_ctrl ∨
          ((min_water_level ≥ M1 ∧ max_water_level < N1 ⇒
          pump_ctrl' = ON) ∧
          (min_water_level > N2 ∧ max_water_level ≤ M2 ⇒
          pump_ctrl' = OFF) ∧
          (min_water_level ≥ N1 ∧ max_water_level ≤ N2 ⇒
          pump_ctrl' = pump_ctrl)))
  end
```

```
// Event in the refined MACHINE M2
event Degraded_Operational  refines Operational
  where
    @grd1 phase = CONT
    @grd3 stop = FALSE
    @grd4 preop_flag = FALSE
    @grd6 wl_sensor_failure = FALSE ∧
            (pump_failure = TRUE ∨ so_sensor_failure = TRUE)
    @grd7 valve_ctrl = CLOSED
    @grd8 WL_critical(min_water_level ↦
                        max_water_level) = FALSE
  then
    @act1 phase ≔ PRED
    @act2 pump_ctrl :| pump_ctrl' ∈ PUMP_MODE ∧
          (pump_failure = TRUE ⇒ pump_ctrl' = pump_ctrl) ∧
          (pump_failure = FALSE ∧ min_water_level ≥ M1 ∧
          max_water_level < N1 ⇒ pump_ctrl' = ON) ∧
          (pump_failure = FALSE ∧ min_water_level > N2 ∧
          max_water_level ≤ M2 ⇒ pump_ctrl' = OFF) ∧
          (pump_failure = FALSE ∧ min_water_level ≥ N1 ∧
          max_water_level ≤ N2 ⇒ pump_ctrl' = pump_ctrl)
end
```

Figure 35: Events *Operational* and *Degraded_Operatinal*

In the MACHINE **M1**, we abstractly model a system failure by the variable *failure*. Then, in the MACHINE **M2**, we substitute this abstract variable and introduce the variables standing for failures of the system components, namely, the water level sensor failure – the variable *wl_sensor_failure*, the pump failure – the variable *pump_failure*, and the steam output sensor failure – the variable *so_sensor_failure*. The precise formal relationships between these new variables and the old one is depicted by the respective gluing invariant. In other words, the gluing invariant added to the MACHINE **M2** relates these concrete variables with the variable *failure* modelling an abstract failure.

The described data refinement directly affects the considered events *Operational* and *Degraded_Operational*. To guarantee that the refinement of the variable *failure* in the event *Degraded_Operational* does not weaken the corresponding guard of the event *Operational*, i.e., **grd2**, the proof obligation of the type GRD is discharged (see Section 4.7). Moreover, to satisfy the requirement **SR-09-lower-level**, we modify the action **act2** as shown in Figure 35. The correctness of this kind of simulation is guaranteed by the proof obligation of the type SIM. This pair of discharged proof obligations serves as the evidence that the consistency relationship between the corresponding hierarchically linked requirements is preserved by refinement.

The resulting instance of the argument pattern is shown in Figure 36. Here the mentioned proof obligations are attached as the safety case evidence – **Sn2.6.1** and **Sn2.6.2** respectively. Due to the large size, we do not show the details of these proof obligations in this paper.

5.3.8 Instantiation of the argument pattern for Class 7

To demonstrate an instantiation of the argument pattern for *Class 7* (i.e., a class representing safety requirements about temporal properties), we consider the following safety requirement

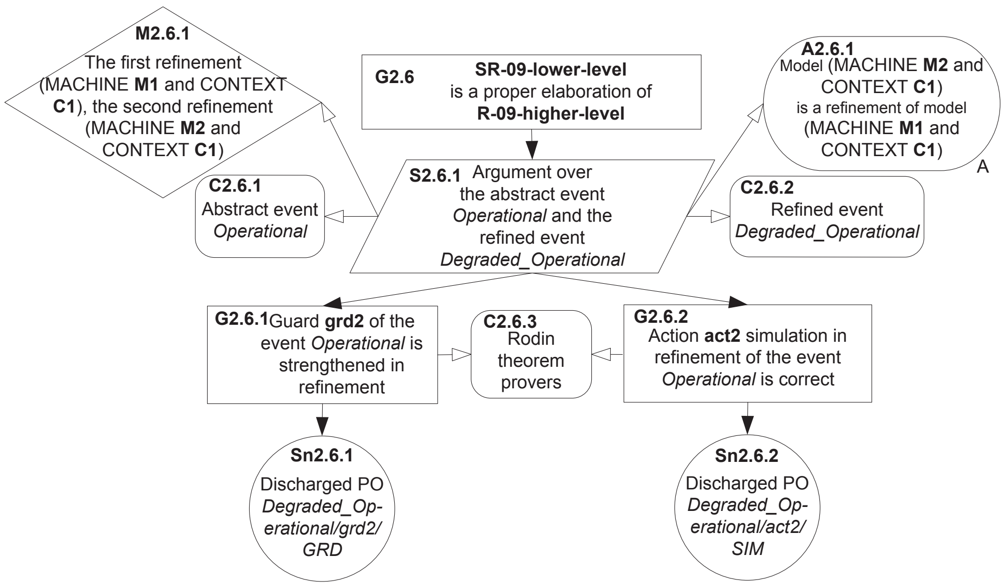

M2.6.1
The first refinement (MACHINE M1 and CONTEXT C1), the second refinement (MACHINE M2 and CONTEXT C1)
G2.6
SR-09-lower-level is a proper elaboration of R-09-higher-level
A2.6.1
Model (MACHINE M2 and CONTEXT C1) is a refinement of model (MACHINE M1 and CONTEXT C1)
A
C2.6.1
Abstract event Operational
S2.6.1 Argument over the abstract event Operational and the refined event Degraded_Operational
C2.6.2
Refined event Degraded_Operational
G2.6.1 Guard grd2 of the event Operational is strengthened in refinement
C2.6.3
Rodin theorem provers
G2.6.2
Action act2 simulation in refinement of the event Operational is correct
Sn2.6.1
Discharged PO Degraded_Operational/grd2/GRD
Sn2.6.2
Discharged PO Degraded_Operational/act2/SIM

Figure 36: A fragment of the safety case corresponding to assurance of hierarchical requirements **R-09-higher-level** and **SR-09-lower-level**

of the steam boiler system:

SR-13 : *If there is no system failure, the system shall continue its operation in a new cycle.*

In our Event-B specification of the steam boiler system, the new cycle starts when the system enables the event *Environment* (Figure 31). Therefore, we have to show that, whenever no failure is detected in the detection phase, the system will start a new cycle by eventually reaching the event *Environment*. According to our pattern, we associate the requirement **SR-13** with a temporal reachability property. The corresponding instance of the generic mapping function F_M for *Class 7* in this case is

$$\textbf{SR-13} \mapsto \{\textbf{temp\_property}\},$$

where **temp_property** is an LTL formula defined as

$$\textbf{temp\_property:}\ \Box\,(after(Detection)\ \wedge\ failure = FALSE\ \rightarrow \Diamond\ before(Environment)).$$

This formula has the following representation in the ProB plug-in:

$$\begin{aligned} G\ (\{(\forall phase', failure'\ \cdot\ & phase' \in PHASE\ \wedge\ failure' \in BOOL \wedge \\ & (\exists\, phase, stop, failure\ \cdot\ phase \in PHASE\ \wedge\ stop \in BOOL \wedge \\ & failure \in BOOL\ \wedge\ phase = DET\ \wedge\ failure = FALSE \wedge \\ & stop = FALSE)\ \wedge\ phase' = CONT\ \wedge\ failure' \in BOOL) \wedge \\ & failure = FALSE\} \\ & \Rightarrow \\ & F\ \{phase = ENV\ \wedge\ stop = FALSE\}). \end{aligned}$$

As a result of the model checking on this property, ProB yields the following outcome: *"no counter-example has been found, all nodes have been visited"*. Therefore, we can attach this result as the evidence for the corresponding fragment of our safety case (**Sn2.7.1**). The resulting instance of the argument pattern is shown in Figure 37.

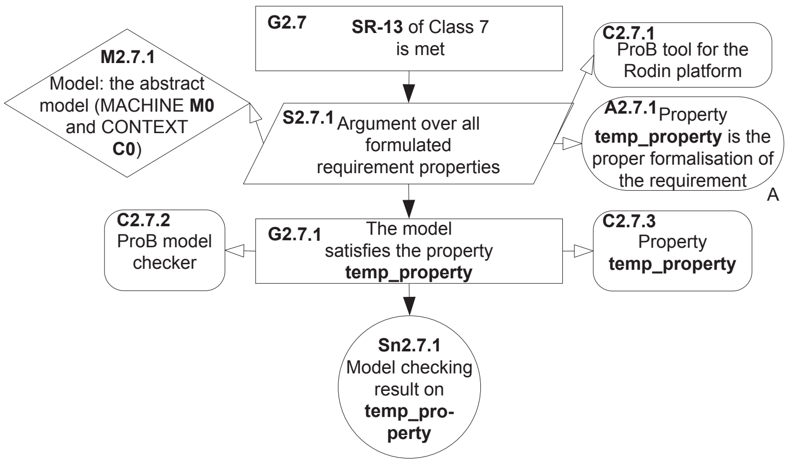

G2.7 SR-13 of Class 7 is met
M2.7.1 Model: the abstract model (MACHINE M0 and CONTEXT C0)
S2.7.1 Argument over all formulated requirement properties
C2.7.1 ProB tool for the Rodin platform
A2.7.1 Property temp_property is the proper formalisation of the requirement
A
C2.7.2 ProB model checker
G2.7.1 The model satisfies the property temp_property
C2.7.3 Property temp_property
Sn2.7.1 Model checking result on temp_pro-perty

Figure 37: A fragment of the safety case corresponding to assurance of **SR-13**

5.3.9 Instantiation of the argument pattern for Class 8

We did not take into account timing constraints imposed on the steam boiler control system while developing the formal system specification in Event-B. Therefore, we could not support the system safety case with a fragment associated with the safety requirements about timing properties (*Class 8*).

5.4 Discussion on the application of the approach

Despite the fact that the accomplished Event-B development of the steam boiler control system is quite complicated and, as a result, a significant number of proof obligations has been discharged, we have not been able to instantiate two argument patterns, namely the patterns for *Class 4* and *Class 8*. First of all, the steam boiler control system is a failsafe system, which means that there is a deadlock in its execution. Consequently, there are no requirements about the absence of system deadlock (*Class 4*). Second of all, timing properties (*Class 8*) were not a part of the given system requirements either. Nevertheless, the presented guidelines on the instantiation of the argument patterns have allowed us to easily construct the corresponding fragments of the system safety case for the remaining safety requirements as well as to demonstrate well-definedness of the overall development of the system.

The use of the Rodin platform and accompanying plug-ins has facilitated derivation of the proof- and model checking-based evidence that the given safety requirements hold for the modelled system. The proof-based semantics of Event-B (a strong relationship between model elements and the associated proof obligations) has given us a direct access to the corresponding proof obligations. It has allowed us to not just claim that the verified theorems were proved but also explicitly include the obtained proof obligations into the resulting safety case.

6 Related work

In this section, we overview related contributions according to the following three directions: firstly, we consider the publications on the use of formal methods for safety cases; secondly, we overview the works that aim at formalising safety requirements; and thirdly, we take a closer look at the approaches focusing on argument patterns.

Formal methods in safety cases. There are two main research directions in utilising formal methods for safety cases. On the one hand, a safety case argument itself can be formally defined and verified. On the other hand, safety requirements can be formalised and formally verified allowing us to determine the safety evidence such as the obtained results of static analysis, model checking or theorem proving. Note that such evidence corresponds to the class of safety evidence called *formal verification results* defined in the safety evidence taxonomy proposed by Nair et al. in [51].

In the former case, soundness of a safety argument can be proved by means of theorem proving in the classical or higher order logic, e.g., using the interactive theorem prover PVS [33, 57]. In particular, Rushby [57] formalises a top-level safety argument to support automated checking of soundness of a safety argument. He proposes to represent a safety case argument in the classical logic as a theorem where antecedents are the assumptions under which a system (or design) satisfies the consequent, whereas the consequent is a specific claim in the safety case that has to be assured. Then, such a theorem can be verified by an automated interactive theorem prover or a model checker.

In the latter case, soundness of an overall safety case is not formally examined. The focus is rather put on the evidence derived from formal analysis to show that the specific goals reflecting safety requirements are met. For example, to support the claim that the source code of a program module does not contain potentially hazardous errors, the authors of [32] use as the evidence the results of static analysis of program code. In [8,9], the authors assure safety of automatically generated code by providing formal proofs as the evidence. They ensure that safety requirements hold in specific locations of software system implementations. In [22, 23], the authors automate generation of heterogeneous safety cases including a manually developed top-level system safety case, and lower-level fragments automatically generated from the formal verification of safety requirements. According to this approach, the implementation is formally verified against a mathematical specification within a logical domain theory. This approach is developed for the aviation domain and illustrated by an unmanned aircraft system. To ensure that a model derived during model-driven development of a safety critical system, namely pacemaker, satisfies all the required properties, the authors of [43] use the obtained model checking results. Our approach proposed in this paper also belongs to this category. Formalisation and verification of safety requirements of a critical system allows us to obtain the proof- and model checking-based evidence that these requirements hold.

Formalisation of safety requirements. Incorporation of requirements in general, and safety requirements in particular, into formal specifications is considered to be a challenging task. We overview some recent approaches that address this problem dividing them into two categories: those that aim at utilising model checking for verification of critical properties, and those that employ theorem proving for this purpose.

For example, a formalisation of safety requirements using the Computation Tree Logic (CTL) and then verification of them using a model checker is presented in [14]. The author classifies the given requirements associating them with the corresponding CTL formulas. A similar approach is presented in [35]. Here safety properties defined as LTL formulas are verified by using the SPIN model checker.

In contrast, the authors of [16] perform a systematic transformation of a Petri net model into an abstract B model for verification of safety properties by theorem proving. Another work that aims at verifying safety requirements by means of theorem proving is presented in [46]. The authors incorporate the given requirements into an Event-B model via applying a set of automated patterns, which are based on Failure Modes and Effects Analysis (FMEA).

Similarly to these works, we take an advantage of using theorem proving and a refinement-based approach to formal development of a system. We gradually introduce the required safety properties into an Event-B model and verify them in the refinement process. This allows us to avoid the state explosion problem commonly associated with model checking, thus making our approach more scalable for systems with higher levels of complexity. Nonetheless, in this paper, we also rely on model checking for those properties that cannot be verified by our framework directly.

Furthermore, there are other works that aim at formalising safety requirements, specifically in Event-B [41, 42, 49, 59]. Some of them propose to incorporate safety requirements as invariants and before-after predicates of events [41, 59], while others, e.g., [49], represent them as invariants or theorems only. Moreover, all these works show the correspondence between some particular requirements and the associated elements of the Event-B structure. However, they neither classify the safety requirements nor give precise guidelines for formal verification of those requirements that cannot be directly verified by the Event-B framework. In contrast, to be able to argue over each given safety requirement by relying on its formal representation, we propose a classification of safety requirements and define a precise mapping of each class onto a set of the corresponding model expressions. Moreover, for some of these classes, we propose bridging Event-B with other tools (model checkers).

Argument patterns. In general, argument patterns (or safety case patterns) facilitate construction of a safety case by capturing commonly used structures and allowing for simplified instantiation. Safety case patterns have been introduced by Kelly and McDermid [45] and received recognition among safety case developers. In [20], the authors give a formal definition for safety case patterns, define formal semantics of patterns, and propose a generic data model and algorithm for pattern instantiation. For example, a safety case pattern for arguing the correctness of implementations developed from a timed automata model using a model-based development approach has been presented in [6]. An instantiation of this pattern has been illustrated on the implementation software of a patient controlled analgesic infusion pump. In [7], the author proposes a set of property-oriented and requirement-oriented safety case patterns for arguing safe execution of automatically generated program code with respect to the given safety properties as well as safety requirements. Additionally, the author defines architecture-oriented patterns for safety-critical systems developed using a model-based development approach.

An approach to automatically integrating the output generated by a formal method or tool into a software safety assurance case as an evidence argument is described in [21]. To capture the reasoning underlying a tool-based formal verification, the authors propose specific safety case patterns. The approach is software-oriented. A formalised requirement is verified to hold at a specific location of code. The proposed patterns allow formal reasoning and evidence to be integrated into the language of assurance arguments. Our approach is similar to the approach presented in [21]. However, we focus on formal system models rather that the code. Moreover, the way system safety requirements are formalised and verified in Event-B varies according to the proposed classification of safety requirements. Consequently, the resulting evidence arguments are also different. Nevertheless, we believe that the approach given in [21] can be used to complement our approach.

In this paper, we contribute to a set of existing safety case patterns and describe in detail their instantiation process for different classes of safety requirements. Moreover, our proposed patterns facilitate construction of safety cases where safety requirements are verified formally and the corresponding formal-based evidence is derived to represent justification of safety assurance. The evidence arguments obtained by applying our approach explicitly reflect the formal reasoning instead of just references to the corresponding proofs or the model checking results.

7 Conclusions

In this paper, we propose a methodology supporting rigorous construction of safety cases from formal Event-B models. It guides the developers starting from informal representation of safety requirements to building the corresponding parts of safety cases via formal modelling and verification in Event-B and the accompanying toolsets.

We believe that the proposed methodology has shown good scalability. In this paper, we have illustrated the application of our methodology both by small examples and a larger case study without major difficulties. Moreover, we have applied the methodology in two different situations: when formal models of systems were developed beforehand, and when the development was performed in parallel with the construction of the associated safety case. Specifically, all the formal models for illustrating the argument patterns in Section 4 were taken as given, while the formal development of the steam boiler system (presented in our previous work [55] and partially in Section 5.3) was done taking into account the proposed classification of safety requirements and the need to produce a safety case of the system. We have additionally observed the fact that, to construct an adequate safety case of a system based on its formal model, a *feedback loop* between two processes, namely, the process of formal system development and construction of safety cases, is required. It means that, if construction of a safety case indicates that the associated formal model is "weak", i.e., it does not contain an adequate formalisation of some safety requirements that need to be demonstrated in the safety case, the developers should be able to react on that by improving the model.

Our main contribution, namely, the proposed methodology for rigorous construction of safety cases, has led us to achieving the following two sub-contributions. Firstly, we have classified safety requirements and shown how they can be formalised in Event-B. To attain this, we have proposed a strict mapping between the given safety requirements and the associated elements of formal models, thus establishing a clear traceability of those requirements. Secondly, we have proposed a set of argument patterns based on the proposed classification, the use of which facilitates the construction of safety cases. Due to the strong relationship between model elements and the associated proof obligations provided by the proof-based semantics of Event-B, we have been able to formally verify the mapped safety requirements and derive the corresponding proofs. Moreover, via developing the argument strategies based on formal reasoning and using the resulting proofs as the evidence for a safety case, we have achieved additional assurance that the desired safety requirements hold.

Furthermore, application of the well-defined Event-B theory for formal verification of safety requirements and formal-based construction of safety cases has clarified the use of particular safety case goals or strategies. It has allowed us to omit the additional explanations why the defined strategies are needed and why the proposed evidence is relevant. Otherwise, we would have needed to extend each proposed argument pattern with multiple instances of a specific GSN element called *justification* [28]. Consequently, this would have led to a significant growth of already large safety cases.

In this work, we have focused on safety aspects however the proposed approach can be extended to cover other dependability attributes, e.g., reliability and availability. We also believe that the generic principles described in this paper by the example of the Event-B formalism are applicable to any other formalism defined as a state transition system, e.g., B, Z, VDM, refinement calculus, etc.

So far, all the proposed patterns and their instantiation examples have been developed manually. However, the larger a system under consideration is, the more difficult this procedure becomes. Therefore, the necessity of automated tool support is obvious. We consider development of a dedicated plug-in for the Rodin platform as a part of our future work. Moreover, the proposed classification of the safety requirements is by no means complete. Consequently, it could be further extended with some new classes and the corresponding argument patterns.

Acknowledgements

Yuliya Prokhorova's work is partially supported by the Foundation of Nokia Corporation. Additionally, the authors would like to thank Prof. Michael Butler for the valuable feedback on the requirements classification.

References

[1] J.-R. Abrial. Steam-Boiler Control Specification Problem. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, pages 500–509, London, UK, 1996. Springer-Verlag.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. Controlling Cars on a Bridge. http://deploy-eprints. ecs.soton.ac.uk/112/, April 2010.

[4] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[5] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer Berlin Heidelberg, 1998.

[6] A. Ayoub, B.G. Kim, I. Lee, and O. Sokolsky. A Safety Case Pattern for Model-Based Development Approach. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*, pages 141–146, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] N. Basir. *Safety Cases for the Formal Verification of Automatically Generated Code*. Doctoral thesis, University of Southampton, 2010.

[8] N. Basir, E. Denney, and B. Fischer. Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. In M.D. Harrison and M.-A. Sujan, editors, *Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 249–262. Springer Berlin Heidelberg, 2008.

[9] N. Basir, E. Denney, and B. Fischer. Deriving Safety Cases from Automatically Constructed Proofs. In *Systems Safety 2009. Incorporating the SaRS Annual Conference, 4th IET International Conference on*, pages 1–6, 2009.

[10] N. Basir, E. Denney, and B. Fischer. Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In E. Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 68–81. Springer Berlin Heidelberg, 2010.

[11] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[12] J. Berthing, P. Boström, K. Sere, L. Tsiopoulos, and J. Vain. Refinement-Based Development of Timed Systems. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin Heidelberg, 2012.

[13] P. Bishop and R. Bloomfield. A Methodology for Safety Case Development. In *Safety-Critical Systems Symposium, Birmingham, UK*. Springer-Verlag, 1998.

[14] F. Bitsch. Classification of Safety Requirements for Formal Verification of Software Models of Industrial Automation Systems. In *Proceedings of 13th International Conference on Software and Systems Engineering and their Applications (ICSSEA'00)*, Paris, France, 2000. CNAM.

[15] F. Bitsch. Safety Patterns - The Key to Formal Specification of Safety Requirements. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP'01)*, pages 176–189, London, UK, UK, 2001. Springer-Verlag.

[16] P. Bon and S. Collart-Dutilleul. From a Solution Model to a B Model for Verification of Safety Properties. *Journal of Universal Computer Science*, 19(1):2–24, 2013.

[17] Claims, Arguments and Evidence (CAE). http://www.adelard.com/ asce/choosing-asce/cae.html, 2014.

[18] D. Cansell, D. Méry, and J. Rehm. Time Constraint Patterns for Event-B Development. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154. Springer Berlin Heidelberg, 2007.

[19] UK Ministry of Defence. 00-56 Safety Management Requirements for Defence Systems, 2007.

[20] E. Denney and G. Pai. A Formal Basis for Safety Case Patterns. In F. Bitsch, J. Guiochet, and M. Kaâniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 21–32. Springer Berlin Heidelberg, 2013.

[21] E. Denney and G. Pai. Evidence Arguments for Using Formal Methods in Software Certification. In *Proceedings of IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13)*, pages 375–380, 2013.

[22] E. Denney, G. Pai, and J. Pohl. Heterogeneous Aviation Safety Cases: Integrating the Formal and the Non-formal. In *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*, pages 199–208, Washington, DC, USA, 2012. IEEE Computer Society.

[23] E.W. Denney, G.J. Pai, and J.M. Pohl. Automating the Generation of Heterogeneous Aviation Safety Cases. NASA Contractor Report NASA/CR-2011-215983, August 2011.

[24] EB2ALL - The Event-B to C, C++, Java and C# Code Generator. http://eb2all.loria.fr/, October 2013.

[25] European Committee for Electrotechnical Standardization (CENELEC). EN 50128 Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. June 2011.

[26] Event-B and the Rodin Platform. http://www.event-b.org/, 2014.

[27] The Flow plug-in. http://iliasov.org/usecase/, 2014.

[28] Goal Structuring Notation Working Group. Goal Structuring Notation Standard. http://www.goalstructuringnotation.info/, November 2011.

[29] J. Groslambert. Verification of LTL on B Event Systems. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 109–124. Springer Berlin Heidelberg, 2006.

[30] I. Habli and T. Kelly. A Generic Goal-Based Certification Argument for the Justification of Formal Analysis. *Electronic Notes in Theoretical Computer Science*, 238(4):27–39, September 2009.

[31] J. Hatcliff, A. Wassyng, T. Kelly, C. Comar, and P. Jones. Certifiably Safe Software-dependent Systems: Challenges and Directions. In *Proceedings of the Track on Future of Software Engineering (FOSE'14)*, pages 182–200, New York, NY, USA, 2014. ACM.

[32] R. Hawkins, I. Habli, T. Kelly, and J. McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety Science*, 59:55–71, 2013.

[33] H. Herencia-Zapana, G. Hagen, and A. Narkawicz. Formalizing Probabilistic Safety Claims. In M. Bobaru, K. Havelund, G.J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 2011.

[34] T.S. Hoang and J.-R. Abrial. Reasoning about Liveness Properties in Event-B. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 456–471. Springer Berlin Heidelberg, 2011.

[35] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[36] IEC61508. International Electrotechnical Commission. IEC 61508, functional safety of electrical/electronic/programmable electronic safety-related systems. April 2010.

[37] A. Iliasov. Use Case Scenarios as Verification Conditions: Event-B/Flow Approach. In *Proceedings of the 3rd International Workshop on Software Engineering for Resilient Systems (SERENE'11)*, pages 9–23, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] A. Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, and T. Latvala. Augmenting Event B Modelling with Real-Time Verification. TUCS Technical Report 1006, 2011.

[39] A. Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, and T. Latvala. Augmenting Event-B Modelling with Real-Time Verification. In *Proceedings of Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA'12)*, pages 51–57, 2012.

[40] International Organization for Standardization. ISO 26262 Road Vehicles Functional Safety. November 2011.

[41] M. Jastram, S. Hallerstede, and L. Ladenberger. Mixing Formal and Informal Model Elements for Tracing Requirements. In *Electronic Communications of the EASST*, volume 46, 2011.

[42] M. Jastram, S. Hallerstede, M. Leuschel, and A.G. Russo Jr. An Approach of Requirements Tracing in Formal Refinement. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'10)*, pages 97–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] E. Jee, I. Lee, and O. Sokolsky. Assurance Cases in Model-Driven Development of the Pacemaker Software. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 2010.

[44] T.P. Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases*. Doctoral thesis, University of York, September 1998.

[45] T.P. Kelly and J.A. McDermid. Safety Case Construction and Reuse Using Patterns. In P. Daniel, editor, *Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP'97)*, pages 55–69. Springer-Verlag London, 1997.

[46] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Report 1003, 2011.

[47] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB/index.php5/LTL_Model_ Checking, 2014.

[48] D. Méry. Requirements for a Temporal B Assigning Temporal Meaning to Abstract Machines ... and to Abstract Systems. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 395–414. Springer London, 1999.

[49] D. Méry and N.K. Singh. Technical Report on Interpretation of the Electrocardiogram (ECG) Signal using Formal Methods. Technical Report INRIA-00584177, 2011.

[50] C. Metayer, J.-R. Abrial, and L. Voisin. Event-B Language. Rigorous Open Development Environment for Complex Systems (RODIN) Deliverable 3.2. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, May 2005.

[51] S. Nair, J.L. de la Vara, M. Sabetzadeh, and L. Briand. Classification, Structuring, and Assessment of Evidence for Safety – A Systematic Literature Review. In *Proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST'13)*, pages 94–103, 2013.

[52] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page, 2014.

[53] Y. Prokhorova, L. Laibinis, E. Troubitsyna, K. Varpaaniemi, and T. Latvala. Deriving a mode logic using failure modes and effects analysis. *International Journal of Critical Computer-Based Systems*, 3(4):305—328, 2012.

[54] Y. Prokhorova and E. Troubitsyna. Linking Modelling in Event-B with Safety Cases. In P. Avgeriou, editor, *Software Engineering for Resilient Systems*, volume 7527 of *Lecture Notes in Computer Science*, pages 47–62. Springer Berlin Heidelberg, 2012.

[55] Y. Prokhorova, E. Troubitsyna, and L. Laibinis. A Case Study in Refinement-Based Modelling of a Resilient Control System. TUCS Technical Report 1086, 2013.

[56] Y. Prokhorova, E. Troubitsyna, L. Laibinis, D. Ilić, and T. Latvala. Formalisation of an Industrial Approach to Monitoring Critical Data. TUCS Technical Report 1070, 2013.

[57] J. Rushby. Formalism in Safety Cases. In C. Dale and T. Anderson, editors, *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, pages 3–17, Bristol, UK, 2010. Springer.

[58] M.R. Sarshogh and M. Butler. Specification and Refinement of Discrete Timing Properties in Event-B. In *Electronic Communications of the EASST*, volume 46, 2011.

[59] S. Yeganefard and M. Butler. Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation. In *Electronic Communications of the EASST*, volume 46, 2011.

Paper VII

A Survey of Safety-Oriented Model-Driven and Formal Development Approaches

Yuliya Prokhorova and Elena Troubitsyna

Originally published in: *International Journal of Critical Computer-Based Systems (IJCCBS)*, Vol. 4, No. 2, pp. 93—118, Inderscience Publishers, 2013.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3174-2
ISSN 1239-1883

Yuliya Prokhorova

Rigorous Development of Safety-Critical Systems

Yuliya Prokhorova

Rigorous Development of Safety-Critical Systems